ББК 32.973.26-018.2.75
Т30
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского *В.А. Коваленко* и *Ю.А. Шпака*
Под редакцией *В.А. Коваленко*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

Тейксейра, Стив, **Пачеко**, Ксавье.

Т30 Borland Delphi 6. Руководство разработчика. : Пер. с англ. — М. : Издательский дом "Вильямс", 2002. — 1120 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0305-X (рус.)

Эта книга предназначена для профессиональных разработчиков программного обеспечения в среде Delphi и написана двумя признанными экспертами в этой области. В текст книги включен исходный код множества прекрасных примеров работоспособных приложений по всем обсуждаемым темам, включая примеры приложений рабочего стола, многоуровневых и Web-ориентированных приложений. Каждый пример подробно комментируется, что делает данную книгу отличным учебником, позволяющим быстро освоить создание разнообразных эффективных приложений. В этой книге описана стратегия создания пользовательских приложений, динамических библиотек, применения многопоточного режима, создания специальных компонентов и многого другого. Вы узнаете как с помощью Delphi создаются приложения, в которых используются технологии CLX™, BizSnap™, DataSnap™, SOAP, ASP и беспроводной связи.

В новом издании авторы постарались сохранить дух и традиции прежних изданий книг этой серии, которые, возможно, сделали их наиболее читаемыми книгами о Delphi в мире, а также двухкратным призером читательских симпатий книг по Delphi.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм. Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Sams Publishing.

Authorized translation from the English language edition published by Sams Publishing, Copyright © 2002. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2002

ISBN 5-8459-0305-X (рус.)
ISBN 0-672-32115-7 (англ.)

© Издательский дом "Вильямс", 2002
© Sams Publishing, 2002

Оглавление

Содержание

Предисловие

"Delphi 6 уже два года, для программного продукта это возраст зрелости."

В компании *Borland* я имею удовольствие работать уже более 16 лет. Я пришел сюда летом 1985 года и сначала принимал участие в разработке нового поколения инструментальных средств программирования (системы UCSD языка Pascal и обычных инструментов командной строки было уже недостаточно). Предполагалось, что достигнутое повышение эффективности труда позволит ускорить разработку программ и снизить нагрузку на программистов. В конечном счете это даст возможность им (включая и меня) больше времени проводить с семьей и друзьями и вообще сделает их жизнь богаче и интересней. Я горжусь тем, что принадлежу ко всемирному сообществу разработчиков *Borland*, воплотившему в жизнь на протяжении последних 18 лет столько передовых технологий разработки и нововведений.

Выход пакета Turbo Pascal 1.0 навсегда изменил представление об инструментах программирования. Это произошло в 1983 году. Выход пакета Delphi сделал это еще раз. Среда Delphi 1.0 была разработана для поддержки объектно-ориентированного программирования, программирования в Windows и создания приложений для работы с базами данных. В последующих версиях Delphi была упрощена разработка приложений для Internet и распределенных приложений. Хотя Web-сайт с описаниями основных свойств наших продуктов существует уже много лет и содержит тысячи страниц печатной документации и мегабайты данных интерактивной справочной системы, существует много другой полезной информации, рекомендаций и советов, которые были бы полезны разработчикам для успешной реализации их проектов.

Что же в Delphi 6 появилось такого нового и универсального, обеспечившего ему превосходство над весьма неплохой Delphi 5? Разве Delphi 5 уже не упрощал процесс создания приложений Internet и распределенных приложений при улучшении производительности программирования? Смогла ли группа разработчиков Delphi превзойти себя и снова удовлетворить запросы сегодняшних и завтрашних разработчиков?

Группа разработчиков Delphi потратила более двух лет, выслушивая пожелания заказчиков, рассматривая применение программного продукта разработчиками, выискивая перспективные направления применения программирования в новом тысячелетии. Они сосредоточили свои усилия на радикальном упрощении процесса разработки следующего поколения приложений электронного бизнеса, Web-приложений, XML- и SOAP-ориентированных Web-служб, интеграции с приложениями B2b/B2C/P2P, межплатформенных приложений, распределенных приложений, включающих интеграцию с серверами приложений и EJB, а также на приложениях для Windows ME/2000 и Microsoft Office 2000.

Стив Тейксейра и Ксавье Пачеко сделали это снова. Они переделали свое руководство разработчика так, чтобы вы смогли воспользоваться всеми широчайшими возможностями программирования в Delphi 6.

Я уже многие годы знаю Стива Тейксейру (некоторые называют его T-Rex — "Тиранозавр Rex") и Ксавье Пачеко (некоторые называют его просто X — "кси") как своих друзей, замечательных работников, непременных ораторов на всех ежегодных конференциях и давних членов сообщества пользователей продуктов компании *Borland*.

Предыдущее издание их “Руководства разработчика” было с энтузиазмом встречено пользователями Delphi всего мира. И вот теперь у вас в руках новейшая версия этого издания, способная доставить немало приятных часов увлекательного чтения.

Читайте, учитесь и развлекайтесь вместе с авторами! Любой из продуктов Delphi выпускался для того, чтобы доставить вам радость творчества, ведущего к успеху и достойному вознаграждению.

Дэвид Интерсаймон (David Intersimone) — “Дэвид I”.

Вице-президент по связям с разработчиками Borland Software Corporation.

davidi@borland.com

Об авторах

Стив Тейксейра (Steve Teixeira) — руководитель компании *Zone Labs*, отвечающий за основные технологии и ведущий специалист в области обеспечения безопасности в Internet. Ранее Стив работал главным техническим специалистом в компании *ThinSpace*, занимавшейся программным обеспечением беспроводной и мобильной связи, а также в *Full Moon Interactive*, предлагавшей решения по разработке программ электронного бизнеса. До этого он работал инженером-исследователем в области разработки программного обеспечения в корпорации *Borland* и принимал участие в создании Delphi и C++Builder. Стив известен как автор четырех книг, завоевавших приз читательских симпатий, и многочисленных статей в журналах по разработке программного обеспечения. Его работы переведены на многие языки и опубликованы во всем мире. Стив часто выступает на отраслевых конференциях и международных семинарах.

Ксавье Пачеко (Xavier Pacheco) — президент и главный консультант *Xapware Technologies Inc.*, Колорадо Спрингс, США. Эта фирма специализируется на предоставлении консультаций и обучении. Ксавье часто выступает на отраслевых конференциях и регулярно пишет для периодических изданий статьи, посвященные Delphi. Он пользуется международным признанием как специалист и консультант по Delphi, является автором четырех книг, переведенных на многие языки мира, и завоевавших приз читательских симпатий. Он также является членом особой группы "TeamB" корпорации *Borland*, в задачи которой входит поддержка начинающих пользователей. Вместе со своей женой Анной и дочерью Амандой Ксавье живет в Колорадо Спрингс.

О соавторах

Боб Сворт (Bob Swart), известный также под именем Доктор Боб (Dr.Bob — `www.drbob42.com`), сотрудник филиала *Borland* в Великобритании (*UK Borland Connections*), независимый автор технической литературы, преподаватель и консультант по Delphi, Kylix и C++Builder, проживающий в Хелмонде (Нидерланды). Боб ведет постоянные колонки в журналах *The Delphi Magazine*, *Delphi Developer*, *UK-BUG Developer's Magazine*, а также такие Web-сайты сообщества *Borland*, как `DevX` и `TechRepublic`. Боб написал главы для *The Revolutionary Guide to Delphi 2*, *Delphi 4 Unleashed*, *C++Builder 4 Unleashed*, *C++Builder 5 Developer's Guide*, *Kylix Developer's Guide*, а теперь и для *Delphi 6 Руководство разработчика* (опубликованного в издательстве *Sams*).

Боб часто выступает на международных семинарах на темы, связанные с *Borland*, Delphi и Kylix, а также готовит учебные материалы для своей собственной клиники Delphi доктора Боба (*Dr.Bob's Delphi Clinics*) в Нидерландах и Великобритании.

В свободное время Боб любит посмотреть записи сериалов "Стар Трек" (*Star Trek Voyager*) и "Дальний космос девять" (*Deep Space Nine*) со своим 7-летним сыном *Эриком Марком Паскалем* (Erik Mark Pascal) и 5-летней дочерью *Наташой Луизой Делфиной* (Natasha Louise Delphine).

Дэн Мизер (Dan Miser) — руководитель проекта группы разработчиков Delphi компании *Borland*. Большую часть своего времени он проводит именно там, занимаясь исследованием технологий, находящихся на стадии становления. Дэн также работал в группе Delphi, где его обязанности заключались в разработке технологии DataSnap. Основное внимание Дэн уделяет изысканию беспрепятственных способов доступа к информации, размещенной на разных платформах, что сделало его экспертом в ряде таких компьютерных технологий корпоративного доступа к данным, как MIDAS, SOAP, DCOM, RMI, J2EE, EJB, Struts и RDS. В качестве технического редактора он оказал неоценимую помощь авторам в создании серии книг *Руководства разработчиков Delphi*, писал статьи в журналах, участвовал в группах новостей *Borland* как член группы TeamB, а также выступал на различных конференциях, посвященных COM и MIDAS.

Дэвид Семпсон (David Sampson), инженер группы разработчиков инструментальных средств *Borland*, отвечал за интеграцию CORBA в продукты RAD. Весьма продолжительное время он занимался разработкой на языках Pascal, Delphi и C++. Дэвид — непременный участник конференции разработчиков *Borland*. Вместе с женой он проживает в городе Росвел (штат Джорджия) и наслаждается хоккеем и айкидо, а иногда помогает жене в ее работе с пакетом Basenjis.

Ник Ходжес (Nick Hodges) — старший инженер-разработчик корпорации *Lemanix Corporation* (Сан-Поль, Миннесота). Он является участником группы "TeamB" корпорации *Borland* и длительное время работал разработчиком Pascal и Delphi. В настоящий момент Ник работает в правлении консультационной службы *Borland*, участвует в конференциях и пишет статьи для сайта сообщества *Borland*. Он живет в городе Сан-Поль вместе с женой и двумя дочерьми. Ник любит читать, занимается бегом, а также помогает жене и детям по дому.

Рей Конопка (Ray Konopka) — основатель компании *Raize Software, Inc.* и главный системный архитектор *CodeSite* и *Raize Components*. Рей является автором весьма популярной книги *Developing Custom Delphi Components* и не менее популярной колонки *Delphi by Design* в журнале *Visual Developer Magazine*. Рей специализируется на проектировании пользовательского интерфейса и разработке компонентов Delphi. Кроме того, он является участником международных конференций разработчиков.

Посвящение

Эта книга посвящена жертвам и героям 11 сентября 2001 года.

Благодарю мою семью – Элен, Купера и Райана. Без их любви, поддержки и самоотверженности я никогда, наверное, не смог бы закончить эту книгу, я просто непременно сошел бы с ума.

– Стив

Благодарю мою семью – Энн, Аманду и Захарию – за их любовь, терпение и поддержку.

– Ксавье

Благодарности

Мы благодарим всех, без чьего содействия эта книга никогда не была бы написана. Кроме того, просим никого не винить в ошибках, если таковые будут замечены в книге. В любом случае все обнаруженные ошибки мы относим на свой счет.

Мы выражаем свою исключительную признательность нашим соавторам, использовавшим весь свой опыт разработчиков программного обеспечения и писательское мастерство для того, чтобы эта книга стала тем, чем она является сейчас. Превосходную главу 13, "Разработка компонентов CLX", написал сам мистер Компонент, *Рей Конопка* (Ray Konopka). Гуру DataSnap, *Дэн Мизер* (Dan Miser), блестяще сподобился написать главу 21, "Разработка приложений DataSnap". Признанный эксперт CORBA, *Дэвид Сампсон* (David Sampson), пожертвовал на общее благо главу 19, "Разработка приложений CORBA". Спасибо тебе, "Доктор Боб", он же *Роберт Сварт* (Robert Swart), за твою талантливую и основополагающую главу 22, "Разработка приложений ASP". И, наконец (но не в последнюю очередь!), "Web-Мастер" — *Ник Ходжес* (Nick Hodges) снова с нами в главе 23, "Разработка приложений WebSnap".

Прежде всего мы хотим поблагодарить наших технических рецензентов и хороших друзей — *Томаса Теобалда* (Thomas Theobald) и *Джона Томаса* (John Thomas). Эти парни сумели уложиться в жесткий график работы несмотря на то, что технические специалисты *Borland* завалили их поистине громадным объемом программного обеспечения.

Во время создания этой книги мы получили множество советов и рекомендаций от своих друзей и сотрудников. В их числе (в алфавитном порядке) "Лино" *Алан Тадрос* (Alan Tadros), *Андерс Хежлсберг* (Anders Hejlsberg), *Андерс Олсон* (Anders Ohlsson), *Шарль Калверт* (Charlie Calvert), *Виктор Хорнбак* (Victor Hornback), *Чак Джаздзевский* (Chuck Jazdzewski), *Дэниел Полищук* (Daniel Polistchuck), *Дэнни Торп* (Danny Thorpe), *Дэвид Стривер* (David Streever), *Элли Питерс* (Ellie Peters), *Джефф Питерс* (Jeff Peters), *Ланс Буллок* (Lance Bullock), *Марк Дункан* (Mark Duncan), *Майк Дуган* (Mike Dugan), *Ник Ходжес* (Nick Hodges), *Паульь Квалльс* (Paul Qualls), *Рич Джонс* (Rich Jones), *Роланд Бюшер* (Roland Bouchereau), *Скотт Фролич* (Scott Frolich), *Стив Биб* (Steve Beebe), *Том Бутт* (Tom Butt) и многие другие — мы просто не в состоянии всех перечислить, но пиво мы им должны.

И, наконец, огромное спасибо всей бригаде, работавшей над книгой: *Кэрол Аккерман* (Carol Ackerman), *Кристине Смит* (Christina Smith), *Дэну Шерфа* (Dan Scherf) и многим другим, которые скромно трудились за сценой, но без которых книга никогда не стала бы реальностью.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать электронное письмо или просто посетить наш Web-сервер, оставив свои замечания, — одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Введение

Вы держите в руках экземпляр пятого издания серии *Delphi Руководство разработчика* — результат поистине тысяч человеко-часов труда в течение более чем семи лет. Стив и Ксавье входили в состав первой группы разработчиков проекта Delphi корпорации *Borland*, а эта работа — итог их более чем пятнадцатилетней совместной деятельности и обобщение огромного опыта разработки программного обеспечения на языке Delphi. В новом издании *Delphi 6 Руководство разработчика* мы постарались сохранить дух и традиции прежних изданий этой серии, которые, возможно, сделали ее наиболее читаемыми книгами о Delphi в мире, а также двухкратным лауреатом приза читательских симпатий книг по Delphi. Эта книга написана разработчиками для разработчиков.

Нам хотелось, чтобы *Delphi 6 Руководство разработчика* органично вошла в подборку книг из серии *Delphi Руководство разработчика*, а также дополнила и развила написанное ранее. Было бы просто замечательно, если бы удалось собрать в одной книге (слегка модифицировав), то, что было изложено в *Delphi 5 Руководство разработчика*, и все новое, относящееся к Delphi 6. Но прежняя книга и так была уже достаточно объемной, а полиграфическая техника имеет, к сожалению, ограничения на их размер. Чтобы избежать проблем, связанных с объемом книги, при этом изложив наиболее ценный материал, раскрывающий все возможности Delphi 6, мы решили опубликовать новую книгу с новой информацией.

Delphi 6 Руководство разработчика содержит несколько абсолютно новых глав, большинство прежних глав было существенно расширено и дополнено, а некоторые из наилучших разделов предыдущего издания вошли практически без изменений. Но информация книги *Delphi 5 Руководство разработчика* не была потеряна полностью, она содержится на CD, прилагаемом к этой книге. В результате, в одной книге содержится по существу две.

Delphi 6 Руководство разработчика разделена на шесть разделов. Часть I, "Основные направления программирования", содержит все основы знаний, необходимые для того, чтобы стать настоящим разработчиком Delphi. Часть II, "Профессиональное программирование", освещает некоторые наиболее общие проблемы программирования нового типа. Таких, например, как потоки и динамически связанные библиотеки. Часть III, "Разработка баз данных", обсуждает многообразие способов доступа к данным в Delphi. Часть IV, "Компонент-ориентированная разработка", демонстрирует разнообразные проявления компонент-ориентированной разработки, включая VCL, CLX, пакеты, COM и API Open Tools. Часть V, "Разработка корпоративных приложений", позволяет приобрести практические знания, необходимые для разработки корпоративных приложений, оснащенных такими технологиями, как COM+, CORBA, SOAP/BizSnap и DataSnap. И, в заключение, часть VI, "Программирование для Internet", демонстрирует пример разработки на Delphi приложений для Internet и беспроводной связи.

Для кого написана эта книга

Как следует из названия, книга предназначена для разработчиков. Поэтому, если вы — разработчик программного обеспечения и используете Delphi, то эта книга — для вас. Мы предполагали, что книга заинтересует три основные группы читателей:

- Разработчиков на Delphi, желающих повысить свой уровень.
- Опытных программистов на языках Pascal, C/C++, Java или Basic, намеревающихся приступить к работе на Delphi.
- Программистов, работающих на языке Delphi, но желающих научиться пользоваться всеми его дополнительными возможностями, а также не всегда очевидными особенностями.

Соглашения, принятые в этой книге

В данной книге использованы следующие типографские соглашения:

- Выводимые на экран сообщения, примеры команд, имена переменных, любой код, присутствующий в тексте, а также унифицированные локаторы ресурсов (URL) Web-страниц будут представлены моноширинным шрифтом.
- Все, что придется вводить с клавиатуры, представлено **полужирным моноширинным** шрифтом.
- Знакоместо в описаниях синтаксиса выделяется курсивом. Это указывает на необходимость заменить знакоместо фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте `BINDSIZE=(`*`максимальная ширина колонки`*`)*(`*`номер колонки`*`)`.
- Пункты меню и названия диалоговых окон представлены следующим образом: Menu Option (Пункт меню).
- Процедуры и функции содержат скобки `()` после их имени. Хотя это и не является стандартным синтаксисом языка Pascal, но помогает отделить функции и процедуры от свойств, переменных и типов.

Текст некоторых абзацев этой книги выделен специальным шрифтом. Это примечания, советы и предостережения, которые помогут обратить внимание на наиболее важные моменты в изложении материала и избежать ошибок при работе.

На прилагаемом компакт-диске находятся все исходные тексты и файлы описанных проектов, а также исходный код примеров, текст которых мы не смогли включить в книгу. CD также содержит несколько мощных утилит и компонентов сторонних производителей. Кроме того, исходные тексты и файлы проектов, представленных в главах, можно увидеть на Web-сайте издательского дома "*Вильямс*" по адресу: `www.williamspublishing.com`.

Дополнения, исправления и обновления книги можно найти на Web-сайте по адресу: `http://www.xapware.com/ddg`.

Итак, приступим

Нам иногда задают вопрос: что заставляет вас писать все новые книги, посвященные Delphi? Это трудно объяснить, но, наверное, хотя бы то, что, когда встречаешься с другим разработчиком Delphi и видишь у него в руках свою потрепанную книгу, понимаешь — жизнь потрачена не зря.

ЧАСТЬ

I

Основные направления программирования

В ЭТОЙ ЧАСТИ...

ГЛАВА

1

Программирование в Delphi

В ЭТОЙ ГЛАВЕ...

В этой главе содержится краткий обзор языка Delphi, в том числе история его развития, основные компоненты и возможности, а также рассказ о той роли, которую играет язык Delphi в мире разработки приложений для Windows. Здесь излагается в общих чертах описание той информации, которой необходимо овладеть, чтобы стать квалифицированным разработчиком на языке Delphi. Помимо сугубо технических характеристик настоящая глава содержит описание *интегрированной среды разработки* (IDE — Integrated Development Environment) Delphi, ее возможностей, достоинств и недостатков, а также ряда настолько малоизвестных особенностей, о которых, возможно, не слышали даже опытные разработчики Delphi.

Эта глава не предназначена для обучения основам разработки программ в среде Delphi. Мы полагаем, что читатель приобрел данную книгу, чтобы узнать что-то новое и интересное, а не просто освежить в памяти содержимое документации, предоставляемой компанией *Borland*. Исходя из этого, мы считаем своей задачей продемонстрировать наиболее мощные возможности данного продукта и пояснить, как их можно использовать для разработки программ коммерческого уровня. Мы надеемся, что чтение настоящей главы (и всей этой книги!) окажется полезным как опытным, так и начинающим разработчикам. В отношении последних необходимо заметить, что освоение Delphi следует начинать не с чтения данной главы, а с рассмотрения документации *Borland* и создания простейших примеров. Изучение настоящей главы принесет вам пользу лишь после уяснения основных механизмов работы IDE и общей процедуры разработки приложений.

Семейство продуктов Delphi

Поставка комплекта разработчика Delphi 6 осуществляется в трех различных модификациях: Delphi 6 Personal (персональная), Delphi 6 Professional (профессиональная) и Delphi 6 Enterprise (корпоративная). Каждый из этих вариантов рассчитан на определенный тип разработчика.

Delphi 6 Personal представляет собой самую простую версию. Она содержит все необходимое для того, чтобы начать создание приложений в среде Delphi. Эта версия идеальна для любителей или студентов (кроме всего прочего, она самая дешевая). Версия включает:

- оптимизирующий 32-разрядный компилятор Object Pascal (в том числе ряд новейших дополнений, расширяющих возможности языка);
- библиотеку визуальных компонентов (VCL — Visual Component Library), содержащую более 85 стандартных компонентов, размещаемых на панели компонентов Component Palette;
- поддержку пакетирования (package support), позволяющую создавать небольшие исполняемые файлы и библиотеки компонентов;
- объектно-ориентированную среду разработки (IDE), в которую входят редактор, отладчик, конструктор форм и набор инструментов повышения производительности;
- дополнение IDE функциями наследования и связывания визуальных форм, древовидным представлением объектов, контролем классов и интерактивной подсказкой (CodeInsight);

- полную поддержку интерфейса API Win32, в том числе COM, GDI, DirectX, многопоточность и различные комплекты разработчика программного обеспечения (SDK — Software Development Kit) от *Microsoft* и других производителей;
- лицензию, позволяющую создавать приложения только для личного использования: никакое коммерческое распространение приложений, созданных на Delphi 6 Personal, не допускается.

Delphi 6 Professional предназначен для профессиональных разработчиков, не использующих корпоративные технологии. Если вы — профессиональный разработчик, занимающийся созданием и распространением приложений или компонентов Delphi, то этот продукт — именно для вас. Помимо всего того, что входит в состав версии Personal, в версию Delphi 6 Professional включены:

- более 225 дополнительных компонентов VCL в составе палитры компонентов;
- более 160 дополнительных компонентов CLX для разработки межплатформенных приложений (для Windows и Linux);
- поддержка баз данных, в том числе архитектура баз данных DataCLX, элементы управления VCL для работы с данными, межплатформенные компоненты и драйверы dbExpress, объекты данных ActiveX (ADO — ActiveX Data Objects), процессор баз данных Borland (BDE — Borland Database Engine) для совместимости с прежними версиями, архитектура виртуальных структур данных (virtual dataset architecture), позволяющая включать в VCL другие типы базы данных, инструмент Database Explorer (проводник баз данных), хранилище данных, а также компоненты InterBase и InterBase Express;
- драйверы InterBase и MySQL для dbExpress;
- архитектура баз данных DataCLX (известная ранее как MIDAS) в комплекте с XML-ориентированным локальным процессором данных MyBase;
- набор мастеров для создания компонентов COM/COM+, таких, например, как элементы управления ActiveX, ActiveForms, серверов автоматизации, страниц свойств и транзакционных компонентов;
- набор инструментальных средств и компонентов стороннего производителя, в том числе инструменты для Internet INDY, инструмент разработки отчетов QuickReports, графические компоненты TeeChart предназначенные для представления данных в виде графиков и диаграмм, а также элементы управления NetMasters FastNet;
- сервер баз данных InterBase 6 с лицензией для пяти пользователей;
- инструмент Web Deployment, облегчающий размещение в Web содержимого компонентов ActiveX;
- инструмент установки приложений InstallShield MSI Light;
- интерфейс API OpenTools для разработки компонентов, интегрированных со средой Delphi, а также с интерфейсом системы контроля версий PVCS;
- утилита WebBroker из набора NetCLX, а также компоненты разработки межплатформенных приложений для Internet;

- исходный код для *библиотеки визуальных компонентов* (VCL — Visual Component Library), *библиотеки межплатформенных компонентов* (CLX — Component Library for Cross-platform), *динамической библиотеки* (RTL — Runtime Library) и редакторов свойств;
- лицензия на коммерческое распространение приложений, разработанных с помощью Delphi 6 Professional.

Delphi 6 Enterprise предназначен для профессиональных разработчиков, создающих приложения масштаба предприятия. Версия Enterprise обладает всеми вышеперечисленными возможностями, а также включает следующие дополнительные компоненты:

- более 300 дополнительных компонентов VCL в составе палитры компонентов;
- технологию BizSnap, предназначенную для создания XML-ориентированных приложений и Web-служб;
- платформу WebSnap, предназначенную для разработки Web-приложений, интегрированных с XML, поддерживающих технологии Web-сценариев (scripting) и т.д.;
- поддержку CORBA для приложений клиент/сервер, в том числе ORB VisiBroker версии 4.0x и Borland AppServer версии 4.5;
- пакет TeamSource — программное обеспечение для управления групповой разработкой программ, допускающее одновременную работу с несколькими версиями продуктов (в том числе поддержку ZIP и PVCS);
- инструменты для перевода и локализации приложений;
- драйверы SQLLinks BDE для Oracle, MS SQL Server, InterBase, Informix, Sybase и DB2;
- драйверы Oracle и DB2 для dbExpress;
- дополнительные инструментальные средства для создания приложений, ориентированных на применение SQL, в том числе SQL Explorer, SQL Monitor, SQL Builder, а также поддержку таблиц ADT.

Delphi: что почем

Нам часто задают вопросы: “В чем главные достоинства Delphi?” или “Чем Delphi превосходит пакет *X*?”. За прошедшие годы мы выработали два варианта ответов на эти вопросы — короткий и длинный. Первый из них заключается в одном слове — “*продуктивность*”. Просто на сегодняшний день работа в Delphi является самым продуктивным методом создания приложений для Windows. Безусловно, существуют категории лиц (начальство и возможные клиенты), для которых такого ответа будет недостаточно. В этом случае мы даем длинный ответ, где подробно описывается сочетание особенностей среды Delphi, делающее ее столь продуктивной. Мы полагаем, что общая продуктивность любых инструментов создания программного обеспечения определяется следующими пятью важнейшими аспектами:

- качеством визуальной среды разработки;
- скоростью работы компилятора и быстродействием откомпилированных программ;

- мощностью языка программирования и его сложностью;
- гибкостью и масштабируемостью используемой архитектуры баз данных;
- наличием поддерживаемых средой разработки шаблонов проектирования и использования.

Безусловно, существует еще немало важных факторов — например, вопросы установки, документация, поддержка сторонних производителей и т.д. Тем не менее, мы пришли к выводу, что и этой упрощенной модели вполне достаточно для объяснения, почему имеет смысл остановить свой выбор на Delphi. Некоторые из упомянутых выше категорий связаны с определенной субъективностью оценки. Как же можно использовать их для оценки продуктивности определенного инструмента разработки? Предлагаемая схема проста. Оцените каждый из пяти показателей анализируемых пакетов по пятибалльной шкале и нанесите соответствующие точки на оси графика, представленного на рис. 1.1. Соедините точки для каждого из пакетов линиями — получится несколько пятиугольников. Чем больше площадь получившегося пятиугольника, тем выше продуктивность данного инструмента разработки.

Рис. 1.1. *Схема построения диаграмм для оценки продуктивности инструментов разработки приложений*

Мы не пытаемся с помощью предложенной формулы навязать свое заранее подготовленное решение — выбор остается за вами! Мы просто предлагаем подробнее остановиться на каждом из аспектов вышеприведенной схемы и оценить соответствующие показатели Delphi в сравнении с другими инструментами разработки приложений для Windows.

Качество визуальной среды разработки

Обычно, визуальная среда разработки состоит из трех взаимосвязанных компонентов: редактора, отладчика и конструктора форм. В любом из современных инструментов *ускоренной разработки приложений* (Rapid Application Development – RAD) эти три компонента должны гармонично взаимодействовать друг с другом. При работе в конструкторе форм Delphi неявно генерирует программный код тех компонентов, которые размещаются или обрабатываются в формах. В окне редактора в код автоматически созданной программы можно внести необходимые дополнения, определяющие специфическое поведение данного приложения. Здесь же, в окне редактора, можно отладить код, внося точки останова, точки просмотра (watches) и т.д.

Редактор Delphi обычно используется параллельно с другими инструментами. Пожалуй, наиболее мощным из них можно считать технологию CodeInsight (интерактивную подсказку), позволяющую существенно уменьшить объем кода, вводимого с клавиатуры. Этот инструмент построен на использовании информации компилятора, а не библиотеки типов (как в Visual Basic), поэтому область его применения значительно шире. Хотя редактор Delphi поддерживает достаточный набор параметров настройки, следует отметить, что возможности настройки редактора пакета Visual Studio несколько шире.

Современная версия отладчика Delphi поддерживает весь набор функциональных возможностей, присущих отладчику пакета Visual Studio. К вновь добавленным функциям относятся средства удаленной отладки, подключения процессов, отладки пакетов и библиотек DLL, средства контроля значений автоматических локальных переменных и поддержки окна CPU. Кроме того, Delphi предоставляет удобные средства управления графической средой отладки. Они позволяют в ходе отладки размещать и объединять окна в любом удобном месте, а также запоминать сведения о полученной конфигурации в виде поименованной группы параметров настройки рабочего стола. Одна из чрезвычайно удобных функций отладчиков, которая широко распространена в среде интерпретаторов (таких как Visual Basic или Java), заключается в возможности изменять программный код и, следовательно, поведение приложения непосредственно в процессе его отладки. К сожалению, в среде компиляторов реализация подобных функций связана с очень большими трудностями, поэтому в нынешней версии она отсутствует.

Конструктор форм является обязательной принадлежностью всех инструментов RAD, включая Delphi, Visual Basic, C++ Builder и PowerBuilder. Классический вариант среды разработки (например Visual C++ и Borland C++) обычно содержит редакторы диалогов, однако эти инструменты менее удобны для интеграции в рабочий поток создания приложения, чем конструкторы форм. Рассмотрев представленную на рис. 1.1 диаграмму, можно сделать вывод, что отсутствие конструктора форм оказывает заметное негативное влияние на общие показатели продуктивности конкретного инструмента разработки приложений.

В последние годы Delphi и Visual Basic оказались втянутыми в постоянную борьбу за первенство в разнообразии возможностей их конструкторов форм. Каждая выпущенная версия этих продуктов просто изумляет своими новыми функциональными возможностями. Например, в конструкторе форм Delphi используется полностью объектно-ориентированная схема построения. В результате внесенные в базовые классы изменения немедленно распространяются и на все классы, производные от них. Этот механизм использован для реализации функции *наследования визуальных*

форм (Visual Form Inheritance — VFI). VFI позволяет динамически порождать новые формы из любых других форм проекта или галереи форм. Кроме того, внесенные в базовую форму изменения будут немедленно распространены и унаследованы всеми ее формами-потомками. Более подробная информация по этой теме приведена в главе 3, "Приключения сообщения".

Скорость работы компилятора и быстродействие откомпилированных программ

Быстрый компилятор позволяет разрабатывать программное обеспечение поэтапно, поскольку допускает многократное внесение небольших изменений в исходную программу, с последующей перекомпиляцией и тестированием. Вследствие этого возникает весьма эффективный цикл разработки. Более медленный компилятор вынуждает разработчика одновременно вносить бо́льший объем изменений, комбинируя несколько отдельных доработок в одном цикле компиляции и отладки. Это, безусловно, снижает эффективность отдельных циклов разработки. Преимущества, достигаемые за счет повышенной эффективности работы откомпилированных программ, очевидны. В любом случае, чем быстрее работает программа и чем меньшее ее объектный код, тем лучше.

Вероятно, наиболее известное преимущество используемого в Delphi компилятора языка Pascal состоит в его быстродействии. Фактически это — самый быстрый компилятор языка высокого уровня из всех, существующих в среде Windows. В последние годы отмечаются заметные улучшения в работе компиляторов языка C++, который традиционно считался самым медленным в смысле скорости компилирования. Успехи были достигнуты за счет пошагового связывания и различных стратегий кэширования, используемых, в частности, в пакетах Visual C++ и C++ Builder. Тем не менее, даже эти улучшенные компиляторы языка C++ работают в несколько раз медленнее, чем компилятор Delphi.

Означает ли столь высокая скорость компилирования обязательное отставание в эффективности создаваемых программ? Безусловно, ответом на этот вопрос будет "Нет". Создаваемый в Delphi объектный код имеет те же показатели эффективности, что и объектный код, созданный транслятором C++ Builder. Отсюда можно сделать вывод, что качество создаваемых программ соответствует уровню, обеспечиваемому очень хорошим компилятором языка C++. По последним достоверным оценкам производительности, программы, созданные компилятором Visual C++, действительно имеют самые высокие показатели скорости выполнения и размеров кода. В основном, это достигается за счет очень хорошей оптимизации. Хотя эти небольшие преимущества и незаметны во время разработки обычных приложений, они могут иметь очень большое значение при создании программ, выполняющих значительный объем вычислений.

С точки зрения используемой технологии компилирования язык Visual Basic уникален. В ходе разработки приложения Visual Basic (VB) используется в интерпретирующем режиме, обеспечивая достаточную скорость работы. При необходимости распространения созданного приложения можно воспользоваться компилятором VB, результатом работы которого является исполняемый файл (`*.EXE`). Этот компилятор весьма медлителен, и его показатели сильно отстают от возможностей инструментов

C++ и Delphi. Во время создания данной книги *Microsoft* подготовила новое поколение компиляторов, и Visual Basic.NET обещает занять достойное место в этом ряду.

Еще одним интересным вариантом является язык Java. Лучшие инструменты этой языковой среды (например, JBuilder и Visual J++) демонстрируют время компиляции, сравнимое с Delphi. Но эффективность создаваемых программ чаще всего оставляет желать лучшего по той простой причине, что язык Java является интерпретируемым языком. Хотя развитие возможностей инструментов Java происходит быстрыми темпами, скорость выполнения создаваемых на нем программ в большинстве случаев сильно уступает Delphi и C++.

Мощность языка программирования и его сложность

Мощность и сложность языка в значительной степени определяются точкой зрения собеседника, поэтому эти категории часто служат поводом для проведения многочисленных перепалок и ожесточенных дискуссий в группах новостей и списках рассылки. То, что совсем просто для одного человека, может оказаться весьма сложным для другого. В свою очередь то, что воспринимается одним человеком как ограничение, может расцениваться другим как самое изящное решение. Поэтому приведенные ниже рассуждения являются изложением точки зрения авторов и отражают их личный опыт и предпочтения.

Наиболее мощным из всех языков является ассемблер. Едва ли существует что-то такое, чего нельзя выполнить с его помощью. Однако создание даже самого простого приложения Windows на ассемблере является весьма сложным заданием, а полученный результат почти наверняка будет содержать ошибки. Кроме того, чаще всего практически невозможно обеспечить сопровождение программ на ассемблере группой разработчиков на сколько-нибудь продолжительный период времени. По мере того как код программ передается от одного исполнителя к другому, выбранные проектные решения и методы становятся все более туманными, и так происходит до тех пор, пока код программы не приобретает совершенно непонятный вид, больше напоминающий священные тексты на санскрите, а не обычную компьютерную программу. Следовательно, в рассматриваемой категории ассемблеру следует поставить очень низкую оценку, несмотря на всю его мощь. Главная причина — чрезмерная сложность использования этого языка для достижения тех целей, которые стоят перед большинством разработчиков приложений.

C++ также является очень мощным языком. С помощью его действительно эффективных инструментов, подобных макросам препроцессора, шаблонам, перегрузке операторов, можно даже создать собственный язык в пределах C++. Если предоставленный разработчикам исключительно широкий набор функциональных возможностей будет использоваться продуманно, то это позволит создавать очень ясные и простые в сопровождении программы. Однако проблема состоит в том, что большинство разработчиков не может противостоять искушению чрезмерного и неоправданного использования существующих возможностей, а это часто приводит к появлению громоздких программ. Фактически на C++ писать плохие программы гораздо легче, чем хорошие, поскольку сам язык не ориентирует разработчика на использование хороших приемов программирования, оставляя такие вопросы полностью на его усмотрение.

Существует два языка, которые, по нашему мнению, очень схожи в том, что в них достигнут оптимальный баланс между сложностью и мощностью. Это — Object Pascal и Java. В обоих языках использован подход, предусматривающий ограничение доступных функциональных возможностей, что позволяет разработчикам перенести основные усилия на логику создаваемых приложений. Например, в обоих языках отсутствует "очень объектно-ориентированная", но способствующая различным злоупотреблениям концепция множественного наследования. В обоих случаях она заменяется реализацией в классах нескольких различных интерфейсов. Оба языка исключают изящную, но в то же время весьма опасную функцию перегрузки операторов. Кроме того, в обоих случаях исходные файлы рассматриваются как основные объекты языка, а не как сырье для компоновщика. Более того, в обоих языках используются такие мощнейшие возможности, как обработка исключений, информация о типах времени выполнения (RTTI — Runtime Type Information) и строковые ресурсы. Возможно, это совпадение, но оба языка были созданы не огромным коллективом разработчиков, а одним человеком или небольшей группой внутри единой организации, имеющей вполне определенное представление о том, что именно должен представлять собой создаваемый язык.

Изначально Visual Basic (простейший) был создан как язык достаточно простой, чтобы начинающие программисты могли быстро его освоить. Но по мере добавления в него новых возможностей, являвшихся ответом на неотложные требования времени, этот язык становился все более сложным. Несмотря на все усилия, прилагаемые для того, чтобы позволить разработчику избежать обременительных подробностей программирования, язык Visual Basic по-прежнему содержит ряд препятствий, которые приходится преодолевать при создании достаточно сложных приложений. Нынешняя версия Visual Basic.NET от *Microsoft*, являющаяся следующим поколением этого продукта, претерпела в данном направлении существенные изменения (в основном за счет совместимости с прежними версиями).

Гибкость и масштабируемость архитектуры баз данных

Поскольку в компании *Borland* отсутствует собственная линия продуктов управления базами данных, в состав Delphi входит инструментарий, который, на наш взгляд, обеспечивает самую гибкую архитектуру поддержки баз данных, по сравнению со всеми остальными представленными на рынке. Безусловно, dbExpress очень эффективен (хоть и за счет дополнительных функциональных возможностей), но выбор драйверов довольно ограничен. Механизм BDE успешно работает и обеспечивает достаточную для большинства типов приложений производительность при взаимодействии с широким диапазоном баз данных, хотя *Borland* постепенно и сокращает этот диапазон. Кроме того, встроенные компоненты ADO обеспечивают эффективные средства связи с базами через ADO или ODBC. Если при каких-либо обстоятельствах InterBase не справится с задачей, то можно воспользоваться его базовым компонентом — IBExpress, который обладает значительно большей эффективностью и способностью установить связь практически с любым сервером баз данных. Если ни один из названных способов не позволяет обеспечить доступ к данным, то остается возможность самостоятельно написать собственный класс доступа к данным, исполь-

зующий абстрактную архитектуру структуры данных, либо приобрести готовое решение у стороннего производителя. При этом DataCLX облегчит доступ к любому из таких источников данных, несмотря на то, что они могут быть разделены на несколько уровней логически или физически.

Следует отметить, что инструменты разработки *Microsoft* логически сфокусированы на поддержке собственных баз данных *Microsoft* и предоставляют соответствующие решения для доступа к их данным, включая средства ODBC, OLE DB и т.д.

Поддержка средой разработки шаблонов проектирования и использования

Это "чудодейственное снадобье" и "святой грааль" всех технологий разработки программного обеспечения, похоже, совершенно игнорируется другими инструментами разработчика. Хотя все элементы Delphi необходимы и важны, самым существенным из них является все-таки библиотека VCL. Возможность манипулирования компонентами непосредственно в процессе проектирования, средства разработки собственных компонентов, наследующих элементы своего поведения от других компонентов с помощью различных объектно-ориентированных технологий, — все это является важнейшими условиями высокого уровня продуктивности, свойственного среде Delphi. При разработке компонентов VCL всегда можно выбрать подходящую к определенному случаю технологию объектно-ориентированного проектирования из числа предоставляемых. Другие среды разработки, поддерживающие работу с компонентами, часто либо слишком жесткие, либо слишком сложные.

Например, элементы управления ActiveX предоставляют практически те же самые возможности, что и компоненты VCL, однако создать новый класс, являющийся производным от элемента управления ActiveX, нельзя. Традиционные среды разработки, обеспечивающие работу с классами (например OWL или MFC), обычно требуют от разработчика глубокого знания их внутренних механизмов. Только в этом случае работа в их среде может быть достаточно продуктивной. Всем им не хватает определенных инструментов поддержки функций проектирования. В настоящий момент *Microsoft* выпустила новую общую библиотеку .NET (.NET common library), что, безусловно, выдвинуло ее на передний план в области компонент-ориентированной разработки. Кроме того, технология .NET оказалась совместима даже с рядом их прежних инструментальных средств, включая C#, Visual C++ и Visual Basic.

Немного истории

Сердцем Delphi является компилятор Pascal. По сути, Delphi представляет собой очередной шаг в эволюции компиляторов Pascal, продолжающейся с тех времен, когда *Андерс Хейлсберг* (Anders Hejlsberg) создал первый компилятор Turbo Pascal. С тех пор прошло уже 17 лет, и программисты не устают восхищаться надежностью, изяществом и, конечно же, скоростью работы компиляторов Pascal от *Borland*. Delphi 6 — не исключение. В нем воплощен более чем десятилетний опыт разработки компиляторов, превративший этот 32-разрядный оптимизирующий компилятор в настоящее произведение искусства. Хотя с течением времени возможности компиляторов постоянно увеличивались, скорость их работы осталась практически неизменной. Более

того, стабильность компилятора Delphi продолжает оставаться эталоном, с которым сравнивают все остальные инструменты разработки.

А теперь настало время для небольшого экскурса в историю, в течение которого мы рассмотрим особенности каждой из версий Delphi, а также кратко остановимся на характерных чертах исторического контекста, сложившегося на момент их выпуска.

Delphi 1

Во времена DOS, которые стали уже историей, программисты стояли перед нелегким выбором между продуктивным, но неэффективным BASIC и эффективным, но непродуктивным ассемблером. Появление компилятора Turbo Pascal, который сочетал простоту структурированного языка программирования с эффективностью настоящего компилятора, во многом разрешило эти проблемы. Перед программистами, работающими под Windows 3.1, стоит выбор, что предпочесть мощный, но сложный и требующий знаний C++ или простой, однако крайне ограниченный Visual Basic? Delphi 1 предложил радикально новый подход к разработке приложений в среде Windows: простой язык, визуальная разработка приложений, создание откомпилированных выполняемых файлов, динамических библиотек, баз данных и многое другое. Delphi 1 был первым инструментом разработки приложений Windows, объединившим в себе оптимизирующий компилятор, визуальную среду программирования и мощные возможности работы с базами данных. Все это вместе впоследствии получило название среды *быстрой разработки приложений* (RAD — Rapid Application Development).

Сочетание компилятора, инструментов RAD и быстрого доступа к базам данных было совершенно неотразимым для множества разработчиков в среде Visual Basic, поэтому Delphi приобрел массу почитателей. Многие из разработчиков, работавших с Turbo Pascal, перешли к работе в среде этого нового инструмента автоматически. Распространилось мнение, что Object Pascal — это уже не тот язык, с которым нас заставляли работать в колледже и который оставлял впечатление, что у работающего с ним связаны руки. Многие из разработчиков перешли к Delphi, чтобы воспользоваться преимуществами надежных элементов визуальной разработки, дополненных мощным языком и необходимыми инструментами. Visual Basic компании *Microsoft* явно проиграл соревнование с Delphi, к выходу которого разработчики Visual Basic оказались совершенно неподготовленными. Медленный, раздутый и ограниченный, Visual Basic 3 ничего не мог противопоставить Delphi 1.

Это был 1995 год. Компания *Borland* выплатила громадную компенсацию компании *Lotus* в связи с судебным иском по использованию элементов интерфейса приложения Lotus 1-2-3 в приложении Quattro. Кроме того, компания *Borland* подвергалась атакам со стороны *Microsoft* за то, что сделала попытку утвердиться в той области коммерческих приложений, которую *Microsoft* считала своей собственностью. Чтобы разрядить ситуацию, *Borland* продает права на Quattro компании *Novell* и нацеливает разработчиков dBASE и Paradox на удовлетворение нужд профессиональных разработчиков баз данных, а не на случайных пользователей-непрофессионалов. Пока *Borland* разбиралась с рынком своих приложений, *Microsoft* спокойно выравнивала положение дел на собственной платформе, стараясь привлечь к своим продуктам тех разработчиков в среде Windows, которые уже использовали продукты *Borland*. Когда внимание *Borland* вновь сосредоточилось на вопросах конкурентной борьбы между приложениями,

предназначенными для разработчиков, выяснилось, что она утратила часть рынка, ранее принадлежавшего Delphi и новой версии Borland C++.

Delphi 2

Годом позже в Delphi 2 было предложено все то же, но на новом уровне современной 32-разрядной операционной системы Windows 95 и Windows NT. Кроме того, Delphi 2 предоставил программисту 32-битовый компилятор, создававший более быстрые и эффективные приложения, мощные библиотеки объектов, улучшенную поддержку баз данных, поддержку OLE, средства Visual Form Inheritance, и при этом обеспечивал совместимость со старыми 16-разрядными приложениями. Delphi 2 стал тем мерилом, по которому равнялись другие RAD.

Это был 1996 год. Наиболее важный этап истории развития операционной системы Windows после выпуска в свет версии 3.0 — 32-разрядная Windows 95, которая появилась на рынке в конце 1995 года. *Borland* твердо намеревалась сделать Delphi пакетом, превосходящим все остальные инструменты разработчика для данной платформы. Интересным историческим фактом является то, что Delphi 2 исходно получил название *Delphi32* — это должно было подчеркнуть тот факт, что он создавался специально для 32-разрядной среды Windows. Однако перед самым выпуском название продукта было изменено на Delphi 2. Предполагалось, что это отметит тот факт, что Delphi 2 является самостоятельным законченным продуктом, а не просто вариантом Delphi 1 для новой платформы.

Microsoft сделала попытку ответить на вызов, выпустив Visual Basic 4, но он обладал низкой производительностью, не обеспечивал совместимость 16- и 32-разрядных приложений, а также имел несколько других заметных недостатков. Тем не менее, впечатляющее количество разработчиков по тем или иным причинам продолжало использовать Visual Basic. Помимо всего прочего, компания *Borland* хотела, чтобы Delphi вышел на рынок высокопроизводительных приложений среды клиент-сервер, занятый такими приложениями, как PowerBuilder. Но эта версия не обладала необходимой мощностью, чтобы сколько-нибудь заметным образом потеснить те продукты, которые полностью захватили корпоративный сектор рынка.

К тому времени компания *Borland* вынуждена была сосредоточить свои интересы на корпоративных пользователях. Это решение во многом было продиктовано сокращением рынка dBASE и Paradox, а также уменьшением доходов, поступающих от продуктов C++. При подготовке к такому сложному шагу *Borland* допустила ошибку, заключавшуюся в приобретении компании *Open Environment Corporation*, которая специализировалась на программном обеспечении среднего уровня и выпустила два продукта. Один из них стал предшественником CORBA, а другой, поддерживавший распределенную технологию OLE, был впоследствии вытеснен DCOM.

Delphi 3

Если в процессе создания Delphi 1 команда разработчиков была занята преимущественно проектированием и реализацией основных инструментов среды разработки, то при создании Delphi 2 основная работа состояла в переходе на 32-разрядную платформу (с одновременным сохранением практически полной совместимости с предыдущей версией). Кроме того, создавались новые средства поддержки баз данных с архитектурой клиент-

сервер, необходимые для выхода на корпоративный рынок. В процессе работы над Delphi 3 команде разработчиков было поручено расширить набор инструментов для того, чтобы обеспечить самые широкие возможности выбора решений тех проблем, с которыми постоянно сталкивались разработчики в среде Windows. В частности, в Delphi 3 было существенно упрощено использование таких сложных технологий, как COM и ActiveX, добавлены средства разработки приложений для World Wide Web, включены средства создания тонких клиентов ("thin client") приложений, а также поддержка баз данных с многоуровневой архитектурой. Модернизация инструмента CodeInsight позволила упростить процесс написания программ, хотя в остальной части используемые для создания приложений технологии остались теми же, что и в Delphi 1.

Это был 1997 год, и конкурентная борьба приняла особенно напряженную форму. На рынок простых приложений *Microsoft* наконец выпустила достойный внимания продукт — Visual Basic 5. Он содержал долгожданный компилятор, призванный разрешить проблемы низкой производительности, отличные средства поддержки технологий COM и ActiveX и еще несколько важных улучшений. На рынке корпоративных приложений Delphi удалось успешно потеснить такие пакеты, как PowerBuilder и Forte.

Во время создания Delphi 3 команда разработчиков потеряла своего главного члена. *Андерс Хейлсберг* (Anders Hejlsberg), ведущий специалист и главный архитектор Delphi, принял решение оставить свой пост и перейти на работу в корпорацию *Microsoft*. Но это не нарушило хода работ, поскольку освободившееся место занял *Чак Яджевски* (Chuck Jazdzewski), долгое время тесно работавший с прежним ведущим специалистом.

Delphi 4

Главной задачей Delphi 4 стало упрощение процедуры разработки приложений. Новая утилита Module Explorer позволила просматривать и редактировать модули с помощью удобного графического интерфейса. Новые средства навигации в программах и используемых классах позволяли вести работу над кодом создаваемого приложения с минимальными усилиями. Визуальную среду разработки перепроектировали и дополнили возможностью перетаскивать панели инструментов (dockable toolbar) и окна, делая процесс разработки более удобным. Существенные улучшения были внесены и в отладчик. Возможности Delphi 4 расширились благодаря средствам поддержки корпоративных многопользовательских решений за счет таких современных технологий, как MIDAS, DCOM, MTS и CORBA.

Это был 1998 год, и Delphi удавалось эффективно защищать свои позиции на рынке. Общая обстановка стабилизировалась, а Delphi продолжал медленно расширять свой сектор рынка. Самым модным решением в то время была технология CORBA, и, в отличие от конкурентов, Delphi обладал ее поддержкой. Кроме того, некоторый успех отмечался и на рынке простых приложений. Заслужив славу самого стабильного инструмента разработки на рынке, Delphi 4 завоевал хорошую репутацию у постоянных пользователей, которые не желали отказываться от предоставленных им стабильности и высококачественных решений.

Выпуск Delphi 4 последовал за приобретением компании *Visigenic* — одного из лидеров в области технологии CORBA. Компания *Borland*, носившая теперь название *Inprise*, присвоенное ей для упрощения выхода на корпоративный рынок, фактически стала лидером в этом секторе (в основном за счет интеграции своих инструментов со средствами технологии CORBA). Для окончательной победы необходимо было обес-

печить ту же простоту применения средств CORBA, которой удалось достигнуть в предыдущих версиях продуктов *Borland* в отношении COM и средств разработки приложений для Internet. Но по разным причинам необходимой степени интеграции достичь не удалось, а интегрированные средства поддержки CORBA составляли лишь незначительную часть общих функциональных возможностей среды разработки.

Delphi 5

В Delphi 5 дальнейшее развитие продукта происходило сразу по нескольким направлениям. Во-первых, была продолжена основная линия улучшений, начатая в Delphi 4. В пакет добавили новые функции, упрощающие выполнение задач, традиционно связанных со значительными затратами времени. Это позволило разработчикам больше сосредотачиваться на том, *что* они хотят написать, а не том, *как* это можно сделать. К новым возможностям можно отнести улучшенный графический интерфейс среды разработки и отладчика, пакет поддержки корпоративной разработки программ TeamSource и инструменты трансляции. Во-вторых, в Delphi 5 включен набор новых функций, упрощающих разработку приложений для Internet. Сюда относится мастер объектов Active Server Object Wizard, предназначенный для создания ASP, компоненты InternetExpress, обеспечивающие поддержку XML, и новые функции MIDAS, допускающие весьма гибкую платформу размещения данных в среде Internet. Наконец, разработчики затратили немало усилий на обеспечение самого важного из показателей Delphi 5 — стабильности его работы. Как и при изготовлении лучших вин, при создании высококачественного программного обеспечения не должно быть излишней поспешности. Поэтому Delphi 5 выпустили в свет только после того, как он был окончательно готов.

Выход в свет Delphi 5 состоялся во второй половине 1999 года. Delphi продолжает все глубже проникать на корпоративный рынок, а в секторе малых приложений он по-прежнему конкурирует с Visual Basic. В целом, положение остается более-менее стабильным. Компания *Inprise* приняла решение вернуть свое прежнее название — *Borland*, что было с радостью воспринято давними почитателями ее продуктов. Кроме того, была проведена определенная реорганизация, основное назначение которой — гарантировать сохранение высокого качества выпускаемых программных продуктов, свойственного прежним разработкам фирмы *Borland*.

Delphi 6

Основной задачей при разработке проекта Delphi 6 была совместимость с Kylix — инструментом разработки *Borland* для Linux. Для этого *Borland* разработал новую *библиотеку межплатформенных компонентов* (CLX — Component Library for Cross-Platform), которая включает VisualCLX (для визуальной разработки), клиентские компоненты доступа к данным DataCLX и компоненты NetCLX (для разработки под Internet). Весь проект реализован с использованием только библиотек CLX и межплатформенных элементов RTL, поэтому он обеспечивает беспроблемную совместимость между операционными системами Windows и Linux.

Новый набор компонентов и драйверов dbExpress — одно из самых больших достижений для обеспечения совместимости с Linux, поскольку он представляет собой реальный и окончательный вариант альтернативы BDE, которая в последнее время действительно начала устаревать.

Дополнительной задачей проекта Delphi 6 был охват всего, что связано с XML. Речь идет об XML для приложений баз данных, Web-ориентированных приложений и SOAP-ориентированных Web-служб. Разработчики Delphi постарались создать инструментальные средства, которые будут полностью соответствовать новой тенденции широкого применения XML, что обещает существенные преимущества с точки зрения возможности разработки Internet-приложений, способных функционировать вне традиционных границ, установленных различием в инструментальных средствах разработки, платформах или базах данных.

Безусловно, вдобавок ко всем этим новым преимуществам, Delphi 6 обладает, как и ожидается обычно от новой версии, дополнительными возможностям в таких основных областях, как VCL, IDE, отладчик, язык Object Pascal и RTL.

Интегрированная среда разработки Delphi

Для подтверждения факта преемственности используемой терминологии на рис 1.2 показан общий вид *интегрированной среды разработки* (IDE — Integrated Development Environment) Delphi. На этом рисунке отмечены все основные компоненты среды разработки: главное окно (Main Window), палитра компонентов (Component Palette), панели инструментов (toolbars), окно конструктора форм (Form Designer), окно редактора кода (Code Editor), окно инспектора объектов (Object Inspector), дерево объектов (Object TreeView) и проводник структуры кода (Code Explorer).

Главное окно

Главное окно можно представить как центр управления IDE Delphi. Это окно обладает всеми стандартными функциональными возможностями главного окна любой другой программы Windows. Оно состоит из трех частей: главного меню, панелей инструментов и палитры компонентов.

Главное меню

Как и в любой программе Windows, к меню обращаются при необходимости открыть, сохранить или создать новый файл, вызвать мастер, перейти в другое окно, изменить параметры настройки и т.д. Каждый элемент главного меню может быть продублирован соответствующей кнопкой на панели инструментов.

Панели инструментов Delphi

Панели инструментов предоставляют доступ к различным функциям главного меню IDE с помощью единственного щелчка на соответствующей кнопке. Обратите внимание, что для каждой кнопки панели инструментов предусмотрен вывод подсказки, содержащей описание ее назначения. Не считая палитры компонентов, в IDE Delphi имеется пять отдельных панелей инструментов: Debug (Отладка), Desktop (Рабочий стол), Standard (Стандартная), View (Вид) и Custom (Пользовательская). На рис. 1.2 показана конфигурация кнопок этих панелей, принимаемая по умолчанию. Но любую из кнопок можно удалить или добавить, выбрав в меню View пункты

Toolbars, Customize. На рис. 1.3 показано диалоговое окно Customize, предназначенное для настройки панелей инструментов. Чтобы добавить новую кнопку на любую панель инструментов, достаточно просто перетащить ее из этого окна. Для удаления кнопки достаточно перетащить ее за пределы панели инструментов.

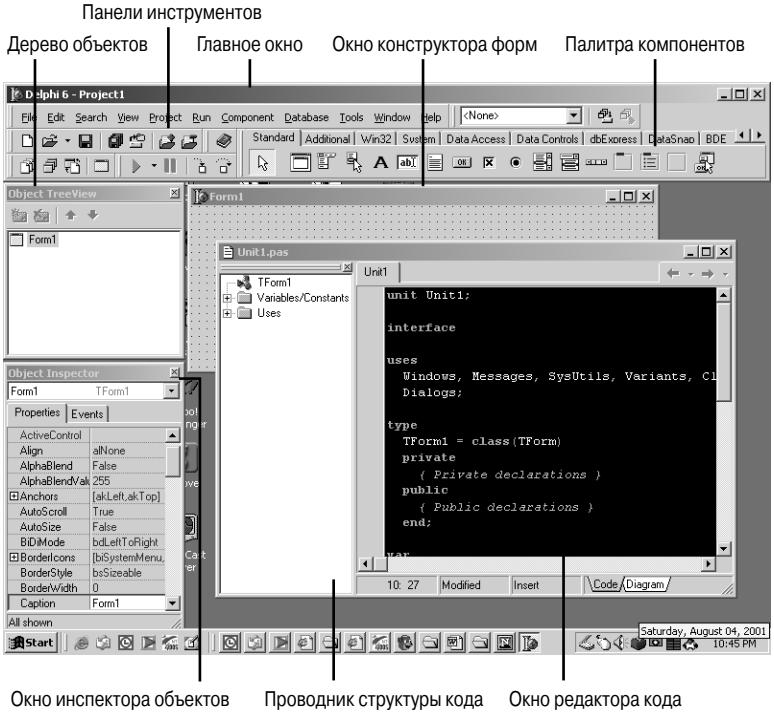

Рис. 1.2. *Общий вид интегрированной среды разработки (IDE) Delphi 6*

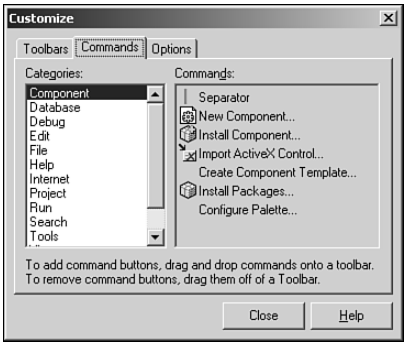

Рис. 1.3. *Диалоговое окно* Customize *предназначено для настройки панелей инструментов*

Возможности настройки панелей инструментов не ограничиваются лишь заданием отображаемых на них кнопок. Любую из панелей инструментов, палитру компонентов и панель меню можно перемещать в пределах главного окна IDE. Для этого достаточно щелкнуть указателем мыши на выпуклой серой полоске в левой части панели и перета-

щить ее в нужное место. Если перетащить панель за пределы главного окна, будет использована еще одна из возможностей настройки — панель инструментов отделится от главного окна и станет плавающей, т.е. расположенной в собственном независимом окне. Вид плавающих панелей инструментов показан на рис. 1.4.

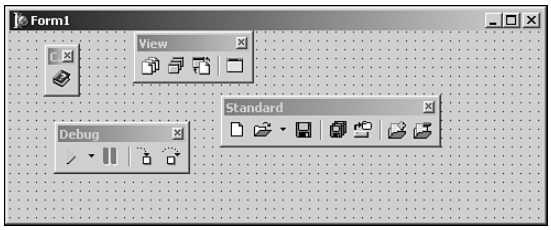

Рис. 1.4. *Плавающие панели инструментов*

Палитра компонентов

Палитра компонентов представляет собой панель инструментов удвоенной высоты, содержащей несколько вкладок, в которых находятся все установленные в среде IDE компоненты VCL и ActiveX. Порядок следования и вид вкладок и компонентов может быть настроен с помощью щелчка правой кнопкой мыши на интересующем объекте или в главном меню (пункты Component, Configure Palette).

Конструктор форм

При запуске конструктор форм (Form Designer) представляет собой пустую панель, готовую к превращению в окно приложения Windows. Его можно рассматривать как холст художника, предназначенный для создания графического интерфейса будущего приложения — здесь определяется, как оно будет выглядеть с точки зрения пользователя. Процесс создания заключается в выборе компонентов на палитре и перетаскивании их в форму. Точное размещение и установку размеров компонентов также можно выполнить с помощью мыши. Кроме того, существует возможность управлять внешним видом и поведением компонентов из окон Object Inspector и Code Editor.

Инспектор объектов

В окне Object Inspector можно изменять свойства компонентов формы или определять события, на которые будет реагировать сама форма или ее компоненты. *Свойства* (properties) представляют собой данные, определяющие, как объект выглядит на экране, — размер, цвет, шрифт и т.д. *События* (events) — это участки кода, выполняемые в ответ на некоторые действия, происходящие в приложении. Примером события может служить поступление сообщения от мыши или передача сообщения окну с требованием его перерисовки. В окне Object Inspector для переключения между работой с событиями и работой со свойствами используется стандартная *технология вкладок* (notebook tab) — для перехода в ту или иную вкладку достаточно щелкнуть на ее корешке. Инспектор показывает события и свойства, относящиеся к той форме или компоненту, который активен в конструкторе форм в настоящее время.

Одной из возможностей Delphi является способность упорядочивать содержимое окна Object Inspector либо по категории, либо по именам (в алфавитном порядке). Для

этого достаточно щелкнуть правой кнопкой мыши в любом месте окна Object Inspector и выбрать в раскрывшемся контекстном меню пункт Arrange (Выстроить). На рис. 1.5 показаны два расположенных рядом окна Object Inspector. В левом окне объекты упорядочены по категории, а в правом — по именам. Кроме того, с помощью пункта View этого же контекстного меню можно определить, какие именно категории объектов необходимо представить в данный момент.

Одним из наиболее ценных источников знаний, которым должен уметь пользоваться любой программист Delphi, является система помощи. Она полностью интегрирована с инспектором объектов, и если когда-нибудь возникнут какие-либо сложности со свойствами или событиями, то достаточно только нажать клавишу <F1> — и WinHelp приходит на помощь.

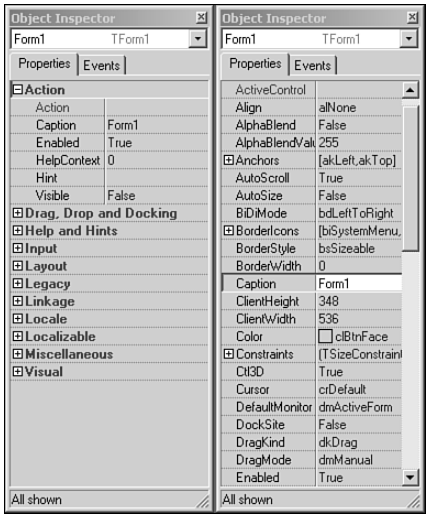

Рис. 1.5. *Представление содержимого Object Inspector по категориям и именам*

Редактор кода

Окно редактора кода Code Editor предназначено для ввода текста программ. Здесь же отображается код, автоматически созданный Delphi для компонентов разработанной формы. Окно Code Editor использует технологию вкладок, причем отдельная вкладка создается для каждого модуля или файла. При каждом добавлении в приложение новой формы создается новый модуль, а в окно Code Editor добавляется соответствующая вкладка. Контекстное меню окна Code Editor предоставляет широкий диапазон команд редактирования, включая команды работы с файлами, создания закладок и поиска символов.

СОВЕТ

Можно работать сразу с несколькими окнами Code Editor. Чтобы открыть новое окно редактора кода, необходимо выбрать в главном меню View пункт New Edit Window.

Проводник структуры кода

В окне Code Explorer можно просматривать модули, представленные во вкладках окна Code Editor, как древовидную структуру. Подобное представление позволяет легче ориентироваться в модулях, а также добавлять новые или переименовывать уже существующие элементы модулей. Очень важно помнить, что между окнами Code Explorer и Code Editor всегда поддерживается связь типа "один к одному". Щелчок правой кнопкой мыши на любом из элементов в окне Code Explorer позволяет вывести контекстное меню с командами, доступными для этого объекта. Кроме того, можно управлять сортировкой и фильтрацией объектов, отображаемых в окне Code Explorer. Для этого используются параметры, расположенные во вкладке Explorer диалогового окна Environment Options (Параметры среды).

Древовидное представление объектов

Древовидное представление объектов (Object TreeView) обеспечивает визуальное представление иерархии компонентов, модулей данных и фреймов, помещенных в форму. Отображаемая структура данных учитывает взаимосвязь между отдельными компонентами на уровне родительский-дочерний, компонент-свойство или свойство-свойство. Кроме средства представления, дерево объектов может послужить удобным инструментом редактирования связей между компонентами. Проще всего это можно сделать перетащив компонент из палитры или дерева в другое место дерева. Это изменит взаимосвязь между двумя компонентами (если это возможно).

Обзор исходного кода проекта

При работе с визуальными компонентами в конструкторе форм Delphi автоматически создает соответствующий код на языке Object Pascal. Простейший путь познакомиться с этой особенностью Delphi — начать новый проект. Выберите в меню File пункт New Application (Новое приложение), и в конструкторе форм будет создана новая форма, а в редакторе кода — каркас (skeleton) исходного кода модуля новой формы, представленный в листинге 1.1.

Листинг 1.1. Исходный код пустой формы

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
```

```
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation;

{$R *.dfm}

end.
```

Обратите внимание, исходный код, ассоциированный с некоторой формой, всегда сохраняется в отдельном модуле. *Каждая форма имеет собственный модуль, но не каждый модуль имеет собственную форму.* (Тем, кто недостаточно хорошо знаком с языком Pascal и применяемой в нем концепцией *модулей* (unit), стоит обратиться за пояснениями к главе 2, "Язык программирования Object Pascal". В данной главе содержится лишь краткое описание языка Object Pascal, предназначенное для тех, кто перешел к Delphi после C++, Visual Basic, Java или другого языка программирования.)

Рассмотрим код этого модуля более подробно. Ниже приведена его верхняя часть.

```
type
  TForm1 = class(TForm);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Этот фрагмент показывает, что собственно объект формы `TForm1` происходит[1] от объекта `TForm`. Здесь же в комментариях отмечены места, предназначенные для объявления собственных переменных — открытых (`public`) и закрытых (`private`). Не беспокойтесь, если термины *class*, *public* и *private* пока не понятны до конца. Более подробная информация об этих и других понятиях языка Object Pascal приведена в главе 2, "Язык программирования Object Pascal".

Следующая строка особенно важна:

```
{$R *.dfm}
```

В Object Pascal директива `$R` используется для загрузки внешнего файла ресурсов. Приведенная строка связывает файл с расширением `.dfm` (расширение *файлов форм Delphi* — Delphi form) с исполняемым модулем. Файл `.dfm` содержит бинарное представление формы, созданной с помощью конструктора форм. Символ шаблона "`*`" в данном случае означает, что имя файла должно быть тем же, что и имя модуля. В данном случае имя модуля определено как `Unit1`, следовательно, его исходный код будет находиться в файле `Unit1.pas`, а значение `*.dfm` в директиве — соответствовать файлу `Unit1.dfm`.

[1] *Точнее является экземпляром класса* `TForm`. – Прим. ред.

НА ЗАМЕТКУ

В IDE Delphi существует возможность сохранения создаваемых файлов DFM в текстовом, а не двоичном виде. Именно этот режим теперь устанавливается по умолчанию. Но его можно отменить, для чего следует сбросить флажок параметра New forms as text (Новые формы как текст) во вкладке Preferences (Свойства) диалогового окна Environment Options. Хотя сохранение форм в текстовом формате менее эффективно с точки зрения размера создаваемых файлов, данный вариант следует считать более предпочтительным по следующим причинам. Во-первых, это позволяет очень легко вносить незначительные изменения в текст описания формы в окне любого текстового редактора. Во-вторых, если файл по какой-либо причине был поврежден, то восстановить содержимое текстового файла гораздо проще, чем двоичного. Не забывайте, что предыдущие версии Delphi работают только с двоичными файлами `.DFM`, поэтому этот режим следует отменить, если создаваемый проект будет обрабатываться и в других версиях Delphi.

Файл проекта приложения также заслуживает внимания. Расширение файла проекта — `.dpr` (от **D**elphi **PR**oject). Он представляет собой обычный файл исходного кода Pascal, но с некоторыми расширениями. В этом файле содержится основная часть программы (с точки зрения Object Pascal). В отличие от других версий Pascal, которые, возможно, знакомы читателю, основная работа программ Delphi осуществляется в модулях, а не в главном файле программы. Текст файла проекта можно отобразить в окне редактора, выбрав в меню Project (Проект) пункт View Source (Вид исходного кода). Вот текст файла нового проекта:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

При добавлении новых модулей и форм в приложение они будут появляться в разделе `uses` файла проекта. Обратите внимание, что в разделе `uses` автоматически созданного кода определение подключенной формы закомментировано (`{Form1}`). Если когда-нибудь возникнут сомнения, в каком из модулей содержится та или иная форма, то их можно будет легко развеять, воспользовавшись окном Project Manager, которое выводится на экран с помощью пункта Project Manager меню View.

НА ЗАМЕТКУ

Каждая форма связана с собственным модулем, кроме того, в программу могут входить модули, содержащие только код и не связанные с какой-либо формой. В Delphi работают, в основном, с модулями, а файл проекта `.dpr` изменяют достаточно редко.

Обзор простейшего приложения

Простое действие, например перетаскивание компонента "Кнопка" в форму, приводит к автоматическому созданию кода данного элемента и добавлению его к объекту формы:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;
```

Таким образом, помещенная в форму кнопка представляет собой экземпляр объекта `TButton` в форме `TForm1`. Напомним, что при обращении к этой кнопке за пределами модуля данной формы имя экземпляра такой формы необходимо указывать в качестве области видимости: `Form1.Button1`. Более подробная информация об областях видимости приведена в главе 2, "Язык программирования Object Pascal".

Поведение выбранной в конструкторе форм кнопки можно изменить в окне Object Inspector. Предположим, что необходимо установить ее исходную ширину равной 100 пикселям и обеспечить, чтобы во время работы программы щелчок на кнопке приводил к удвоению ее высоты. Для изменения ширины кнопки выберите в окне Object Browser ее свойство `Width` (ширина) и установите его значение равным `100`. Внесенное изменение вступит в силу и будет отображено в конструкторе форм только после нажатия клавиши <Enter> или выхода из строки свойства `Width`. Для того, чтобы обеспечить реакцию кнопки на щелчок, в окне Object Inspector перейдите во вкладку Events (События). В ней содержится список событий, на которые может реагировать объект. Дважды щелкните на пункте списка `OnClick`, при этом Delphi создаст заготовку процедуры, которая будет вызвана при щелчке на данной кнопке, и перейдет к участку кода, содержащему ее текст (в данном случае это процедура `TForm1.Button1Click()`). Теперь осталось только ввести в заготовку между операторами `begin..end` оператор удвоения высоты кнопки:

```
Button1.Height := Button1.Height * 2;
```

Чтобы убедиться в работоспособности "приложения", нажмите клавишу <F9> и посмотрите, что произойдет.

НА ЗАМЕТКУ

Delphi постоянно поддерживает связь между созданными процедурами и теми элементами управления, которым они соответствуют. При компиляции или сохранении исходного кода модуля Delphi просматривает его и удаляет все пустые заготовки процедур. Это означает, что если в заготовку процедуры `TForm1.Button1Click()` не внести ни одной строки кода между логическими скобками `begin` и `end`, то Delphi полностью удалит ее из модуля. Таким образом, если процедура больше не нужна, просто удалите ее тело, после чего Delphi удалит оболочку процедуры собственными средствами.

После того как кнопка в созданной форме примет достаточно большие размеры, завершите работу программы и вернитесь в среду IDE Delphi. Следует заметить, что обеспечить реакцию кнопки на щелчок мышью можно сразу же после помещения данного элемента управления в форму. Двойной щелчок мышью на любом компоненте формы автоматически вызывает окно редактора кода с текстами подпрограмм, связанных с данным компонентом. В большинстве случаев будет автоматически создан обработчик для первого из событий того компонента, который первым упоминается в списке окна Object Inspector.

События и сообщения

Те, кому уже приходилось создавать приложения Windows традиционными методами, безусловно оценят простоту использования событий Delphi. Исключается всякая необходимость организации перехвата сообщений Windows, их обработки, анализа и т.д. Более подробная информация по этой теме приведена в главе 3, "Приключения сообщения".

Как правило, события Delphi происходят в ответ на сообщения Windows. Так, например, событие `OnMouseDown` (кнопка мыши нажата) объекта `TButton` фактически инкапсулирует сообщение Windows `WM_xBUTTONDOWN`. Обратите внимание, что событие выполняет предварительную обработку сообщения и предоставляет информацию о том, какая именно из кнопок мыши была нажата и где находился курсор мыши, когда это произошло. Подобную информацию — но уже о нажатии клавиши клавиатуры — предоставляет событие `OnKeyDown`. Ниже приведен созданный Delphi код обработчика этого события:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
                             Shift: TShiftState);
begin
end;
```

Вся необходимая для написания обработчика события информация — уже в ваших руках. Обладая опытом программирования в Windows, можно оценить, насколько такая форма удобнее самостоятельного разбора параметров сообщения `LParams` или `WParams`. Вся эта работа выполняется автоматически, и остается только воспользоваться ее результатами, которые значительно понятнее, чем стандартные параметры сообщений Windows. Более подробная информация о том, как в Delphi функционирует внутренняя служба обработки сообщений, приведена в главе 3, "Приключения сообщения".

Необязательность программирования

Одним из самых значительных (хоть и спорных) достоинств системы событий Delphi (по сравнению со стандартной системой обработки Windows) является то, что теперь *необязательно* программировать обработчики событий (технология *contract free*). В отличие от обработки стандартных сообщений Windows, не требуется вызывать ни унаследованный обработчик, ни передавать какую-либо информацию назад в Windows после обработки сообщения.

Обратная сторона медали заключается в том, что нельзя сделать больше, чем позволено. Программист оказывается полностью во власти того, кто спроектировал это

событие, поскольку именно этот человек определил тот уровень возможностей, который будет предоставлен при обработке данного события в приложении. Так, в обработчике `OnKeyPress` можно изменить или просто удалить информацию о нажатых клавишах, в то время как обработчик `OnResize` (при изменении размеров) только извещает о происшедшем событии, не позволяя ни предотвратить его, ни повлиять на устанавливаемые размеры объекта.

Но не стоит расстраиваться, Delphi вовсе не запрещает обращаться непосредственно к сообщениям Windows. Это не так просто, как работа с системой событий, поскольку обработка сообщений подразумевает наличие у программиста определенного опыта и знаний по обработке сообщений Windows, но это вполне возможно. Ключевое слово `message` (сообщение) предоставляет полную свободу перехватывать и обрабатывать все сообщения Windows. Более подробная информация об обработчиках сообщений Windows приведена в главе 3, "Приключения сообщения".

В этом — весь Delphi. Он универсален и предоставляет каждому свое, обеспечивая простоту и ясность при высокоуровневом программировании и оставляя, в случае необходимости, возможность эффективного и нестандартного низкоуровневого программирования.

Упрощение разработки прототипов

Поработав с Delphi некоторое время, можно заметить, что предъявляемые этим пакетом требования к уровню знаний достаточно размыты. Создание в Delphi даже самого первого приложения приносит новичкам заметные дивиденды. Это выражается в сокращении времени разработки и получении надежных устойчивых продуктов. Delphi обладает просто неограниченными возможностями в том аспекте разработки приложений, который доставлял программистам в среде Windows больше всего неприятностей. Речь идет о создании *пользовательского интерфейса* (UI — User Interface).

Иногда процедуру разработки пользовательского интерфейса и общего макета окна программы называют "созданием *прототипа*" (prototyping). В невизуальной среде программирования создание прототипа приложения зачастую требует существенно бо́льших усилий, чем создание собственно рабочей части программы или ее внутренней структуры (back end). Тем не менее, именно реализация основного плана программы является главной целью ее создания. Безусловно, создание привлекательного и интуитивно понятного интерфейса также является важнейшим условием успеха. Однако кому будет нужна коммуникационная программа с превосходно выполненными диалоговыми окнами и удобным интерфейсом, но не способная переслать данные через модем? Программы во многом подобны людям — приятная внешность всегда располагает, но этого мало для того, чтобы данный человек занял сколько-нибудь заметное место в нашей жизни. Поэтому оставим дизайн приложений без комментариев.

Delphi позволяет создавать программу, вкладывая максимум сил и умения в ее рабочую часть, сократив до минимума затраты времени на реализацию пользовательского интерфейса. Потратив некоторое время на обучение работе с Delphi, можно легко и просто разрабатывать пользовательские интерфейсы, не сравнимые ни с какими другими, созданными с помощью традиционного инструментария. Более того, они станут не просто элегантными, но и оригинальными, т.е. будут обладать "изюминкой" в виде разработанных пользователем новых элементов управления.

Расширяемость Delphi

Благодаря объектно-ориентированной природе Delphi можно создавать собственные компоненты не только с нуля, но и на основе уже существующего богатого набора стандартных компонентов. Более подробная информация о пользовательских и других типах компонентов приведена в части IV, "Компонент-ориентированная разработка".

Помимо создания и интеграции в среду разработки новых компонентов, Delphi также допускает интеграцию в IDE готовых подпрограмм, называемых *экспертами* (experts). Интерфейс Expert Interface позволяет дополнять визуальную среду разработки своими пунктами меню и диалоговыми окнами, предназначенными для расширения ее функциональных возможностей. Примером подобного эксперта является утилита Database Form Expert, вызвать которую можно в меню Database (База данных) главного окна. Процесс создания собственного эксперта и интеграции его в IDE Delphi описан в главе 17, "Применение интерфейса API Open Tools".

Десять важнейших возможностей IDE Delphi

Прежде чем продолжить изучение этой книги, авторы хотели бы удостовериться, что читатель знаком со всеми важнейшими инструментами, которые могут потребоваться в работе. Поэтому ниже приводится краткое описание десяти инструментов графической среды разработки Delphi, которые мы считаем наиболее важными и необходимыми.

1. Автозавершение классов

Ничто не отнимает у разработчика времени больше, чем необходимость набирать этот надоедающий код! Как часто бывает, абсолютно точно знаешь, что необходимо написать, но пальцы не успевают летать по клавишам. Пока еще не существует технологии, способной полностью освободить разработчика от необходимости общения с клавиатурой, но Delphi обладает возможностью, называемой *автозавершением классов* (class completion), которое довольно ощутимо облегчает рутинные работы.

Возможно, наиболее важной особенностью функции автозавершения классов является то, что она способна работать автономно и не нуждается в контроле со стороны человека. Достаточно частично объявить класс и нажать магическую комбинацию клавиш <Ctrl+Shift+C> и функция автозавершения классов попытается определить, что именно предполагалось ввести, и создаст соответствующий код. Например, если поместить в класс объявление процедуры `Foo`, а затем вызвать автозавершение классов, то описание этого метода автоматически будет внесено в раздел реализации данного модуля. Если объявить новое свойство, значение которого считывается из поля, а записывается с помощью метода, то после автозавершения будет создано определение поля, а также код описания и реализации метода.

Если вышесказанное не произвело достаточного впечатления, попробуйте применить эту функцию на практике. Очень скоро она окажется просто незаменимой.

2. Навигатор AppBrowser

Иногда, глядя на строку кода программы, мучительно вспоминаешь, где именно был объявлен используемый в ней метод. Самый простой способ выяснить это — нажать клавишу <Ctrl> и щелкнуть мышью на интересующем слове. IDE используют заранее собранную компилятором отладочную информацию и перейдет к строке объявления указанной функции. Очень удобно. Подобно Web-броузеру, эта функция обладает хронологическим стеком, который позволяет с помощью небольших стрелок, расположенных справа от вкладок редактора кода, перемещаться вперед и назад по списку.

3. Навигатор Interface/Implementation

Для перемещения между объявлением (interface) и реализацией (implementation) метода достаточно поместить на его имя курсор и нажать комбинацию клавиш <Ctrl+Shift+Стрелка вверх> или <Ctrl+Shift+Стрелка вниз>.

4. Стыковка окон

Визуальная среда разработки позволяет организовать на экране единое окно, составленное из нескольких окон, каждое из которых будет выглядеть как одна из панелей общего окна. Это называется *стыковкой* (docking). Полученное составное окно можно перемещать по экрану как единое целое. Отличить составное окно от обычного очень просто — при перемещении на экране оно особым образом пульсирует. Окно редактора кода позволяет стыковать другие окна с трех его сторон — слева, снизу и справа. Стыковка окна осуществляется при перетаскивании одного окна вплотную к границе или в центр другого. Завершив перекомпоновку, не забудьте сохранить результаты с помощью кнопок панели инструментов Desktops (Рабочий стол). Для предотвращения случайной стыковки окон при их перемещении нажмите и удерживайте клавишу <Ctrl>. Щелкнув в окне правой кнопкой мыши, можно сбросить в раскрывшемся контекстном меню флажок Dockable (Стыкуемое).

СОВЕТ

Еще одна приятная скрытая возможность. Если щелкнуть правой кнопкой мыши на корешке вкладки состыкованного окна, то его можно будет переместить в верхнюю, нижнюю, левую или правую часть общего окна.

5. Броузер объектов

В Delphi версий 1–4 использовался очень примитивный броузер объектов. Если о нем никто ничего даже не слышал, то это и неудивительно — многие вообще предпочитали им не пользоваться, поскольку возможности его были весьма ограничены. Теперь броузер объектов полностью переработан. Его окно, которое можно вывести на экран, выбрав в меню View пункт Browser, показано на рис. 1.6. Здесь отображена древовидная структура, позволяющая получать доступ к глобальным переменным, классам и модулям, а также контролировать области видимости, цепочки наследования и ссылки на символьные строки.

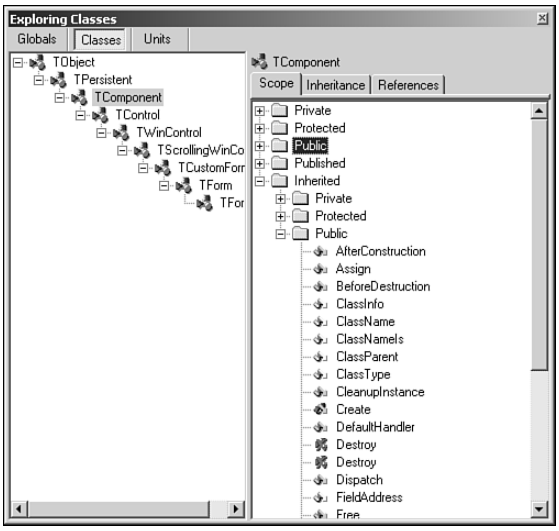

Рис. *1.6. Общий вид окна броузера объектов*

6. Новый GUID

Нажатие комбинации клавиш <Ctrl+Shift+G> позволяет поместить в окно редактора кода новое уникальное значение GUID (уникального глобального идентификатора интерфейса). Эта возможность оказывается весьма полезной при создании нового интерфейса.

7. Подсветка синтаксиса C++

Если в процессе работы в среде Delphi приходится часто просматривать файлы с программами на языке C++ (например файлы заголовков SDK), то эту встроенную функцию редактора кода можно оценить по достоинству. Поскольку Delphi и C++ Builder совместно используют тот же самый редактор, то исходный код одного из них можно не только просматривать в другом, но и подсвечивать текст "инородных" файлов (в частности C++). Достаточно просто загрузить в редактор файл C++ (`.CPP` или `.H`), а все остальное будет сделано автоматически.

8. Список To Do

Используйте список To Do List (Список To Do) для управления ходом работы над исходными файлами проекта. Чтобы вывести содержимое этого списка, необходимо в меню View выбрать пункт To Do List. В этот список автоматически помещаются все комментарии из создаваемых программ, которые начинаются со слова `TODO`. Окно To Do Items (Элементы To Do) можно использовать, чтобы установить владельца, приоритет и категорию любого из элементов списка To Do. Общий вид этого окна показан на рис. 1.7.

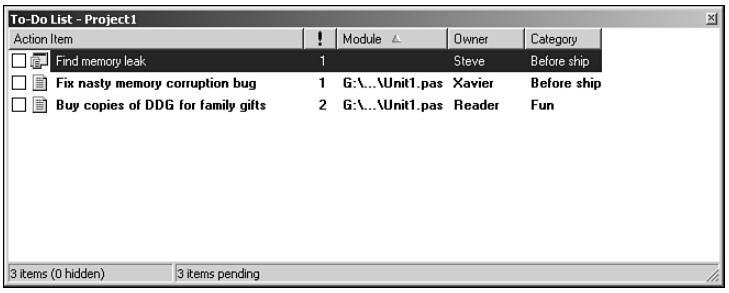

Рис. 1.7. *Окно To Do Items*

9. Использование диспетчера проектов

Диспетчер проектов (окно Project Manager) может оказаться очень удобным инструментом при работе над большими проектами — особенно над теми, которые предусматривают создание нескольких файлов `.EXE` или `.DLL`. Но многие пользователи просто забывают о его существовании. Это окно можно вывести на экран, выбрав в меню View пункт Project Manager. Теперь функциональные возможности данного окна существенно расширены. В частности, появилась возможность вставки и копирования элементов из одного проекта в другой с помощью простого перетаскивания.

10. Использование Code Insight

Технология Code Insight (осознание кода) применяется для автоматического завершения объявлений и ввода параметров. После ввода ключевого слова **`Identifier.`** (с точкой в конце) на экран автоматически выводится окно со списком свойств, методов, событий и полей, доступных для этого идентификатора. Щелкнув правой кнопкой мыши, можно отсортировать этот список по именам или по области видимости. Если окно исчезло с экрана до того, как удалось прочитать его содержимое, то для повторного его вывода достаточно нажать комбинацию клавиш <Ctrl+Пробел>.

Практически невозможно запомнить все параметры всех функций, поэтому функция Code Insight окажется очень полезна. Как только в окне редактора кода будет введено имя любой функции в сопровождении открывающей скобки (**`FunctionName(`**), на экране автоматически появится окно интерактивной подсказки со списком ее параметров. Если это окно исчезло с экрана слишком быстро, можно повторно вывести его с помощью комбинации клавиш <Ctrl+Shift+Пробел>.

Резюме

В этой главе была описана история развития системы программирования Delphi 6, ее интегрированная среда разработки, а также рассказано о ее роли среди средств разработки для Windows в целом. Эта глава должна была ознакомить с основными концепциями Delphi, используемыми на протяжении всей книги. Теперь можно перейти к этапу собственно технического содержания. Прежде чем углубиться в дальнейшее изучение книги, удостоверьтесь, что полностью понимаете назначение элементов IDE и умеете работать с небольшими проектами.

ГЛАВА

2

Язык программирования Object Pascal

В ЭТОЙ ГЛАВЕ...

В настоящей главе не рассматриваются разнообразные визуальные элементы среды разработки Delphi. Она посвящена языку программирования Object Pascal, лежащему в основе этого пакета. Сначала речь пойдет о базовых элементах языка, его правилах и конструкциях, а затем будут рассмотрены более сложные вопросы — классы и обработка исключений. Поскольку данная книга не предназначена для начинающих программистов, предполагается, что читатель знаком с другими языками программирования высокого уровня, такими, например, как C, C++ или Visual Basic. Мы будем часто проводить сравнение тех или иных положений языка Object Pascal с другими языками программирования. К концу главы необходимо иметь представление об основополагающих концепциях программирования (т.е. переменных, типах, операторах, циклах, исключениях и объектах) в Object Pascal и их отличиях от таких языков, как Java, C++ и Visual Basic.

НА ЗАМЕТКУ

Упоминая в этой главе язык C, мы имеем в виду все, что является общим для языков C и C++. Об особенностях, специфичных только для языка C++, рассказывается отдельно.

Даже для опытных программистов эта глава будет очень полезной, так как только в ней можно найти полную информацию о синтаксисе и семантике Object Pascal.

Комментарии

Начнем рассмотрение темы с того, каким образом следует вносить комментарии в программу на языке Object Pascal. Этот язык поддерживает три типа комментариев — с использованием фигурных скобок, пар скобка–звездочка и двойной наклонной черты (в стиле C++). Ниже приведены три примера комментариев Object Pascal:

```
{  Комментарий с использованием фигурных скобок }
(* Комментарий с использованием пары скобка+звездочка *)
// Комментарий с использованием двойной наклонной черты
```

Поведение первых двух типов комментариев идентично. Object Pascal считает комментарием *все*, что находится между символом (или парой символов) начала комментария и символом завершения комментария, включая символы перехода на новую строку. Комментарий в виде двойной косой черты продолжается только до конца строки.

НА ЗАМЕТКУ

В Object Pascal нельзя использовать вложенные комментарии одного и того же типа. И хотя синтаксис Object Pascal позволяет создавать вложенные комментарии различного типа, но пользоваться этой возможностью не рекомендуется.

```
{ (* Допустимо *) }
(* { Допустимо } *)
(* (* Недопустимо *) *)
{ { Недопустимо } }
```

Дополнительные возможности процедур и функций

Поскольку процедуры и функции — довольно обширная тема, а речь здесь идет лишь о сравнительном анализе различных языков программирования, не будем слишком углубляться в данный вопрос, а отметим лишь малоизвестные и новые возможности Delphi в этой области.

Скобки при вызове функций

Хотя добавление скобок при вызове функции или процедуры без параметров не является новинкой, но про такую возможность знают далеко не все. В Object Pascal считаются корректными оба варианта вызова:

```
Form1.Show;
Form1.Show();
```

Это, конечно, возможность не из тех, которые сильно удивляют. Однако программисты, вынужденные делить свое рабочее время между разными языками (например Delphi, C и Java), оценят ее по достоинству, ибо им не придется задумываться о различии синтаксиса вызова функций в различных языках.

Перегрузка

В Delphi 4 была реализована концепция *перегрузки функций* (overloading), которая позволяет иметь несколько различных функций или процедур с одинаковым именем, но разными списками параметров. Такие процедуры и функции должны быть описаны с применением директивы `overload`:

```
procedure Hello(I: Integer); overload;
procedure Hello(S: String);  overload;
procedure Hello(D: Double);  overload;
```

Заметим, что правила перегрузки методов класса, сформулированные в соответствующей главе книги, несколько отличаются от правил перегрузки обычных процедур и функций. Хотя предоставления возможности перегрузки функций программисты требовали с момента выхода в свет Delphi 1, тем не менее это не самая безопасная возможность языка. Наряду с уже имевшейся возможностью использования в разных модулях функций с одним и тем же именем, применение перегруженных функций может стать неиссякаемым источником трудноуловимых ошибок в программе. Поэтому сто раз подумайте, прежде чем использовать перегрузку. Помните золотое правило: “Можно потреблять, но не злоупотреблять!”.

Значения параметров по умолчанию

В Delphi 4 была введена еще одна полезная возможность — использование значений параметров по умолчанию. Она позволяет установить принимаемое по умолчанию значение параметра процедуры или функции. Это значение будет использовать-

ся в тех случаях, когда вызов процедуры или функции осуществляется без указания значения данного параметра. В объявлении процедуры либо функции принимаемое по умолчанию значение параметра указывается после знака равенства, следующего после его имени. Поясним это на следующем примере:

```
procedure HasDefVal(S: string; I: Integer = 0);
```

Процедура `HasDefVal()` может быть вызвана двумя способами. В первом случае — как обычно, с указанием обоих параметров:

```
HasDefVal('hello', 26);
```

Во втором случае можно задать только значение параметра `S`, а для параметра `I` использовать значение, установленное по умолчанию:

```
HasDefVal('hello');  // Для I используется значение по умолчанию
```

При использовании значений параметров по умолчанию следует помнить о нескольких приведенных ниже правилах.

- Параметры, имеющие значения по умолчанию, должны располагаться в конце списка параметров. Параметр без значения по умолчанию не должен встречаться в списке после параметра, имеющего значение по умолчанию.
- Значения по умолчанию могут присваиваться только параметрам обычных типов, указателям или множествам.
- Значение по умолчанию может передаваться только по значению либо с модификатором `const`. Оно не может быть ссылкой или нетипизированным параметром.

Одним из важных преимуществ применения значений параметров по умолчанию является простота расширения функциональных возможностей уже имеющихся процедур и функций с соблюдением обратной совместимости. Предположим, на рынок программных продуктов была выпущена программа, ключевым звеном которой является функция сложения двух целых величин:

```
function AddInts(I1, I2: Integer): Integer;
begin
  Result := I1 + I2;
end;
```

Допустим, что маркетинговые исследования показали целесообразность добавления в программу возможности сложения трех чисел. Замена существующей функции на другую, способную складывать три числа, приведет к необходимости исправить весь код программы, который перестанет компилироваться из-за внесения в функцию еще одного параметра. Но при использовании значений параметров по умолчанию проблема решается легко и просто. Достаточно изменить объявление функции так, как показано ниже.

```
function AddInts(I1, I2: Integer; I3: Integer = 0);
begin
  Result := I1 + I2 + I3;
end;
```

Переменные

У начинающих программистов бытует мнение: “Если мне понадобится еще одна переменная — я опишу ее прямо здесь, посреди кода, в том месте, где она понадобилась”. Для таких языков, как Java, C или Visual Basic, — это вполне допустимо. Но в языке Object Pascal *все* переменные обязательно должны быть описаны в *соответствующем разделе* `var`, расположенном в начале процедуры, функции или программы. Предположим, на C++ код функции выглядит так:

```
void foo(void)
{
    int x = 1;
    x++;
    int y = 2;
    float f;
    //...и так далее...
}
```

Аналогичная функция на Object Pascal должна выглядеть следующим образом:

```
Procedure Foo;
var
  x, y: Integer;
  f: Double;
begin
  x := 1;
  inc(x);
  y := 2;
  //...и так далее...

end;
```

НА ЗАМЕТКУ

Object Pascal (как и Visual Basic), в отличие от Java и C, не чувствителен к регистру символов. Использование строчных и прописных символов сводится к тому, чтобы сделать код более удобочитаемым. Например, сложно воспринять имя процедуры, записанное как:

```
procedure thisprocedurenamemakenosense;
```

Значительно понятнее имена процедур, которые записаны в таком виде:

```
procedure ThisProcedureNameIsMoreClear;
```

Возникает вполне резонный вопрос: в чем же тогда преимущества и удобства Object Pascal? Вскоре вы убедитесь, что такая строгая структура программы делает ее более понятной, удобочитаемой и, как следствие, она проще в отладке и сопровождении, чем у других языков, которые полагаются скорее на общепринятые соглашения, а не на жесткие правила.

Обратите внимание, Object Pascal позволяет объявить более одной переменной одинакового типа в одной строке:

```
VarName1, VarName2: SomeType;
```

Не забывайте, что объявление переменной в Object Pascal состоит из ее имени, двоеточия и типа переменной. Запомните: в Object Pascal переменная *всегда* инициализируется *отдельно* от своего объявления.

Начиная с Delphi 2.0, язык позволяет инициализировать глобальные переменные в разделе их объявления (var). Приведенный ниже пример демонстрирует синтаксис такого объявления.

```
var
  i: Integer = 10;
  S: string  = 'Hello world';
  D: Double  = 3.141579;
```

НА ЗАМЕТКУ

Предварительная инициализация возможна только для глобальных переменных и недопустима для переменных, являющихся локальными для процедуры или функции.

СОВЕТ

Компилятор Delphi автоматически инициализирует все глобальные переменные нулевым значением. Таким образом, при запуске программы все глобальные переменные целого типа примут значение 0, переменные с плавающей точкой — 0.0, строки окажутся пустыми, а указатели будут иметь значение nil. Поэтому в исходном коде специальная инициализация глобальных переменных нулевым значением не требуется.

Константы

Константы в Object Pascal определяются в специальном разделе директивой const, действие которой аналогично ключевому слову const в языке C. Вот пример объявления трех констант в языке C:

```
const float ADecimalNumber = 3.14;
const int i = 10;
const char * ErrorString = "Danger, Danger, Danger!";
```

Основное отличие между константами в языках C и Object Pascal заключается в том, что в Object Pascal (как и в Visual Basic) не требуется указывать тип константы. Компилятор Delphi автоматически выделяет необходимую память, основываясь на инициализирующем значении (а в случае скалярных констант, таких как целое значение, он просто подставляет его в нужные места в программе, совсем не выделяя памяти). Объявления тех же констант в Object Pascal выглядят следующим образом:

```
const
  ADecimalNumber = 3.14;
  i = 10;
  ErrorString = 'Danger, Danger, Danger!';
```

НА ЗАМЕТКУ

Выделение памяти для констант происходит так: целые значения размещаются в наименьшей, но достаточной для этого области памяти (в частности, значение 10 будет размещено как переменная типа `ShortInt`, 32000 — как `SmallInt` и т.д.). Алфавитно-цифровые значения, а попросту — строки, размещаются как тип `Char` или текущий тип `String`, определенный директивой `$H`. Значения с плавающей точкой размещаются как данные типа `extended`, кроме значений, имеющих менее четырех десятичных знаков, — в этом случае используется тип `Comp`. Массивы целых (`Integer`) или символьных (`Char`) данных хранятся в собственном представлении.

Хоть это и необязательно, но в объявлении константы можно указать ее тип. Это позволит компилятору осуществлять полный контроль типов данных при работе с такими константами:

```
const
  ADecimalNumber: Double = 3.14;
  I: Integer = 10;
  ErrorString: string = 'Danger, Danger, Danger!';
```

Object Pascal допускает применение в объявлениях констант (`const`) и переменных (`var`) функций, вычисляемых во время компиляции. К этим функциям относятся: `Ord()`, `Chr()`, `Trunc()`, `Round()`, `High()`, `Low()` и `SizeOf()`. Так, все объявления в следующем примере абсолютно корректны:

```
type
  A = array[1..2] of Integer;

const
  w: Word = SizeOf(Byte);

var
  i: Integer = 8;
  j: SmallInt = Ord('a');
  L: Longint = Trunc(3.14159);
  x: ShortInt = Round(2.71828);
  B1: Byte = High(A);
  B2: Byte = Low(A);
  C: char = Chr(46);
```

СОВЕТ

Поведение констант в 32-битовом Delphi отличается от 16-битового Delphi 1.0. В Delphi 1.0 константа рассматривалась как предварительно инициализированная переменная, называемая *типизированной константой*. Начиная с Delphi 2.0, константы стали истинными константами, а для обратной совместимости была введена директива компилятора `$J`. Для совместимости с кодом Delphi 1 этот режим по умолчанию включен, но, чтобы избавиться от устаревших типизированных констант, рекомендуется отключать его (флажок Assignable typed constants (Присваиваемые типизированные константы), расположенный во вкладке Compiler окна Project Options).

Если попробовать присвоить константе значение, то компилятор выдаст сообщение об ошибке, поясняющее, что значение константы не может быть изменено. Исходя из того факта, что константы доступны только для чтения, компилятор Object Pascal оптимизирует использование памяти, располагая константы в незаполненных частях страниц кода. Более подробная информация по этой теме приведена в предыдущем издании — *"Delphi 5. Руководство разработчика, том 1"*, в главе 3, "Win32 API".

НА ЗАМЕТКУ

Object Pascal не имеет препроцессора, как C или C++. Таким образом, в Object Pascal отсутствует концепция макросов, а значит — и эквивалент директивы препроцессора `#define`, используемой в C для описания констант. Хотя язык Object Pascal и позволяет выполнять условную компиляцию с использованием директивы компилятора `$define`, но ее нельзя применять для определения констант, как в C. Там, где в языке C применялась директива `#define`, в языке Object Pascal рекомендуется объявлять константы (`const`).

Операторы

Оператор — это один или несколько символов кода, с помощью которых выполняются определенные действия с данными различных типов. Простейшим примером могут служить операторы сложения, вычитания, умножения и деления арифметических типов данных; другим примером может быть оператор для доступа к определенному элементу массива. В данном разделе рассматриваются операторы Object Pascal и их отличие от операторов Java, C и Visual Basic.

Оператор присвоения

Это один из наиболее часто используемых операторов. Оператор присвоения языка Object Pascal (`:=`), аналогичный оператору `=` в языках Java, C и Visual Basic, применяется для присвоения значения переменной. Иногда программисты называют его оператором *возвращения значения* (get). Рассмотрим пример оператора присвоения:

```
Number1 := 5;
```

Эту строку можно прочитать как "переменная `Number` получает значение `5`" либо как "переменной `Number` присвоено значение `5`".

Операторы сравнения

Для тех, кто работал с Visual Basic, использование операторов сравнения в Object Pascal не вызовет никаких проблем, так как они идентичны соответствующим операторам Visual Basic.

Object Pascal использует оператор `=` для логического сравнения двух выражений или значений. В языках Java и C ему аналогичен оператор `==`, пример применения которого показан ниже.

```
if (x == y)
```

На языке Object Pascal этот же оператор будет выглядеть следующим образом:

```
if x = y
```

НА ЗАМЕТКУ

Запомните: в Object Pascal оператор `=` используется только для сравнения. Чтобы присвоить значение, следует воспользоваться оператором `:=`.

В Delphi оператор "не равно" выглядит так: `<>`. В языке C ему аналогичен оператор `!=`. Вот пример проверки того, что два значения не равны друг другу:

```
if x <> y thenDoSomething
```

Логические операторы

Object Pascal использует ключевые слова `and` и `or` в качестве логических операторов "и" и "или" (в языках C и Java для этого применяются соответственно операторы `&&` и `||`). В основном эти операторы применяются как элементы оператора `if` или цикла. Например:

```
if (Condition 1) and (Condition 2) then
    DoSomething;

while (Condition 1) or (Condition 2) do
    DoSomething;
```

Оператор "не" в Object Pascal выглядит как `not` (он аналогичен оператору `!` в языках Java и C). В основном оператор `not` применяется в составе оператора `if`; это демонстрирует следующий пример:

```
if not (condition) then (do something); // Если условие condition
                                        // ложно, то выполнить
                                        // оператор do something
```

В табл. 2.1 приведен список операторов Object Pascal и соответствующие им операторы языков Java, C и Visual Basic.

Таблица 2.1. Операторы присвоения, сравнения и логические

Оператор	*Pascal*	*Java/C*	*Visual Basic*
Присвоение	`:=`	`=`	`=`
Сравнение	`=`	`==`	`=` или `Is*`
Неравенство	`<>`	`!=`	`<>`
Меньше, чем	`<`	`<`	`<`
Больше, чем	`>`	`>`	`>`
Меньше или равно	`<=`	`<=`	`<=`
Больше или равно	`>=`	`>=`	`>=`

Окончание табл. 2.1

Оператор	*Pascal*	*Java/C*	*Visual Basic*
Логическое И	`and`	`&&`	`And`
Логическое ИЛИ	`or`	`||`	`Or`
Логическое НЕ	`not`	`!`	`Not`

** Оператор сравнения* `Is` *используется только для объектов, а оператор сравнения* `=` *применяется для всех остальных типов данных.*

Арифметические операторы

Большинство арифметических операторов Object Pascal должно быть знакомо всем, так как они идентичны операторам, используемым в языках Java, C, и Visual Basic. В табл. 2.2 приведены арифметические операторы Object Pascal и соответствующие им операторы языков Java, C и Visual Basic.

Таблица 2.2. Арифметические операторы

Оператор	*Pascal*	*Java/C*	*Visual Basic*
Сложение	`+`	`+`	`+`
Вычитание	`-`	`-`	`-`
Умножение	`*`	`*`	`*`
Деление с плавающей точкой	`/`	`/`	`/`
Деление целых чисел	`div`	`/`	`\`
Деление по модулю	`mod`	`%`	`Mod`
Возведение в степень	Отсутствует	Отсутствует	`^`

Как видно из таблицы, Object Pascal отличается от других языков программирования тем, что в нем существуют разные операторы для деления целых чисел и чисел с плавающей точкой. Оператор `div` автоматически отсекает остаток при делении двух целых выражений.

НА ЗАМЕТКУ

Не забывайте использовать правильный оператор деления для соответствующих типов выражений. Компилятор выдаст сообщение об ошибке при попытке деления двух чисел с плавающей точкой с помощью оператора `div` или двух целых чисел с помощью оператора `/`, например:

```
var
  i: Integer;
  r: Real;
begin
  i := 4 / 3;          // Эта строка приведет к ошибке компиляции
  f := 3.4 div 2.3;    // Эта строка тоже приведет к ошибке
end;
```

В большинстве других языков программирования не делается различия между делением целочисленным и делением с плавающей точкой. Как правило, всегда выполняется деление с плавающей точкой, а, при необходимости, результат конвертируется в целое число. Но при таком подходе возможно существенное снижение производительности программы. Специализация оператора `div` языка Pascal несколько выше, а потому и выполняется быстрее.

Побитовые операторы

Побитовыми называются операторы, позволяющие работать с отдельными битами заданной переменной. Чаще всего побитовые операторы используются для сдвига битов влево или вправо, их инверсии, а также побитовых операций "и", "или" и "исключающее или" (`xor`) между двумя числами. Операторы сдвига влево и вправо в Object Pascal имеют вид `shl` и `shr` соответственно, они аналогичны операторам `<<` и `>>` в языках Java и C. Запомнить остальные операторы тоже достаточно легко — это `not`, `and`, `or` и `xor`. В табл. 2.3 приведены побитовые операторы и соответствующие им операторы языков C и Visual Basic.

Таблица 2.3. Побитовые операторы

Оператор	*Pascal*	*Java/C*	*Visual Basic*
И	`and`	`&`	`And`
Не	`not`	`~`	`Not`
Или	`or`	`|`	`Or`
Исключающее или	`xor`	`^`	`Xor`
Сдвиг влево	`shl`	`<<`	Нет
Сдвиг вправо	`shr`	`>>`	Нет

Процедуры инкремента (приращения) и декремента

Процедуры инкремента и декремента генерируют оптимизированный код для добавления или вычитания единицы из целой переменной. Object Pascal не предоставляет таких широких возможностей, как постфиксные и префиксные операторы `++` и `--` в языках Java и C. Тем не менее, процедуры `Inc()` и `Dec()` обычно компилируются в одну команду процессора.

Процедуры `Inc()` и `Dec()` можно вызывать и с одним, и с двумя параметрами. Ниже приведен пример их вызова с одним параметром.

```
Inc(variable);
Dec(variable);
```

Эти операторы будут скомпилированы в ассемблерные инструкции `add` и `sub`. Ниже приведен пример вызова таких процедур с двумя параметрами. Здесь переменная `variable` увеличивается, а затем уменьшается на `3`:

```
Inc(variable, 3);
Dec(variable, 3);
```

В табл. 2.4 приведены данные об операторах инкремента и декремента в разных языках программирования.

НА ЗАМЕТКУ

Если для компилятора установлен режим оптимизации, то процедуры `Inc()` и `Dec()` обычно порождают такой же машинный код, что и выражения *`variable:=variable + 1`*. Поэтому для увеличения или уменьшения значения целой переменной можно использовать тот вариант записи, который покажется более удобным.

Таблица 2.4. Операторы инкремента и декремента

Operator	*Pascal*	*Java/C*	*Visual Basic*
Инкремент	`Inc()`	`++`	Нет
Декремент	`Dec()`	`--`	Нет

Операторы присвоения с действием

Весьма удобные *операторы присвоения с действием* (do-and-assign operator), присущие языкам Java и C, в языке Object Pascal отсутствуют. Это такие операторы, как присвоение с суммой `+=` и присвоение с умножением `*=`. Они выполняют арифметическую операцию (в данном случае сложение и умножение) перед операцией присвоения. В Object Pascal, этот тип операций следует выполнять, используя два разных оператора. Например, следующий код C

```
x += 5;
```

на языке Object Pascal выглядит так.

```
x := x + 5;
```

Типы данных Object Pascal

Одним из самых больших достоинств языка Object Pascal является строгая типизация данных. В частности, это означает, что все реальные переменные, передаваемые в качестве параметров в функцию или процедуру, должны абсолютно точно соответствовать типу формальных параметров в объявлении данной функции или процедуры. В Object Pascal не существует предупреждений компилятора о подозрительных преобразованиях указателя, с которыми хорошо знакомы программисты на языке C. Компилятор Object Pascal не допустит вызова функции с типом указателя, отличным от того, который описан в объявлении этой функции. (Однако в функцию, параметр которой описан как нетипизированный указатель, можно передавать указатели любых типов.) В целом, строгая типизация данных в языке Pascal предназначена для предупреждения попыток заткнуть квадратной пробкой круглую дыру.

Сравнение типов данных

Основные типы данных Object Pascal схожи с типами данных языков Java, C и Visual Basic. В табл. 2.5 приведено сравнение типов данных этих языков. Настоятельно рекомендуем пометить настоящую страницу закладкой — она содержит отличный справочный материал, который будет полезен при вызове функций из динамически компонуемых библиотек, созданных другими компиляторами.

Таблица 2.5. Сравнение типов данных разных языков программирования

Тип переменной	*Pascal*	*Java*	*C/C++*	*Visual Basic*
8-битовое целое со знаком	`ShortInt`	`byte`	`char`	Нет
8-битовое целое без знака	`Byte`	Нет	`BYTE`, `unsigned short`	`Byte`
16-битовое целое со знаком	`SmallInt`	`short`	`short`	`Short`
16-битовое целое без знака	`Word`	Нет	`unsigned short`	Нет
32-битовое целое со знаком	`Integer`, `Longint`	`int`	`int`, `long`	`Integer`, `Long`
32-битовое целое без знака	`Cardinal`, `LongWord`	Нет	`unsigned long`	Нет
64-битовое целое со знаком	`Int64`	`long`	`__int64`	Нет
4-байтовое вещественное	`Single`	`float`	`float`	`Single`
6-байтовое вещественное	`Real48`	Нет	Нет	Нет
8-байтовое вещественное	`Double`	`double`	`double`	`Double`
110-байтовое вещественное	`Extended`	Нет	`long. double`	Нет
64-битовое денежное	`currency`	Нет	Нет	`Currency`
8-битовое дата/время	`TDateTime`	Нет	Нет	`Date`
16-байтовый вариант	`Variant`, `OleVariant`, `TVarData`	Нет	`VARIANT**`, `Variant***`, `OleVariant***`	`Variant` (По умолчанию)
1-байтовый символ	`Char`	Нет	`char`	Нет

Окончание табл. 2.5.

Тип переменной	*Pascal*	*Java*	*C/C++*	*Visual Basic*
2-байтовый символ	`WideChar`	`char`	`WCHAR`	Нет
Строка фиксированной длины	`ShortString`	Нет	Нет	Нет
Динамическая строка	`AnsiString`	Нет	`Ansi-String***`	`String`
Строка с завершающим нулевым символом	`PChar`	Нет	`char*`	Нет
Строка 2-байтовых символов с завершающим нулевым символом	`PWideChar`	Нет	`LPCWSTR`	Нет
Динамическая 2-байтовая строка	`WideString`	`String**`	`Wide-String***`	Нет
1-байтовое булево	`Boolean`, `ByteBool`	`boolean`	(Любое 1-байтовое)	Нет
2-байтовое булево	`WordBool`	Нет	(Любое 2-байтовое)	`Boolean`
4-байтовое булево	`BOOL`, `LongBool`	Нет	`BOOL`	Нет

*** Не соответствующий элемент языка, а обычно используемая структура или класс*

**** Классы Borland C++ Builder для эмуляции соответствующих типов Object Pascal*

НА ЗАМЕТКУ

При переносе 16-битового кода из Delphi 1.0 помните, что размер типов `Integer` и `Cardinal` вырос с 16 до 32 бит. Впрочем, это утверждение не совсем точно: в Delphi 2 и 3 тип `Cardinal` трактовался как 31-битовое целое без знака — в соответствии с теми результатами, которые могли быть получены при выполнении целочисленных операций. Но уже в Delphi 4 тип `Cardinal` представлял собой истинное 32-битовое целое без знака.

СОВЕТ

В Delphi 1, 2 и 3 тип `Real` определял 6-байтовое вещественное число. Такой тип присущ только языку Pascal и не совместим с другими языками программирования. В Delphi 4 этот тип стал синонимом типа `Double`. Старый 6-байтовый тип остался в языке под именем `Real48`, но с помощью директивы `{$REALCOMPATIBILITY ON}` можно заставить компилятор трактовать тип `Real` как 6-байтовый.

Символьные типы

Delphi поддерживает три символьных типа данных.

- `AnsiChar` — хорошо всем известный стандартный 1-байтовый символ ANSI.
- `WideChar` — 2-байтовый символ Unicode.
- `Char` — сейчас это тип, эквивалентный `AnsiChar`, но *Borland* предупреждает, что в последующих версиях он может измениться и стать эквивалентным `WideChar`.

Имейте в виду, поскольку символ (`Char`) не является больше гарантированно однобитовым, не стоит рассчитывать, что длина строки в символах будет всегда соответствовать ее размеру в байтах. Поэтому, чтобы выяснить настоящий размер строки, используйте функцию `SizeOf()`.

НА ЗАМЕТКУ

Стандартная функция `SizeOf()` возвращает размер в байтах переменной любого типа или экземпляра любого класса.

Многообразие строк

Строки представляют собой типы данных, используемые для представления групп символов. Каждый язык по-своему решает проблему размещения в памяти и использования строк. Object Pascal поддерживает несколько различных типов строк, и, исходя из конкретной ситуации, можно выбирать тот или иной строковый тип.

- `AnsiString` — строковый тип Object Pascal по умолчанию. Состоит из символов `AnsiChar`, длина практически неограничена. Совместим также со строками с завершающим нулевым символом.
- `ShortString` — остался в языке для совместимости с Delphi 1. Максимальная длина составляет 255 символов.
- `WideString` — по сути сходен с `AnsiString`. Единственное отличие заключается в том, что эта строка состоит из символов типа `WideChar`.
- `PChar` — представляет собой указатель на строку с завершающим нулевым символом, состоящую из символов типа `Char`. Аналог типов `char*` или `lpstr` в языке C.
- `PAnsiChar` — указатель на строку `AnsiChar` с завершающим нулевым символом.
- `PWideChar` — указатель на строку `WideChar` с завершающим нулевым символом.

По умолчанию при объявлении в коде строковой переменной (тип `string`) компилятор полагает, что создается строка типа `AnsiString`:

```
var
  S: string;   // Переменная S имеет тип AnsiString
```

Для изменения принимаемого по умолчанию типа строки используется директива компилятора `$H`. Ее положительное (по умолчанию) значение определяет использова-

ние в качестве стандартного строкового типа AnsiString, отрицательное — ShortString. Вот пример использования директивы $H для изменения строкового типа, выбираемого по умолчанию:

```
var
  {$H-}
  S1: string;  // Переменная S1 имеет тип ShortString
  {$H+}
  S2: string;  // Переменная S2 имеет тип AnsiString
```

Исключением из этого правила являются строки, объявленные с заранее установленным фиксированным размером. Если заданная длина не превышает 255 символов, такие строки всегда имеют тип ShortString:

```
var
  S: string[63];     // Это строка типа ShortString
                     // размером 63 символа
```

Тип AnsiString

Тип AnsiString, известный как "*длинная строка*" (long string), был введен в Delphi 2.0 в ответ на требования пользователей отменить 255-символьное ограничение на длину строки.

Хотя в применении тип AnsiString практически ничем не отличается от своего предшественника, память для него выделяется динамически, а для ее освобождения применяется технология автоматической "уборки мусора" (garbage collect). Благодаря этому AnsiString является типом *с управляемым временем жизни* (lifetime-managed). Object Pascal сам автоматически выделяет память для временных строк, так что об этом не придется беспокоиться (как и у языка C), а также единолично заботится о том, чтобы строка всегда была с завершающим нулевым символом (обеспечивая тем самым ее совместимость с интерфейсом API Win32). На самом деле тип AnsiString реализован как указатель на структуру, размещенную в *распределяемой памяти* (heap), и схематически показан на рис. 2.1.

Рис. 2.1. *Размещение* AnsiString *в памяти*

СОВЕТ

Полный внутренний формат длинной строки компанией *Borland* не документирован, а следовательно, она оставляет за собой право изменять его в дальнейших версиях языка. Поэтому приведенная здесь информация о структуре AnsiString служит только для того, чтобы облегчить понимание ее функционирования, однако использовать эти сведения для создания реальных программ не стоит.

Разработчики программ, которые избегали использования внутренней структуры строк, при переходе от Delphi 1 к Delphi 2 смогли перекомпилировать свои программы без проблем. Те же, кто опирались на внутренний формат строки (например на то, что нулевой символ строки содержит ее длину), должны были соответствующим образом изменить свой код.

Как показано на рис. 2.1, размещенная в памяти строка типа `AnsiString` обладает *счетчиком ссылок* (reference counted), содержащим количество строковых переменных, ссылающихся на одно и то же место в физической памяти. Таким образом копирование строки выполняется очень быстро, поскольку копируется не сама строка, а лишь указатель на нее (значение счетчика при этом увеличивается). Если две или более переменных типа `AnsiString`, обладающих ссылкой на одну и ту же запись в физической памяти, пытаются модифицировать ее, то *диспетчер памяти Delphi* (Delphi memory manager) использует методику *копирования для записи* (copy-on-write), которая позволяет второму процессу не ждать, пока модификация строки будет завершена, а перенести ссылку на вновь созданную в памяти физическую строку. Следующий пример иллюстрирует эту концепцию:

```
var
  S1, S2: string;
begin
  // поместить строку в S1, счетчик ссылок S1 равен 1
  S1 := 'And now for something... ';
  S2 := S1;   // Теперь S2 ссылается на S1.
              // Счетчик ссылок S1 равен 2.

  // Строка S2 изменяется, поэтому ее содержимое копируется в
  // новую область памяти, а значение счетчика ссылок S1
  // уменьшается на единицу
  S2 := S2 + 'completely different!';
```

Типы с управляемым временем жизни

Помимо `AnsiString`, в Delphi существует несколько других типов данных с управляемым временем жизни. Это динамические массивы `WideString`, `Variant`, `OleVariant`, `interface` и `dispinterface`. Далее в этой главе они будут описаны подробнее, а сейчас нас интересует только один вопрос: что такое управляемое время жизни и как это работает?

Подобные типы иногда называют типами "*с уборкой мусора*" (garbage-collected types). Они используют некоторые ресурсы компьютера и автоматически освобождают их при выходе из области видимости. Естественно, то, какие именно ресурсы требуются тому или иному типу, зависит исключительно от него. Так, тип `AnsiString` использует память компьютера для хранения своих данных, и при выходе из области видимости эта память должна быть освобождена.

Для глобальных переменных решение данной проблемы достаточно тривиально — соответствующий код освобождения ресурсов помещается в завершающий код приложения. Поскольку при начальной инициализации происходит инициализация глобальных переменных нулевыми значениями, т.е. фактически значениями "не используется", завершающий код может принимать решение о необходимости освобождения тех или иных ресурсов по состоянию глобальной переменной и выполнять "уборку мусора" только там, где это необходимо.

Для локальных переменных этот процесс заметно сложнее. Во-первых, каждый раз, при объявлении локальной переменной с управляемым временем жизни, компилятору необходимо внести код, который инициализирует переменную при входе в функцию или процедуру. Во-вторых, компилятор генерирует блок обработки исключений `try..finally` вокруг всего тела функции. И, наконец, компилятор помещает код "уборки мусора" в часть `finally` блока. (Более подробная информация об обработке

исключений приведена в настоящей главе далее.) Чтобы лучше представить себе эти процессы, рассмотрим простейший пример:

```
procedure Foo;
  var
    S: string;
  begin
    // Тело процедуры. Переменная
    // S используется здесь
  end;
```

Хотя процедура и выглядит достаточно просто, но на самом деле компилятор вынужден будет создать следующий код:

```
procedure Foo;

  var
    S: string;
  begin
    S := '';
    try
      // Тело процедуры. Переменная
      // S используется здесь
    finally
      // освобождение S происходит здесь
    end;
  end;
```

Строковые операции

Сложить две строки можно либо с помощью оператора +, либо с помощью функции `Concat()`. Предпочтительнее использовать оператор +, так как функция `Concat()` сохранена в основном из соображений совместимости с прежними версиями. Вот примеры использования оператора + и функции `Concat()`:

```
{ Применение оператора + }
var
  S, S2: string
begin
  S := 'Cookie ';
  S2 := 'Monster';
  S := S + S2;   { Cookie Monster }
end.

{ Применение функции Concat() }
var
  S, S2: string;
begin
  S := 'Cookie ';
  S2 := 'Monster';
  S := Concat(S, S2); { Cookie Monster }
end.
```

НА ЗАМЕТКУ

При работе со строками в Object Pascal не забывайте, что использовать необходимо *одинарные* кавычки (`'Это строка'`).

СОВЕТ

Функция `Concat()` — это одна из многих "волшебных палочек компилятора", подобных функциям `ReadLn()` и `WriteLn()`, которые в Object Pascal не имеют определения. Такие функции и процедуры специально предназначены для того, чтобы принимать неопределенное количество параметров или необязательные параметры. Поэтому эти функции не могут быть определены в терминах языка Object Pascal, а компилятор вынужден сделать исключение для каждой из них и создать код специального обращения к одной из "волшебных палочек" — *вспомогательной функции* (helper function), которые определены в модуле `System`. Вспомогательные функции реализованы на языке ассемблер специально для того, чтобы обойти правила языка Pascal.
Кроме функций поддержки работы со строками, Delphi обладает некоторыми другими функциями модуля `SysUtils`, предназначенными для упрощения работы со строками. О них можно узнать в разделе "String-handling routines (Pascal-style)" справочной системы Delphi.

Длина строк и размещение в памяти

Вначале, при объявлении, строка `AnsiString` не имеет длины, а значит, память для ее содержимого не выделяется. Чтобы выделить строке память, необходимо либо присвоить ей некоторое строковое значение (например литерал), либо использовать процедуру `SetLength()`, как показано ниже.

```
var
  S: string;         // Вначале строка не имеет длины
begin
  S := 'Doh!';       // Выделяется память, необходимая для
                     // размещения строки
  { или }
  S := OtherString   // Увеличить счетчик ссылок другой строки
  // (Предполагается, что строка OtherString уже имеет значение)
  { или }
  SetLength(S, 4);   // Выделяется достаточно памяти, по крайней
                     // мере для размещения 4 символов
end;
```

Строку типа `AnsiString` можно рассматривать как массив символов, но будьте осторожны — индекс элемента массива не может превышать длину строки. Так, показанный ниже код приведет к ошибке.

```
var
  S: string;
begin
  S[1] := 'a';  // Ошибка! Память для S не выделена!
end;
```

А этот код будет работать правильно:

```
var
  S: string;
begin
  SetLength(S, 1);
  S[1] := 'a';       // Теперь у S достаточно места, чтобы
                     // содержать один символ
end;
```

Совместимость с Win32

Как уже говорилось, строка типа AnsiString всегда завершается нулевым символом (null). Поэтому такая строка вполне совместима с функциями интерфейса API Win32 или любыми другими, использующими параметры типа PChar. Все, что в данном случае необходимо, — это преобразование типа string в тип PChar (о преобразовании типов в Object Pascal можно узнать далее в этой главе). Приведенный ниже пример демонстрирует вызов функции Win32 GetWindowsDirectory(), которой в качестве параметров передается указатель на буфер PChar и его размер.

```
var
  S: string;
begin
  SetLength(S, 256); // Важно! Сначала выделяем память для строки
  // Теперь, после вызова функции, S будет содержать имя каталога
  GetWindowsDirectory(PChar(S), 256);
end;
```

Использовав строку AnsiString в качестве аргумента функции, ожидавшей тип PChar, необходимо вручную установить длину строковой переменной, равной длине строки с завершающим нулевым символом. Для этого можно воспользоваться процедурой RealizeLength() из упомянутого ранее модуля StrUtils.

```
procedure RealizeLength(var S: string);
begin
  SetLength(S, StrLen(PChar(S)));
end;
```

Вызов процедуры RealizeLength() завершает преобразование строки AnsiString в строку PChar:

```
var
  S: string;
begin
  SetLength(S, 256); // Важно! Сначала выделяем память для строки
  // Теперь, после вызова функции, S будет содержать имя каталога
  GetWindowsDirectory(PChar(S), 256);
  RealizeLength(S);  // установить длину S по символу null
end;
```

СОВЕТ

Проявляйте определенную осторожность при приведении типов string к типу PChar. Поскольку string — это тип с управляемым временем жизни, следует обратить внимание на область видимости соответствующих переменных. Так, если сделать присвоение типа P := PChar(Str), а область видимости P больше, чем Str, результат может оказаться плачевным.

Вопросы переносимости

При переносе на новую 32-битовую платформу старых 16-битовых приложений, созданных с помощью Delphi 1.0 и использующих тип AnsiString, необходимо помнить следующее.

- Там, где применялся тип `PString` (указатель на `ShortString`), следует использовать тип `String`. Помните, что `AnsiString` представляет собой указатель на строку.
- Больше нельзя использовать нулевой элемент строки (как массива) для возвращения ее длины. Теперь для этого необходимо прибегать к функции `Length()`, а для определения или установки нового значения длины строки — к процедуре `SetLength()`.
- Для приведения типов строк между `String` и `PChar` больше не требуются функции `StrPas()` и `StrPCopy()`. Как уже было сказано, можно выполнять прямое преобразование типа `AnsiString` в `PChar`. При копировании содержимого `PChar` в `AnsiString` можно использовать обычное присвоение: `StringVar := PCharVar;`.

СОВЕТ

Не забывайте, что для установки длины строк типа `AnsiString` должна использоваться процедура `SetLength()`. Прежняя практика прямого обращения к нулевому элементу коротких строк здесь не применима. Это замечание следует помнить при переносе 16-разрядных приложений Delphi 1.0 в 32-разрядную среду.

Тип ShortString

Ветераны Delphi помнят, что `ShortString` — это тип `String` в Delphi 1.0, он же тип с *байтом длины строки* (length-byte strings), он же тип *строки Pascal* (Pascal string). Еще раз напомним, что директива компилятора `$H` позволяет выбрать, какой именно тип будет подразумеваться под именем `String` — `AnsiString` или `ShortString`.

В памяти строка `ShortString` представляет собой массив символов, причем нулевой элемент данного массива содержит длину строки. Последующие символы представляют собой содержимое строки, при этом ее максимальная длина не может превышать 255 байт, а вся строка в памяти не может занимать более 256 байт (255 байт строки и один байт длины). Как и в случае с типом `AnsiString`, при работе со строками `ShortString` не нужно заботиться о выделении памяти для временных строк или ее освобождении — всю эту работу компилятор берет на себя (в отличие от языка C).

На рис. 2.2 показано размещение строки `ShortString` в памяти.

Рис. 2.2. *Размещение строки типа `ShortString` в памяти*

Ниже приведено объявление и инициализация строки типа `ShortString`.

```
var
  S: ShortString;
begin
  S := 'Bob the cat.';
end.
```

При необходимости строке можно выделить менее 255 байт памяти, для чего в ее объявление следует поместить явное указание длины:

```
var
  S: string[45];  { 45-символьная строка }
begin
  S := 'This string must be 45 or fewer characters.';
     // В этой строке не более 45 символов.
end.
```

В приведенном примере строка типа `ShortString` была создана вне зависимости от состояния директивы компилятора `$H`. Следовательно, ее максимально допустимая длина составляет 255 символов.

Не следует сохранять в строке символов больше, чем составляет ее реальная длина. Если строка была описана как `String[8]`, то при попытке сохранить в ней значение `'a_pretty_darn_ long_string'` строка будет обрезана до восьми символов, а остальная часть данных потеряна.

При обращении к отдельным символам в строке `ShortString` как к элементам массива необходимо следить за тем, чтобы индекс элемента не выходил за пределы размера строки, а следовательно, и выделенной для нее памяти. В противном случае полученные результаты будут ошибочными (при чтении) либо произойдет повреждение других данных, расположенных в памяти рядом (при записи). Предположим, что переменная была объявлена следующим образом:

```
var
  Str: string[8];
```

Если выполнить запись в десятый элемент такой строки, то в результате будут повреждены размещенные в памяти данные, принадлежащие другой переменной (вероятнее всего `i`, поскольку она должна быть размещена в памяти сразу за `Str`, если диспетчер памяти Delphi не имеет ничего против).

```
var
  Str: string[8];
  i: Integer;
begin
  i := 10;
  Str[i] := 's';  // повредит данные в памяти
```

Компилятор можно заставить встраивать в исполняемый файл специальный код, который осуществляет перехват и обработку ошибок выхода индекса за допустимые пределы в процессе выполнения программы. Для этого необходимо установить флажок Range Checking (Проверка диапазона) во вкладке Compiler диалогового окна Project Options.

СОВЕТ

Несмотря на то, что включение в программу *механизма проверки диапазона* (range-checking) помогает избежать ошибок при работе со строками, в целом это немного снижает производительность приложения. Поэтому проверку диапазона используют обычно во время разработки и отладки приложения, а при компиляции окончательной версии его отключают.

В отличие от строк типа `AnsiString`, строки `ShortString` не совместимы со строками в формате с завершающим нулевым символом. Чтобы использовать их в функциях интерфейса API Win32, необходима определенная предварительная обработка. Для это-

го можно воспользоваться функцией ShortStringAsPChar(), которая входит в состав уже упомянутого модуля STRUTILS.PAS. Текст такой функции приведен ниже.

```
function ShortStringAsPChar(var S: ShortString): PChar;
{ Функция добавляет к исходной строке завершающий нулевой символ, чтобы
она могла быть передана тем функциям, которые ожидают строку типа PChar.
Если строка окажется длиннее 254 символов, то она будет усечена. }
begin
  if Length(S) = High(S) then Dec(S[0]);
  { Усечение строки S, если она слишком длинная }
  S[Ord(Length(S)) + 1] := #0;
  { Добавление нулевого символа в конец строки }
  Result := @S[1];
  { Возвращает строку типа Pchar }
end;
```

СОВЕТ

Функции и процедуры интерфейса API Win32 работают только со строками с завершающим нулем. Не пытайтесь передать им строки типа ShortString непосредственно — программа просто не будет компилироваться. Можно существенно облегчить себе жизнь, если при работе с функциями интерфейса API использовать только длинные строки типа AnsiString.

Тип WideString

Это тип строк с управляемым временем жизни, подобный типу AnsiString. Данные обоих типов располагаются в динамической памяти, обладают автоматической "уборкой мусора" и даже по назначению схожи. Но между типами WideString и AnsiString существует три таких важных отличия:

- Строка WideString состоит из символов WideChar, что делает ее совместимой со строками Unicode.
- Строки WideString используют память, выделенную с помощью функции интерфейса API SysAllocStrLen(), что делает их совместимыми со строками OLE BSTR.
- В строках WideString отсутствует счетчик ссылок, поэтому любое присвоение одной строки другой приводит к выделению памяти и копированию строки, что делает этот тип строк менее эффективным, чем AnsiString, с точки зрения производительности и использования памяти.

Компилятор способен автоматически конвертировать строки типов WideString и AnsiString при присвоении, что и показано в следующем примере:

```
var
  W: WideString;
  S: string;
begin
  W := 'Margaritaville';
  S := W;  // Преобразование WideString в AnsiString
  S := 'Come Monday';
  W := S;  // Преобразование AnsiString в WideString
end;
```

Для обеспечения работы со строками этого типа Object Pascal перегружает функции `Concat()`, `Copy()`, `Insert()`, `Length()`, `Pos()` и `SetLength()`, а также операторы +, = и <> для работы с `WideString`. Таким образом, следующий код синтаксически безупречен:

```
var
  W1, W2: WideString;
  P: Integer;
begin
  W1 := 'Enfield';
  W2 := 'field';
  if W1 <> W2 then P := Pos(W1, W2);
end;
```

Как и в случае строк типа `AnsiString` или `ShortString`, для ссылок на отдельные символы строки `WideString` можно использовать квадратные скобки:

```
var
  W: WideString;
  C: WideChar;
begin
  W := 'Ebony and Ivory living in perfect harmony';
  C := W[Length(W)];  // C содержит последний символ строки W
end;
```

Строки с завершающим нулевым символом

Ранее уже говорилось, что в Delphi существует три различных типа строк с завершающим нулевым символом: `PChar`, `PAnsiChar` и `PWideChar`. Как и следует из их названий, это строки с завершающим нулем, соответствующие трем основным типам строк Delphi. Впоследствии в настоящей главе под типом `PChar` будем подразумевать именно эти типы строк. В основном тип `PChar` обеспечивает совместимость с Delphi 1.0 и работу с интерфейсом API Win32, широко использующим строки с завершающим нулевым символом. Тип `PChar` определяется как указатель на строку с завершающим нулевым символом. (Более подробная информация об указателях приведена в этой главе далее.) В отличие от строк `AnsiString` и `WideString`, память для строк типа `PChar` автоматически не выделяется (и не освобождается). Это означает, что для тех строк, на которые указывают данные указатели, память потребуется выделять вручную — с помощью одной из существующих в Object Pascal функций выделения памяти. Теоретически длина строки `PChar` может достигать 4 Гбайт. Ее размещение в памяти показано на рис. 2.3.

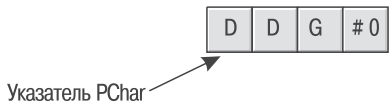

Рис. 2.3. *Размещение строки типа `PChar` в памяти*

СОВЕТ

Поскольку в большинстве случаев вместо типа `PChar` можно использовать тип `AnsiString`, рекомендуется применять его везде, где это возможно. При этом удается избежать множества неприятностей, связанных с выделением и освобождением памяти, поскольку при работе с `AnsiString` компилятор берет заботу об этом на себя.

Как уже отмечалось, переменные типа `PChar` требуют непосредственного выделения и освобождения буферов, содержащих соответствующие строки. Обычно выделение памяти осуществляется с помощью функции `StrAlloc()`, однако для создания буферов `PChar` можно применять и другие из них, такие, например, как `AllocMem()`, `GetMem()`, `StrNew()` или даже функции API `VirtualAlloc()`. Каждой функции соответствует функция освобождения памяти (табл. 2.6).

Таблица 2.6. Функции выделения и освобождения памяти

Функция выделения памяти	*Функция освобождения памяти*
`AllocMem()`	`FreeMem()`
`GlobalAlloc()`	`GlobalFree()`
`GetMem()`	`FreeMem()`
`New()`	`Dispose()`
`StrAlloc()`	`StrDispose()`
`StrNew()`	`StrDispose()`
`VirtualAlloc()`	`VirtualFree()`

Вот пример выделения памяти при работе с данными типа `PChar` и `String`:

```
var
  P1, P2: PChar;
  S1, S2: string;
begin
  P1 := StrAlloc(64 * SizeOf(Char));
     // P1 указывает на участок памяти для размещения 63 символов
  StrPCopy(P1, 'Delphi 6 ');  // Копирует набор символов в P1
  S1 := 'Developer''s Guide'; // Помещает текст в строку S1
  P2 := StrNew(PChar(S1));    // P1 указывает на копию S1
  StrCat(P1, P2);             // Объединяет P1 и P2
  S2 := P1;                   // Теперь S2 содержит строку
                              // 'Delphi 6 Developer's Guide'
  StrDispose(P1);             // Освобождает буферы P1 и P2
  StrDispose(P2);
end.
```

Обратите особое внимание на то, что при выделении памяти для `P1` использовалась функция `SizeOf(Char)`. Вполне вероятно, что в будущих версиях Delphi размер символа может измениться с одного до двух байт, а такой способ выделения памяти для строки будет корректно работать при любых изменениях в размере символа.

Для объединения двух строк использовалась функция StrCat() — при работе с PChar нельзя применять оператор +, как это делается в случае со строками типа AnsiString или ShortString.

Функция StrNew() использовалась для выделения памяти строки P2 и одновременного копирования в нее строки S1. Но не следует забывать, что память при этом выделяется в количестве, необходимом для хранения лишь строки конкретного размера. Попытка записи в такой буфер более длинной строки приведет к ошибке памяти. Следующий пример иллюстрирует подобную ситуацию.

```
var
P1, P2: Pchar;
begin
  P1 := StrNew('Hello ');
  P2 := StrNew('World');
  // Выделена память только для P1 и P2
  StrCat(P1, P2);          // ОСТОРОЖНО, запись за пределы
                           // выделенной памяти!
  .
  .
  .
end;
```

СОВЕТ

Как и в случае с другими типами строк, для работы с типом PChar Object Pascal предоставляет различные функции и процедуры. Подробные сведения о них можно найти в разделе “String-handling routines (null-terminated)” справочной системы Delphi.

Тип Variant

В Delphi 2.0 был введен новый мощный тип данных — Variant. В основном его назначение заключалось в поддержке автоматизации OLE (OLE Automation), где тип данных Variant используется очень широко. Фактически тип Variant языка Object Pascal является инкапсуляцией *вариантов* OLE. Как мы вскоре убедились, реализация в Delphi вариантов оказалась полезной и с точки зрения других аспектов программирования. Object Pascal является единственным компилируемым языком, в котором для работы с вариантами OLE введен специализированный тип данных, представляемый как динамический во время выполнения программы и как статический во время ее компиляции.

В Delphi 3 для этой же цели был введен еще один новый тип данных — OleVariant, полностью идентичный типу Variant, за исключением того, что он может содержать лишь те типы, которые совместимы с OLE. Далее в этом разделе тип Variant и отличия между типами данных OleVariant и Variant рассматривается более подробно.

Variant — динамическое изменение типа

Основное назначение типа Variant — предоставить возможность объявить переменную, тип которой неизвестен на момент компиляции. Это означает, что настоящий тип такой переменной может изменяться в процессе выполнения программы.

Например, приведенная ниже программа корректна и будет компилироваться и выполняться без ошибок.

```
var
  V: Variant;             // V - переменная типа Variant
begin
  V := 'Delphi is Great!';// V содержит строку (тип string)
  V := 1;                 // V - целое число (тип Integer)
  V := 123.34;            // V - вещественное число (тип float)
  V := True;              // V - логическое значение (тип boolean)
  V := CreateOleObject('Word.Basic'); // V содержит объект OLE
end;
```

Варианты способны поддерживать все простые типы Object Pascal, такие как целые и вещественные числа, строки, значения даты и времени и т.п. Кроме того, они могут содержать объекты автоматизации OLE. Однако отметим, что они не могут быть ссылками на объекты Object Pascal, но могут ссылаться на неоднородные массивы, состоящие из переменного количества элементов, имеющих различный тип (в том числе они могут быть и другими массивами вариантов).

Структура определения данных типа Variant

Структура определения данных типа `Variant` описана в модуле `System` и выглядит следующим образом:

```
TVarType = Word;
PVarData = ^TVarData;
{$EXTERNALSYM PVarData}
TVarData = packed record
  VType: TVarType;
  case Integer of
    0: (Reserved1: Word;
      case Integer of
        0: (Reserved2, Reserved3: Word;
          case Integer of
            varSmallInt: (VSmallInt: SmallInt);
            varInteger:  (VInteger: Integer);
            varSingle:   (VSingle: Single);
            varDouble:   (VDouble: Double);
            varCurrency: (VCurrency: Currency);
            varDate:     (VDate: TDateTime);
            varOleStr:   (VOleStr: PWideChar);
            varDispatch: (VDispatch: Pointer);
            varError:    (VError: LongWord);
            varBoolean:  (VBoolean: WordBool);
            varUnknown:  (VUnknown: Pointer);
            varShortInt: (VShortInt: ShortInt);
            varByte:     (VByte: Byte);
            varWord:     (VWord: Word);
            varLongWord: (VLongWord: LongWord);
            varInt64:    (VInt64: Int64);
            varString:   (VString: Pointer);
```

```
            varAny:      (VAny: Pointer);
            varArray:    (VArray: PVarArray);
            varByRef:    (VPointer: Pointer);
          );
        1: (VLongs: array[0..2] of LongInt);
      );
    2: (VWords: array [0..6] of Word);
    3: (VBytes: array [0..13] of Byte);
end;
```

Структура TVarData занимает 16 байт памяти. Первых два байта этой структуры содержат слово, значение которого определяет, на какой именно тип данных ссылается вариант. Ниже приведены конкретные значения, соответствующие различным типам данных, которые могут быть помещены в поле VType записи TVarData. Следующие 6 байт записи не используются. Последние 8 байт содержат либо настоящие данные, либо указатель на данные, представляемые этим вариантом. Необходимо отметить, что данная структура точно соответствует требованиям, предъявляемым к реализации вариантов OLE.

```
{ Коды типов Variant (wtypes.h) }

  varEmpty    = $0000; { vt_empty       }
  varNull     = $0001; { vt_null        }
  varSmallint = $0002; { vt_i2          }
  varInteger  = $0003; { vt_i4          }
  varSingle   = $0004; { vt_r4          }
  varDouble   = $0005; { vt_r8          }
  varCurrency = $0006; { vt_cy          }
  varDate     = $0007; { vt_date        }
  varOleStr   = $0008; { vt_bstr        }
  varDispatch = $0009; { vt_dispatch    }
  varError    = $000A; { vt_error       }
  varBoolean  = $000B; { vt_bool        }
  varVariant  = $000C; { vt_variant     }
  varUnknown  = $000D; { vt_unknown     }
//varDecimal  = $000E; { vt_decimal     } { Не поддерживается }
                       { undefined  $0f } { Не поддерживается }
  varShortInt = $0010; { vt_i1          }
  varByte     = $0011; { vt_ui1         }
  varWord     = $0012; { vt_ui2         }
  varLongWord = $0013; { vt_ui4         }
  varInt64    = $0014; { vt_i8          }
//varWord64   = $0015; { vt_ui8         } { Не поддерживается }

{ При добавлении новых элементов необходимо модифицирвать varLast,
BaseTypeMap и OpTypeMap вариантов }
  varStrArg   = $0048; { vt_clsid       }
  varString   = $0100; { Строка Pascal; с OLE несовместима }
  varAny      = $0101; { Любой Corba }
  varTypeMask = $0FFF;
  varArray    = $2000;
  varByRef    = $4000;
```

НА ЗАМЕТКУ

Как можно заметить из вышеприведенного кода, `Variant` не может содержать ссылку на данные типа `Pointer` или `class`.

Из описания структуры записи `TVarData` видно, что она действительно может содержать данные любого типа. Следует отличать приведенную выше запись `TVarData` от действительных данных типа `Variant`. Хотя запись переменного состава и данные типа `Variant` (вариант) внешне схожи по своему назначению, они представляют две совершенно разные конструкции. Запись `TVarData` позволяет хранить данные различных типов в одной и той же области памяти (подобно объединениям — `union` языка C/C++). Более подробная информация по этой теме приведена далее в настоящей главе. Оператор `case` в описании записи `TVarData` служит для определения типов данных, которые может представлять вариант. Так, если в поле `VType` содержится значение `varInteger`, то только четыре из восьми байтов данных записи будут содержать хранимое переменной целое значение. Подобным образом, если в поле `VType` содержится значение `varByte`, только один из восьми байтов области данных будет использоваться для хранения значения.

Обратите внимание, если `VType` содержит значение `varString`, то восемь байтов данных в записи содержат не реальную строку, а лишь указатель на нее. Это очень важно понимать, поскольку при работе с вариантами можно получить непосредственный доступ к значениям любого поля, как показано в приведенном ниже примере.

```
var
  V: Variant;
begin
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 2;
end;
```

Следует отдавать себе отчет в том, что такое использование варианта — опасная игра, так как при этом, например, можно легко разрушить указатель на строку или другой объект с управляемым временем жизни. В результате объект станет недоступным для процессов "уборки мусора", что приведет к *утечке памяти* (leaking memory) и потере других ресурсов приложения. Более подробная информация о процессах "уборки мусора" приведена в следующем разделе.

Варианты — объекты с управляемым временем жизни

Delphi автоматически выделяет и освобождает память для данных типа `Variant`. Рассмотрим приведенный ниже пример, в котором варианту присваивается строка.

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := S;
  ShowMessage(V);
end;
```

Как уже отмечалось в этой главе, в отношении объектов с управляемым временем жизни, при выполнении программы осуществляется несколько действий, которые не вполне очевидны. Прежде всего Delphi инициализирует вариант пустым значением. Затем, в ходе присвоения значения, в поле VType помещается значение varString, а в поле VString копируется указатель на строку. При этом счетчик ссылок строки S увеличивается. Когда вариант покидает область видимости (в данном случае — при выходе из процедуры), он удаляется и счетчик ссылок строки S уменьшается. Delphi реализует это, помещая тело процедуры в блок try..finally, как показано ниже.

```
procedure ShowVariant(S: string);
var
  V: Variant
begin
  V := Unassigned;  // Инициализирует вариант как "пустой"
  try
    V := S;
    ShowMessage(V);
  finally
    // Теперь можно освободить ресурсы, связанные с вариантом
  end;
end;
```

Точно такое же неявное освобождение ресурсов происходит и при присвоении варианту данных нового типа. Рассмотрим следующий пример:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  V := S;
  V := 34;
end;
```

Данный код сводится к приведенному ниже псевдокоду:

```
procedure ChangeVariant(S: string);
var
  V: Variant
begin
  Освободить вариант V, инициализировав его как "пустой"
  try
    V.VType := varString; V.VString := S; Inc(S.RefCount);
    Освободить вариант V, сославшись на другие данные;
    V.VType := varInteger; V.VInteger := 34;
  finally
    Освободить ресурсы, связанные с вариантом
  end;
end;
```

Разобравшись в работе этого примера, можно понять, почему не рекомендуется непосредственно манипулировать полями записи TVarData, как в данном случае:

```
procedure ChangeVariant(S: string);
var
```

```
  V: Variant
begin
  V := S;
  TVarData(V).VType := varInteger;
  TVarData(V).VInteger := 32;
  V := 34;
end;
```

Хотя все кажется вполне корректным, это далеко не так — ведь уменьшение счетчика ссылок строки S не выполняется, что приводит к утечке памяти. Одним словом, никогда не работайте с полями TVarData напрямую, а если все же приходится делать это, то следует четко понимать, что именно происходит и к каким последствиям это может привести.

Преобразование типов для вариантов

Приведение к типу Variant можно осуществлять явно. Так, выражение Variant(X) дает в результате вариант, тип которого соответствует результату выражения X. Последнее должно быть целым, вещественным, строкой, символом, денежным или булевым типом.

Можно преобразовывать тип Variant и в другие типы. Рассмотрим такой пример:

```
V := 1.6;
```

Здесь V переменная типа Variant, а следующие выражения представляют различные преобразования типов, возможные для него:

```
S := string(V);  // S будет содержать строку '1.6'
I := Integer(V); // I будет равно ближайшему целому, т.е. 2.
B := Boolean(V); // B содержит False (если V равно 0) иначе - True
D := Double(V);  // D получает значение 1.6
```

Результаты всех этих преобразований получены в соответствии с определенными правилами приведения типов, применимых к переменным типа Variant. Более подробную информацию о преобразовании типов можно получить из руководства по Delphi "*Object Pascal Language Guide*".

Заметим, что в предыдущем примере явное приведение типов не являлось неизбежным. Показанный ниже код по своим возможностям идентичен предыдущему.

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

В этом примере преобразование типов происходит неявно. Однако за счет того, что оно осуществляется непосредственно в процессе выполнения программы, в состав последней включается соответствующий дополнительный код. Какой метод предпочесть — зависит от вас. Если тип реальных данных варианта известен на стадии компиляции, то лучше использовать явное преобразование типов, которое более эффективно (так как выполняется на стадии компиляции). Особенно справедливо это замечание в тех случаях, когда вариант используется в выражениях, речь о которых пойдет ниже.

Использование вариантов в выражениях

Варианты можно использовать в выражениях со следующими операторами: +, -, =, *, /, div, mod, shl, shr, and, or, xor, not, :=, <>, <, >, <=, >=.

Используя варианты в выражениях, Delphi принимает решение о том, как именно должны выполняться операторы, на основании текущего типа содержимого варианта. Например, если варианты V1 и V2 содержат целые числа, то результатом выражения V1+V2 станет их сумма. Если они содержат строки, то результатом будет их объединение. А что произойдет, если типы данных различны? В этом случае Delphi использует некоторые правила, чтобы определить, какие именно преобразования необходимо выполнить. Так, если V1 содержит строку '4.5', а V2 — вещественное число, то V1 будет преобразовано в число 4.5 и добавлено к значению V2. Рассмотрим следующий пример:

```
var
  V1, V2, V3: Variant;
begin
  V1 := '100';  // Строковый тип
  V2 := '50';   // Строковый тип
  V3 := 200;    // Тип Integer
  V1 := V1 + V2 + V3;
end;
```

Исходя из сделанного выше замечания, можно предположить, что в результате будет получено целое значение 350. Но это не так. Поскольку вычисление выражений выполняется слева направо, то при первом сложении (V1 + V2) складываются две *строки* и в результате должна получиться тоже *строка*, имеющая значение '10050'. А уже затем полученный результат будет преобразован в целое значение и просуммирован с третьим целочисленным операндом, в результате чего будет получено значение 10250.

В Delphi для успешного выполнения вычислений данные типа Variant всегда преобразуются в самый высокий тип данных, присутствующих в выражении. Но если в операции участвуют два варианта, для которых Delphi не в состоянии подобрать подходящий тип, генерируется и передается исключение *invalid variant type conversion* (недопустимое преобразование типов варианта). Вот простейший пример такой ситуации:

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  V1 := V1 / V2;  // Возникает исключение.
end;
```

Как уже отмечалось, использовать явное преобразование во время компиляции — хорошая мысль. Так, в случае даже такой обычной операции, как V4 := V1*V2/V3, в процессе выполнения программы Delphi вынужден рассматривать множество вариантов преобразования типов для того, чтобы найти наиболее подходящий. Явное же указание типов, например V4 := Integer(V1)*Double(V2)/Integer(V3), позволяет принять решение еще на стадии компиляции, избежав тем самым излишних затрат времени при работе программы. Правда, при этом следует точно знать, какой именно тип данных будет содержится в варианте.

Пустое значение и значение Null

Два специальных значения поля VType вариантов заслуживают отдельного обсуждения. Первое — varEmpty (вариант пуст или неопределен) — означает, что варианту пока не назначено никакого значения. Это начальное значение варианта, которое компилятор устанавливает при входе переменной в область видимости. Второе значение — varNull (вариант содержит ничто) — отличается от varEmpty тем, что оно представляет реально существующее значение переменной, которое равно Null. Это отличие особенно важно при работе с базами данных, где отсутствие значения и значение Null — абсолютно разные вещи. Более подробная информация о применении вариантов в контексте приложений для работы с базами данных приведена в части III, "Разработка баз данных".

Еще одно отличие этих значений состоит в том, что любая попытка вычисления выражений с пустым вариантом приведет к возникновению исключения *invalid variant operation* (недопустимая операция с вариантом). Но при использовании выражения варианта со значением Null этого не произойдет, поскольку результат вычисления любого выражения, в состав которого входит значение Null, всегда будет равен Null.

Если необходимо присвоить или сравнить вариант с одним из таких значений, то используется один из двух предопределенных в модуле System специальных вариантов — Unassigned и Null, у которых поля VType соответственно имеют значения varEmpty и varNull.

СОВЕТ

За все в этой жизни приходится расплачиваться, и варианты — не исключение. Удобство работы и высокая гибкость достигаются ценой увеличения размера и замедления работы приложения. Кроме того, повышается сложность сопровождения создаваемого программного обеспечения. Естественно, бывают ситуации, когда без вариантов трудно обойтись. В частности, благодаря их гибкости, они достаточно широко применяются в визуальных компонентах, особенно в элементах управления ActiveX и компонентах для работы с базами данных. Тем не менее, в большинстве случаев рекомендуется работать с обычными типами данных. Старайтесь использовать варианты только в тех ситуациях, когда без них действительно нельзя обойтись и когда увеличение размера и замедление работы приложения — разумная плата за гибкость. Не забывайте, что использование неоднозначных типов данных приводит к появлению неоднозначных ошибок.

Варианты и массивы

Как отмечалось ранее, варианты способны представлять *неоднородные* (nonhomogeneous) массивы, поэтому следующий код синтаксически абсолютно корректен:

```
var
  V: Variant;
  I, J: Integer;
begin
  I := V[J];
end;
```

Object Pascal предоставляет несколько функций, позволяющих создать вариант, который является массивом, в частности функции VarArrayCreate() и VarArrayOf().

Функция VarArrayCreate()

Функция `VarArrayCreate()` определена в модуле `System` следующим образом:

```
function VarArrayCreate(const Bounds: array of Integer;
                        VarType: Integer): Variant;
```

Функции `VarArrayCreate()` в качестве параметров передаются границы создаваемого массива и код типа варианта элементов создаваемого массива. (Первый параметр представляет собой открытый массив, речь о котором пойдет далее в этой главе.) Например, в приведенном ниже фрагменте кода создается вариант-массив целых чисел, после чего его членам присваиваются необходимые значения.

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 4], varInteger); // Создает массив
                                           // из 4 элементов
  V[1] := 1;
  V[2] := 2;
  V[3] := 3;
  V[4] := 4;
end;
```

Но это еще не все: передав в качестве параметра тип `varVariant`, можно создать *вариант-массив вариантов* (variant array of variants), который представляет собой массив, элементы которого могут содержать значения разных типов. Кроме того, точно так же можно создавать и многомерные массивы, для чего достаточно просто указать дополнительные значения границ. Например, в приведенном ниже фрагменте кода создается двухмерный вариант-массив размерностью `[1..4, 1..5]`.

```
V := VarArrayCreate([1, 4, 1, 5], varInteger);
```

НА ЗАМЕТКУ

Модуль `Variants` был добавлен в RTL лишь в Delphi 6, поскольку только теперь поддержка вариантов была перенесена из модуля `System` в отдельный модуль. Кроме всего прочего, подобное физическое разделение кода позволило плавно перейти к совместимости с Kylix, а кроме того обеспечило поддержку вариантов для пользовательских типов данных.

Функция VarArrayOf()

Эта функция определена в модуле `System` следующим образом:

```
function VarArrayOf(const Values: array of Variant): Variant;
```

Она возвращает одномерный массив, элементы которого были заданы в параметре `Values`. Ниже приведен пример создания массива из трех элементов: целого, строки и вещественного числа.

```
V := VarArrayOf([1, 'Delphi', 2.2]);
```

Функции и процедуры для работы с вариантами-массивами

Помимо функций `VarArrayCreate()` и `VarArrayOf()`, существуют другие функции и процедуры, предназначенные для работы с вариантами-массивами. Они определены в модуле `System` таким образом:

```
procedure VarArrayRedim(var A: Variant; HighBound: Integer);
function VarArrayDimCount(const A: Variant): Integer;
function VarArrayLowBound(const A: Variant;
                          Dim: Integer): Integer;
function VarArrayHighBound(const A: Variant;
                           Dim: Integer): Integer;
function VarArrayLock(const A: Variant): Pointer;
procedure VarArrayUnlock(const A: Variant);
function VarArrayRef(const A: Variant): Variant;
function VarIsArray(const A: Variant): Boolean;
```

Функция `VarArrayRedim()` позволяет изменять верхнюю границу размерности самого варианта-массива. Функция `VarArrayDimCount()` возвращает размерность варианта-массива, а функции `VarArrayLowBound()` и `VarArrayHighBound()` — соответственно возвращают нижнюю и верхнюю границы варианта-массива. Более подробная информация о функциях `VarArrayLock()` и `VarArrayUnlock()` приведена в следующем разделе этой главы.

Функция `VarArrayRef()` необходима для передачи вариантов-массивов серверу автоматизации OLE. Проблемы возникают в том случае, когда методу автоматизации в качестве параметра передается вариант, содержащий вариант-массив. Например:

```
Server.PassVariantArray(VA);
```

Между передачей варианта-массива и передачей варианта, содержащего вариант-массив, существуют принципиальные различия. Если серверу передать вариант-массив, а не ссылку на подобный объект, то при использовании приведенной выше записи сервер выдаст сообщение об ошибке. Функция `VarArrayRef()` предназначена для приведения передаваемого значения к виду и типу, ожидаемому сервером:

```
Server.PassVariantArray(VarArrayRef(VA));
```

Функция `VarIsArray()` представляет собой простую проверку, возвращающую значение `True`, если передаваемый ей параметр является массивом.

Инициализация большого массива с помощью VarArrayLock() и VarArrayUnlock()

Варианты-массивы широко используются в средствах автоматизации OLE, так как они представляют собой единственный способ передачи серверу автоматизации произвольных двоичных данных. Отметим, что указатели не являются допустимым типом данных в среде автоматизации OLE. Более подробная информация по этой теме приведена в главе 15, “Разработка приложений COM”. Но при неверном использовании варианты-массивы могут оказаться весьма неэффективным средством обмена данными. Рассмотрим небольшой пример:

```
V := VarArrayCreate([1, 10000], VarByte);
```

Здесь создается вариант-массив размером в 10 000 байт. Предположим, что существует другой, обычный, массив того же размера и необходимо скопировать его содержимое в созданный вариант-массив. Естественный путь такого копирования — цикл, подобный тому, который приведен ниже.

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  for i := 1 to 10000 do V[i] := A[i];
end;
```

Проблема здесь заключается в огромной и совершенно излишней работе программы по определению типа и совместимости каждого элемента, его инициализации и др. Суть в том, что присвоение значений элементам массива осуществляется во время выполнения программы. Для исключения всех этих ненужных проверок предназначена функция `VarArrayLock()` и процедура `VarArrayUnlock()`.

Функция `VarArrayLock()` блокирует массив в памяти так, чтобы его нельзя было переместить или изменить его размеры. Она возвращает указатель на данные массива. Процедура `VarArrayUnlock()` деблокирует массив, разрешая его перемещение в памяти и изменение его размеров. После блокировки массива для заполнения его данными можно воспользоваться более эффективными методами, например, такими как процедура `Move()`. Приведенный ниже пример позволяет инициализировать тот же массив значительно более эффективным методом.

```
begin
  V := VarArrayCreate([1, 10000], VarByte);
  P := VarArrayLock(V);
  try
    Move(A, P^, 10000);
  finally
    VarArrayUnlock(V);
  end;
end;
```

Вспомогательные функции

Существует еще несколько функций и процедур, предназначенных для работы с вариантами. Ниже приведены их определения в модуле `System`.

```
procedure VarClear(var V: Variant);
procedure VarCopy(var Dest: Variant; const Source: Variant);
procedure VarCast(var Dest: Variant; const Source: Variant;
                  VarType: Integer);
function VarType(const V: Variant): Integer;
function VarAsType(const V: Variant; VarType: Integer): Variant;
function VarIsEmpty(const V: Variant): Boolean;
function VarIsNull(const V: Variant): Boolean;
function VarToStr(const V: Variant): string;
function VarFromDateTime(DateTime: TDateTime): Variant;
function VarToDateTime(const V: Variant): TDateTime;
```

Процедура `VarClear()` очищает вариант и помещает в его поле `VType` значение `varEmpty`. Процедура `VarCopy()` копирует вариант `Source` в вариант `Dest`. Процедура `VarCast()` предназначена для преобразования варианта в некоторый тип и со-

хранения результата в другом варианте. Функция VarType() возвращает значение типа var*XXXX* для заданного варианта. Функция VarAsType() имеет то же самое назначение, что и процедура VarCast(). Функция VarIsEmpty() возвращает значение True, если поле VType варианта равно VarEmpty, а функция VarIsNull() возвращает это же значение, если данное поле равно VarNull. Функция VarToStr() конвертирует вариант в строку (пустую, если вариант пуст или нулевой). С помощью функции VarFromDateTime() создается вариант, содержащий заданное значение типа TDateTime. Функция VarToDateTime() возвращает значение типа TDateTime, содержащееся в указанном варианте.

Тип OleVariant

Тип данных OleVariant практически во всем идентичен рассмотренному выше типу Variant, за одним исключением — он допускает использование только тех типов данных, которые совместимы со средствами автоматизации OLE. В настоящее время единственное отличие проявляется при работе со строками (тип VarString предназначен для строк типа AnsiString). В то время как тип Variant работает с подобными строками, при присвоении строкового значения типа AnsiString переменной типа OleVariant происходит автоматическое конвертирование этой строки в тип OLE BSTR и сохранение ее в варианте как тип VarOleStr.

Тип Currency

Этот тип впервые был введен в Delphi 2 и представляет собой десятичное число с фиксированной точкой, имеющее 15 значащих цифр до десятичной точки и 4 после. Данный формат идеально подходит для финансовых вычислений, поскольку позволяет избежать ошибок округления, свойственных операциям над числами с плавающей точкой. Настоятельно рекомендуем заменить все участвующие в финансовых расчетах переменные типа Single, Real, Double и Extended на тип Currency.

Пользовательские типы данных

Таких типов, как целые, строки и вещественные числа, зачастую недостаточно для адекватного представления данных, с которыми приходится работать при решении реальных задач. Нередко приходится использовать и другие типы данных, более точно отражающие реальную действительность, моделируемую конкретной программой. В Object Pascal подобные пользовательские типы данных обычно принимают вид записей или объектов. Объявление этих типов осуществляется с помощью ключевого слова Type.

Массивы

Object Pascal позволяет создавать *массивы* (array) переменных любого типа (кроме файлов). Например, ниже объявлена переменная, представляющая собой массив из восьми целых чисел.

```
var
  A: Array[0..7] of Integer;
```

Этот оператор эквивалентен следующему объявлению в языке C:

```
int A[8];
```

Соответствующий оператор языка Visual Basic выглядит таким образом:

```
Dim A(8) as Integer
```

Массивы Object Pascal имеют одно существенное отличие от массивов языков C или Visual Basic (и многих других языков) — они *не обязаны* начинаться с определенного номера элемента. Например, можно определить массив из трех элементов, начинающийся с элемента под номером 28:

```
var
  A: Array[28..30] of Integer;
```

Поскольку в Object Pascal элементы массива не обязательно начинаются с нулевого или первого элемента, следует принимать необходимые меры при организации итераций, например в цикле `for`. Для этого компилятор предоставляет две встроенные функции — `High()` и `Low()`, возвращающие верхнюю и нижнюю границы заданного массива. Программа станет более устойчивой к ошибкам и возможным изменениям в описании массива, если в ней будут использоваться эти функции. Например:

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do  // Не используйте для цикла
    A[i] := i;                   // жесткий код!
end;
```

СОВЕТ

Всегда начинайте массивы символов с нулевого элемента — такие массивы могут быть переданы в качестве параметров функциям, использующим параметры с типом `PChar`. Это — особая дополнительная возможность, предоставляемая компилятором.

Чтобы определить многомерные массивы, пользуйтесь в описании запятой в качестве разграничителя списка размерностей:

```
var
  // Двумерный массив целых чисел:
  A: array[1..2, 1..2] of Integer;
```

Для доступа к элементам такого массива используют индексы, задаваемые через запятую:

```
I := A[1, 2];
```

Динамические массивы

Динамический массив (dynamic array) — это массив, память для которых выделяется динамически. Размерность динамического массива на момент компиляции не извест-

на. Для объявления такого массива применяется обычное описание, но без указания размерности:

```
var
  // Динамический массив строк:
  SA: array of string;
```

Перед использованием динамического массива следует с помощью процедуры `SetLength()` определить его размер, что приведет к выделению необходимой памяти:

```
begin
  // Выделить место в памяти для 33 элементов:
  SetLength(SA, 33);
```

После этого с элементами динамического массива можно работать как с элементами любого другого массива:

```
  SA[0] := 'Pooh likes hunny';
  OtherString := SA[0];
```

НА ЗАМЕТКУ

Динамические массивы всегда начинаются с нулевого элемента.
Динамические массивы — это тип данных с управляемым временем жизни, поэтому можно не заботиться о своевременном освобождении выделенной им памяти. Она будет освобождена автоматически, когда данная переменная покинет область видимости. Но это не означает, что при необходимости память нельзя освободить самостоятельно (например, если массив использовал слишком большое ее количество). Для этого достаточно просто присвоить массиву значение `nil`:

```
SA := nil;  // Освобождение памяти, выделенной массиву SA
```

Для работы с динамическими массивами применяется семантика ссылок (как и у типа `AnsiString`), а не значений (как у обычных массивов). Вот маленький тест: чему равен элемент `A1[0]` после выполнения следующего фрагмента кода?

```
var
  A1, A2: array of Integer;
begin
  SetLength(A1, 4);
  A2 := A1;
  A1[0] := 1;
  A2[0] := 26;
```

Правильный ответ — `26`. Поскольку присвоение `A2 := A1` не создает новый массив, а приводит к тому, что элементы массива `A1` и `A2` ссылаются на один и тот же участок памяти, то изменение элемента массива `A2` приводит и к изменению соответствующего элемента массива `A1` (впрочем, более точным будет утверждение, что это просто один и тот же элемент). Если же необходимо создать именно копию массива, то придется воспользоваться стандартной процедурой `Copy()`:

```
A2 := Copy(A1);
```

После выполнения данной строки программы `A2` и `A1` будут представлять собой два отдельных массива, первоначально содержащих одинаковые данные. Кроме того, совсем не обязательно копировать весь массив — можно выбрать определенный диа-

пазон его элементов, как показано в приведенном примере, где копируются два элемента, начиная с первого:

```
// Копируются только два элемента, начиная с первого:
A2 := Copy(A1, 1, 2);
```

Динамические массивы тоже могут быть многомерными. Чтобы определить такой массив, добавьте дополнительное описание `array of` для каждой дополнительной размерности:

```
var
    // Двумерный динамический массив целых чисел:
    IA: array of array of Integer;
```

При выделении памяти для многомерного динамического массива функции `SetLength()` следует передать дополнительный параметр:

```
begin
    // IA будет массивом целых чисел размерностью 5 x 5
    SetLength(IA, 5, 5);
```

Обращение к элементам многомерного динамического массива ничем не отличается от обращения к элементам обычного массива:

```
IA[0,3] := 28;
```

Записи

Пользовательские структуры, определяемые пользователем, в Object Pascal называются *записями* (record). Они эквивалентны типам данных `struct` в языке C или `Type` — в языке Visual Basic:

```
{ Pascal }
Type
  MyRec = record
       i: Integer;
       d: Double;
  end;

/* C */
typedef struct {
    int i;
    double d;
} MyRec;

'Visual Basic
Type MyRec
    i As Integer
    d As Double
End Type
```

При работе с записями доступ к их полям можно получить с помощью символа точки (точечный оператор):

```
var
  N: MyRec;
```

```
begin
  N.i := 23;
  N.d := 3.4;
end;
```

Object Pascal поддерживает также *вариантные записи* (variant records), которые обеспечивают хранение разнотипных данных в одной и той же области памяти. Не путайте эти записи с рассмотренным выше типом `Variant` — вариантные записи позволяют независимо получать доступ к каждому из перекрывающихся полей данных. Те, кто знаком с языком C, могут представить себе вариантные записи как аналог концепции `union` в структурах языка C. Приведенный ниже код показывает вариантную запись, в которой поля типа `Double`, `Integer` и `Char` занимают одну и ту же область памяти.

```
type
  TVariantRecord = record
    NullStrField: PChar;
    IntField: Integer;
    case Integer of
      0: (D: Double);
      1: (I: Integer);
      2: (C: char);
  end;
```

НА ЗАМЕТКУ

Правила Object Pascal запрещают размещать в вариантной части записи какие-либо типы данных с управляемым временем жизни.

Вот эквивалент приведенного кода в языке C++:

```
struct TUnionStruct
{
    char * StrField;
    int IntField;
    union u
    {
        double D;
        int i;
        char c;
    };
};
```

Множества

Множества (sets) — уникальный для языка Pascal тип данных, который не имеет аналогов в языках Visual Basic, C или C++ (хотя в Borland C++ Builder реализован шаблонный класс `Set`, эмулирующий поведение множеств в Pascal). Множества обеспечивают эффективный способ представления коллекций чисел, символов или других перечислимых значений. Новый тип множества можно определить с помощью ключевого слова `set of` с указанием перечислимого типа или диапазона в некотором множестве допустимых значений:

```
type
  TCharSet = set of char;   // Допустимы элементы: #0 - #255
  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;  // Любая комбинация членов TEnum
  TSubrangeSet = set of 1..10; // Допустимы элементы: 1 - 10
  TAlphaSet = set of 'A'..'z'; // Допустимы элементы: 'A' - 'z'
```

Заметим, что множество может содержать не более 256 элементов. Кроме того, в множествах после ключевого слова `set of` могут указываться только перечислимые типы данных. Таким образом, следующие объявления некорректны:

```
type
  TIntSet = set of Integer;  // Слишком много элементов
  TStrSet = set of string;   // Неперечислимый тип данных
```

Внутренне элементы множеств хранятся как отдельные биты, что делает их весьма эффективными в плане скорости обработки и использования памяти. Множества, насчитывающие менее чем 32 базовых элемента, могут храниться и обрабатываться в регистрах процессора, что способствует еще большей эффективности. Множества с 32 (или более) элементами (например множество символов `char` из 255 элементов) хранятся в памяти. Поэтому для достижения максимальной скорости обработки целесообразно определять множества не более чем из 32 базовых элементов.

Использование множеств

Для доступа к элементам множества используются квадратные скобки. Приведенный ниже пример демонстрирует объявление множества и присвоение значений его элементам.

```
type
  TCharSet = set of char;   // Допустимы элементы: #0 - #255
  TEnum = (Monday, Tuesday, Wednesday, Thursday,
           Friday, Saturday, Sunday);
  TEnumSet = set of TEnum;  // Любая комбинация членов TEnum

var
  CharSet: TCharSet;
  EnumSet: TEnumSet;
  SubrangeSet: set of 1..10; // Допустимы элементы: 1 - 10
  AlphaSet: set of 'A'..'z'; // Допустимы элементы: 'A' - 'z'

begin
  CharSet := ['A'..'J', 'a', 'm'];
  EnumSet := [Saturday, Sunday];
  SubrangeSet := [1, 2, 4..6];
  AlphaSet := [];  // Пусто, нет элементов
end;
```

Операторы для работы со множествами

Object Pascal предоставляет несколько операторов для работы со множествами (например для определения принадлежности к множеству, добавления или удаления элементов множеств).

Принадлежность к множеству

Оператор `in` используется для определения принадлежности данного элемента тому или иному множеству. В частности, следующий код используется для определения того, содержится ли символ "S" в множестве `CharSet`:

```
if 'S' in CharSet then
  // выполнить действия
```

В приведенном ниже примере осуществляется проверка отсутствия члена `Monday` в множестве `EnumSet`.

```
if not (Monday in EnumSet) then
  // выполнить действия
```

Добавление и удаление элементов

Операторы `+` и `-` или процедуры `Include()` и `Exclude()` могут быть использованы для добавления (union) или удаления (difference) элементов множеств:

```
Include(CharSet, 'a');           // добавить 'a' в множество
CharSet := CharSet + ['b'];      // добавить 'b' в множество
Exclude(CharSet, 'x');           // удалить 'z' из множества
CharSet := CharSet - ['y', 'z']; // удалить 'y' и 'z' из множества
```

СОВЕТ

Везде, где только это возможно, для добавления или удаления элементов из множества вместо операторов `+` и `-` используйте процедуры `Include()` и `Exclude()`. Каждая из этих процедур реализуется одной машинной командой, в то время как для реализации операторов `+` и `-` требуется по `13+6n` команд (здесь `n` — количество битов в множестве).

Пересечение множеств

Оператор `*` используется для вычисления пересечения двух множеств. В результате операции `Set1 * Set2` получается множество с элементами, содержащимися в обоих множествах одновременно. Приведенный ниже код является эффективным способом определения того, содержит ли множество несколько заданных элементов.

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then
  // выполнить действия
```

Объекты

Объекты в Object Pascal можно представить как записи, которые помимо данных содержат процедуры и функции. Но поскольку объектная модель Delphi детально обсуждается в следующем разделе этой главы, здесь остановимся только на синтаксисе объявления объектов Object Pascal. Объект объявляется таким образом:

```
Type
  TChildObject = class(TParentObject);
    SomeVar: Integer;
    procedure SomeProc;
  end;
```

Хотя объекты Delphi не совсем идентичны объектам языка C++, это объявление очень близко к описанию, используемому в языке C++:

```
class TChildObject : public TParentObject
{
    int SomeVar;
    void SomeProc();
};
```

Методы определяются подобно обычным процедурам и функциям (речь о которых пойдет далее в этой главе). Единственное отличие — это добавление имени объекта и точки перед именем метода:

```
procedure TChildObject.SomeProc;
begin
  { здесь располагается код процедуры }
end;
```

Символ точки (.) в Object Pascal по назначению похож на оператор "." в языке Visual Basic или на оператор "::" в языке C++. Следует отметить, что, хотя все три языка позволяют использовать классы, только Object Pascal и C++ дают возможность создавать новые классы, полностью соответствующие парадигме объектно-ориентированного программирования.

НА ЗАМЕТКУ

Размещение объектов Object Pascal в памяти отличается от размещения объектов C++, поэтому невозможно использовать объекты C++ непосредственно в Delphi, и наоборот.
Исключение составляют классы Borland C++ Builder, внедренные в Object Pascal и использующие собственную директиву `__declspec(delphiclass)`. Они совместимы с объектами Delphi (но не совместимы с обычными объектами C++).

Указатели

Указатель (pointer) представляет собой переменную, содержащую местоположение (адрес) участка памяти. В этой главе уже встречался указатель, когда рассматривался тип `PChar`. Тип указателя в Object Pascal называется `Pointer`, или нетипизированный указатель, поскольку он содержит адрес определенного места в памяти, а компилятору ничего не известно о данных, располагающихся по этому адресу. Но применение таких "указателей вообще" противоречит концепции строгого контроля типов, поэтому в основном придется работать с типизированными указателями, т.е. с указателями на данные конкретного типа.

НА ЗАМЕТКУ

Указатели — тема достаточно сложная для новичка, а приложения Delphi можно создавать и не будучи досконально знакомым с ней. Но, по мере ее освоения, указатели

могут стать одним из самых мощных инструментов программирования, доступных в Delphi.

Типизированные указатели объявляются в разделе Type программы с использованием символа ^ (оператора указателя) или ключевого слова Pointer. Типизированные указатели позволяют компилятору отслеживать, с каким именно типом данных происходят действия и не выполняются ли при этом некорректные операции. Вот примеры объявлений типизированных указателей:

```
Type
  PInt = ^Integer;          // PInt - указатель на Integer
  Foo = record              // Тип  - запись
    GobbledyGook: string;
    Snarf: Real;
  end;
  PFoo = ^Foo;              // PFoo - указатель на объект типа foo
var
  P: Pointer;               // Нетипизированный указатель
  P2: PFoo;                 // Экземпляр PFoo
```

НА ЗАМЕТКУ

Программисты на языке C++ могут заметить схожесть оператора ^ Object Pascal и оператора * языка C++. Тип Pointer в Object Pascal соответствует типу void * языка C.

Запомните, что переменная типа Pointer всегда содержит только адрес области памяти. О выделении памяти для структуры, на которую будет указывать такой указатель, программист должен позаботиться сам (для этого можно воспользоваться одной из функций, перечисленных в табл. 2.6).

НА ЗАМЕТКУ

Когда указатель не указывает ни на что (его значение равно 0), то о таком указателе говорят, что его значение равно Nil, а сам указатель называют *нулевым*, или *пустым* указателем.

Если необходимо получить доступ к данным, на которые указатель указывает, то можно воспользоваться оператором ^, следующим за именем этой переменной. Такой метод называется *разрешением указателя* (dereferencing), а также *косвенным доступом*, *взятием значения*, *разыменованием* и *ссылкой*. Ниже приведен пример работы с указателями.

```
Program PtrTest;

Type
  MyRec = record
    I: Integer;
    S: string;
    R: Real;
  end;
  PMyRec = ^MyRec;
```

```
var
  Rec: PMyRec;
begin
  New(Rec);       // Выделить память для новой записи Rec
  Rec^.I := 10;   // Поместить в нее данные. Обратите внимание
                  // на оператор ^
  Rec^.S := 'And now for something completely different.';
            { Теперь помещаем в нее данные другого типа }
  Rec^.R := 6.384;
  { Теперь запись Rec заполнена }
  Dispose(Rec);  // Не забывайте освобождать память!
end.
```

Когда использовать функцию `New()`

Функция `New()` используется при выделении памяти для указателя на структуру данных известного размера. Поскольку компилятору известен размер структуры, для которой необходимо выделить память, то при выполнении функции `New()` будет распределено достаточное количество байтов, причем такой способ выделения более корректен и безопасен, чем вызов функции `GetMem()` или `AllocMem()`. В то же время, никогда не используйте функцию `New()` для выделения памяти для типов `Pointer` или `PChar`, так как в этом случае компилятору не известно, какое количество памяти должно быть выделено. И не забывайте использовать функцию `Dispose()` для освобождения памяти, выделенной с помощью функции `New()`.

Для выделения памяти структурам, размер которых на этапе компиляции еще не известен, используются функции `GetMem()` и `AllocMem()`. Например, компилятор не может определить заранее, сколько памяти потребуется выделить для структур, задаваемых переменными типа `PChar` или `Pointer`, что связано с самой природой этого типа данных. Самое важное — не пытаться манипулировать количеством памяти, большим, чем было выделено реально, поскольку наиболее вероятным результатом таких действий будет ошибка доступа к памяти (access violation). Для освобождения памяти, выделенной с помощью вышеупомянутых функций, используйте функцию `FreeMem()`. Кстати, для выделения памяти лучше пользоваться функцией `AllocMem()`, так как она всегда инициализирует выделенную память нулевыми значениями.

Один из аспектов работы с указателями в Object Pascal, который существенно отличается от работы с ними в языке C, — это их строжайшая типизация. Так, в приведенном ниже примере переменные `a` и `b` не совместимы по типу.

```
var
  a: ^Integer;
  b: ^Integer;
```

В то же время в эквивалентном описании на языке C эти переменные вполне совместимы:

```
int *a;
int *b
```

Object Pascal создает уникальный тип для каждого объявления *указателя на тип* (pointer-to-type), поэтому для совместимости по типу следует объявлять не только переменные, но и их тип:

```
type
  PtrInteger = ^Integer;  // Создать именованный тип

var
  a, b: PtrInteger;       // Теперь a и b совместимы по типу
```

Псевдоним типа

Object Pascal позволяет присвоить новое имя уже существующему типу данных, т.е. создать его *псевдоним* (alias). Например, если обычному типу `Integer` необходимо присвоить новое имя `MyReallyNiftyInteger`, то можно использовать следующий код:

```
type
  MyReallyNiftyInteger = Integer;
```

Новый тип аналогичен оригиналу. Это означает, что везде, где использовался тип `Integer`, есть возможность применять тип `MyReallyNiftyInteger`.

Но можно создать и *строго типизированный* (strongly typed) псевдоним (т.е. псевдоним, не совместимый с оригиналом), для чего используется дополнительное ключевое слово `type`.

```
type
  MyOtherNeatInteger = type Integer;
```

При этом новый тип будет преобразовываться в оригинальный при таких операциях, как присвоение, но как параметр он будет не совместим с оригиналом. Следующий код синтаксически корректен:

```
var
  MONI: MyOtherNeatInteger;
  I: Integer;
begin
  I := 1;
  MONI := I;
```

Но код, приведенный ниже, не может быть откомпилирован из-за ошибки совместимости типов:

```
procedure Goon(var Value: Integer);
begin
  // Некоторый код
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);  // Ошибка: тип переменной M не совместим с Integer
```

Помимо усиленного контроля за совместимостью типов данных, для строго типизированных псевдонимов компилятор также генерирует информацию о типах времени выполнения программы. Это позволяет создавать уникальные редакторы свойств для простых типов. Более подробная информация по данной теме приведена в главе 12, "Создание расширенного компонента VCL".

Приведение и преобразование типов

Приведение типов (typecasting) — эта технология, способная заставить компилятор рассматривать переменную одного типа как переменную другого типа. Pascal является строго типизированным языком, который весьма требователен в отношении соответствия типов формальных параметров функции и реальных параметров, передаваемых ей при вызове. Поэтому в некоторых случаях переменную одного типа необходимо привести к некоторому другому типу, чтобы требования компилятора были удовлетворены. Предположим, необходимо присвоить символьное значение переменной типа `byte`:

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := c;   // здесь компилятор выдаст сообщение об ошибке
end.
```

В приведенном выше примере требуется привести переменную `c` к типу `byte`. Фактически выполнение приведения типа явно указывает компилятору, что программист точно знает, что он делает, требуя преобразовать один тип данных в другой:

```
var
  c: char;
  b: byte;
begin
  c := 's';
  b := byte(c);   // Теперь компилятор доволен
end.
```

НА ЗАМЕТКУ

Использовать приведение типов можно только при совпадении размеров данных обоих типов. Привести тип `Double` к типу `Integer` невозможно, поэтому для подобного преобразования придется воспользоваться функцией `Trunc()` или `Round()`. Но, чтобы преобразовать целое число в вещественное, можно пользоваться обычным оператором присвоения:

```
FloatVar := IntVar
```

Object Pascal предоставляет широчайшие возможности приведения типов объектов с помощью оператора `as`. Более подробная информация по данной теме приведена далее в этой главе.

Строковые ресурсы

В Delphi 3 появилась возможность внесения строковых ресурсов непосредственно в исходный код с помощью ключевого слова `resourcestring`. *Строковые ресурсы* (string resources) представляют собой литеральные строки (обычно это — сообщения программы пользователю), которые физически расположены в ресурсных файлах, присоединенных к приложению, или в отдельной библиотеке, а не внедрены в исходный код программы. В частности, подобное отделение строк от исходного кода упрощает перевод приложения на другой язык (локализацию): для этого достаточно просто присоединить к приложению строковые ресурсы на необходимом языке, без перекомпиляции самого приложения. Строковые ресурсы описываются в виде пар значений *идентификатор = строковый литерал*, как показано ниже.

```
resourcestring
  ResString1 = 'Resource string 1';
  ResString2 = 'Resource string 2';
  ResString3 = 'Resource string 3';
```

В программе строковые ресурсы используются так же, как и обычные строковые константы:

```
resourcestring
  ResString1 = 'hello';
  ResString2 = 'world';

var
  String1: string;

begin
  String1 := ResString1 + ' ' + ResString2;
  .
  .
  .
end;
```

Условные операторы

В этом разделе операторы `if` и `case` языка Object Pascal сравниваются с соответствующими конструкциями языков C и Visual Basic. Предполагается, что читатель уже знаком с подобными операторами, а потому не будем тратить время на подробное объяснение их назначения и функционирования.

Оператор if

Оператор `if` (условного перехода) позволяет проверить, выполняется ли некоторое условие, и, в зависимости от результатов этой проверки, выполнить тот или иной блок кода. В качестве примера ниже приведены простейший фрагмент кода с использованием оператора `if` и его аналоги на языках C и Visual Basic:

```
{ Pascal }
if x = 4 then y := x;

/* C */
if (x == 4) y = x;

'Visual Basic
If x = 4 Then y = x
```

НА ЗАМЕТКУ

При наличии в операторе `if` нескольких условий поместите каждое из них в скобки. Это сделает код, как минимум, более ясным и удобочитаемым. Например:

```
if (x = 7) and (y = 8) then
```

Избегайте записи, подобной следующей (поскольку компилятор поймет ее неправильно):

```
if x = 7 and y = 8 then
```

В языке Pascal ключевые слова `begin` и `end` (называемые также логическими скобками) используются аналогично фигурным скобкам ({ и }) в языках C и C++ для определения блока кода. Ниже приведен пример использования подобной конструкции в операторе `if`:

```
if x = 6 then begin
  DoSomething;
  DoSomethingElse;
  DoAnotherThing;
end;
```

Конструкция `if..else` позволяет объединить проверку нескольких условий с исполнением нескольких блоков кода:

```
if x =100 then
  SomeFunction
else if x = 200 then
  SomeOtherFunction
else begin
  SomethingElse;
  Entirely;
end;
```

Оператор case

Оператор `case` в языке Pascal подобен операторам `switch` в языках C или C++. Он позволяет осуществить выбор одного варианта из нескольких возможных без использования сложных конструкций, состоящих из нескольких вложенных операторов `if..else if..else`. Вот пример оператора `case`:

```
case SomeIntegerVariable of
  101: DoSomething;
  202: begin
```

```
        DoSomething;
        DoSomethingElse;
      end;
  303: DoAnotherThing;
  else DoTheDefault;
end;
```

НА ЗАМЕТКУ

В операторе `case` проверяемая переменная обязана принадлежать к перечислимому типу. Использование других типов данных, в частности типа `string`, не допускается.

На языке C этот пример выглядит так:

```
switch (SomeIntegerVariable)
{
    case 101: DoSomeThing();
        break;
    case 202: DoSomething();
              DoSomethingElse();
        break;
    case 303: DoAnotherThing();
        break;
    default: DoTheDefault();
}
```

Циклы

Цикл (loop) представляет собой конструкцию, которая позволяет выполнять повторяющиеся действия. Циклы языка Pascal во многом подобны циклам других языков, потому не будем останавливаться на деталях их построения и работы. В этом разделе описываются все конструкции циклов, существующие в языке Pascal.

Цикл for

Цикл `for` идеален для организации повторения некоторых действий заранее определенное количество раз. Ниже приведен код цикла `for` (хоть и не слишком полезного), в котором к значению переменной `X` десять раз прибавляется значение счетчика цикла `I` (далее следуют аналоги этого цикла на языках C и Visual Basic).

```
var
  I, X: Integer;
begin
  X := 0;
  for I := 1 to 10 do inc(X, I);
end.
```

На языке C данный пример выглядит так:

```
void main(void) {
    int x, i;
    x = 0;
```

```
    for(i=1; i<=10; i++)
        x += i;
}
```

На языке Visual Basic этот пример имеет следующий вид:

```
X = 0
For I = 1 to 10
    X = X + I
NextI
```

СОВЕТ

Начиная с версии Delphi 2 присвоение значений счетчику цикла в теле самого цикла недопустимо. Это связано с оптимизацией циклов в 32-разрядном компиляторе.

Цикл while

Цикл `while` позволяет повторять выполнение блока кода до тех пор, пока заданное условие остается истинным. Условие проверяется *перед* выполнением блока кода, поэтому классическим примером использования данной конструкции является выполнение некоторых действий с файлом (например чтение его данных), продолжающееся до тех пор, пока не будет достигнут его конец. Ниже приведен пример подобного цикла, в котором при каждом выполнении его тела одна строка файла считывается и выводится на экран:

```
Program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile;  // Текстовый файл
  s: string;
begin
  AssignFile(f, 'foo.txt');
  Reset(f);
  while not EOF(f) do begin
    readln(f, S);
    writeln(S);
  end;
  CloseFile(f);
end.
```

Цикл `while` языка Pascal по своим свойствам не отличается от цикла `while` языка C и цикла `Do..While` языка Visual Basic.

Цикл repeat..until

Цикл `repeat..until` выполняет ту же задачу, что и цикл `while`, но несколько иначе. С его помощью блок кода выполняется до тех пор, пока заданное условие не станет истинным; причем вначале выполняется блок кода, а уже *затем* проверяется истинность условия. Этот цикл аналогичен циклу `do..while` языка C.

Вот пример такого цикла. Счетчик X будет увеличиваться до тех пор, пока его значение не превысит 100:

```
var
  x: Integer;
begin
  X := 1;
  repeat
    inc(x);
  until x >100;
end.
```

Процедура Break()

Вызов процедуры Break() внутри цикла while, for или repeat..until организует немедленный выход программы из цикла. Подобный метод полезен в тех случаях, когда во время выполнения цикла возникло некоторое обстоятельство, требующее немедленного завершения его выполнения. Эта процедура аналогична оператору break в языке C и оператору exit в языке Visual Basic. В следующем далее примере выполнение цикла прекращается после пяти итераций:

```
var
  i: Integer;
begin
  for i := 1 to 1000000 do begin
    MessageBeep(0);              // Подача звукового сигнала
    if i = 5 then Break;
  end;
end;
```

Процедура Continue()

Вызов процедуры Continue() прерывает выполнение тела цикла и осуществляет переход к началу следующей итерации, пропуская при этом все оставшиеся операторы блока. В частности, в приведенном ниже примере при первой итерации второй оператор вывода строки выполнен не будет:

```
var
  i: Integer;
begin
  for i := 1 to 3 do begin
    writeln(i, '. Before continue'); // До continue
    if i = 1 then Continue;
    writeln(i, '. After continue');  // После continue
  end;
end;
```

Процедуры и функции

Всем программистам хорошо знакомы основные положения использования функций и процедур. Процедура представляет собой отдельную часть программы, которая решает определенную задачу, а затем возвращает управление в точку вызова. Функция работает практически точно так же, за одним исключением — она возвращает какое-то значение, которое может быть использовано в вызвавшем функцию коде.

Те, кто знаком с языком программирования C или C++, могут рассматривать процедуру языка Pascal как функцию языка C, возвращающую значение `void`. Функции языка Pascal аналогичны остальным функциям языка C.

В листинге 2.1 приведена небольшая программа, в которой используются процедура и функция.

Листинг 2.1. Пример использования процедур и функций

```
Program FuncProc;

{$APPTYPE CONSOLE}

procedure BiggerThanTen(i: Integer);
{ Выводит на экран сообщение, если I больше 10 }
begin
  if I > 10 then
    writeln('Funky.');
end;

function IsPositive(I: Integer): Boolean;
{ Возвращает True, если I больше или равно 0 }
begin
  if I < 0 then
    Result := False
  else
    Result := True;
end;

var
  Num: Integer;
begin
  Num := 23;
  BiggerThanTen(Num);
  if IsPositive(Num) then
    writeln(Num, 'Is positive.')
  else
    writeln(Num, 'Is negative.');
end.
```

НА ЗАМЕТКУ

Локальная переменная `Result` (результат) функции `IsPositive()` имеет специальное назначение. В каждой функции языка Object Pascal существует локальная пере-

менная с этим именем, предназначенная для размещения возвращаемого значения. Обратите внимание, в отличие от языков C и C++, выполнение функции не прекращается при присвоении значения переменной `Result`.
Вернуть значение из функции можно присвоив его имени функции внутри кода самой функции. Это стандартный синтаксис языка Pascal, сохранившийся от его предыдущих версий. При использовании в теле функции ее имени будьте внимательны, поскольку существует принципиальное различие между использованием имени функции слева от оператора присвоения и использованием его в любом другом месте кода. Имя функции слева от знака равенства означает присвоение функции возвращаемого значения, а имя функции в любом другом месте внутри кода функции привет к ее рекурсивному[1] вызову!
Имейте в виду, что использование переменной `Result` недопустимо при сброшенном флажке Extended Syntax (Расширенный синтаксис), расположенном во вкладке Compiler (Компилятор) диалогового окна Project Options, или при указании директивы компилятора `{$X-}`.

Передача параметров

Pascal позволяет передавать параметры в функции и процедуры либо *по значению* (by value), либо *по ссылке* (by reference). Передаваемый параметр может иметь любой встроенный или пользовательский тип либо являться открытым массивом (они рассматриваются в этой главе далее). Параметр также может быть константой, если его значение в процедуре или функции не изменяется.

Передача параметров по значению

Этот режим передачи параметров принят по умолчанию. Если параметр передается по значению, создается локальная копия данной переменной, которая и предоставляется для обработки в процедуру или функцию. Рассмотрим следующий пример:

```
procedure Foo(s: string);
```

При вызове указанной процедуры будет создана копия передаваемой ей в качестве параметра строки `s`, с которой и будет работать процедура `Foo()`. При этом все внесенные в переданную строку изменения никак не отразятся на исходной строке `s`.

Передача параметров по ссылке

Pascal позволяет также передавать параметры в функции или процедуры по ссылке — такие параметры называются *переменными параметрами* (variable parameter), или `var`-*параметрами*. Передача параметра по ссылке означает, что функция или процедура сможет изменять значения полученных параметров. Для передачи параметров по ссылке используется ключевое слово `var`, помещаемое в список параметров процедуры или функции:

```
procedure ChangeMe(var x: longint);
begin
  x := 2;
  { теперь в процедуре можно изменить исходный x }
end;
```

[1] *Рекурсия – вызов себя самого* – Прим. ред.

Вместо создания копии переменной x ключевое слово var требует передачи адреса самой переменной x, что позволяет процедуре непосредственно изменять ее значение.

Для передачи параметров по ссылке в языке C++ используется оператор &.

Параметры-константы

Если нет необходимости изменять передаваемые функции или процедуре данные, то можно описать параметр как константу. Ключевое слово const не только защищает параметр от изменения, но и позволяет компилятору создать более оптимальный код для переданных строк и записей. Вот пример объявления параметра-константы:

```
procedure Goon(const s: string);
```

Использование открытых массивов

Открытый массив параметров позволяет передавать в функцию или процедуру различное количество параметров. В качестве параметра можно передать либо открытый массив элементов одинакового типа, либо массив констант различного типа. В приведенном ниже примере объявляется функция, которой в качестве параметра должен передаваться открытый массив целых чисел:

```
function AddEmUp(A: array of Integer): Integer;
```

В открытом массиве можно передавать переменные, константы или выражения из констант. Далее продемонстрирован вызов функции AddEmUp() с передачей ей нескольких различных элементов:

```
Var
  i, Rez: Integer;
const
  j = 23;
begin
  i := 8;
  Rez := AddEmUp([i, 50, j, 89]);
```

Чтобы работать с открытым массивом внутри функции или процедуры, необходима информация о нем. Получить информацию об открытом массиве можно с помощью функций High(), Low() и SizeOf(). Для иллюстрации их использования ниже приведен код функции AddEmUp(), которая возвращает сумму всех переданных ей элементов массива A.

```
function AddEmUp(A: array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := Low(A) to High(A) do inc(Result, A[i]);
end;
```

Object Pascal поддерживает также тип array of const, который позволяет передавать в одном массиве данные различных типов. Для объявления функций и процедур, использующих в качестве параметра такой массив, применяется следующий синтаксис:

```
procedure WhatHaveIGot(A: array of const);
```

Объявленная выше функция допускает даже такой синтаксис вызова:

```
WhatHaveIGot(['Tabasco', 90, 5.6, @WhatHaveIGot,
              3.14159, True, 's']);
```

Компилятор автоматически конвертирует переданный функции или процедуре массив констант в тип `TVarRec`, объявленный в модуле `System` следующим образом:

```
type
  PVarRec = ^TVarRec;
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
      vtWideString: (VWideString: Pointer);
      vtInt64:      (VInt64: PInt64);
  end;
```

Поле `VType` определяет тип содержащихся в этом экземпляре записи данных `TVarRec` и может принимать одно из приведенных ниже значений.

```
const
  { Значения TVarRec.VType }
  vtInteger    = 0;
  vtBoolean    = 1;
  vtChar       = 2;
  vtExtended   = 3;
  vtString     = 4;
  vtPointer    = 5;
  vtPChar      = 6;
  vtObject     = 7;
  vtClass      = 8;
  vtWideChar   = 9;
  vtPWideChar  = 10;
  vtAnsiString = 11;
  vtCurrency   = 12;
  vtVariant    = 13;
  vtInterface  = 14;
  vtWideString = 15;
  vtInt64      = 16;
```

Поскольку массив констант (тип `array of const`) способен передавать данные различных типов, это может вызвать определенные затруднения при создании обрабатывающей полученные параметры функции или процедуры. В качестве примера работы с таким массивом рассмотрим реализацию процедуры `WhatHaveIGot()`, которая просматривает элементы полученного массива параметров и выводит их тип на экран.

```
procedure WhatHaveIGot(A: array of const);
var
  i: Integer;
  TypeStr: string;
begin
  for i := Low(A) to High(A) do begin
    case A[i].VType of
      vtInteger    : TypeStr := 'Integer';
      vtBoolean    : TypeStr := 'Boolean';
      vtChar       : TypeStr := 'Char';
      vtExtended   : TypeStr := 'Extended';
      vtString     : TypeStr := 'String';
      vtPointer    : TypeStr := 'Pointer';
      vtPChar      : TypeStr := 'PChar';
      vtObject     : TypeStr := 'Object';
      vtClass      : TypeStr := 'Class';
      vtWideChar   : TypeStr := 'WideChar';
      vtPWideChar  : TypeStr := 'PWideChar';
      vtAnsiString : TypeStr := 'AnsiString';
      vtCurrency   : TypeStr := 'Currency';
      vtVariant    : TypeStr := 'Variant';
      vtInterface  : TypeStr := 'Interface';
      vtWideString : TypeStr := 'WideString';
      vtInt64      : TypeStr := 'Int64';
    end;
    ShowMessage(Format('Array item %d is a %s', [i, TypeStr]));
  end;
end;
```

Область видимости

Область видимости (scope) — это определенный участок программы, на протяжении которого данная функция или переменная известна компилятору. Глобальные константы видимы в любой точке программы, в то время как локальные переменные видны только в той процедуре, где они были объявлены. Рассмотрим листинг 2.2.

Листинг 2.2. Иллюстрация понятия области видимости

```
program Foo;

{$APPTYPE CONSOLE}

const
  SomeConstant = 100;

var
  SomeGlobal: Integer;
  R: Real;

procedure SomeProc(var R: Real);
var
  LocalReal: Real;
begin
  LocalReal := 10.0;
  R := R - LocalReal;
end;

begin
  SomeGlobal := SomeConstant;
  R := 4.593;
  SomeProc(R);
end.
```

Здесь переменные `SomeConstant`, `SomeGlobal` и `R` имеют глобальную область видимости, поэтому их значения известны компилятору в любой точке программы. Процедура `SomeProc()` имеет две собственные локальные переменные: `R` и `LocalReal`. Любая попытка обращения к переменной `LocalReal` вне процедуры `SomeProc()` приведет к сообщению об ошибке. Обращение к переменной `R` внутри функции `SomeProc()` вернет значение ее локального экземпляра, тогда как обращение к переменной `R` вне этой функции вернет значение глобальной переменной с этим же именем.

Модули

Модуль (unit) представляет собой отдельную единицу исходного кода, вся совокупность которых составляет программу на языке Object Pascal. В модуле обычно размещается определенная группа функций и процедур, которые могут быть вызваны из основной программы. Для того чтобы считаться модулем, файл с текстом исходного кода должен состоять как минимум из трех частей, приведенных ниже.

- Оператор `unit`. Каждый модуль должен начинаться со строки, объявляющей, что данный блок текста является модулем, и задающей имя этого модуля. Имя модуля всегда должно соответствовать имени его файла. Например, если файл называется `FooBar`, то его первая строка должна выглядеть так:

  ```
  Unit FooBar;
  ```

- Раздел интерфейса (`interface`). После оператора `unit` следующей функциональной строкой должен быть оператор `interface`. Все, что находится между этой строкой и оператором `implementation` данного модуля, доступно извне

и может использоваться другими модулями и программами. Именно в таком разделе описываются типы данных, константы, переменные, процедуры и функции, которые должны быть доступны программе или другим модулям. В этом разделе допустимы *только объявления* (но не реализация!) функций и процедур. Сам оператор `interface` занимает отдельную строку и содержит единственное ключевое слово:

```
interface
```

- Раздел реализации (`implementation`) следует за разделом интерфейса и начинается оператором `implementation`. Хотя основное содержимое этой части модуля составляют тела описанных ранее процедур и функций, здесь также можно определять типы данных, константы и переменные, которые будут доступны только в пределах данного модуля. Сам оператор `implementation` занимает отдельную строку и содержит единственное ключевое слово:

```
implementation
```

В состав модуля могут входить еще два необязательных раздела.

- Раздел инициализации (`initialization`). Располагается после раздела реализации и содержит код, необходимый для инициализации данного модуля. Этот код будет выполнен только один раз перед началом выполнения основной программы.
- Раздел завершения (`finalization`). Располагается между разделом инициализации и оператором `end.` модуля. Он содержит код, необходимый для завершения работы модуля. Этот код будет выполнен только один раз — при завершении работы программы. Раздел завершения впервые появился в Delphi 2.0. В Delphi 1.0 для выполнения действий, завершающих работу модуля, следовало создать особую процедуру завершения и зарегистрировать ее с помощью функции `AddExitProc()`.

НА ЗАМЕТКУ

При наличии разделов `initialization/finalization` сразу в нескольких модулях выполнение их кода происходит в том порядке, в котором к этим модулям обращается компилятор (первый модуль в разделе `uses` основной программы, затем первый модуль в разделе `uses` этого модуля и т.д.). Но не следует создавать код инициализации/завершения модулей, работа которых полагается на некоторый жесткий порядок их выполнения, — это плохой стиль программирования. Малейшее изменение в любом разделе `uses` может привести к ошибке, которую будет чрезвычайно трудно обнаружить!

Раздел uses

В разделе `uses` перечисляются модули, которые будут включены в данную программу (или в модуль). Например, если в программе `FooProg` используются функции или типы данных из двух модулей — `UnitA` и `UnitB`, — то раздел `uses` этой программы должен выглядеть следующим образом:

```
Program FooProg;
```

```
uses
  UnitA, UnitB;
```

Модули могут содержать два раздела uses: один — в разделе interface, а другой — в разделе implementation.

Вот пример простейшего модуля:

```
Unit FooBar;

interface

uses BarFoo;

  { Объявления открытых элементов (public) }

implementation

uses BarFly;

  { Объявления закрытых элементов (private) }

initialization
  { Код инициализации }

finalization
  { Код завершения }
end.
```

Взаимные ссылки

Иногда может возникнуть ситуация, когда модуль UnitA использует модуль UnitB, а тот в свою очередь — модуль UnitA. Обычно наличие подобных *взаимных ссылок* (circular unit reference) свидетельствует о просчетах на этапе проектирования структуры приложения. Этого следует избегать. Устранить данную проблему можно создав третий модуль, в состав которого войдут все необходимые функции и процедуры обоих модулей. Если же по каким-либо соображениям это невозможно (например, модули достаточно велики), переместите один из разделов uses в раздел implementation модуля, а другой оставьте в разделе interface. Зачастую это решает проблему.

Пакеты

Пакеты (packages) Delphi позволяют размещать части приложения в различных модулях, которые могут затем совместно использоваться несколькими приложениями. Те, кто имеют опыт работы с Delphi 1 или 2, оценят преимущества новых пакетов по достоинству, поскольку ими можно воспользоваться без каких-либо изменений исходного кода.

Пакеты можно понимать как коллекции модулей, сохраняемых в отдельных файлах *библиотек пакетов Borland* (BPL — Borland Package Library), подобных файлам DLL. Приложение можно связать с пакетированными модулями непосредственно во время выполнения, а не во время его компиляции и компоновки. Естественно, при этом

размер исполняемого файла уменьшается, так как часть кода и данных располагается в файле BPL. Delphi позволяет создавать пакеты четырех типов.

- *Пакет времени выполнения* (runtime package). Этот тип пакетов содержит модули, используемые программой во время ее выполнения. Приложение, скомпилированное для взаимодействия с данным пакетом, не будет работать без него. Примером пакета такого типа может служить пакет Delphi `VCL60.BPL`.
- *Пакеты разработки* (design package). Пакет этого типа содержит элементы, необходимые для разработки приложения (например компонентов редактора свойств и компонентов программ-экспертов). Эти пакеты можно включить в библиотеку компонентов Delphi, выбрав в меню Component (Компонент) пункт Install Package (Установка пакетов). В качестве примера можно привести пакет Delphi `DCL*.BPL`. Более подробная информация о пакетах этого типа приведена в главе 11, "Разработка компонентов VCL".
- *Пакеты разработки и времени выполнения* (runtime and design package). Такие пакеты применяются как пакеты одновременно обоих указанных выше типов. Использование подобных пакетов упрощает создание и распространение приложения. Но этот тип пакетов менее эффективен, поскольку помимо необходимой для работы исполняемой части они содержат также поддержку среды разработки.
- *Пакеты, не относящиеся ни к одному из перечисленных типов.* Такие пакеты могут применяться только другими пакетами и не используются непосредственно приложением или средой разработки.

Использование пакетов Delphi

Создание приложений, работающих с пакетами Delphi, представляет собой несложную задачу. Для этого достаточно установить флажок Build with Runtime Packages во вкладке Packages диалогового окно Project Options. Данный параметр позволяет компилятору создавать приложения, динамически связанные с пакетами времени выполнения, вместо статической компоновки всех модулей в общий файл `.EXE` или `.DLL`. В результате размер файла приложения станет намного меньше. Однако не забывайте, что в этом случае при установке приложения придется установить и необходимые пакеты.

Синтаксис описания пакетов

Обычно пакеты создаются с помощью редактора Package Editor, для вызова которого необходимо выбрать в меню File пункты New и Package. Этот редактор создает исходный файл пакета DPK (Delphi Package Source), который затем компилируется в пакет. Синтаксис, используемый при создании файла DPK, очень прост и имеет следующий формат:

```
package PackageName

requires Package1, Package2, ...;

contains
```

```
  Unit1 in 'Unit1.pas',
  Unit2 in 'Unit2.pas',
  ...;

end.
```

Пакеты, перечисленные в разделе `requires`, необходимы для работы текущего пакета. Обычно эти пакеты содержат модули, используемые теми модулями, которые перечислены в разделе `contains` (последние будут скомпилированы в данный пакет). При этом следует помнить, что модули, объявленные в разделе `contains` этого пакета, не должны упоминаться в разделах `contains` пакетов, перечисленных в разделе `requires` данного модуля. Заметим также, что в пакет будет неявно включен любой модуль, используемый другим модулем, указанным в разделе `contains` пакета (если только он уже не содержится в одном из пакетов, указанных в разделе `requires`).

Объектно-ориентированное программирование

Объектно-ориентированному программированию (OOP — Object-Oriented Programming) посвящены многие тома, в дискуссиях о нем сломано немало копий, и уже начинает казаться, что это не методология программирования, а религия. Не являясь ортодоксами объектно-ориентированного программирования, авторы книги не будут пытаться обратить вас в ту или иную веру. Поэтому мы просто излагаем здесь основные принципы OOP, которые положены в основу языка Object Pascal.

Объектно-ориентированным называется такой метод программирования, при котором в качестве основных элементов программ используются отдельные объекты, содержащие и данные, и код для их обработки. Облегчение создания и сопровождения программ не являлось основной целью разработки методологии объектно-ориентированного программирования — это, скорее, побочный эффект ее применения на практике. Кроме того, хранение данных и кода в одном объекте позволяет минимизировать воздействие одного объекта на другой, а следовательно, и облегчить поиск и исправление ошибок в программе. Традиционно объектно-ориентированные языки реализуют, по меньшей мере, три основные концепции.

- *Инкапсуляция* (encapsulation). Работа с данными и детали ее реализации скрыты от внешнего пользователя объекта. Достоинства инкапсуляции заключаются в модульности и изоляции кода объекта от другого кода программы.
- *Наследование* (inheritance). Возможность создания новых объектов, которые обладают свойствами и поведением родительских объектов. Такая концепция позволяет создавать иерархии объектов (например VCL), включающие наборы объектов, порожденных от одного общего предка и обладающих все большей специализацией и функциональностью по сравнению со своими предшественниками.

  Достоинства наследования состоят, в первую очередь, в совместном использовании многими объектами общего кода. На рис. 2.4 представлен пример наследования: один корневой объект `fruit` (фрукт) является предком для всех остальных плодов,

включая `melon` (дыня), которая, в свою очередь, является предком таких объектов, как `watermelon` (арбуз) и `honeydew` (зимняя дыня).

- *Полиморфизм* (polymorphism). Слово полиморфизм означает "много форм". В данном случае под этим подразумевается, что вызов метода объекта для переменной приведет к исполнению кода, конкретного экземпляра объекта, соответствующего данной переменной.

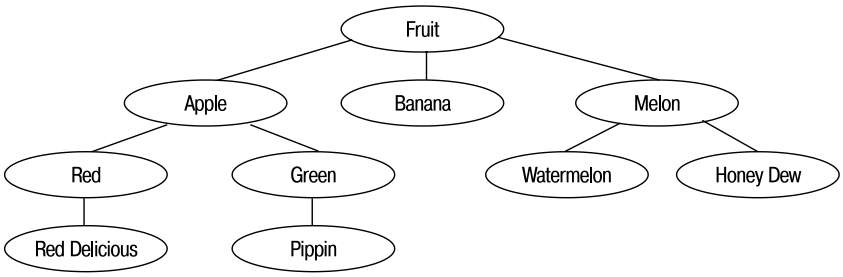

Рис. 2.4. *Пример иерархии наследования*

Замечания о множественном наследовании

Object Pascal не поддерживает множественного наследования, присущего языку C++. *Множественное наследование* (multiple inheritance) означает, что некоторый объект может быть потомком двух и более других различных объектов, т.е. содержать в себе *все* данные и код объектов-предков.

В рамках приведенной на рис. 2.4 иллюстрации это можно представить как класс "яблочное варенье", предками которого окажутся уже существующий класс `apple` (яблоко) и некоторый другой класс "варенье". Хотя подобная функциональность на первый взгляд кажется полезной, она часто вносит в программы множество проблем и служит причиной снижения их эффективности.

Object Pascal предоставляет два возможных пути решения проблемы множественного наследования. Первый состоит в создании объекта одного класса, который *содержит* (contain) объект другого класса (такое решение использовалось при построении VCL). В рамках аналогии "яблочного варенья" можно представить себе некий объект "варенье", который содержится в объекте "яблоко". Второе решение заключается в использовании *интерфейсов* (interfaces), речь о которых пойдет в этой главе далее. При использовании интерфейсов появляется возможность создать объект, который будет иметь оба интерфейса — и "яблока", и "варенья".

Чтобы рассматриваемый далее материал не вызывал затруднений, уточним значение нескольких терминов.

- *Поле* (field). Называется также *переменной экземпляра* (instance variable), представляет собой переменную с данными, содержащуюся внутри объекта. Поле в объекте сходно с полем в записи Pascal. В языке C++ для них иногда используется термин *данные-члены* (data members).
- *Метод* (method). Процедуры и функции, принадлежащие объекту. В языке C++ для них используется термин *функция-член* (member function).

- *Свойство* (property). Свойства представляют собой сущности, обеспечивающие доступ к данным и коду, содержащемуся в объекте. Тем самым они избавляют конечного пользователя от деталей реализации объекта.

НА ЗАМЕТКУ

В объектно-ориентированном подходе прямой доступ к полям объекта считается обычно плохим стилем программирования. В первую очередь, это связано с тем, что детали реализации объекта могут со временем измениться. Предпочтительнее работать со свойствами, представляющими собой стандартный интерфейс объекта, скрывающий его конкретную реализацию.

Объектно-основанное или объектно-ориентированное программирование

Некоторые языки программирования позволяют работать с отдельными сущностями, которые можно считать объектами, но не позволяют создавать собственные объекты. Хорошим примером этого могут служить элементы управления ActiveX (бывшие OCX) в языке Visual Basic. Хотя элементы управления ActiveX можно использовать в приложениях Visual Basic, их нельзя создавать заново или порождать один элемент управления ActiveX из другого. Среда программирования с такими свойствами называется *объектно-основанной* (object-based).

Delphi представляет собой полностью объектно-ориентированную среду. Это означает, что в Delphi новые объекты можно создавать либо с самого начала, либо используя уже имеющиеся. Это относится ко всем объектам Delphi — как визуальным, так и невизуальным (и даже к формам).

Использование объектов Delphi

Как уже говорилось, объекты (называемые также *экземплярами класса*) представляют собой сущности, которые могут содержать данные и код. Объекты Delphi предоставляют программисту все основные возможности объектно-ориентированного программирования, такие как наследование, инкапсуляция и полиморфизм.

Объявление и создание экземпляра

Безусловно, перед тем как использовать объект, его следует объявить. В Object Pascal это делается с помощью ключевого слова `class`. Объявления объектов помещаются в раздел объявления типов (`type`) модуля или программы:

```
type
  TFooObject = class;
```

После объявления типа объекта можно выполнить объявление переменных этого типа (называемых также *экземплярами* (instance)) в разделе `var`:

```
var
  FooObject: TFooObject;
```

В Object Pascal экземпляр объекта создается с помощью вызова одного из *конструкторов* (constructor) этого объекта. Конструктор отвечает за создание экземпляра объекта, а также за выделение памяти и необходимую инициализацию полей. Он не только создает объект, но и приводит его в состояние, необходимое для его дальнейшего использования. Каждый объект содержит по крайней мере один конструктор `Create()`, который может иметь различное количество параметров разного типа — в зависимости от типа объекта. В этой главе рассматривается только простейший конструктор `Create()` (без параметров).

В отличие от языка C++, конструкторы в Object Pascal не вызываются автоматически. Создание каждого объекта с помощью вызова его конструктора входит в обязанности программиста. Синтаксис вызова конструктора следующий:

```
FooObject := TFooObject.Create;
```

Обратите внимание на уникальную особенность вызова конструктора — он вызывается с помощью ссылки на тип, а не на экземпляр типа (в отличие от других методов, которые вызываются с помощью ссылки на экземпляр). На первый взгляд это может показаться нелепостью, однако в этом есть глубокий смысл — ведь экземпляр объекта `FooObject` в момент вызова конструктора еще не создан. Но код конструктора класса `TFooObject` статичен и находится в памяти. Он относится к типу, а не его экземпляру, поэтому такой вызов вполне корректен.

Вызов конструктора для создания экземпляра объекта часто называют *созданием экземпляра* (instantiation).

НА ЗАМЕТКУ

При создании экземпляра объекта с помощью конструктора компилятор гарантирует, что все поля экземпляра будут инициализированы. Все числовые поля будут обнулены, указатели примут значение `nil`, а строки будут пусты.

Уничтожение

По окончании использования экземпляра объекта следует освободить выделенную для него память с помощью метода `Free()`. Этот метод сначала проверяет, не равен ли экземпляр объекта значению `Nil`, и затем вызывает *деструктор* (destructor) объекта — метод `Destroy()`. Понятно, что действие деструктора обратно действию конструктора, т.е. он освобождает всю выделенную память и выполняет другие действия по освобождению захваченных конструктором объекта ресурсов. Синтаксис вызова метода `Free()` прост:

```
FooObject.Free;
```

Обратите внимание, что, в отличие от вызова конструктора, вызов деструктора выполняется с помощью ссылки на экземпляр, а не на тип. Кроме этого, запомните еще один совет — никогда не используйте непосредственный вызов метода `Destroy()`. Более безопасно и корректно вызывать метод `Free()`.

СОВЕТ

В языке C++ деструктор экземпляра статически созданного объекта вызывается автоматически, когда этот экземпляр покидает область видимости. В случае динамического создания экземпляра (с помощью оператора `new`) объект необходимо уничтожить самостоятельно, использовав оператор `delete`. То же правило действует и в Object Pascal, но с одной поправкой: в нем все экземпляры объекта — динамические, и программист должен удалить их сам. Возьмите себе за правило уничтожать и освобождать все, что было создано вами в программе. Исключением из этого правила являются объекты, принадлежащие другим объектам. Этот тип объектов уничтожается автоматически. Еще одним исключением являются объекты с управляемым временем жизни, имеющие собственный счетчик ссылок (например производные от классов `TInterfacedObject` или `TComObject`), которые автоматически удаляются после ликвидации последней ссылки на них.

Могут возникнуть вопросы: откуда берутся все эти конструкторы и деструкторы; неужели необходимо всегда описывать их — даже для самого маленького объекта? Конечно, нет. Дело в том, что все классы Object Pascal неявно наследуют функциональные возможности базового класса `TObject`, причем, независимо от того, указано ли это наследование явно или нет. Рассмотрим следующее объявление класса:

```
Type TFoo = Class;
```

Это полностью эквивалентно следующему объявлению класса:

```
Type TFoo = Class(TObject);
```

Методы

Методы представляют собой процедуры и функции, принадлежащие объекту. Можно сказать, что методы определяют поведение объекта. Выше были рассмотрены два важнейших метода объектов: конструктор и деструктор. Можно самостоятельно создать произвольное количество любых других методов, необходимых для решения конкретных задач.

Создание метода — процесс из двух этапов. Сначала следует описать метод в объявлении типа, а затем создать код его реализации. Вот пример описания и определения метода:

```
type
  TBoogieNights = class
    Dance: Boolean;
    procedure DoTheHustle;
  end;

procedure TBoogieNights.DoTheHustle;
begin
  Dance := True;
end;
```

Отметим, что при определении тела метода необходимо использовать его полное имя с указанием объекта. Вторая важная деталь: метод способен непосредственно обратиться к *любому* полю объекта.

Типы методов

Методы объекта могут быть описаны как статические (static), виртуальные (virtual), динамические (dinamic) или как методы обработки сообщения (message). Рассмотрим следующий пример:

```
TFoo = class
  procedure IAmAStatic;
  procedure IAmAVirtual; virtual;
  procedure IAmADynamic; dynamic;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

Статические методы

Статический метод (static) IAmAStatic работает подобно обычной процедуре или функции. Этот тип методов устанавливается по умолчанию. Адрес такого метода известен уже на стадии компиляции, и компилятор в коде программы оформляет все вызовы данного метода как статические. Такие методы работают быстрее других, однако не могут быть перегружены в целях полиморфизма объектов.

НА ЗАМЕТКУ

Хотя Object Pascal и поддерживает статические методы, но он не поддерживает статические переменные-члены подобно C++ или Java. Для достижения такого эффекта в Object Pascal применяют глобальные переменные. Если необходимо, чтобы глобальная переменная вела себя как закрытая, достаточно поместить ее объявление в раздел реализации (implementation).

Виртуальные методы

Метод IAmAVirtual объявлен как *виртуальный* (virtual). Вызов таких методов из-за возможности их перегрузки немного сложнее, чем вызов статического метода, так как во время компиляции адрес конкретного вызываемого метода не известен. Для решения этой задачи компилятор строит *таблицу виртуальных методов* (VMT — Virtual Method Table), обеспечивающую определение адреса метода в процессе выполнения программы. VMT содержит все виртуальные методы предка и виртуальные методы самого объекта, поэтому виртуальные методы используют несколько больший объем памяти, чем методы динамические, однако их вызов происходит быстрее.

Динамические методы

Динамический (dynamic) метод IAmADynamic в целом подобен виртуальным методам, но обслуживается другой диспетчерской системой. Каждому динамическому методу компилятор назначает уникальное число и использует его вместе с адресом метода для построения таблицы динамических методов (DMT — Dynamic Method Table). В отличие от VMT, DMT содержит методы лишь данного объекта, благодаря чему обеспечивается экономия используемой памяти, но замедляется вызов метода, поскольку для поиска его адреса, скорее всего, будет пересмотрена не одна DMT в иерархии объектов.

Методы обработки сообщения

Метод *обработки сообщений* (message-handling) `IAmAMessage`. Значение после ключевого слова `message` определяет сообщение, в ответ на которое вызывается данный метод. Такие методы создаются для реакции на те или иные сообщения Windows. Они никогда не вызываются непосредственно из программы. Более подробная информация об обработке сообщений приведена в главе 3, "Приключения сообщения".

Переопределение методов

Переопределение (overriding) метода в Object Pascal реализует концепцию полиморфизма. Оно позволяет изменять поведение метода от наследника к наследнику. Переопределение метода возможно только в том случае, если первоначально он был объявлен как `virtual` или `dynamic`. Для переопределения метода при его объявлении вместо ключевых слов `virtual` или `dynamic` следует указать ключевое слово `override`. Ниже приведен пример переопределения методов `IAmAVirtual` и `IAmADynamic`.

```
TFooChild = class(TFoo)
  procedure IAmAVirtual; override;
  procedure IAmADynamic; override;
  procedure IAmAMessage(var M: TMessage); message wm_SomeMessage;
end;
```

Директива `override` приводит к замещению строки описания исходного метода в VMT строкой описания нового метода. Если объявить новые функции с ключевым словом `virtual` или `dynamic`, а не `override`, то вместо замещения старых будут созданы *новые* методы. В случае переопределения статического метода, новый вариант просто полностью заменит статический метод родителя.

Перегрузка метода

Подобно обычным процедурам и функциям, методы могут быть перегружены таким образом, чтобы класс содержал несколько методов с одним именем, но с различными списками параметров. Перегруженные методы должны быть объявлены с указанием директивы `overload` (использовать эту директиву при описании первого перегруженного метода необязательно). Вот пример объявления объекта с перегруженными методами:

```
type
  TSomeClass = class
    procedure AMethod(I: Integer); overload;
    procedure AMethod(S: string); overload;
    procedure AMethod(D: Double); overload;
end;
```

Дублирование имен методов

Иногда может понадобиться к одному из классов добавить метод, замещающий метод с тем же именем, но принадлежащий предку этого класса. В данном случае требуется не переопределить исходный метод, а полностью его заменить. Если просто добавить такой метод в новый класс, то компилятор выдаст предупреждение о том, что новый метод скрывает метод базового класса с тем же именем. Для устранения этой ошибки в новом методе укажите директиву `reintroduce`:

```
type
  TSomeBase = class
    procedure Cooper;
  end;

  TSomeClass = class
    procedure Cooper; reintroduce;
  end;
```

Указатель Self

Во всех методах объекта доступна неявная переменная `Self`, представляющая собой указатель на тот экземпляр объекта, который был использован при данном вызове этого метода. Переменная `Self` передается методу компилятором в качестве скрытого параметра.

Свойства

Свойства объекта (properties) — это специализированные средства доступа к полям объекта, позволяющие изменять данные его полей и выполнять код его методов. По отношению к компонентам свойства являются теми элементами, сведения о которых отображаются в окне Object Inspector. Вот простой пример объекта, для которого определено свойство:

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
public
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

Класс `TMyObject` представляет собой объект, содержащий одно поле — целое по имени `SomeValue`, один метод — процедуру `SetSomeValue`, а также одно свойство по имени `Value`. Назначение процедуры `SetSomeValue` состоит в присвоении полю `SomeValue` некоторого значения. Свойство `Value` не содержит данных — оно используется в качестве средства доступа к полю `SomeValue`. Когда запрашивают значение свойства `Value`, оно считывает его из поля `SomeValue` и возвращает. При попытке присвоить свойству `Value` некоторое значение вызывается процедура `SetSomeValue`, предназначенная для изменения значения поля `SomeValue`. Вся эта технология имеет два основных преимущества. Во-первых, она создает для конечного пользователя некий интерфейс, полностью скрывающий реализацию объекта и обеспечивающий контроль за доступом к объекту. Во-вторых, она позволяет замещать методы в производных классах, что обеспечивает полиморфизм поведения объектов.

Определение области видимости

Object Pascal предоставляет дополнительный контроль степени доступа к членам классов (полей и методов) с помощью директив `protected`, `private`, `public`, `published` и `automated`, открывающих соответствующие разделы объявлений. Синтаксис использования этих директив следующий:

```
TSomeObject = class
private
  APrivateVariable: Integer;
  AnotherPrivateVariable: Boolean;
protected
  procedure AProtectedProcedure;
  function ProtectMe: Byte;
public
  constructor APublicContructor;
  destructor APublicKiller;
published
  property AProperty read APrivateVariable write APrivateVariable;
end;
```

За каждой из директив может следовать любое необходимое количество объявлений полей или методов. Требования хорошего тона предполагают наличие отступа — аналогично тому, как это делается для имен классов. Что же означают эти разделы?

- `Private` (закрытый). Объявленные в данном разделе переменные и методы доступны только для того кода, который находится в блоке реализации самого объекта. Директива `private` скрывает особенности реализации объекта от пользователей и защищает члены этого объекта от непосредственного доступа и изменения извне.
- `Protected` (защищенный). Члены объекта, объявленные в разделе `protected`, доступны объектам, производным от такого класса. Это позволяет скрыть внутреннее устройство объекта от пользователя и в то же время обеспечить необходимую гибкость, а также эффективность доступа к полям и методам объекта для его потомков.
- `Public` (открытый). Объявленные в этом разделе члены объекта доступны в любом месте программы. Конструкторы и деструкторы всегда должны быть объявлены как `public`.
- `Published` (публикуемый). Для членов объекта, объявленных в данном разделе, при компиляции будет создана информация о типах времени выполнения (RTTI — Runtime Type Information). Это позволит другим элементам приложения получать информацию об элементах объекта, объявленных как `published`. В частности, подобная информация используется утилитой Object Inspector при построении списков свойств объектов.
- `Automated` (автоматизированный). Этот раздел сохранен только для обеспечения совместимости с Delphi 2. Более подробная информация по данной теме приведена в главе 15, “Разработка приложений COM”.

Ниже показано объявление класса TMyObject (использовавшееся в разделе, посвященном свойствам объекта), которое дополнено элементами, повышающими его целостность и логичность.

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
published
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

Теперь ни один из пользователей этого класса не сможет изменить значение SomeValue непосредственно и вынужден будет использовать только интерфейс, предоставляемый свойством Value.

Дружественные классы

В языке C++ реализована концепция *дружественных классов* (friend classes), т.е. классов, которым разрешен доступ к закрытым (private) данным и функциям другого класса. В языке C++ этот механизм реализуется с помощью ключевого слова friend. Хотя, по большому счету, в Object Pascal нет ни такого ключевого слова, ни такого понятия, тем не менее фактически того же эффекта можно добиться, просто описав объекты в одном модуле. Описанные в одном модуле объекты имеют право доступа к закрытым данным и функциям друг друга, благодаря чему и обеспечивается "дружественность" в пределах модуля.

Внутреннее представление объектов

Все экземпляры классов Object Pascal на самом деле представляют собой 32-битовые указатели на данные этого экземпляра объекта, расположенные в динамической памяти. При доступе к полям, методам или свойствам объекта компилятор автоматически выполняет скрытые действия, приводящие к возвращению данных по этим ссылкам. В результате для постороннего взгляда объект всегда выглядит как статическая переменная. Однако использование подобного механизма означает, что, в отличие от языка C++, язык Object Pascal не предоставляет реального метода для размещения данных класса вне динамической памяти (например в сегменте данных приложения).

Базовый класс TObject

Все объекты языка Object Pascal являются производными (наследниками) базового (родительского) класса TObject, а следовательно, автоматически наследуют все его методы. В результате любой объект способен, например, сообщить свое имя, тип и даже указать, является ли он производным от того или класса. Самым замечательным во всем этом механизме является тот факт, что прикладной программист избавлен от

необходимости беспокоиться о реализации стандартных функций — он может просто использовать их по своему усмотрению.

TObject является особым объектом, определение которого расположено в модуле System и всегда доступно компилятору:

```
type
  TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString):
                                                  Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry(const IID: TGUID):
                                       PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
                       ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

Документацию каждого из перечисленных методов можно найти в интерактивной справочной системе Delphi.

Обратите внимание на методы, объявление которых начинается со слова class. Это означает, что метод может быть вызван как обычная процедура или функция без создания экземпляра класса, членом которого является данный метод. (Аналоги таких методов в языке C++ — это методы, объявленные как static.) Будьте осторожны при создании подобных методов — они не должны использовать никакой информации экземпляра класса, поскольку это приведет к ошибке компиляции.

Интерфейсы

Возможно, наиболее важным дополнением к языку Object Pascal стала поддержка *интерфейсов* (interfaces), введенная в Delphi 3. Интерфейс определяет набор функций и процедур, которые могут быть использованы для взаимодействия программы с объектом. Определение конкретного интерфейса известно и разработчику, и его пользовате-

лю и воспринимается как соглашение о правилах объявления и использования этого интерфейса. В классе может быть реализовано несколько интерфейсов. В результате объект становится "многоликим", являя клиенту каждого интерфейса свое особое лицо.

Интерфейсы представляют собой только определения того, каким образом могут сообщаться между собой объект и его клиент. Данная концепция схожа с концепцией чистых виртуальных классов (`PURE VIRTUAL`) языка C++. Класс, обеспечивающий поддержку некоторого интерфейса, обязан обеспечить и реализацию всех его функций и процедур.

В настоящей главе речь пойдет лишь о синтаксических элементах интерфейсов. Более подробная информация по этой теме приведена в главе 15, "Разработка приложений COM".

Определение интерфейса

Подобно тому, как все классы языка Object Pascal происходят от класса `TObject`, все интерфейсы явно или неявно являются потомками интерфейса `IUnknown`, определение которого в модуле `System` выглядит следующим образом:

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): Integer;
                                                        stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

Как видите, синтаксис определения интерфейса очень похож на синтаксис определения класса. Основное различие между ними заключается в том, что интерфейс, в отличие от класса, может быть связан с *глобальным уникальным идентификатором* (Globally Unique Identifier — GUID). Определение интерфейса `IUnknown` "происходит" из спецификации *модели компонентных объектов* (COM — Component Object Model) *Microsoft*. Более подробная информация по этой теме приведена в главе 15, "Разработка приложений COM".

Для того, кто умеет создавать классы, определение собственного интерфейса в Delphi не представляет особых сложностей. Ниже приведен пример определения нового интерфейса `IFoo`, который реализует единственный метод `F1()`.

```
type
  IFoo = interface
    ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;
```

СОВЕТ

Интегрированная среда разработки Delphi создает новый GUID при нажатии комбинации клавиш <Ctrl+Shift+G>.

Следующий код определяет новый интерфейс `IBar`, производный от `IFoo`:

```
type
  IBar = interface(IFoo)
    ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
    function F2: Integer;
  end;
```

Реализация интерфейса

Приведенный ниже фрагмент кода демонстрирует реализацию интерфейсов `IFoo` и `IBar` в классе `TFooBar`.

```
type
  TFooBar = class(TInterfacedObject, IFoo, IBar)
    function F1: Integer;
    function F2: Integer;
  end;

function TFooBar.F1: Integer;
begin
  Result := 0;
end;

function TFooBar.F2: Integer;
begin
  Result := 0;
end;
```

Обратите внимание, в первой строке объявления класса, после указания базового класса, может быть перечислено несколько интерфейсов. Связывание функции интерфейса с определенной функцией класса осуществляется компилятором тогда, когда он обнаруживает метод класса, сигнатура которого совпадает с сигнатурой метода интерфейса. Класс должен содержать собственные объявления и реализации всех методов интерфейса, в противном случае компилятор выдаст соответствующее сообщение об ошибке и программа не будет откомпилирована.

Если класс реализует несколько интерфейсов, которые имеют методы с одинаковыми сигнатурами, то избежать неоднозначности позволяет назначение методам *псевдонимов* (alias), как показано в следующем примере:

```
type
  IFoo = interface
    ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;

  IBar = interface
    ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;

  TFooBar = class(TInterfacedObject, IFoo, IBar)
    // Назначение методам псевдонимов
    function IFoo.F1 = FooF1;
    function IBar.F1 = BarF1;
```

```
    // Методы интерфейса
    function FooF1: Integer;
    function BarF1: Integer;
  end;

function TFooBar.FooF1: Integer;
begin
  Result := 0;
end;

function TFooBar.BarF1: Integer;
begin
  Result := 0;
end;
```

Директива implements

Директива `implements` впервые была введена в Delphi 4. Она позволяет делегировать реализацию методов интерфейса другим классам или интерфейсам. Она всегда указывается как последняя директива в объявлении свойства класса или интерфейса:

```
type
  TSomeClass = class(TInterfacedObject, IFoo)
    // ...
    function GetFoo: TFoo;
    property Foo: TFoo read GetFoo implements IFoo;
    // ...
  end;
```

В приведенном примере использование директивы `implements` говорит о том, что методы, реализующие интерфейс `IFoo`, следует искать в свойстве `Foo`. Тип свойства должен быть классом, содержащим методы `IFoo`, интерфейс `IFoo` или его потомков. В директиве `implements` можно использовать не один, а целый список интерфейсов (элементы списка должны отделяться запятыми). В таком случае класс, используемый для представления свойства, должен содержать методы, реализующие все приведенные в списке интерфейсы.

Директива `implements` позволяет выполнять бесконфликтную агрегацию. *Агрегация* (aggregation) — это концепция COM, отвечающая за комбинацию нескольких классов для выполнения одной задачи. (Более подробная информация по данной теме приведена в главе 15, “Разработка приложений COM”.) Другое немаловажное достоинство применения директивы `implements` заключается в том, что она позволяет отложить использование необходимых для реализации интерфейса ресурсов до того момента, когда они потребуются реально. Предположим, что в реализации некоторого интерфейса необходим мегабайт памяти для хранения растрового изображения, но этот интерфейс применяется достаточно редко. Вряд ли можно считать эффективным решение постоянно выделять мегабайт памяти, который будет использоваться лишь время от времени. Директива `implements` позволяет реализовать интерфейс в отдельном классе, экземпляр которого будет создаваться только по прямому запросу пользователя.

Использование интерфейсов

При использовании переменных типа интерфейса (экземпляров) в приложениях следует помнить о нескольких важных правилах. Во-первых, не забывайте, что интерфейс является типом данных с управляемым временем жизни. А это означает, что он всегда инициализируется значением `nil`, обладает счетчиком ссылок и автоматически уничтожается при выходе за пределы области видимости. Вот небольшой пример, иллюстрирующий управление временем жизни экземпляра интерфейса:

```
var
  I: ISomeInterface;
begin
  // I инициализируется значением nil
  I := FunctionReturningAnInterface;  // Счетчик ссылок
                                      // увеличивается на 1
  I.SomeFunc;   // Счетчик ссылок уменьшается на 1.
                // Когда значение счетчика станет равным 0,
                // объект I будет автоматически уничтожен.
end;
```

К тому же всегда нужно помнить, что переменные типа интерфейса совместимы по присвоению с классами, реализующими интерфейсы. В приведенном ниже примере корректно используется объявленный ранее класс `TFooBar`.

```
procedure Test(FB: TFooBar)
var
  F: IFoo;
begin
F := FB;  // Корректно, поскольку FB поддерживает IFoo
  .
  .
  .
```

Последнее правило гласит, что для переменной типа интерфейса может быть использован оператор преобразования типов `as` как вызов метода `QueryInterface`, возвращающего адрес другой переменной типа интерфейса (не отчаивайтесь, если это правило пока не до конца понятно, — более подробная информация по данной теме приведена в главе 15, "Разработка приложений COM"). Указанное правило можно проиллюстрировать следующим примером:

```
var
  FB: TFooBar;
  F: IFoo;
  B: IBar;
begin
  FB := TFooBar.Create
  F := FB;  // Допустимо, так как FB поддерживает IFoo
  B := F as IBar;  // Вызов функции QueryInterface F для IBar
  .
  .
  .
```

Если запрошенный интерфейс не поддерживается, то будет передано соответствующее исключение.

Структурная обработка исключений

Структурная обработка исключений (SEH — Structured Exception Handling) представляет собой метод обработки ошибок. Благодаря ему можно восстановить нормальную работу приложения после сбоя в работе программы, который в противном случае стал бы фатальным. Исключения были введены в язык Object Pascal в Delphi 1.0, но только начиная с Delphi 2.0 они стали частью интерфейса API Win32. То, что исключения являются не более чем классами, содержащими информацию о месте и характере каждой ошибки, делает их применение в Object Pascal простым и общедоступным. Это позволяет реализовать и использовать исключения в приложениях наравне с любым другим классом.

Delphi имеет предопределенные классы исключений, предназначенные для обработки стандартных ошибок, таких как нехватка памяти, деление на нуль, переполнение числа или ошибки ввода/вывода. Delphi также позволяет создавать собственные классы исключений, предназначенные для применения в создаваемых приложениях.

В листинге 2.3 приведен пример обработки исключения при файловых операциях ввода/вывода.

Листинг 2.3. Обработка исключений файловых операций

```
Program FileIO;

uses Classes, Dialogs;

{$APPTYPE CONSOLE}

var
  F: TextFile;
  S: string;
begin
  AssignFile(F, 'FOO.TXT');
  try
    Reset(F);
    try
      ReadLn(F, S);
    finally
      CloseFile(F);
    end;
  except
    on EInOutError do
      ShowMessage('Error Accessing File!');
  end;
end.
```

В данном листинге внутренний блок `try..finally` используется для гарантии того, что файл будет закрыт в любом случае, т.е. независимо от того, допущена ошибка или нет. Иначе говоря, это звучит так: “Программа, попытайся выполнить код между операторами `try` и `finally`. Независимо от того, возникла проблема или нет, выполни код между операторами `finally` и `end`. Но если проблема все же возникла, обратись к блоку обработки”.

НА ЗАМЕТКУ

Операторы после ключевого слова `finally` в блоке `try..finally` выполняются независимо от того, произошло исключение в процессе выполнения этого блока или нет. Создавая этот код, убедитесь, что он не зависит от того, передавалось исключение или нет. Кроме того, в связи с тем, что оператор `finally` не прекращает продвижения исключения, выполнение программы возобновится в расположенном далее по тексту программы блоке обработки исключения.

Внешний блок `try..except` используется для обработки исключения, которое может произойти в программе. После закрытия файла во внутреннем блоке `finally` блок `except` выводит сообщение об ошибке (если она произошла).

Одно из главных преимуществ системы обработки исключений по сравнению с традиционными методами обработки ошибок состоит в отделении процедуры обнаружения ошибки от процедуры ее обработки. Это позволяет сделать код более удобным для чтения и понимания и в каждом случае сосредоточиться на отдельном аспекте работы программы.

Безусловно, блок `try..finally` не сможет перехватить абсолютно все исключения. Помещая в программу блок `try..finally`, программист заботится не только об обнаружении некоторого конкретного исключения. Основная цель состоит в том, чтобы выполнить все необходимые действия для корректного выхода из *любой* нештатной ситуации, которая может произойти при выполнении этого фрагмента программы. Блок `finally` является идеальным местом для выполнения действий по освобождению любых распределенных ресурсов, поскольку он выполняется всегда, включая и случай возникновения ошибки. Тем не менее, во многих ситуациях обработка ошибок может быть связана с выполнением определенных действий, зависящих от типа возникшей ошибки. Для этого и предназначен блок обработки исключений `try..except`, пример использования которого показан в листинге 2.4.

Листинг 2.4. Блок обработки исключений `try..except`

```
Program HandleIt;

{$APPTYPE CONSOLE}

var
  R1, R2: Double;
begin
  while True do begin
    try
      Write('Enter a real number: ');
      ReadLn(R1);
      Write('Enter another real number: ');
      ReadLn(R2);
      Writeln('Now divide the first number by the second...');
      Writeln('The answer is: ', (R1 / R2):5:2);
    except
      On EZeroDivide do
```

```
        Writeln('You cannot divide by zero!');
      On EInOutError do
        Writeln('That is not a valid number!');
    end;
  end;
end.
```

В блоке try..except, помимо перехвата конкретных исключений, с помощью конструкции else можно организовать обработку всех остальных типов ошибок. Синтаксис конструкции try..except..else показан в следующем примере:

```
try
  Statements // Вероятный источник исключений
except
  On ESomeException do Something; // Обработчик конкретного
                                  // исключения
else
  { Стандартный обработчик для всех остальных исключений }
end;
```

СОВЕТ

При использовании конструкции try..except..else следует учитывать, что в блоке else будут обрабатываться *все* типы ошибок, включая и те, которых никто не ожидал, например ошибки выделения памяти или некоторые типы ошибок выполнения программы. Поэтому при использовании ключевого слова else проявляйте осторожность, ведь в случае некорректной обработки исключения возникает еще одно исключение. Более подробная информация по этой теме приведена в настоящей главе далее.

Того же самого эффекта, что и конструкция try..except..else (перехват всех типов исключений), можно достичь и с помощью конструкции try..except без указания типа обрабатываемой ошибки:

```
try
  Statements // Вероятный источник исключений
except
  HandleException // Аналог оператора else
end;
```

Классы исключений

Исключения (exceptions) представляют собой экземпляры специальных объектов. Они создаются при передаче исключения и уничтожаются после обработки. Базовым классом для всех исключений является класс Exception, определяемый следующим образом:

```
type
  Exception = class(TObject)
  private
    FMessage: string;
    FHelpContext: Integer;
```

```
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string;
                          const Args: array of const);
    constructor CreateRes(Ident: Integer); overload;
    constructor CreateRes(ResStringRec: PResStringRec); overload;
    constructor CreateResFmt(Ident: Integer;
                             const Args: array of const);
                             overload;
    constructor CreateResFmt(ResStringRec: PResStringRec;
    const Args: array of const); overload;
    constructor CreateHelp(const Msg: string;
                           AHelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string;
                              const Args: array of const;
                              AHelpContext: Integer);
    constructor CreateResHelp(Ident: Integer;
                              AHelpContext: Integer); overload;
    constructor CreateResHelp(ResStringRec: PResStringRec;
                              AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(ResStringRec: PResStringRec;
                                 const Args: array of const;
                                 AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(Ident: Integer;
                                 const Args: array of const;
                                 AHelpContext: Integer); overload;
    property HelpContext: Integer read FHelpContext
                                 write FHelpContext;
    property Message: string read FMessage write FMessage;
  end;
```

Важным элементом объекта `Exception` является свойство `Message`, имеющее строковый тип. Оно содержит дополнительную информацию или разъяснение сути произошедшей ошибки. Вид информации в свойстве `Message` зависит от типа возникшего исключения.

СОВЕТ

При определении собственного класса исключения обязательно объявляйте его как производный от уже существующего класса исключения. Это может быть класс `Exception` или любой его потомок. Суть такого требования заключается в том, что в данном случае новый объект исключения гарантированно будет перехвачен стандартным обработчиком исключений.

При обработке конкретного типа исключения в блоке `except` его обработчик будет перехватывать также и все его производные типы. Например, если исключение `EMathError` является базовым для нескольких типов исключений, в том числе для `EZeroDivide` и `EOverflow`, то все они будут перехвачены обработчиком для исключения `EMathError`:

```
try
  Statements
```

```
except
  on EMathError do  // Перехватит исключение класса EMathError и
                    // все производные от него
    HandleException
end;
```

Любое исключение, которое не будет перехвачено и обработано самой программой, перехватит и обработает стандартный обработчик исключений, входящий в состав динамической библиотеки Delphi. Этот обработчик выводит диалоговое окно с сообщением, извещающим пользователя о возникшем исключении.

При обработке исключения может потребоваться доступ к самому экземпляру объекта исключения — например, чтобы получить дополнительную информацию о случившемся, содержащуюся в свойстве Message. Для этого есть два пути: использование необязательного идентификатора в конструкции on ESomeException или обращение к функции ExceptObject().

Добавив необязательный идентификатор в конструкцию on ESomeException блока except, можно получить идентификатор текущего объекта исключения. Синтаксис использования необязательного идентификатора заключается в указании самого идентификатора и двоеточия непосредственно *перед* типом исключения:

```
try
  Something
except
  on E:ESomeException do
    ShowMessage(E.Message);
end;
```

Идентификатор (в данном случае E) становится экземпляром текущего исключения. Этот идентификатор всегда имеет тот же тип, что и породившее его исключение.

Кроме того, можно воспользоваться функцией ExceptObject(), которая возвращает экземпляр текущего объекта исключения. Однако следует отметить, что типом возвращаемого функцией значения всегда является класс TObject и для работы с ним потребуется выполнить преобразование типов. Вот пример применения этой функции:

```
try
  Something
except
  on ESomeException do
    ShowMessage(ESomeException(ExceptObject).Message);
end;
```

В отсутствии активного исключения вызов функции ExceptObject() возвращает Nil.

Синтаксис генерации исключения подобен синтаксису создания экземпляра объекта. Так, для создания пользовательского исключения типа EBadStuff используется следующий синтаксис:

```
Raise EBadStuff.Create('Описание возникшей ошибки');
```

Процесс обработки исключений

После возникновения и передачи объекта исключения нормальный ход выполнения программы прерывается и управление начинает передаваться от одного обработчика исключений к другому до тех пор, пока исключение не будет обработано, а экземпляр объекта исключения уничтожен. Этот процесс построен на обработке стека вызовов и, следовательно, имеет глобальный характер в пределах всей программы, а не только в рамках текущей процедуры или модуля. В листинге 2.5 приведен пример, иллюстрирующий указанный принцип обработки исключений. Здесь представлен главный модуль приложения Delphi, содержащего единственную форму с одной кнопкой. Если щелкнуть на кнопке, то метод `Button1Click()` (обработки этого события) вызывает процедуру `Proc1()`, которая, в свою очередь, вызывает процедуру `Proc2()`, вызывающую процедуру `Proc3()`. Исключение передается именно в этой, последней, наиболее глубоко вложенной процедуре `Proc3()`, что позволяет проследить весь процесс прохождения исключения через каждый из блоков `try..finally` до тех пор, пока оно не будет обработано внутри метода `Button1Click()`.

СОВЕТ

Если запустить эту программу из интегрированной среды разработки Delphi, то проследить процесс обработки исключения будет проще, если предварительно отключить встроенный обработчик исключений отладчика. Для этого следует сбросить флажок Stop on Delphi Exceptions во вкладке Language Exceptions диалогового окна Debugger Options. Доступ к этому диалоговому окну находится в меню Tools пункт Debugger Options.

Листинг 2.5. Демонстрация передачи и обработки исключений

```
unit Main;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
      Button1: TButton;
      procedure Button1Click(Sender: TObject);
    private
      { Закрытые объявления }
    public
      { Открытые объявления }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

type
  EBadStuff = class(Exception);

procedure Proc3;
begin
  try
    raise EBadStuff.Create('Up the stack we go!');
  finally
    ShowMessage('Exception raised. Proc3 sees the exception');
  end;
end;

procedure Proc2;
begin
  try
    Proc3;
  finally
    ShowMessage('Proc2 sees the exception');
  end;
end;

procedure Proc1;
begin
  try
    Proc2;
  finally
    ShowMessage('Proc1 sees the exception');
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
const
  ExceptMsg = 'Exception handled in calling procedure. The message
is "%s"';
begin
  ShowMessage('This method calls Proc1 which calls Proc2 which
calls Proc3');
  try
    Proc1;
  except
    on E:EBadStuff do
    ShowMessage(Format(ExceptMsg, [E.Message]));
  end;
end;
```

```
end.
```

Повторная передача исключения

Если во внутреннем блоке `try..except` создается пользовательский обработчик исключения, выполняющий специальные действия, но не прекращающий дальнейшую передачу исключения вплоть до стандартного обработчика, то можно воспользоваться технологией *повторной передачи исключения* (reraising the exception). Листинг 2.6 демонстрирует пример повторной передачи исключения.

Листинг 2.6. Повторная передача исключения

```
try              // Внешний блок
  { Оператор }
  { Оператор }
  { Оператор }
  try            // специальный внутренний блок
    { действия, требующие специальной обработки исключения }
  except
    on ESomeException do
    begin
      { Специальный внутренний обработчик исключения }
      raise;     // Повторная передача исключения во внешний блок
    end;
  end;
except
  // Внешний блок со стандартным обработчиком
  on ESomeException do Something;
end;
```

Информация о типах времени выполнения

Информация о типах времени выполнения (RTTI — Runtime Type Information) представляет собой способность языка предоставлять приложениям Delphi информацию об объектах непосредственно во время выполнения программы. Эта же функция используется для обмена информацией между компонентами Delphi и графической средой разработки.

Будучи потомками класса `TObject`, все объекты Object Pascal содержат указатель на информацию о типе и некоторые встроенные методы для работы с ней, обеспечивающие функциональные возможности RTTI. Ниже приведены некоторые методы класса `TObject`, предназначенные для получения информации о конкретном экземпляре объекта.

Таблица 2.7. Методы класса `TObject`, используемые для RTTI

Функция	*Возвращаемый тип*	*Информация*
`ClassName()`	`string`	Имя класса объекта
`ClassType()`	`TClass`	Тип объекта
`InheritsFrom()`	`Boolean`	Логический индикатор (`Boolean`) происхождения одного класса от другого
`ClassParent()`	`TClass`	Тип базового класса
`InstanceSize()`	`word`	Размер экземпляра в байтах
`ClassInfo()`	`Pointer`	Указатель на RTTI объекта в памяти

Object Pascal обладает двумя операторами — `is` и `as`, — которые позволяют проводить сравнение и преобразование типов с помощью средств RTTI.

Оператор `as` позволяет привести объект базового класса к типу производного (при невозможности такого преобразования будет передано исключение). Предположим, что существует процедура, которой может быть передан в качестве параметра объект любого типа. Ниже приведен возможный вариант объявления такой процедуры:

```
Procedure Foo(AnObject: TObject);
```

Если необходимо не только получить объект в качестве параметра, а и сделать с его помощью нечто полезное, то потребуется привести его к типу производного класса, обладающего необходимыми возможностями. Предположим, что полученный объект `AnObject` опознан как производный от класса `TEdit` и необходимо изменить текст, содержащийся в этом объекте. (В Delphi класс `TEdit` является элементом управления VCL, представляющим собой окно текстового редактора.) Для него можно воспользоваться таким кодом:

```
(Foo as TEdit).Text := 'Hello World.';
```

Кроме того, для непосредственного сравнения типов объектов можно использовать оператор `is`. Этот оператор позволяет сравнить характеристики неизвестного объекта с характеристиками известного типа данных или экземпляра объекта. На основе такого сравнения можно будет сделать выводы о возможных свойствах и поведении неизвестного объекта. Так, до выполнения приведения типа (см. предыдущий пример) можно проверить, является ли полученный объект `AnObject` совместимым с классом `TEdit`:

```
If (Foo is TEdit) then
  TEdit(Foo).Text := 'Hello World.';
```

Обратите внимание, что в данном примере во второй строке вместо оператора `as` использовано стандартное приведение типа. Это сделано для повышения эффективности работы программы, поскольку в первой строке было уже установлено, что тип объекта `Foo` относится к `TEdit`, и дополнительная проверка допустимости преобразования, выполняемая при использовании оператора `as`, уже не нужна.

Резюме

В этой главе рассматривался базовый синтаксис и семантики языка Object Pascal, включая переменные, операторы, функции, процедуры, типы данных, конструкции и стили программирования. Кроме того, обсуждалась концепция объектно-ориентированного программирования, в том числе понятия объектов, полей, свойств, методов, интерфейсов, обработки исключений, RTTI, а также базовый для всех классов Delphi класс `TObject`.

Теперь, имея общее представление об объектно-ориентированном языке Object Pascal и его работе, можно перейти к обсуждению приложений Delphi и концепциям их разработки.

ГЛАВА

3

Приключения сообщения

В ЭТОЙ ГЛАВЕ...

Хотя компоненты библиотеки VCL (Visual Component Library) и преобразуют большинство сообщений Win32 в события языка Object Pascal, иметь понятие о работе системы сообщений Windows желательно для всех программистов Win32.

Потребности программиста Delphi практически полностью удовлетворяются возможностями работы с событиями, предоставляемыми VCL. Но при создании серьезных нестандартных приложений, а особенно при разработке компонентов Delphi, безусловно, потребуется непосредственная обработка сообщений Windows, позволяющая самостоятельно создавать события, соответствующие этим сообщениям.

НА ЗАМЕТКУ

Возможности обмена сообщениями, описанные в этой главе, специфичны для библиотеки VCL и не поддерживаются библиотекой CLX. Более подробная информация об архитектуре CLX приведена в главе 10, "Архитектура компонентов: VCL и CLX".

Что такое сообщение?

Сообщение (message) представляет собой извещение о некотором событии, посылаемое системой Windows в адрес приложения. Любые действия пользователя — щелчок мышью, изменение размеров окна приложения, нажатие клавиши на клавиатуре — вынуждают Windows отправить приложению сообщение, извещающее о том, что произошло в системе.

Сообщение представляет собой *запись* (record), передаваемую приложению системой Windows. Эта запись содержит информацию о типе произошедшего события и дополнительную информацию, специфическую для данного сообщения. Например, для щелчка мышью запись содержит дополнительную информацию о координатах указателя мыши на момент щелчка и номер нажатой кнопки. Тип записи, используемый Delphi для представления сообщений Windows, называется `TMsg`. Он определен в модуле `Windows` следующим образом:

```
type
  TMsg = packed record
    hwnd: HWND;     // Дескриптор (handle) окна-получателя
    message: UINT;  // Идентификатор сообщения
    wParam: WPARAM; // 32 бита дополнительной информации
    lParam: LPARAM; // 32 бита дополнительной информации
    time: DWORD;    // Время создания сообщения
    pt: TPoint;     // Позиция мыши в момент создания сообщения
  end;
```

Что же содержится в сообщении?

Ниже приведено более подробное описание формата записи сообщения.

- `hwnd` — 32-битовый *дескриптор окна* (window handle), которому предназначено сообщение. Окно может быть практически любым типом объекта на экране, поскольку Win32 поддерживает дескрипторы окон для большинства визуальных объектов (окон, диалоговых окон, кнопок, полей ввода и т.п.).

- `message` — константа, соответствующая некоторому стандартному сообщению. Системные константы сообщений (для Windows) определены в модуле `Windows`, а константы для пользовательских сообщений программист должен определить сам.
- `wParam` — это поле часто содержит какую-то константу, значение которой определяется типом сообщения. Кроме того, оно может содержать идентификатор окна или элемента управления, связанного с данным сообщением.
- `lParam` — это поле чаще всего хранит индекс или указатель на некоторые данные в памяти. Так как поля `WParam`, `LParam` и `Pointer` имеют один и тот же размер, равный 32 битам, между ними допускается взаимное преобразование типов.

Теперь, ознакомившись с тем, что представляют собой сообщения Windows в целом, можно более подробно рассмотреть их различные типы.

Типы сообщений

Интерфейс прикладных программ Windows 32 (API Win32 – Application Programming Interface Win32) является интерфейсом операционной системы для взаимодействия с большинством приложений MS Windows. Здесь каждому сообщению Windows поставлено в соответствие определенное значение, которое заносится в поле `message` записи `TMsg`. В Delphi все эти константы определены в модуле `Messages`, и большая их часть задокументирована в интерактивной справочной системе. Обратите внимание, что имя каждой константы, представляющей тип сообщения, начинается с символов *WM* (т.е. ***W**indows **M**essage*). В табл. 3.1 представлены некоторые наиболее распространенные сообщения Windows и их числовые коды.

Таблица 3.1. Основные типы сообщений Windows

Идентификатор сообщения	*Значение*	*Сообщает окну о том, что...*
`wm_Activate`	`$00l6`	Оно активизируется или дезактивируется
`wm_Char`	`$0102`	От некоторой клавиши были посланы сообщения `wm_KeyDown` или `wm_KeyUp`
`wm_Close`	`$0010`	Оно должно быть закрыто
`wm_KeyDown`	`$0100`	На клавиатуре была нажата клавиша
`wm_KeyUp`	`$0101`	Клавиша на клавиатуре была отпущена
`wm_LButtonDown`	`$0201`	Пользователь нажал левую кнопку мыши
`wm_MouseMove`	`$0200`	Указатель мыши переместился
`WM_PAINT`	`$000F`	Необходимо перерисовать клиентскую область окна
`wm_Timer`	`$0113`	Произошло событие таймера
`wm_Quit`	`$0012`	Программа должна быть завершена

Принципы работы системы сообщений Windows

Система сообщений Windows состоит из трех компонентов.

- *Очередь сообщений* (Message queue). Windows поддерживает отдельную очередь сообщений для каждого приложения. Приложение Windows должно получать сообщения из этой очереди и передавать их соответствующему окну.
- *Цикл сообщений* (Message loop). Группа циклически выполняемых операторов приложения, осуществляющая выборку сообщения из очереди и передачу его соответствующему окну для обработки. Цикл выборки и передачи сообщений выполняется на протяжении всего периода работы приложения.
- *Процедура окна* (Window procedure). Каждое окно приложения имеет собственную процедуру, которая получает все передаваемые данному окну сообщения. В ответ на полученное сообщение процедура должна выполнить определенные действия. Эта процедура является процедурой обратного вызова и обычно возвращает Windows некоторое значение по завершении обработки сообщения.

НА ЗАМЕТКУ

Функция обратного вызова (callback function) представляет собой функцию в программе, обращение к которой осуществляет операционная система Windows или некоторый другой внешний модуль.

Путешествие сообщения из пункта А (где произошло породившее его событие) в пункт Б (окно пользовательского приложения, отвечающее на это сообщение) состоит из пяти этапов.

1. В системе происходит некоторое событие.
2. Windows превращает это событие в соответствующее сообщение и помещает его в очередь сообщений приложения.
3. Приложение получает сообщение из очереди и помещает его в запись типа `TMsg`.
4. Приложение передает сообщение процедуре соответствующего окна приложения.
5. Процедура окна выполняет некоторые действия в ответ на сообщение.

Этапы 3 и 4 являются *циклом сообщений*, который фактически представляет собой "сердце" программы Windows. Именно этот цикл обеспечивает взаимодействие программы с внешним миром, получая сообщения из очереди и передавая их соответствующему окну приложения. Если очередь сообщения данного приложения будет исчерпана, система Windows обратится к другим приложениям и позволит им выполнить обработку сообщений в их очередях. Все описанные выше действия показаны на рис. 3.1.

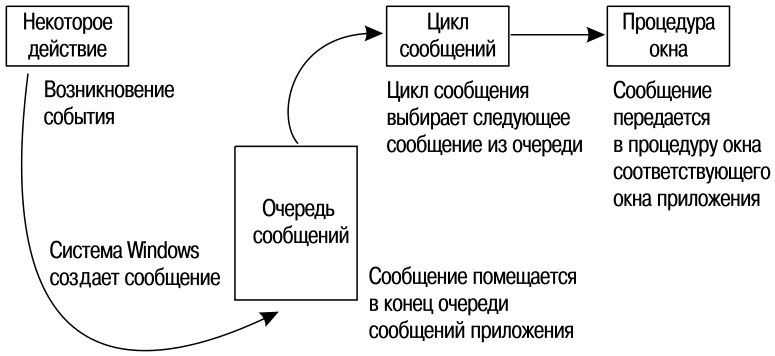

Рис. 3.1. *Схема системы сообщений Windows*

Система сообщений Delphi

Подпрограммы библиотеки VCL выполняют существенную часть обработки сообщений Windows в приложении. В частности, цикл сообщений встроен в модуль `Forms` библиотеки VCL, благодаря чему прикладному программисту не нужно беспокоиться о выборке сообщений из очереди и передаче их соответствующим процедурам окон. Кроме того, Delphi помещает информацию из записи типа `TMsg` в собственную запись типа `TMessage`, определение которой приведено ниже.

```
type
  TMessage = record
    Msg: Cardinal;
    case Integer of
      0: (
        WParam: Longint;
        LParam: Longint;
        Result: Longint);
      1: (
        WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;
```

Обратите внимание, что в записи `TMessage` содержится меньше информации, чем в исходной записи `TMsg`. Это связано с тем, что Delphi берет часть обработки сообщений Windows на себя, и в запись `TMessage` помещается только та часть информации, которая необходима для дальнейшей обработки.

Важно отметить, что в записи `TMessage` содержится поле `Result` (результат). Как уже говорилось, некоторые сообщения требуют возврата значения из процедуры окна после завершения их обработки. В Delphi такая задача решается совсем просто — достаточно поместить возвращаемое значение в поле `Result` записи `TMessage`. Более подробная информация по этой теме приведена далее в разделе “Возврат результата обработки сообщения”.

Специализированные записи

В дополнение к обычной записи типа TMessage в Delphi определен набор специализированных записей для всех типов сообщений Windows. Они предназначены для того, чтобы обеспечить программисту возможность работы со всей содержащейся в исходном сообщении Windows информацией без необходимости декодировать значения полей wParam и lParam. Определения всех типов этих записей находятся в модуле Messages. Ниже приведен пример записи, используемой для большинства типов сообщений Windows о событиях мыши.

```
type
  TWMMouse = packed record
    Msg: Cardinal;
    Keys: Longint;
    case Integer of
      0: (
        XPos: Smallint;
        YPos: Smallint);
      1: (
        Pos: TSmallPoint;
        Result: Longint);
  end;
```

Все типы записей для конкретных сообщений о событиях мыши (например WM_LBUTTONDOWN или WM_RBUTTONUP) определяются просто как записи типа TWMMouse, что и показано ниже.

```
TWMRButtonUp = TWMMouse;
TWMLButtonDown = TWMMouse;
```

НА ЗАМЕТКУ

Специализированные записи сообщений определены практически для всех стандартных сообщений Windows. Соглашение о присвоении имен требует, чтобы имя записи соответствовало имени сообщения с префиксом *T* в верблюжьей нотации (т.е. разделение слов в имени прописными буквами, а не символами подчеркивания). К примеру, запись для сообщения WM_SETFONT должна иметь имя TWMSetFont.

Следует отметить, что запись типа TMessage создается для всех типов сообщений Windows и в любых ситуациях. Но работать с такими записями не так удобно, как со специализированными.

Обработка сообщений

Обработка сообщений (handling) означает, что приложение тем или иным образом реагирует на получаемые от операционной системы сообщения. В стандартном приложении Windows обработка сообщений выполняется в процедурах окна. Но Delphi, частично обрабатывая сообщения, упрощает работу программиста, позволяя вместо одной процедуры для обработки всех типов сообщений создавать независимые процедуры для обработки сообщений каждого типа. Любая процедура обработки сообщений должна отвечать трем требованиям.

- Процедура должна быть методом класса.
- Процедура должна принимать по ссылке (определенный как var) один параметр типа TMessage или любого другого типа специализированного сообщения.
- Объявление процедуры должно содержать ключевое слово message, за которым должна следовать константа, задающая тип обрабатываемого сообщения.

Вот пример объявления процедуры, обрабатывающей сообщение WM_PAINT:

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

НА ЗАМЕТКУ

Соглашение по присвоению имен требует присваивать обработчику сообщений то же имя, что и имя обрабатываемого им сообщения, но без символа подчеркивания и с указанием первых знаков имени прописными буквами (верблюжья нотация).

В качестве примера напишем простую процедуру обработки сообщения WM_PAINT, которая вместо перерисовки окна будет подавать звуковой сигнал.

Создайте новый пустой проект. Добавьте в окне редактора заголовок функции WMPaint, поместив его в раздел private объект TForm1:

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

Теперь в раздел implementation модуля добавьте определение процедуры. В этом случае указание ключевого слова message не требуется. Не забудьте определить область видимости новой процедуры, указав с помощью точечного оператора (.) на принадлежность к методам класса TForm1.

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  Beep;
  inherited;
end;
```

Обратите внимание на использование в процедуре ключевого слова inherited (унаследованный), которое позволяет передать сообщение обработчику, принадлежащему базовому классу. В данном случае по ключевому слову inherited сообщение передается обработчику сообщения WM_PAINT класса TForm.

НА ЗАМЕТКУ

В отличие от вызова обычных унаследованных методов, в данном случае имя базового метода не указывается. Дело в том, что при обработке сообщений имя метода не имеет значения, так как Delphi отыскивает процедуру обработки сообщения по значению, задаваемому в интерфейсе класса ключевым словом message.

В листинге 3.1 приведен пример простой формы, в которой будет обрабатываться сообщение WM_PAINT. Подготовить данный проект очень просто — достаточно создать новый проект и добавить код процедуры WMPaint в описание объекта TForm.

Листинг 3.1. `GetMess` — пример обработки сообщения

```
unit GMMain;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
  inherited;
end;

end.
```

Каждый раз, получая сообщение `WM_PAINT`, приложение будет вызывать процедуру `WMPaint`. Она просто проинформирует об этом звуковым сигналом, издаваемым с помощью процедуры `MessageBeep()`, а затем передаст управление унаследованному обработчику данного сообщения.

Процедура `MessageBeep()` — отладчик для бедных

Упомянув процедуру `MessageBeep()`, позволим себе небольшое отступление от основной темы. Эта процедура является одним из самых простых и полезных элементов интерфейса API Win32. Она очень проста в использовании: достаточно лишь передать ей некоторую заранее определенную константу — и система Windows пошлет звуковой сигнал на динамик компьютера (или, при наличии звуковой платы, воспроизведет соответствующий файл `.WAV`). "Ну и что?" — спросите вы. Хоть это и не очевидно с первого взгляда, но процедура `MessageBeep()` очень удобна для отладки программ.

Если необходимо быстро, без привлечения отладчика и задания точек останова, уточнить, достигает ли программа в ходе своего выполнения некоторого места в ее коде, поместите туда вызов процедуры `MessageBeep()`. Эта процедура не требует передачи ей в качестве параметра дескриптора окна или некоторого другого ресурса Windows, поэтому ее можно вызвать практически из любого места программы. Если компьютер имеет звуковую плату, можно передавать этой процедуре несколько заранее определенных значений параметра (они описаны в интерактивной справочной сис-

теме Win32 API) и определять по воспроизводимой мелодии, что же именно произошло.
Если не хочется каждый раз писать такое длинное название процедуры, используйте процедуру `Beep()` из модуля `SysUtils`, которая представляет собой вызов процедуры `MessageBeep()` с параметром `0`.

Обработка сообщений — условие обязательное

В отличие от событий Delphi, обработка сообщений является обязательной и не может выполняться или не выполняться по усмотрению программиста. Обычно, если в программе объявлено о проведении собственной обработки сообщений, то система Windows ожидает выполнения некоторых связанных с ней предопределенных действий. Чаще всего компоненты VCL выполняют бо́льшую часть обработки своими встроенными средствами, для доступа к которым программисту достаточно лишь вызвать обработчик базового класса с помощью директивы `inherited`. Общая схема выглядит примерно так: собственно в обработчике сообщения должны выполняться лишь те действия, которые нужны для приложения, а для выполнения всех остальных действий, необходимых системе Windows, следует с помощью директивы `inherited` вызвать унаследованный метод.

НА ЗАМЕТКУ

Однако простого вызова унаследованного обработчика сообщения в некоторых случаях может оказаться недостаточно. При обработке сообщений Windows на подпрограмму их обработки в определенных случаях могут быть наложены дополнительные ограничения. Например, при обработке сообщения `WM_KILLFOCUS` передача фокуса ввода другому элементу управления вызовет аварийное прекращение работы приложения.

Для демонстрации важности вызова унаследованного обработчика попробуйте убрать директиву `inherited` из метода `WMPaint()` рассмотренного ранее примера. Процедура обработки при этом будет выглядеть так:

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
end;
```

Поскольку системе Windows никогда не предоставляется возможность выполнить основную обработку сообщения `WM_PAINT`, форма этого приложения никогда не будет перерисовываться. Фактически все может закончится тем, что в очереди скопится несколько сообщений `WM_PAINT`, которые, включив звуковой сигнал, будут ожидать до тех пор, пока очередь не очистится.

Впрочем, в некоторых случаях вызов унаследованного обработчика действительно может быть нежелателен. Например, этот подход можно использовать при обработке сообщения `WM_SYSCOMMAND` для запрета свертывания и развертывания окна.

Возврат результата обработки сообщения

При обработке определенных сообщений, Windows ожидает возврата некоторого результирующего значения. Классическим примером может служить сообщение WM_CTLCOLOR. От обработчика этого сообщения Windows ожидает получить дескриптор кисти (brush handle), с помощью которой должно быть прорисовано диалоговое окно или элемент управления. (Для своих компонентов Delphi предоставляет свойство Color, обеспечивающее выполнение упомянутых действий, так что пример имеет лишь иллюстративный характер.) Можно и самостоятельно вернуть требуемый дескриптор кисти из процедуры обработки сообщений, назначив соответствующее значение полю Result записи TMessage (или записи сообщения другого типа) после вызова унаследованного обработчика (inherited). Например, при обработке сообщения WM_CTLCOLOR необходимый тип кисти можно указать Windows следующим образом:

```
procedure TForm1.WMCtlColor(var Msg: TWMCtlColor);
var
  BrushHand: hBrush;
begin
  inherited;
  {Создать дескриптор кисти и назначить его переменной BrushHand}
  Msg.Result := BrushHand;
end;
```

Событие OnMessage класса TApplication

Другой метод обработки сообщений заключается в использовании события OnMessage класса TApplication. При назначении обработчика этому событию он будет вызываться всякий раз при получении сообщения из очереди и подготовке его к обработке. Обработчик события OnMessage всегда вызывается до того, как система Windows получит возможность обработать сообщение. Обработчик TApplication.OnMessage имеет тип TMessageEvent и объявляться со списком параметров, приведенным ниже.

```
procedure SomeObject.AppMessageHandler(var Msg: TMsg;
                                       var Handled: Boolean);
```

Все параметры сообщения передаются обработчику события OnMessage в параметре Msg. Этот параметр имеет тип TMsg (записи сообщения Windows), который был описан в настоящей главе ранее). Параметр Handled имеет тип Boolean и используется для указания того, обработано сообщение или нет.

Чтобы создать обработчик события OnMessage, можно воспользоваться компонентом TApplicationEvents, расположенном во вкладке Additional (Дополнительные) палитры компонентов. Ниже приведен пример такого обработчика события:

```
var
  NumMessages: Integer;

procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
                                           var Handled: Boolean);
begin
  Inc(NumMessages);
```

```
  Handled := False;
end;
```

Событие OnMessage обладает одним существенным ограничением, заключающемся в том, что оно перехватывает только те сообщения, которые выбираются из очереди, и *не перехватывает* те, которые передаются непосредственно процедурам окон приложения. С более подробной информацией о том, как обойти это ограничение, можно ознакомиться в предыдущем издании данной книги *Delphi 5 Руководство разработчика*, в главе 13 — "Дополнительный инструментарий разработчика".

СОВЕТ

Событие OnMessage перехватывает все сообщения, направляемые в адрес всех окон, относящихся к данному приложению. Обработчик этого события будет самой загруженной подпрограммой приложения, поскольку таких событий очень много — до тысяч в секунду. Поэтому избегайте выполнения в данной процедуре любых продолжительных операций, иначе работа всего приложения может существенно замедлиться. В частности, это одно из самых неподходящих мест для размещения точек останова при отладке.

Использование собственных типов сообщений

Аналогично тому, как система Windows посылает свои сообщения различным окнам приложения, в самом приложении также может появиться необходимость обмена сообщениями между его собственными окнами и элементами управления. Delphi предоставляет разработчику несколько способов осуществления обмена сообщениями в пределах приложения: метод Perform() (работающий независимо от API Windows), а также функции интерфейса API Win32 SendMessage() и PostMessage().

Метод Perform()

Этим методом обладают все потомки класса TControl, входящие в состав библиотеки VCL. Метод Perform() позволяет послать сообщение любой форме или элементу управления, заданному именем экземпляра требуемого объекта. Методу Perform() передается три параметра — само сообщение и соответствующие ему параметры lParam и wParam. Определение данного метода имеет такой вид:

```
function TControl.Perform(Msg: Cardinal;
                          wParam, lParam: Longint): Longint;
```

Чтобы послать сообщение форме или элементу управления, применяется следующий синтаксис:

```
RetVal := ControlName.Perform(MessageID, wParam, lParam);
```

При вызове функции Perform() управление вызывающей программе не возвратится до тех пор, пока сообщение не будет обработано. Метод Perform() упаковывает переданные ему параметры в запись типа TMessage, а затем вызывает метод Dispatch()

указанного объекта, чтобы передать сообщение, минуя систему передачи сообщений API Windows. Более подробная информация о методе `Dispatch()` приведена в этой главе далее.

Функции API SendMessage() и PostMessage()

Иногда возникает необходимость послать сообщение окну, для которого не существует экземпляра объекта Delphi. Например, когда требуется послать сообщение окну, созданному не Delphi, а другой системой, и вся информация, которая доступна программе, — это его дескриптор. Для этого используются две функции API Windows — `SendMessage()` и `PostMessage()`, — являющиеся практически идентичными, за исключением того, что первая из них, подобно методу `Perform()`, посылает сообщение непосредственно процедуре окна и ожидает окончания его обработки, а функция `PostMessage()` просто помещает сообщение в очередь сообщений Windows и немедленно возвращает управление вызвавшей ее программе, не дожидаясь результатов обработки.

Функции `SendMessage()` и `PostMessage()` объявлены следующим образом:

```
function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
                     lParam: LPARAM): LRESULT; stdcall;
function PostMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
                     lParam: LPARAM): BOOL; stdcall;
```

где

- `hWnd` — дескриптор окна получателя сообщения;
- `Msg` — идентификатор сообщения;
- `wParam` — 32 бита дополнительной информации сообщения;
- `lParam` — еще 32 бита дополнительной информации сообщения.

НА ЗАМЕТКУ

Хотя функции `SendMessage()` и `PostMessage()` очень похожи, они возвращают различные по своему смыслу значения. Функция `SendMessage()` возвращает значение, полученное в результате обработки сообщения, тогда как функция `PostMessage()` возвращает значение типа `Boolean`, указывающее, удалось ли поместить сообщение в очередь сообщений соответствующего окна или нет. Иными словами, функция `SendMessage()` является синхронной "операцией", а `PostMessage()` — асинхронной.

Нестандартные сообщения

До сих пор обсуждались обычные сообщения Windows (начинающиеся префиксом `WM_XXX`). Теперь обсудим две другие немаловажные категории: уведомляющие сообщения и пользовательские сообщения.

Уведомляющие сообщения

Уведомляющие сообщения (notification messages), или *уведомления* (notifications), представляют собой сообщения, посылаемые родительскому окну в том случае, когда в од-

ном из его дочерних элементов управления происходит нечто, заслуживающее внимания родителя. Эти сообщения рассылаются только стандартными элементами управления Windows (кнопки, списки, раскрывающиеся списки, поля редактирования) и общими элементами управления Windows (дерево объектов, список объектов и т.п.). Щелчок или двойной щелчок на элементе управления, выбор текста в поле редактирования или перемещение ползунка полосы прокрутки — вот примеры событий, посылающих уведомляющие сообщения.

Обработка уведомлений осуществляется с помощью соответствующих процедур обработки, принадлежащих той форме, в которой содержится данный элемент управления. В табл. 3.2 приведен список уведомляющих сообщений для стандартных элементов управления Win32.

Таблица 3.2. Уведомления стандартных элементов управления

Уведомление	*Смысл*
Уведомления кнопки	
BN_CLICKED	Пользователь щелкнул на кнопке
BN_DISABLE	Кнопка переведена в неактивное состояние
BN_DOUBLECLICKED	Пользователь дважды щелкнул на кнопке
BN_HILITE	Пользователь выделил кнопку
BN_PAINT	Кнопка должна быть перерисована
BN_UNHILITE	Выделение кнопки должно быть отменено
Уведомления раскрывающегося списка	
CBN_CLOSEUP	Раскрытый список был закрыт пользователем
CBN_DBLCLK	Пользователь дважды щелкнул на строке
CBN_DROPDOWN	Список был раскрыт
CBN_EDITCHANGE	Пользователь изменил текст в поле ввода
CBN_EDITUPDATE	Требуется вывести измененный текст
CBN_ERRSPACE	Элементу управления не хватает памяти
CBN_KILLFOCUS	Список потерял фокус ввода
CBN_SELCHANGE	Выбран новый элемент списка
CBN_SELENDCANCEL	Пользователь отказался от сделанного им выбора
CBN_SELENDOK	Выбор пользователя корректен
CBN_SETFOCUS	Список получил фокус ввода

Окончание табл. 3.2.

Уведомление	*Смысл*
Уведомления поля ввода	
EN_CHANGE	Требуется обновление после внесения изменений
EN_ERRSPACE	Элементу управления не хватает памяти
EN_HSCROLL	Пользователь щелкнул на горизонтальной полосе прокрутки
EN_KILLFOCUS	Поле ввода потеряло фокус ввода
EN_MAXTEXT	Введенный текст был обрезан
EN_SETFOCUS	Поле ввода получило фокус ввода
EN_UPDATE	Требуется отображение введенного текста
EN_VSCROLL	Пользователь щелкнул на вертикальной полосе прокрутки
Уведомления списков	
LBN_DBLCLK	Пользователь дважды щелкнул на строке
LBN_ERRSPACE	Элементу управления не хватает памяти
LBN_KILLFOCUS	Список потерял фокус ввода
LBN_SELCANCEL	Отмена выделения
LBN_SELCHANGE	Изменение выделения
LBN_SETFOCUS	Список получил фокус ввода

Внутренние сообщения компонентов VCL

Компоненты библиотеки VCL используют обширный набор собственных внутренних сообщений и уведомлений. Хотя в создаваемых приложениях Delphi вряд ли потребуется работать с этими сообщениями непосредственно, тем не менее разработчику компонентов будет весьма полезно познакомиться с ними. Имена этих сообщений всегда начинаются с префикса CM_ (*сообщения компонентов* — component messages) или CN_ (*уведомления компонентов* — component notification). Они используются компонентами библиотеки VCL для управления состоянием их внутренних свойств — например для передачи фокуса, установки цвета, изменения состояния видимости, выдачи требования перерисовки окна, поддержки операций перетаскивания и т.д. Полный список этих сообщений можно найти в разделе интерактивной справочной системы Delphi, посвященном созданию пользовательских компонентов.

Ординарная задача — как обнаружить, что курсор мыши переместился в область элемента управления или покинул ее? Это можно узнать перехватив и обработав специальные сообщения: CM_MOUSEENTER и CM_MOUSELEAVE. Рассмотрим следующий компонент:

```
TSpecialPanel = class(TPanel)
  protected
  procedure CMMouseEnter(var
                         Msg: TMessage); message CM_MOUSEENTER;
  procedure CMMouseLeave(var
                         Msg: TMessage); message CM_MOUSELEAVE;
  end;
  .
  .
  .
procedure TSpecialPanel.CMMouseEnter(var Msg: TMessage);
begin
  inherited;
  Color := clWhite;
end;

procedure TSpecialPanel.CMMouseLeave(var Msg: TMessage);
begin
  inherited;
  Color := clBtnFace;
end;
```

Этот компонент, обрабатывающий специальные сообщения, изменяет цвет панели на белый (`clWhite`), если курсор мыши находится над ее поверхностью, а затем, когда он покидает панель компонента, возвращает ему исходный цвет (`clBtnFace`).

Пользовательские сообщения

При разработке приложений может возникнуть ситуация, при которой приложению потребуется послать сообщение либо самому себе, либо другому пользовательскому приложению. У некоторых предыдущее утверждение вызовет недоумение: зачем приложению посылать сообщение самому себе, если можно просто вызвать соответствующую процедуру? Это хороший вопрос, и на него существует несколько ответов. Во-первых, использование сообщений представляет собой механизм поддержки реального полиморфизма, поскольку не требует наличия каких-либо знаний о типе объекта получающего сообщение. Таким образом, технология работы с сообщениями имеет ту же мощь, что и механизм виртуальных методов, но обладает существенно большей гибкостью. Во-вторых, сообщения допускают необязательную обработку — если объект-получатель не обработает поступившее сообщение, то ничего страшного не произойдет. И, в-третьих, сообщения позволяют осуществлять широковещательное обращение к нескольким получателям и организовывать параллельное прослушивание, что достаточно трудно реализовать с помощью механизма вызова процедур.

Использование сообщений внутри приложения

Заставить приложение послать сообщение самому себе очень просто — достаточно либо воспользоваться функциями интерфейса API Win32 `SendMessage()` или `PostMessage()`, либо методом `Perform()`. Сообщение должно обладать идентификатором в диапазоне от `WM_USER+100` до `$7FFFF` (этот диапазон Windows резервирует для сообщений пользователя). Например:

```
const
  SX_MYMESSAGE = WM_USER + 100;

begin
  SomeForm.Perform(SX_MYMESSAGE, 0, 0);
  { или }
  SendMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
  { или }
  PostMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
  .
  .
  .
end;
```

Затем, для перехвата этого сообщения, создайте обычную процедуру-обработчик, выполняющую в форме необходимые действия:

```
TForm1 = class(TForm)
 .
 .
 .
private
  procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
end;

procedure TForm1.SXMyMessage(var Msg: TMessage);
begin
  MessageDlg('She turned me into a newt!',
             mtInformation, [mbOk], 0);
end;
```

Как видно из примера, различия в обработке собственного сообщения и стандартного сообщения Windows невелики. Они заключаются в использовании идентификаторов в диапазоне от `WM_USER+100` и выше, а также в присвоении каждому сообщению имени, которое каким-то образом будет отражать его смысл.

СОВЕТ

Никогда не посылайте сообщений, значение `WM_USER` которых превосходит `$7FFF`, если не уверены абсолютно точно, что получатель способен правильно обработать это сообщение. Поскольку каждое окно может независимо выбирать используемые им значения, весьма вероятно появление трудноуловимых ошибок, если только заранее не составить таблицы идентификаторов сообщений, с которыми будут работать все отправители и получатели сообщений.

Обмен сообщениями между приложениями

При необходимости организовать обмен сообщениями между двумя или более приложениями в них следует использовать функцию API `RegisterWindowMessage()`. Этот метод гарантирует, что для заданного типа сообщений каждое приложение будет использовать один и тот же *номер сообщения* (message number).

Функция `RegisterWindowMessage()` принимает в качестве параметра строку с завершающим нулевым символом и возвращает для нового сообщения идентификатор в

диапазоне $C000 – $FFFF. Это означает, что вызова данной функции с одной и той же строкой в качестве параметра в любом приложении будет достаточно, чтобы гарантировать одинаковые номера сообщений во всех приложениях, принимающих участие в обмене. Еще одно преимущество такой функции состоит в том, что система гарантирует уникальность идентификатора, назначенного любой заданной строке. Это позволяет посылать широковещательные сообщения всем существующим в системе окнам, не опасаясь нежелательных побочных эффектов. Недостатком данного метода является некоторое усложнение обработки подобных сообщений. Суть заключается в том, что идентификатор сообщения становится известным только при работе приложения, поэтому использование стандартных процедур обработки сообщений оказывается невозможным. Для работы с подобными сообщениями потребуется переопределить стандартные методы WndProc() или DefaultHandler() элементов управления либо соответствующие процедуры класса окна.

НА ЗАМЕТКУ

Число, возвращаемое функцией RegisterWindowMessage(), создается динамически и может принимать различные значения в разных сессиях Windows, а значит, не может быть определено до момента выполнения программы.

Широковещательные сообщения

Любой класс, производный от класса TWinControl, позволяет с помощью метода Broadcast() послать *широковещательное сообщение* (broadcast message) любому элементу управления, владельцем которого он является. Эта технология используется в тех случаях, когда требуется послать одно и то же сообщение группе компонентов. Например, чтобы послать пользовательское сообщение по имени um_Foo всем элементам управления, принадлежащим объекту Panel1, можно воспользоваться следующим кодом:

```
var
  M: TMessage;
begin
  with M do begin
    Message := UM_FOO;
    wParam := 0;
    lParam := 0;
    Result := 0;
  end;
  Panel1.Broadcast(M);
end;
```

Анатомия системы сообщений: библиотека VCL

Система обмена сообщениями не ограничивается библиотекой VCL, которая обрабатывает сообщения с помощью директивы message. После того как Windows отправит сообщение, оно пройдет несколько последовательных этапов обработки, прежде чем достигнет процедуры обработки сообщений создаваемого приложения. (Кроме того, оно может пройти еще несколько этапов обработки и после выполнения

такой процедуры.) В принципе, сообщение остается доступным на протяжении всего этого долгого пути.

Любое сообщение Windows, посланное без требования получения результатов обработки, первоначально обрабатывается методом `Application.ProcessMessage()`, в котором реализован главный цикл обработки сообщений средствами библиотеки VCL. Следующий этап состоит в передаче сообщения обработчику события `Application.OnMessage`. Событие `OnMessage` генерируется при выборке сообщения из очереди приложения в методе `ProcessMessage()`. Поскольку сообщения, посылаемые без требования получения результатов обработки, не помещаются в очередь, событие `OnMessage` для них не генерируется.

Для обработки сообщений, посылаемых с требованием предоставления результатов их обработки, неявно вызывается функция API Win32 `DispatchMessage()`, которая, в свою очередь, передает каждое сообщение функции `StdWndProc()`. Для отправляемых сообщений эта функция вызывается непосредственно системой Win32. Функция `StdWndProc()` представляет собой написанную на ассемблере функцию низкого уровня, которая принимает сообщение от Windows и переправляет его назначенному объекту.

Метод объекта, который получает сообщение, называется `MainWndProc()`. Начиная с этого момента в программе можно выполнять любую специальную обработку сообщения, какая только может потребоваться. Обычно в обработку сообщения на данном этапе вмешиваются только для того, чтобы не допустить стандартной обработки сообщения средствами библиотеки VCL.

По завершении работы метода `MainWndProc()` сообщение передается методу `WndProc()` объекта, а затем поступает в распоряжение механизма диспетчеризации сообщений. Этот механизм, функционирующий в рамках метода `Dispatch()`, переправляет сообщение дальше, некоторой конкретной процедуре обработки сообщений, определенной программистом или уже имеющейся в составе средств библиотеки VCL.

Наконец, сообщение достигает определенной в программе специализированной процедуры обработки сообщений данного типа. После выполнения всех действий, предусмотренных пользовательской процедурой обработки и всеми унаследованными процедурами, которые вызываются с помощью ключевого слова `inherited`, сообщение передается методу `DefaultHandler()`. Этот метод осуществляет любые завершающие действия по обработке сообщения и передает его функции API Win32 `DefWindowProc()` (или другой стандартной процедуре обработки, например `DefMDIProc()`) для стандартной обработки средствами Windows. Общая схема механизма обработки сообщений средствами библиотеки VCL представлена на рис. 3.2.

НА ЗАМЕТКУ

При обработке сообщений всегда следует вызывать унаследованный метод, за исключением тех случаев, когда есть абсолютная уверенность в том, что стандартная обработка сообщения не требуется.

СОВЕТ

Поскольку все необработанные сообщения доставляются обработчику `DefaultHandler()`, этот метод представляет собой самое подходящее место для выполнения обработки сообщений из других приложений, идентификаторы которых назначены с помощью процедуры `RegisterWindowMessage()`.

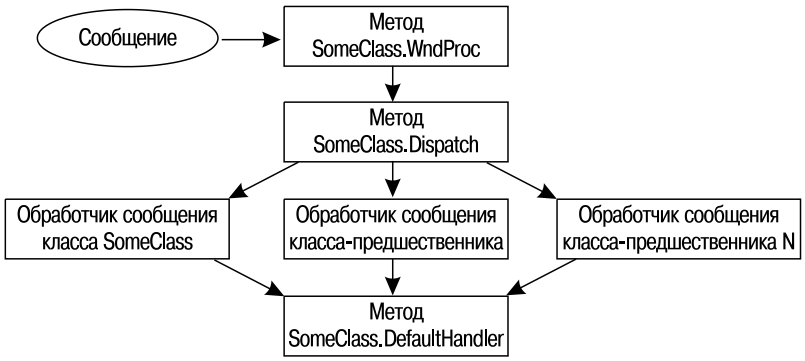

Рис. 3.2. *Схема механизма обработки сообщений средствами библиотеки VCL*

Чтобы лучше понять схему обработки сообщений средствами библиотеки VCL, рассмотрим небольшую программу, в которой обработка сообщений выполняется с помощью методов `Application.OnMessage` и `WndProc()`, на стадии процедуры стандартной обработки сообщений (`DefaultHandler()`). Проект называется `CatchIt`. Общий вид его главной формы показан на рис. 3.3.

Рис. 3.3. *Главная форма приложения* `CatchIt`

Ниже приведен код обработчиков события `OnClick` для объектов кнопок `PostMessButton` и `SendMessButton`. В первом из них для отправки пользовательского сообщения в адрес формы приложения используется функция `PostMessage()`, а во втором для этой же цели применяется функция `SendMessage()`. Для того чтобы в приложении можно было отличить сообщения, посланные с помощью функции `PostMessage()`, от сообщений, посланных с помощью функции `SendMessage()`, нужно в первом случае в поле `wParam` сообщения поместить значение `1`, а во втором — `0`.

```
procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ Отправить (post) сообщение форме }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;

procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ Отправить (send) сообщение форме }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0);
end;
```

Это приложение позволяет "проглотить" сообщение в обработчике события OnMessage, методе WndProc(), методе специализированной обработки сообщения или в методе DefaultHandler(). В любом случае унаследованный обработчик сообщения (которому сообщение передается с помощью директивы inherited) выполнен не будет, а следовательно, цикл прохождения сообщением всей цепочки механизма обработки сообщений средствами библиотеки VCL останется незавершенным. В листинге 3.2 приведен полный код главного модуля приложения, иллюстрирующего процессы обработки сообщений в приложениях Delphi.

Листинг 3.2. Исходный код CIMain.PAS

```
unit CIMain;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls, Menus;

const
  SX_MYMESSAGE = WM_USER;     // Идентификатор пользовательского
                              // сообщения
  MessString = '%s message now in %s.';  // Срока, оповещающая
                              // пользователя о сообщении
type
  TMainForm = class(TForm)
    GroupBox1: TGroupBox;
    PostMessButton: TButton;
    WndProcCB: TCheckBox;
    MessProcCB: TCheckBox;
    DefHandCB: TCheckBox;
    SendMessButton: TButton;
    AppMsgCB: TCheckBox;
    EatMsgCB: TCheckBox;
    EatMsgGB: TGroupBox;
    OnMsgRB: TRadioButton;
    WndProcRB: TRadioButton;
    MsgProcRB: TRadioButton;
    DefHandlerRB: TRadioButton;
    procedure PostMessButtonClick(Sender: TObject);
    procedure SendMessButtonClick(Sender: TObject);
    procedure EatMsgCBClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure AppMsgCBClick(Sender: TObject);
  private
    { Обработка сообщения на уровне приложения }
    procedure OnAppMessage(var Msg: TMsg; var Handled: Boolean);
    { Обработка сообщения на уровне процедуры WndProc }
    procedure WndProc(var Msg: TMessage); override;
    { Обработка сообщения после диспетчеризации }
    procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
    { Стандартный обработчик сообщения }
```

```
    procedure DefaultHandler(var Msg); override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
// Строки для указания способа передачи сообщения (sent или post)
  SendPostStrings: array[0..1] of String = ('Sent', 'Posted');

procedure TMainForm.FormCreate(Sender: TObject);
{ Обработчик события OnCreate главной формы }
begin
  // Определение метода OnAppMessage как обработчика
  // события OnMessage
  Application.OnMessage := OnAppMessage;
  // Использование свойства Tag объекта флажка для сохранения
  // ссылки на связанные с ним переключатели
  AppMsgCB.Tag := Longint(OnMsgRB);
  WndProcCB.Tag := Longint(WndProcRB);
  MessProcCB.Tag := Longint(MsgProcRB);
  DefHandCB.Tag := Longint(DefHandlerRB);
  // Использование свойства Tag переключателей для сохранения
  // ссылки на связанный с ними флажок
  OnMsgRB.Tag := Longint(AppMsgCB);
  WndProcRB.Tag := Longint(WndProcCB);
  MsgProcRB.Tag := Longint(MessProcCB);
  DefHandlerRB.Tag := Longint(DefHandCB);
end;

procedure TMainForm.OnAppMessage(var Msg: TMsg;
                                 var Handled: Boolean);
{ Обработчик события OnMessage на уровне приложения }
begin
  // проверить, не является ли сообщение пользовательским
  if Msg.Message = SX_MYMESSAGE then begin
    if AppMsgCB.Checked then begin
      // Известить пользователя о поступлении сообщения и
      // установить соответствующий флажок Handled
      ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
                  'Application.OnMessage']));
      Handled := OnMsgRB.Checked;
    end;
  end;
end;

procedure TMainForm.WndProc(var Msg: TMessage);
{ Процедура формы WndProc }
var
```

```
  CallInherited: Boolean;
begin
  CallInherited := True;   // Возможно, придется вызвать
                           // унаследованный обработчик
  if Msg.Msg = SX_MYMESSAGE then   // Проверить, не наше ли это
  begin                            // сообщение
    if WndProcCB.Checked then      // Если флажок WndProcCB
    begin                          // установлен...
      // Известить пользователя о сообщении.
      ShowMessage(Format(MessString,
                  [SendPostStrings[Msg.WParam], 'WndProc']));
      // Вызвать унаследованный обработчик, если не удастся
      // проглотить сообщение.
      CallInherited := not WndProcRB.Checked;
    end;
  end;
  if CallInherited then inherited WndProc(Msg);
end;

procedure TMainForm.SXMyMessage(var Msg: TMessage);
{ Процедура обработки пользовательского сообщения }
var
  CallInherited: Boolean;
begin
  CallInherited := True;           // Возможно, придется вызвать
                                   // унаследованный обработчик
  if MessProcCB.Checked then       // Проверить, не наше ли это
  begin                            // сообщение
    // Известить пользователя о сообщении.
    ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
                'Message Procedure']));
    // Вызвать унаследованный обработчик, если не удастся
    // проглотить сообщение.
    CallInherited := not MsgProcRB.Checked;
  end;
  if CallInherited then Inherited;
end;

procedure TMainForm.DefaultHandler(var Msg);
{ Обработчик сообщения по умолчанию }
var
  CallInherited: Boolean;
begin
  CallInherited := True;             // Возможно, придется вызвать
                                     // унаследованный обработчик
  if TMessage(Msg).Msg = SX_MYMESSAGE then // Проверить, не
    begin                 // пользовательское ли это сообщение
    if DefHandCB.Checked then        // Если флажок DefHandCB
    begin                            // установлен...
      // Известить пользователя о сообщении.
      ShowMessage(Format(MessString,
                  [SendPostStrings[TMessage(Msg).WParam],
                  'DefaultHandler']));
```

```
      // Вызвать унаследованный обработчик, если не удается
      // проглотить сообщение.
      CallInherited := not DefHandlerRB.Checked;
    end;
  end;
  if CallInherited then inherited DefaultHandler(Msg);
end;

procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ Отправить форме сообщение методом post }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;

procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ Отправить форме сообщение методом send }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0); // Отправить сообщение
end;

procedure TMainForm.AppMsgCBClick(Sender: TObject);
{ Установить/снять переключатели в зависимости от флажков }
begin
  if EatMsgCB.Checked then begin
    with TRadioButton((Sender as TCheckBox).Tag) do begin
      Enabled := TCheckbox(Sender).Checked;
      if not Enabled then Checked := False;
    end;
  end;
end;

procedure TMainForm.EatMsgCBClick(Sender: TObject);
{ Установить/снять флажки в зависимости от переключателей }
var
  i: Integer;
  DoEnable, EatEnabled: Boolean;
begin
  // Получить состояние флажков
  EatEnabled := EatMsgCB.Checked;
  // Итерация по дочерним элементам управления объекта GroupBox
  // для установки состояния флажков и переключателей
  for i := 0 to EatMsgGB.ControlCount - 1 do
    with EatMsgGB.Controls[i] as TRadioButton do begin
      DoEnable := EatEnabled;
      if DoEnable then DoEnable := TCheckbox(Tag).Checked;
      if not DoEnable then Checked := False;
      Enabled := DoEnable;
    end;
end;

end.
```

СОВЕТ

Обычно для передачи сообщения унаследованному обработчику достаточно ключевого слова `inherited`, но в обработчиках, использующихся в процедурах `WndProc()` и `DefaultHandler()`, эта методика не работает. Поэтому приходится явно указывать имя унаследованной процедуры или функции, в данном случае `inherited WndProc(Msg);`

Обратите внимание: процедура `DefaultHandler()` несколько отличается от других — она принимает один *нетипизированный* `var`-параметр. Все дело в том, что первое слово в значении параметра процедуры `DefaultHandler()` будет идентификатором сообщения, остальная часть сообщения процедурой игнорируется. Поэтому из сообщения можно легко извлечь его идентификатор с помощью явного приведения параметра к типу `TMessage`.

Взаимосвязь сообщений и событий

Теперь уже сказано достаточно, чтобы понять, что система событий Delphi представляет собой интерфейс для взаимодействия с сообщениями Windows, по крайней мере — с некоторой их частью. Многие события компонентов библиотеки VCL непосредственно связаны с сообщениями Windows типа `WM_XXX`. В табл. 3.3 приведены основные события компонентов библиотеки VCL и соответствующие им сообщения Windows.

Таблица 3.3. События VCL и соответствующие им сообщения Windows

Событие VCL	*Сообщение Windows*
`OnActivate`	`wm_Activate`
`OnClick`	`wm_XButtonDown`
`OnCreate`	`wm_Create`
`OnDblClick`	`wm_XButtonDblClick`
`OnKeyDown`	`wm_KeyDown`
`OnKeyPress`	`wm_Char`
`OnKeyUp`	`wm_KeyUp`
`OnPaint`	`WM_PAINT`
`OnResize`	`wm_Size`
`OnTimer`	`wm_Timer`

Табл. 3.3 можно рассматривать как краткий справочник, полезный при поиске событий, непосредственно соответствующих тем или иным сообщениям Windows.

СОВЕТ

Никогда не пишите собственный обработчик сообщения, если вместо него можно использовать предопределенное событие. В связи с необязательным характером событий проблем при их обработке будет меньше, чем при обработке сообщений.

Резюме

В этой главе рассматривалась работа системы обмена сообщений Win32 и инкапсуляции обработки этих сообщений средствами библиотеки VCL. Хотя система обработки событий Delphi обладает достаточно широкими возможностями, знание механизмов обмена сообщениями Windows является важным требованием к любому программисту, работающему в среде Win32.

ЧАСТЬ

II

Профессиональное программирование

В ЭТОЙ ЧАСТИ...

ГЛАВА

4

Переносимость кода

В ЭТОЙ ГЛАВЕ...

Эта глава предназначена тем, кто переходит к Delphi 6 с одной из предыдущих версий. Первый ее раздел посвящен вопросам, связанным с переходом на Delphi 6 от предыдущих версий. Второй раздел содержит полезные советы по обеспечению совместимости между Delphi (платформа Win32) и Kylix (платформа Linux). В остальной части главы речь идет о незначительных различиях между существующими версиями Delphi и о том, каких их учитывать при написании переносимого кода или при переходе с одной версии на другую. Несмотря на то, что *Borland* сделала все возможное для совместимости версий, вполне понятно, что для обеспечения прогресса должны иметь место некоторые изменения. Поэтому, для успешной компиляции и правильной работы в последней версии Delphi, любое приложение, переносимое из прежних версий, должно претерпеть определенные изменения.

Общая совместимость

Общая совместимость между различными версиями Delphi, C++Builder и Kylix связана с большим количеством проблем и обстоятельств. Изучив эти подробности и особенности компиляторов, можно научиться создавать и поддерживать такой код, который с небольшими доработками может быть перенесен не только с одной версии компилятора на другую, но и на иную платформу.

Определение версии

Несмотря на то что большинство программ Delphi будет компилироваться во всех версиях компилятора, в некоторых случаях отличия языка или подпрограмм библиотеки VCL вынуждают вносить незначительные изменения в код, предназначенный для выполнения определенных задач, с целью приведения его в соответствие конкретной версии данного программного продукта. Время от времени один и тот же код приходится компилировать на различных версиях Delphi. Для этого компилятор каждой версии содержит идентификатор версии `VER`*xxx*, который позволяет проверить версию кода. Поскольку Borland C++ Builder и Kylix комплектуются новыми версиями компилятора Delphi, то и эта среда разработки содержит упомянутый идентификатор версии. Идентификаторы версий различных компиляторов Delphi представлены в табл. 4.1.

Таблица 4.1. Идентификаторы версий компиляторов *Borland*

Продукт	*Идентификатор версии*
Delphi 1	`VER80`
Delphi 2	`VER90`
C++Builder 1	`VER95`
Delphi 3	`VER100`
C++Builder 3	`VER110`
Delphi 4	`VER120`
C++Builder 4	`VER120`
Delphi 5	`VER130`
C++Builder 5	`VER130`
Kylix 1	`VER140`
Delphi 6	`VER140`

Используя идентификаторы, можно разместить исходный код, предназначенный для разных версий компилятора, в отдельных блоках. Каждый из блоков будет выполняться только тем компилятором, для которого он предназначен:

```
{$IFDEF VER80}
    Код для Delphi 1 находится здесь
{$ENDIF}
{$IFDEF VER90}
    Код для Delphi 2 находится здесь
{$ENDIF}
{$IFDEF VER95}
    Код для C++Builder 1 находится здесь
{$ENDIF}
{$IFDEF VER100}
    Код для Delphi 3 находится здесь
{$ENDIF}
{$IFDEF VER110}
    Код для C++Builder 3 находится здесь
{$ENDIF}
{$IFDEF VER120}
    Код для Delphi 4 и C++Builder 4 находится здесь
{$ENDIF}
{$IFDEF VER130}
    Код для Delphi и C++Builder 5 находится здесь
{$ENDIF}
{$IFDEF VER140}
    Код для Delphi 6 и Kylix находится здесь
{$ENDIF}
```

НА ЗАМЕТКУ

Некоторых может удивить тот факт, что компилятор Delphi 1 считается версией 8, Delphi 2 — версией 9 и т.д. Это связано с тем, что Delphi 1 укомплектован 8-й версией компилятора Borland Pascal. Предыдущей была версия компилятора Turbo Pascal 7, а в Delphi 1 эту линию продуктов продолжили.

Модули, компоненты и пакеты

Формат откомпилированных модулей Delphi (файлы `.dcu`) имеет тенденцию изменяться от версии к версии компилятора. Это означает, что, если необходимо применять один и тот же модуль в нескольких версиях Delphi, следует иметь исходный код каждого модуля, используемого приложением. В этом случае невозможно воспользоваться компонентами (неважно, кем они изготовлены), если не будет в наличии их исходного кода. При отсутствии исходного кода какого-либо компонента, разработанного независимым производителем, обратитесь к распространителю за версией компонента, соответствующей вашей версии Delphi.

НА ЗАМЕТКУ

Вопрос соответствия версий компилятора и файлов модулей не нов. Если вы распространяете или покупаете компоненты без исходного кода, то должны понимать, что эти файлы предназначены для определенной версии компилятора и, вероятно, потребуют пересмотра при появлении последующих версий.
Более того, вопрос версий модулей DCU не обязательно связан только с компилятором. Даже если в новой версии компилятор останется прежним, изменения и усовершенствования ядра библиотеки VCL, вполне вероятно, могут потребовать перекомпиляции исходного кода этих модулей.

Пакеты (packages) впервые появились в Delphi 3 как реализация идеи хранения нескольких модулей в одном бинарном файле. Начиная с Delphi 3 библиотека компонентов стала больше походить на коллекцию пакетов, чем на единый монолитный файл динамически компонуемой библиотеки (DLL). Как и модули, пакеты не совместимы с различными версиями продукта, а следовательно, потребуется выполнить их повторное построение для использования в Delphi 5. За обновленными версиями используемых пакетов нужно обратиться к их производителям.

Проблемы IDE

Различия в работе IDE, вероятно, будут первой проблемой, которую придется преодолеть при переносе приложения. Основные из них перечислены ниже.

- Символьный файл отладчика Delphi (RSM) не всегда совместим с форматом данного файла в других версиях. Следствием этого факта может стать появление сообщения "`Error reading symbol file`" (Ошибка чтения символьного файла). Чтобы выйти из данной ситуации, следует перекомпилировать приложение.
- В Delphi 5 файлы форм по умолчанию сохраняются в текстовом виде. Если необходимо обеспечить совместимость с файлами DFM предыдущих версий, то придется установить режим сохранения файлов форм в двоичном формате. Для этого следует сбросить флажок New Form As Text во вкладке Preferences диалогового окна Environment Options.
- Генератор кода, используемый для создания библиотек типов при импортировании компонентов, был изменен. Помимо прочих незначительных улучшений, в новый генератор кода внедрили поддержку средств отображения символических имен. Появилась возможность настройки библиотек типов на использование символических имен, принятых в языке Pascal. Для этого достаточно отредактировать файл `tlibimp.sym`. Подробное описание этой функции можно найти в статье "Mapping Symbol Names in the Type Library" интерактивной справочной системы Delphi.

Совместимость Delphi и Kylix

Реализуя приложение с высокой степенью совместимости между Delphi и Kylix, необходимо учитывать тот факт, что технология VCL применима только для Windows. Если необходимо создать межплатформенное приложение или компонент, то придется

воспользоваться *библиотекой компонентов для X-платформы* (CLX – Component Library for X-platform), которая в настоящий момент поддерживает и Delphi 6, и Kylix. Более подробная информация о платформе CLX приведена в главах 10, "Архитектура компонентов: VCL и CLX" , и 13, "Разработка компонентов CLX". В библиотеку CLX входят четыре основных компонента.

- BaseCLX содержит основные системные компоненты.
- DataCLX представляет собой технологию, аналогичную dbExpress. Она обеспечивает простой и эффективный доступ к данным и управление ими. Более подробная информация о dbExpress приведена в главе 8, "Применение dbExpress при разработке баз данных".
- NetCLX включает в себя компоненты и мастера для создания сетевых приложений с архитектурой клиент/сервер. Возможно, NetCLX представляет собой наилучшую и очень надежную среду разработки Web-приложений, в которую входит и технология WebBroker, появившаяся в предыдущей версии. NetCLX применима и для Linux, и для Windows, как для клиентов, так и для серверов.
- VisualCLX обеспечивает возможность создания межплатформенного графического пользовательского интерфейса (GUI). Внешне VisualCLX очень похож на VCL, но на самом деле построен на базе библиотеки Qt производства *Troll Tech* (`http://www.trolltech.com`), в отличие от VCL, построенной на базе API Win32. Qt представляет собой межплатформенную среду разработки, которая позволяет создавать GUI для нескольких платформ, в том числе Windows и Linux.

Создав новое приложение CLX (меню File, пункты New, CLX Application), в разделе `uses` модуля главной формы можно увидеть имена нескольких подключаемых модулей, имена которых начинаются буквой `Q`, например `QGraphics`, `QControls`, `QForms` и так далее. Эти модули похожи по содержанию и возможностям на аналогичные по именам модули VCL, но предназначены для приложений, независимых от платформы.

НА ЗАМЕТКУ

Хотя современные версии CLX поддерживают только Windows и Kylix, они разработаны так, чтобы в будущем их можно было легко модернизировать для поддержки других платформ. Например, Qt поддерживает около дюжины различных платформ.

Только не в Linux

Следует отметить: технологии, специфические для Windows, нельзя перенести на платформу Linux. Это означает, что таким прекрасным технологиям, как ADO, COM/COM+, BDE, MAPI, и многим другим нет места в межплатформенном приложении. Следовательно, придется избегать применения таких модулей, как `Windows`, `ComObj`, `ComServ`, `ActiveX` и `AdoDb`, а также таких специфических для платформы WIN32 функций API, как `RaiseLastWin32Error()`, `Win32Check()` и так далее. Кроме того, существует много технологий Delphi 6, которые не доступны в Kylix 1, но будут, вероятно, присутствовать в будущих версиях Kylix. К ним относятся DataSnap, BizSnap (SOAP) и WebSnap.

Особенности языка и компилятора

Хоть оба компилятора, Delphi и Kylix, рассчитаны на архитектуру процессора x86, между ними существует множество принципиальных различий, о которых необходимо помнить при создании переносимых приложений.

Директива условной компиляции LINUX

Компилятор Kylix отрабатывает участки кода, помеченные в *директиве условной компиляции* (conditional define) `$IFDEF` идентификатором `LINUX`, а Delphi — участки кода, помеченные идентификатором `MSWINDOWS` или `WIN32`. Таким образом, директива условной компиляции (`$IFDEF`) позволяет объединить в одном файле исходного кода участки, предназначенные для обработки на разных платформах различными компиляторами. Такой код может иметь следующий вид:

```
{$IFDEF LINUX}
  // Здесь расположен код, предназначенный для Linux
{$ENDIF}
{$IFDEF MSWINDOWS}
  // Здесь расположен код, предназначенный для Windows
{$ENDIF}
```

Так как идентификатор `LINUX` определен только в Kylix, его компилятор отработает блок `{$IFDEF LINUX}..{$ENDIF}`, но не сможет отработать блок `{$IFDEF MSWINDOWS}..{$ENDIF}`, поскольку идентификатор `MSWINDOWS` в Kylix не определен. Аналогично работает и компилятор Delphi, только в этом случае будет определен идентификатор `MSWINDOWS`.

Формат PIC

Компилятор Linux создает исполняемые файлы в формате PIC (Position Independent Code), который несколько отличается от формата исполняемых файлов, создаваемых компилятором Windows. Если код написан только на языке Pascal, то это отличие может быть незначительным или вовсе отсутствовать, но ассемблерные вставки и компоновка с внешними ассемблерными модулями может привести к поистине драматическим последствиям. Кроме того, формат PIC требует, чтобы доступ ко всем глобальным данным был организован относительно регистра `EBX`. Например, следующая строка кода в Delphi

```
mov eax, SomeVar
```

для формата PIC будет написана так:

```
mov eax [ebx].SomeVar
```

Из-за жесткой привязки к регистру `EBX`, код PIC требует, чтобы значение регистра `EBX` сохранялось при вызове функций и восстанавливалось перед внешними обращениями. Используя директиву условной компиляции `$IFDEF`, можно, при необходимости, отделить ассемблерный код, специфичный для компилятора PIC, от остального кода:

```
{$IFDEF PIC}
  // Здесь расположен код PIC
{$ENDIF}
```

Соглашения о вызовах

Заметим, что соглашения о вызовах `stdcall` и `safecall` в Kylix не существуют. В Kylix функции таких директив выполняет соглашение о вызовах `cdecl`. Но это создаст проблемы только в том случае, если проект содержит ассемблерный код, зависящий от порядка передачи параметров или очистки стека.

Несколько мелочей

Создавая межплатформенное приложение, следует остерегаться множества мелочей в программном коде, связанных с особенностями конкретных платформ. Вот небольшой перечень элементов, на которые необходимо обращать внимание, чтобы не попасть впросак:

- В Linux не существует понятия имен диска (`C:`).
- В Linux разделитель каталогов представляет собой наклонную черту вправо (`/`), а не влево (`\`), как у Windows. Для этого случая Delphi располагает специальной константой, `PathSeparator`, которая принимает необходимое значение в зависимости от платформы.
- В Linux разделитель списка каталогов представляет собой двоеточие (`:`), а не точку с запятой (`;`) как у Windows.
- Пути имен UNC[1] существуют только в Windows.
- Избегайте указания пути каталогов в абсолютном виде, поскольку они зависят от платформы. Например `c:\winnt\system32` или `/usr/bin`.

Новые возможности Delphi 6

В новую версию Delphi внесены некоторые улучшения, особенно в область самого языка и компилятора, что существенно облегчает разработку приложения. Однако необходимо иметь в виду, что применение новых возможностей сделает код приложения несовместимым с компилятором прежних версий.

Варианты

Ранее поддержку типа данных `Variant` осуществлял сам компилятор, теперь она реализована в составе специального модуля `Variants`, отвечающего за пользовательские типы данных.

Значения перечислимого типа

Для большей совместимости с языком C++ нынешний компилятор способен присваивать значения элементам перечислимого типа, как в данном примере:

```
type
  TFoo = (fTwo=2, fFour=4, fSix=6, fEight=8);
```

---

[1] *Метод идентификации расположения файла на удаленном сервере с помощью соглашения Uniform Naming Convention (UNC). Имена UNC начинаются с символов* \\. – Прим. ред.

Директива $IF

Еще одна долгожданная возможность — директива `$IF` дополнена директивой `$ELSEIF`, что позволяет не только проверять наличие предопределенных идентификаторов, но и выполнять сложные логические операции с использованием констант, как и показано ниже:

```
{$IF Defined(MSWINDOWS) and SomeConstant >= 6}
  // Если определен идентификатор MSWINDOWS и SomeConstant больше
  // или равно 6, то выполнять операторы, расположенные здесь
{$ELSEIF SomeConstant < 2}
  // В противном случае, если SomeConstant меньше 2, то выполнять
  // операторы, расположенные здесь
{$ELSE}
  // Во всех остальных случаях выполнять операторы, расположенные
  // здесь
{$ENDIF}
```

Потенциальная несовместимость бинарных DFM

Механизм хранения и загрузки форм Delphi из потока (DFM — Delphi Forms Mechanism) изменился, преимущественно в области представления символов ASCII, номера которых больше 127 (вторая половина таблицы). Прежние версии Delphi не смогут прочитать бинарный код DFM, содержащий символы из второй половины таблицы ASCII. Совместимы лишь текстовые версии файлов хранения форм.

Переход от Delphi 5

Хотя совместимость между Delphi 5 и 6 достаточно высока, существует все-таки несколько незначительных отличий, которые необходимо учитывать при переходе на новую версию.

Переприсвоение типизированных констант

Теперь, в отличие от предыдущих версий, переключатель компилятора `$J` (известный также как `$WRITEABLECONST`) по умолчанию выключен. Это означает, что попытки присваивать значение типизированной константе приведет к возникновению ошибки во время компиляции, если не разрешить такую возможность явно, применив директиву `$J+`.

Унарное вычитание

До версии 6 Delphi использовал для выполнения операции унарного вычитания чисел типа `Cardinal` 32-разрядную арифметику. Это могло привести к непредвиденным результатам. Рассмотрим следующий фрагмент кода:

```
var
  c: Cardinal;
  i: Int64;
begin
  c := 4294967294;
  i := -c;
  WriteLn(i);
end;
```

Delphi 5 отобразит значение `i` как `2`. Хоть это и неправильно, но на подобное поведение иногда и рассчитывают при разработке программ. Если подобный код имеет место, то при переходе на Delphi 6 возникнут проблемы, поскольку теперь эта ошибка исправлена: приведение типа `Cardinal` к `Int64` будет выполнено *до* вычитания. Поэтому результирующим значением переменной `i` в Delphi 6 будет `4294967294`.

Переход от Delphi 4

Настоящий раздел посвящен ряду проблем, с которыми можно столкнуться при переходе от Delphi 4.

Проблемы RTL

Единственная проблема, с которой придется иметь дело, связана с установкой управляющего слова обработки вещественных чисел (FPU) в библиотеках DLL. В предыдущих версиях Delphi подпрограммы библиотек DLL имели право устанавливать управляющее слово FPU, а следовательно, и изменять его значение, заданное при запуске основного приложения. В новой версии в коде инициализации библиотек DLL установить значение управляющего слова FPU больше нельзя. Если установка этого управляющего слова необходима для достижения некоторого особого поведения FPU, следует выполнить ее вручную, вызвав функцию `Set8087()`, определенную в модуле `System`.

Проблемы VCL

При работе с библиотекой VCL могут возникнуть некоторые осложнения, но в большинстве случаев для переноса приложения в новую среду потребуется лишь слегка отредактировать исходный код проекта. Перечень возможных проблем приведен ниже.

- Тип свойства, представляющего индекс в списке графических изображений, был изменен с `Integer` на тип `TImageIndex`. Тип `TImageIndex` — это строго типизированное целое, определяемое в модуле `ImgList` следующим образом:

  ```
  TImageIndex = type Integer;
  ```

  Проблемы такого рода могут возникнуть только в тех случаях, когда необходимо точное соответствие типов (например при передаче `var`-параметров).

- В методе `TCustomTreeview.CustomDrawItem()` появился новый `var`-параметр типа `Boolean`, называющийся `PaintImages`. Если в приложении этот метод переопределен, то потребуется добавить в него этот параметр, иначе успешная компиляция в Delphi 5 или выше будет невозможна.

- Если приложение Delphi 4 выводит контекстные меню в ответ на сообщение `WM_RBUTTONUP` или при обработке события `OnMouseUp`, то после его компиляции в Delphi 5 и выше оно будет выводить либо "удвоенные" контекстные меню, либо совсем прекратит их вывод. Теперь в Delphi для организации вывода контекстного меню используется сообщение `WM_CONTEXTMENU`.

Проблемы приложений для Internet

Для тех, кто занят разработкой приложений для Internet, у нас есть как плохие новости, так и хорошие.

- Компонент `TWebBrowser`, инкапсулирующий элементы управления ActiveX для Microsoft Internet Explorer, заменил в новой версии Delphi компонент `THTML` компании *Netmasters*. Компонент `TWebBrowser` имеет существенно больший диапазон возможностей, но при замене им компонента `THTML` потребуется значительная переработка приложений, так как интерфейсы этих компонентов различны. Если нет желания вносить изменения в уже созданные приложения, можно обеспечить использование в них прежнего компонента, для чего необходимо импортировать файл `HTML.OCX`, расположенный в каталоге `\Info\Extras\NetManage` на инсталляционном компакт-диске.
- Теперь при создании библиотек DLL ISAPI и NSAPI обеспечивается поддержка пакетов. Чтобы воспользоваться этой возможностью, достаточно заменить в разделе `uses` модуль `HTTPApp` модулем `WebBroker`.

Проблемы баз данных

При переводе из Delphi 4 приложений, работающих с базами данных, перечень возможных осложнений совсем невелик и включает в себя переименование нескольких существовавших ранее имен в соответствии с новой архитектурой приложений DataSnap (прежде вызывавшейся MIDAS).

- Тип события `TDatabase.OnLogin` был изменен с `TLoginEvent` на `TDatabaseLoginEvent`. Это изменение едва ли способно вызвать проблемы, поскольку они возможны только в том случае, когда в приложении присутствует обработчик события `OnLogin`.
- Глобальные подпрограммы `FMTBCDToCurr()` и `CurrToFMTBCD()` были заменены новыми: `BCDToCurr()` и `CurrToBCD()`. (Соответствующие им закрытые методы класса `TDataSet` были заменены закрытым и недокументированным методом `DataConvert`.)
- Со времени появления Delphi 4 DataSnap (прежде MIDAS) подвергся ряду существенных изменений. Более подробная информация об этих изменениях и новых возможностях приведена в главе 21, "Разработка приложений DataSnap".

Переход от Delphi 3

Несмотря на то что перечень областей, в которых Delphi 3 и последующие версии этого продукта не совместимы, довольно короток, но иногда такая несовместимость служит потенциальным источником более серьезных проблем, нежели те, которые возникают при переходе от любых предыдущих версий к последующим. Большинство проблем несовместимости касается использования новых типов данных и изменения поведения некоторых уже существовавших.

32-разрядные беззнаковые целые

В Delphi 4 введен тип `LongWord` — 32-разрядный беззнаковый целый тип. В предыдущих версиях наибольшим целым типом был 32-разрядный знаковый целый тип. По этой причине многие типы, которые должны принимать только беззнаковые (положительные) значения, такие как `DWORD`, `UINT`, `HResult`, `HWND`, `HINSTANCE`, и некоторые типы дескрипторов, ранее определявшиеся просто как `Integer`, в Delphi 4 и последующих версиях переопределены как `LongWord`. Кроме того, тип `Cardinal`, ранее определявшийся в диапазоне `0..MaxInt`, сейчас также представляет собой тип `LongWord`. За исключением некоторых описанных ниже случаев, все эти изменения не оказывают никакого влияния на поведение программы.

- При передаче var-параметров типы `Integer` и `LongWord` не совместимы. Нельзя передавать значение типа `LongWord` в var-параметр типа `Integer`, и наоборот. В подобном случае компилятор сообщает об ошибке, поэтому для решения этой проблемы следует либо изменить тип параметра или переменной, либо выполнить явное приведение типов.
- Литеральные константы со значениями от `$80000000` до `$FFFFFFFF` рассматриваются как тип `LongWord`. Если такую константу необходимо присвоить целому типу, то потребуется приведение к типу `Integer`:

```
var
  I: Integer;
begin
  I := Integer($FFFFFFFF);
```

- Любой литерал, имеющий отрицательное значение, выходит за пределы диапазона `LongWord`. Поэтому перед присвоением литерала с отрицательным значением переменной типа `LongWord` необходимо выполнить соответствующее приведение типов:

```
var
  L: LongWord;
begin
  L := LongWord(-1);
```

- Если знаковые и беззнаковые целые участвуют в одном арифметическом выражении или в одной операции сравнения, то для вычисления этого выражения или выполнения сравнения компилятор автоматически приводит каждый операнд к типу `Int64`. Это неявное действие может послужить источником трудновыявляемых ошибок. Рассмотрим следующий код:

```
var
  I: Integer;
  D: DWORD;
begin
  I := -1;
  D := $FFFFFFFF;
  if I = D then DoSomething;
```

В Delphi 3 будут выполнены действия оператора *DoSomething*, так как `-1` и `$FFFFFFFF` имеют одинаковое значение при переводе в тип `Integer`. В Delphi 4 каждый операнд преобразуется в тип `Int64` для выполнения более точного сравнения, поэтому результирующий код будет сравнивать `$FFFFFFFFFFFFFFFF` и `$00000000FFFFFFFF`, которые определенно не равны между собой, и операторы *DoSomething* выполняться не будут.

СОВЕТ

Компилятор Delphi 4 и последующих версий выдает много новых подсказок, предупреждений и сообщений об ошибках, в том числе и относящихся к описанным проблемам совместимости и неявного преобразования типов. Удостоверьтесь, что режим подсказок и предупреждений при компилировании включен, это позволит компилятору сообщить о вышеописанных ошибках.

64-разрядный целый тип

В Delphi 4 также был введен новый тип, называемый `Int64`, который представляет собой 64-разрядный знаковый целый тип. Он широко используется в библиотеках RTL и VCL. Так, например, стандартные функции `Trunc()` и `Round()` возвращают значение типа `Int64`. Кроме того, появились новые версии функций `IntToStr()`, `IntToHex()` и других связанных с ними функций, работающих с `int64`.

Действительный тип

В Delphi версии 4 и последующих тип `Real` является псевдонимом типа `Double`. В предыдущих версиях Delphi и Turbo Pascal тип `Real` представлял собой 6-байтовый тип чисел с плавающей точкой. Это не станет причиной проблем, если только значения типа `Real` не сохранялись записанными в двоичных файлах (например в `file of record`), созданных приложениями предыдущих версий, и код программ не зависит от конкретной организации значений типа `Real` в памяти. Можно принудительно сделать тип `Real` прежним, 6-байтовым типом, включив директиву `{$REALCOMPATIBILITY ON}` в модули, в которых нужно использовать старое представление действительных типов данных. Если же требуется просто "заставить" некоторое ограниченное количество экземпляров типа `Real` иметь прежний размер, то вместо этой директивы следует воспользоваться типом `Real48`.

Переход от Delphi 2

На удивление высокая степень совместимости Delphi 2 с последующими версиями означает возможность довольно гладкого перехода на них. Тем не менее, с момента выпуска Delphi 2 произошли некоторые изменения и в языке, и в подпрограммах библиотеки VCL, поэтому следует знать о том, каким образом правильно перейти от Delphi 2 к последней версии и как воспользоваться всеми преимуществами ее новых возможностей.

Изменения в булевых типах

Реализация булевых типов в Delphi 2 (`Boolean`, `ByteBool`, `WordBool`, `LongBool`) подразумевает, что значение `True` соответствует значению `1`, а значение `False` — `0`. Для обеспечения лучшей совместимости с интерфейсом API Win32 реализация `ByteBool`, `WordBool` и `LongBool` немного изменилась: значение `True` теперь представляется целым — `-1` (`$FF`, `$FFFF` и `$FFFFFFFF` соответственно). Тип же `Boolean` остался прежним. Эти изменения могут сказаться на поведении программ, но только если в них используются числовые представления значений таких типов. Взгляните на следующее объявление:

```
var
  A: array[LongBool] of Integer;
```

В Delphi 2 данный код вполне безобиден — он объявляет массив целых `array[False..True]` или `[0..1]` с двумя элементами. Но в Delphi 3 это объявление может привести к некоторым весьма неожиданным результатам: поскольку `True` определено для `LongBool` как `$FFFFFFFF`, то подобное объявление приводит к созданию массива целых чисел `array[0..$FFFFFFFF]` или массива из четырех миллиардов (!) целых типа `Integer`. Во избежание этой проблемы в качестве индекса массива используйте тип `Boolean`.

Как ни странно, но необходимость в таком изменении возникла в связи с тем, что получившие широкое распространение элементы управления ActiveX и контейнеры элементов управления (как у Visual Basic) проверяют значение булевых переменных, сравнивая их с `-1`, а не с нулем или значением, отличным от нуля.

СОВЕТ

Во избежание ошибок и проблем совместимости никогда не пишите код подобный этому:

```
if BoolVar = True then ...
```

Вместо этого всегда проверяйте логические типы следующим образом:

```
if BoolVar then ...
```

Строковые ресурсы ResourceString

Если в приложении используются строковые ресурсы, рекомендуем воспользоваться преимуществом `ResourceString`. Более подробная информация по данной теме приведена в главе 2, “Язык программирования Object Pascal”. Это не повлияет ни

на размер приложения, ни на скорость его выполнения, но упростит процесс трансляции. `ResourceString` и связанные с ним методы использования ресурсов DLL важны при написании многоязычных (локализованных) приложений, работающих с единым пакетом ядра библиотеки VCL.

Изменения в RTL

Некоторые изменения, внесенные в динамическую библиотеку (RTL — Runtime Library) после выхода версии Delphi 2, могут привести к возникновению проблем при переносе приложений в новую среду. Во-первых, изменилось значение глобальной переменной `HInstance` — теперь она содержит дескриптор экземпляра текущей DLL, исполняемого файла или пакета. Для получения дескриптора экземпляра основного приложения используйте новую глобальную переменную `MainInstance`.

Второе важное изменение касается глобальной переменной `IsLibrary`. В Delphi 2 ее значение проверялось для выяснения способа выполнения данного кода — из контекста библиотеки `.DLL` или исполняемого файла `.EXE`. Переменная `IsLibrary` не обслуживает пакеты, поэтому не следует полагаться на нее при определении способа вызова программы (из исполняемого файла, библиотеки DLL или модуля пакета). Для этого лучше использовать глобальную переменную `ModuleIsLib`, которая возвращает `True` при вызове программы из библиотеки DLL или пакета. Кроме того, переменную `ModuleIsLib` можно применять в комбинации с глобальной переменной `ModuleIsPackage`, что позволит отличить библиотеку DLL от пакета.

Класс TCustomForm

В библиотеке VCL Delphi 3 появился новый промежуточный класс между классами `TScrollingWinControl` и `TForm`, называемый `TCustomForm`. Сам по себе этот факт не усложняет перенос приложений Delphi 2 в новую среду, но, если программа работает с экземплярами класса `TForm`, то ее код следует обновить так, чтобы вместо класса `TForm` использовался класс `TCustomForm`. В частности, подобные действия необходимы при вызове в программе функций `GetParentForm()`, `ValidParentForm()`, а также при любом использовании класса `TDesigner`.

СОВЕТ

Со времен Delphi 2 несколько изменилась семантика методов `GetParentForm()`, `ValidParentForm()` и других методов компонентов библиотеки VCL, возвращающих указатели `Parent`. Теперь они могут возвращать значение `nil`, даже если компонент имеет контекст родительского окна, в котором выполняется прорисовка. Например, если компонент инкапсулирован в элемент управления ActiveX, то он может иметь родительское окно, но не элемент управления `Parent`. Это означает, что необходимо проследить, чтобы в коде Delphi 2 не встречались подобные вещи:

```
with GetParentForm(xx) do ...
```

Теперь метод `GetParent()` может вернуть `nil`, в зависимости от того, каким образом инкапсулирован компонент.

Метод GetChildren()

Разработчики компонентов! Имейте в виду, что объявление `TComponent.GetChildren()` изменилось и выглядит сейчас так:

```
procedure GetChildren(Proc: TGetChildProc;
                      Root: TComponent); dynamic;
```

Новый параметр `Root` содержит указатель на владельца корневого компонента, т.е. на тот компонент, который можно получить, пройдя по цепочке владельцев компонента до самого начала, когда свойство `Owner` окажется равным `nil`.

Серверы автоматизации

Код, необходимый для использования функций автоматизации, значительно изменился по сравнению с тем, каким он был в Delphi 2. (Создание серверов автоматизации в Delphi описано в главе 15, "Разработка приложений COM".) Авторы полагают, что нет смысла подробно описывать эти различия, достаточно сказать, что стили создания серверов автоматизации, применяемые в Delphi 2, не следует использовать в последующих версиях Delphi.

В Delphi 2 автоматизация осуществляется с использованием возможностей модулей `OleAuto` и `Ole2`. Эти модули присутствуют в последующих версиях Delphi лишь в целях совместимости с прежними версиями, и их не рекомендуется использовать в новых проектах. В настоящее время те же функции обеспечивают модули `ComObj`, `ComServ` и `ActiveX`. Не следует смешивать старые модули с новыми в одном проекте.

Переход от Delphi 1

Если кому-то повезло оказаться обладателем антикварного кода, который необходимо откомпилировать и выполнять одновременно и под 16, и под 32-разрядными версиями Windows, то примите наши соболезнования, поскольку точек несовместимости между Delphi 1 и более поздними версиям очень много. Они расположены практически везде, начиная с несовпадения большинства основных типов данных и до библиотек VCL и API Windows. Однако из-за относительно малого количества разработчиков, продолжающих создавать и поддерживать приложения для 16-разрядных платформ, эта информация исключена из текста настоящей книги, но она есть в предыдущем издании в главе 15 – "Переход на Delphi 5".

Резюме

Информация, приведенная в данной главе, поможет быстро и без осложнений перенести проекты из предыдущих версий Delphi в Delphi 6. Приложив некоторые дополнительные усилия, можно создавать проекты, прекрасно работающие в различных версиях Delphi.

ГЛАВА

5

Создание многопоточных приложений

В ЭТОЙ ГЛАВЕ...

В операционной системе Win32 предусмотрена возможность выполнения в приложении нескольких потоков. Именно в этом и состоит наиболее весомое преимущество Win32 перед 16-разрядной Windows. Многопоточность может стать одной из главных причин перехода к 32-разрядной версии Delphi. В настоящей главе приведена информация, которая поможет достичь максимальной эффективности приложений за счет распределения выполняемой в нем работы между несколькими потоками.

Концепция потоков

Поток (thread) — это объект операционной системы, который представляет собой отдельный путь выполнения программы внутри определенного процесса. Каждое приложение Win32 имеет, по крайней мере, один поток, обычно называемый *первичным*, или *главным*, но приложения имеют право создавать дополнительные потоки, предназначенные для выполнения других задач.

С помощью потоков реализуются средства одновременного выполнения нескольких различных подпрограмм. Конечно, если компьютер оснащен только одним процессором, то о настоящей одновременности работы двух потоков говорить не приходится. Но когда для обработки каждого потока операционная система поочередно выделяет определенное время (измеряемое в мельчайших долях секунды[1]), создается впечатление одновременной работы нескольких приложений.

СОВЕТ

Потоки никогда не поддерживались в среде 16-разрядной Windows. Это означает, что ни одна 32-разрядная программа Delphi, написанная с использованием потоков, никогда не будет совместимой с Delphi 1. Данный момент обязательно нужно учитывать при разработке приложений, предназначенных для работы на обеих платформах.

Типы многозадачности

Понятие потока во многом отличается от многозадачности, поддерживаемой в среде 16-разрядной Windows. Возможно, читателю приходилось слышать, что Win32 называют операционной системой с *приоритетной* (preemptive), или *вытесняющей, многозадачностью*, а Windows 3.1 — средой с *кооперативной многозадачностью* (cooperative multitasking).

В чем же разница? В среде приоритетной многозадачности управление возложено на операционную систему — именно она отвечает за то, какой поток должен выполняться в данный момент времени (и обладать более высоким приоритетом). Когда первый поток приостанавливается системой ради передачи второму потоку нескольких циклов работы процессора, то о первом потоке говорят, что он *выгружен*. И если к тому же окажется, что первый поток выполняет бесконечный цикл, то, как правило, это не приводит к трагической ситуации, поскольку операционная система будет продолжать выделение процессорного времени для всех других потоков.

В среде Windows 3.1 ответственность за возвращение управления операционной системе по завершении выполнения приложения лежит целиком на разработчике. Поэтому, когда приложению не удается корректно завершить работу и вернуть управ-

[1] *Обычно в единицах по 100 наносекунд.* – Прим. ред.

ление операционной системе, кажется, что операционная система "зависает" — все хорошо знакомы с такой печальной ситуацией. Если вдуматься, то это может показаться даже забавным — устойчивость работы всей системы 16-разрядной Windows полностью зависит от поведения *всех* ее приложений, которые должны самостоятельно защитить себя от попадания в бесконечные циклы, замкнутую цепь рекурсии и другие неприятные ситуации. А поскольку все приложения для достижения корректной работы системы должны работать согласованно, этот тип многозадачности называется *кооперативным*.

Использование многопоточности в приложениях Delphi

Не секрет, что потоки — серьезное подспорье для программистов Windows. Вторичные потоки в приложениях можно создавать везде, где требуется выполнение некоторых фоновых задач. Типичным примерами таких действий является вычисление значений отдельных ячеек электронных таблиц или помещение в буфер документа MS Word при выводе его на печать. В этом случае задача разработчика — организовать необходимую обработку фоновых процессов, сохранив при этом приемлемое время реакции пользовательского интерфейса приложения.

Бо́льшая часть компонентов библиотеки VCL построена на предположении, что доступ к ним в каждый конкретный момент времени будет выполняться только из одного потока. Хотя это замечание особенно справедливо в отношении компонентов библиотеки VCL, используемых для создания пользовательского интерфейса, тем не менее важно понимать, что оно справедливо и для большинства других типов компонентов.

Невизуальные компоненты библиотеки VCL

Существует всего несколько областей, в которых подпрограммы библиотеки VCL гарантированно поддерживают многопоточность. Возможно, самым заметным в этом отношении является механизм поддержки свойств поточного ввода/вывода компонентов VCL. Особенности его реализации дают полную уверенность, что потоки данных в компонентах могут эффективно считываться и записываться сразу несколькими процессами. Помните, что даже важнейшие базовые классы в библиотеке VCL (например класс `TList`) спроектированы без поддержки доступа к ним одновременно из нескольких потоков. В некоторых случаях подпрограммы библиотеки VCL предоставляют "потокобезопасное" альтернативное решение, к которому можно обратиться в случае необходимости. В частности, если несколько потоков должны одновременно манипулировать некоторым списком, то для его реализации следует использовать класс `TThreadList`, а не обычный класс `TList`.

Визуальные компоненты библиотеки VCL

Подпрограммы библиотеки VCL требуют, чтобы все управление пользовательским интерфейсом происходило в контексте основного потока приложения (исключение составляет "потокобезопасный" класс `TCanvas`, речь о котором пойдет в настоящей главе далее). Конечно, существуют решения, позволяющие выполнять обновление пользовательского интерфейса со стороны вторичных потоков (подробнее это будет описано ниже). Но данное ограничение заставляет разработчиков приложений ис-

пользовать потоки более осторожно. В примерах, приведенных в этой главе, демонстрируются некоторые идеальные варианты использования многопоточности в приложениях Delphi.

Неправильное использование потоков

В любом деле важно не переборщить. Это утверждение справедливо и в отношении потоков. Несмотря на то что потоки оказывают неоценимую помощь в решении одних вопросов, они могут принести с собой целый букет новых проблем. Предположим, например, что необходимо создать интегрированную среду разработки, компилятор которой будет работать в своем собственном потоке, а программист сможет продолжать работу с приложением во время компиляции программы. Здесь могут возникнуть проблемы вот какого плана. Что будет, если внести изменения в файл, компилируемый в данный момент? Можно предложить несколько вариантов решения этой проблемы. Например, можно создать временную копию файла в момент его компиляции или сделать недоступными для редактирования файлы, компиляция которых еще не завершена. Все дело в том, что потоки не являются панацеей. Решая те или иные проблемы разработки, они неминуемо порождают новые, чреватые появлением ошибок, которые очень трудно устранить при отладке. Разработка и реализация кода, обеспечивающего устойчивую работу потоков, — задача не из легких, поскольку в этом случае приходится учитывать гораздо больше факторов, чем при разработке однопоточного приложения.

Объект TThread

Фактически Delphi инкапсулирует в объекте Object Pascal `TThread` объект потока API. Хотя класс `TThread` инкапсулирует практически все важнейшие функции объекта потока API в одном едином объекте, возможны ситуации, когда придется непосредственно обращаться к функциям API. Чаще всего это будет связано с необходимостью синхронизации потоков. В настоящем разделе обсудим работу объекта `TThread` и его применение в создаваемых приложениях.

Принципы работы объекта TThread

Класс `TThread` находится в модуле `Classes` и определен следующим образом:

```
  TThread = class
  private
    FHandle: THandle;
{$IFDEF MSWINDOWS}
    FThreadID: THandle;
{$ENDIF}
{$IFDEF LINUX}
    // ** FThreadID не соответствует THandle в Linux **
    FThreadID: Cardinal;
    FCreateSuspendedSem: TSemaphore;
    FInitialSuspendDone: Boolean;
{$ENDIF}
    FCreateSuspended: Boolean;
    FTerminated: Boolean;
```

```
    FSuspended: Boolean;
    FFreeOnTerminate: Boolean;
    FFinished: Boolean;
    FReturnValue: Integer;
    FOnTerminate: TNotifyEvent;
    FMethod: TThreadMethod;
    FSynchronizeException: TObject;
    FFatalException: TObject;
    procedure CheckThreadError(ErrCode: Integer); overload;
    procedure CheckThreadError(Success: Boolean); overload;
    procedure CallOnTerminate;
{$IFDEF MSWINDOWS}
    function GetPriority: TThreadPriority;
    procedure SetPriority(Value: TThreadPriority);
    procedure SetSuspended(Value: Boolean);
{$ENDIF}
{$IFDEF LINUX}
    // ** В Linux значение приоритета (Priority) имеет тип Integer
    function GetPriority: Integer;
    procedure SetPriority(Value: Integer);
    function GetPolicy: Integer;
    procedure SetPolicy(Value: Integer);
    procedure SetSuspended(Value: Boolean);
{$ENDIF}
  protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer read FReturnValue
                                write FReturnValue;
    property Terminated: Boolean read FTerminated;
  public
    constructor Create(CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure AfterConstruction; override;
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: LongWord;
    property FatalException: TObject read FFatalException;
    property FreeOnTerminate: Boolean read FFreeOnTerminate
                                    write FFreeOnTerminate;
    property Handle: THandle read FHandle;
{$IFDEF MSWINDOWS}
    property Priority: TThreadPriority read GetPriority
                                     write SetPriority;
{$ENDIF}
{$IFDEF LINUX}
    // ** Priority - тип Integer **
    property Priority: Integer read GetPriority write SetPriority;
    property Policy: Integer read GetPolicy write SetPolicy;
{$ENDIF}
    property Suspended: Boolean read FSuspended writeSetSuspended;
```

```
{$IFDEF MSWINDOWS}
    property ThreadID: THandle read FThreadID;
{$ENDIF}
{$IFDEF LINUX}
    // ** ThreadId - тип Cardinal **
    property ThreadID: Cardinal read FThreadID;
{$ENDIF}
    property OnTerminate: TNotifyEvent read FOnTerminate
                                         write FOnTerminate;
  end;
```

Как можно увидеть из объявления, класс `TThread` — прямой потомок (производный) от класса `TObject`, следовательно, он не является компонентом. Сравнив блоки кода `IFDEF`, нетрудно заметить, что класс `TThread` разработан так, чтобы различия между Delphi и Kylix были минимальными, хотя полностью их избежать все же не удалось. Кроме того, метод `TThread.Execute()` объявлен абстрактным (`abstract`). Это означает, что класс `TThread` сам является абстрактным. Таким образом, можно создавать экземпляры классов, производных от `TThread`, а экземпляр самого класса `TThread` создать нельзя. Проще всего создать потомок класса `TThread` с помощью пиктограммы Thread Object в диалоговом окне New Items (рис. 5.1), которое можно открыть, выбрав в меню File пункт New.

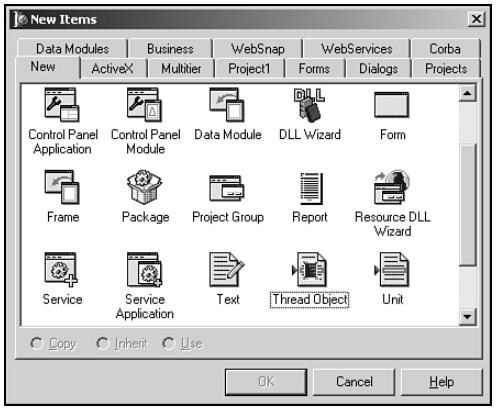

Рис. 5.1. *Пиктограмма Thread Object диалогового окна New Items*

После выбора элемента Thread Object в диалоговом окне New Items откроется диалоговое окно New Thread Object, в котором будет предложено ввести имя нового объекта. Например, можно ввести имя **`TTestThread`**. Затем Delphi создаст новый модуль, содержащий этот объект. Вот как будет выглядеть его объявление:

```
type
  TTestThread = class(TThread)
  private
    { Закрытые объявления }
  protected
    procedure Execute; override;
  end;
```

Как видно из примера, для создания функционального потомка класса `TThread` *необходимо* переопределить единственный метод — `Execute()`. Предположим теперь,

что внутри класса TTestThread требуется выполнить сложные вычисления. В таком случае метод Execute() можно было бы определить следующим образом:

```
procedure TTestThread.Execute;
var
  i, Answer: integer;
begin
  Answer := 0;
  for i := 1 to 2000000 do
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
end;
```

Единственная цель этих вычислений — затратить максимум времени на их выполнение.

Теперь можно выполнить данный пример потока, вызвав конструктор Create(). В данном случае для этого достаточно щелкнуть на кнопке главной формы, как показано в следующем фрагменте программы (во избежание ошибок компилятора не забудьте включить модуль, содержащий объект TTestThread, в раздел uses модуля TForm1):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewThread: TTestThread;
begin
  NewThread := TTestThread.Create(False);
end;
```

Если запустить приложение и щелкнуть на вышеуказанной кнопке, то можно убедиться, что по-прежнему существует возможность работать с формой, т.е. перемещать ее или изменять ее размеры, на фоне выполнения упомянутых вычислений.

НА ЗАМЕТКУ

Единственный логический параметр, который передается в конструктор Create() класса TThread, называется CreateSuspended. Он определяет, следует ли начинать работу потока с перевода его в приостановленное состояние. Если этот параметр равен False, то метод Execute() создаваемого объекта будет вызван автоматически и без промедления. Если данный параметр равен True, то для действительного запуска потока в определенной точке приложения потребуется вызвать его метод Resume(). Это приведет к выполнению метода Execute() потока и активизирует его. Обычно параметр CreateSuspended устанавливается равным True только в том случае, если перед запуском потока требуется установить дополнительные свойства его объекта. Установка свойств объекта после запуска потока может привести к возникновению проблем.

При более внимательном изучении работы конструктора Create() оказывается, что он вызывает функцию библиотеки RTL Delphi BeginThread(), которая, в свою очередь, вызывает функцию интерфейса API Win32 CreateThread() для создания нового потока. Значение параметра CreateSuspended показывает, нужно ли передавать функции CreateThread() флаг CREATE_SUSPENDED.

Экземпляры потока

Теперь вернемся к методу `Execute()` класса `TTestThread`. Обратите внимание: он содержит локальную переменную `i`. Давайте рассмотрим, что произойдет с переменной `i`, если создать два экземпляра класса `TTestThread`. Может ли значение такой переменной одного потока перезаписать значением одноименной переменной другого? Имеет ли первый поток преимущество перед вторым? Происходит ли в этом случае аварийное завершение потока? Ответы на все три вопроса одинаковы: нет, нет и нет. Система Win32 поддерживает для каждого работающего в системе потока отдельный стек. Это означает, что при создании нескольких экземпляров класса `TTestThread` каждый из них будет иметь собственную копию переменной `i` в своем собственном стеке. Следовательно, в этом отношении все потоки будут действовать независимо друг от друга[2].

Однако необходимо заметить, что понятие "одной и той же" переменной (которая в своем потоке действует независимо от "коллег") отнюдь не переносится на глобальные переменные. Более подробная информация по этой теме приведена в настоящей главе далее.

Завершение потока

Поток `TThread` считается завершенным, когда завершается выполнение его метода `Execute()`. В этот момент вызывается стандартная процедура Delphi `EndThread()`, которая, в свою очередь, вызывает функцию API Win32 `ExitThread()`. Данная функция должным образом освобождает стек потока и сам объект потока API. По завершении ее работы поток перестает существовать и все использованные им ресурсы будут освобождены.

По окончании использования объекта `TThread` нужно гарантированно уничтожить и соответствующий объект Object Pascal. Только в таком случае можно быть уверенным в корректном освобождении всей памяти, занимаемой данным объектом. И хотя это происходит автоматически после завершения процесса, возможно придется заняться освобождением объекта несколько раньше, чтобы исключить возникновение утечки памяти в приложении. Простейший способ гарантированно освободить объект `TThread` состоит в установке его свойства `FreeOnTerminate` равным значению `True`. Причем это можно сделать в любое время, до завершения выполнения метода `Execute()`. Например, для объекта `TTestThread` такое свойство устанавливается внутри метода `Execute()`:

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  FreeOnTerminate := True;
  for i := 1 to 2000000 do
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
end;
```

[2] *Как, впрочем, и экземпляры любых других классов.* – Прим. ред.

Объект TThread поддерживает также событие OnTerminate, которое происходит при завершении работы потока. Допускается освобождения объекта TThread внутри обработчика этого события.

НА ЗАМЕТКУ

Событие OnTerminate объекта TThread вызывается в контексте основного потока приложения. Это означает, что внутри обработчика данного события доступ к свойствам и методам VCL разрешается выполнять свободно, не прибегая к услугам метода Synchronize(), речь о котором пойдет в следующем разделе.

Следует иметь в виду, что метод Execute() объекта потока самолично несет ответственность за проверку состояния свойства Terminated с целью определения необходимости в досрочном завершении. Хотя это означает еще одно усложнение, о котором следует помнить при работе с потоками, достоинством этой архитектуры класса является полная гарантия того, что никакая "нечистая сила" не выдернет коврик из-под ваших ног в самый неподходящий момент. А значит, всегда будет существовать возможность выполнить все необходимые операции очистки по окончании работы потока. Подобную проверку состояния свойства Terminated довольно легко добавить в метод Execute() объекта TTestThread — она будет выглядеть следующим образом:

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  FreeOnTerminate := True;
  for i := 1 to 2000000 do begin
    if Terminated then Break;
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
  end;
end;
```

СОВЕТ

В случае аварийной ситуации для завершения выполнения потока можно также использовать функцию API Win32 TerminateThread(). К этой функции следует обращаться только в том случае, если другие варианты отсутствуют — например, когда поток попадает в бесконечный цикл и перестает реагировать на сообщения. Эта функция определяется следующим образом:

```
function TerminateThread(hThread:THandle;dwExitCode:DWORD);
```

Свойство THandle объекта TThread содержит дескриптор потока, присвоенный ему функциями API, поэтому к данной функции можно обращаться и с помощью следующего синтаксиса:

```
TerminateThread(MyHosedThread.Handle, 0);
```

При использовании этой функции следует помнить о крайне неприятных побочных эффектах, которые она способна вызывать. Во-первых, она может вести себя по-разному в операционных системах Windows NT/2000 и Windows 95/98. Под управлением Windows 95/98 функция TerminateThread() освобождает стек, связанный с потоком, а в Windows NT стек не освобождается до тех пор, пока не завершится процесс. Во-вторых, во всех опера-

ционных системах Win32 функция `TerminateThread()`, невзирая ни на что, просто останавливает выполнение в случайном месте и не позволяет

блоку `try..finally` освободить ресурсы. Это означает, что файлы, открытые потоком, могут остаться незакрытыми, память, выделенная потоком, может не быть освобождена и т.д. Кроме того, динамически компонуемые библиотеки (DLL), загруженные данным процессом, не получают соответствующего уведомления при завершении потока с помощью функции `TerminateThread()`, что может вызвать проблемы при закрытии DLL. Более подробная информация о возможности уведомления DLL о работе потоков приведена в главе 6 "Динамически компонуемые библиотеки".

Синхронизация с подпрограммами библиотеки VCL

Как уже неоднократно упоминалось в настоящей главе, прямой доступ к свойствам или методам компонентов библиотеки VCL следует выполнять только из основного потока приложения. Это означает, что любой код, который получает доступ или обновляет данные пользовательского интерфейса в приложении, должен выполняться только в контексте основного потока. Недостатки такой архитектуры вполне очевидны, и может показаться, что это требование слишком уж связывает руки разработчику, однако уже одно то, что о данном ограничении известно, дает определенное преимущество.

Преимущества однопоточного пользовательского интерфейса

Прежде всего, наличие только одного потока, получающего доступ к пользовательскому интерфейсу, значительно уменьшает сложность приложения. В соответствии с требованиями системы Win32, каждый поток, создающий диалоговое окно, должен иметь собственный цикл обработки сообщений на основе использования функции `GetMessage()`. Нетрудно представить, что наличие сообщений, приходящих от различных источников, существенно затруднит отладку приложения. Поскольку очередь сообщений приложения предусматривает их последовательную обработку, т.е. выполнение полной обработки одного сообщения до перехода к следующему, то в большинстве случаев приложение попадает в зависимость от последовательности поступления определенных сообщений, приходящих до или после других. Добавление еще одного цикла сообщений совершенно нарушает правила игры, существующие при последовательном вводе, открывая дорогу потенциальным проблемам синхронизации и вызывая необходимость добавления более сложной программной реализации, предусматривающей решение этих проблем.

Кроме того, поскольку компоненты VCL могут работать только при условии доступа к ним лишь одного потока в каждый момент времени, становится очевидной необходимость создания кода для синхронизации нескольких потоков внутри подпрограмм VCL. Общий итог всего сказанного состоит в том, что общая эффективность приложения прямо зависит от простоты архитектуры, выбранной при его построении.

Метод Synchronize()

В классе TThread определен метод Synchronize(), который позволяет вызывать некоторые из методов этого класса прямо из основного потока приложения. Определение метода Synchronize() имеет следующий вид:

```
procedure Synchronize(Method: TThreadMethod);
```

Параметр Method имеет тип TThreadMethod (означающий процедурный метод, не использующий никаких параметров), который определяется следующим образом:

```
type
  TThreadMethod = procedure of object;
```

Метод, передаваемый в качестве параметра Method, и является как раз тем методом, который затем выполняется из основного потока приложения. Возвращаясь к примеру с классом TTestThread, предположим, что необходимо отобразить результат вычислений в строке редактирования главной формы. Это можно сделать введя в класс TTestThread метод, который вносит необходимые изменения в свойство Text строки редактирования, и осуществив последующий вызов данного метода с помощью процедуры Synchronize().

Предположим, что этот метод называется GiveAnswer(). В листинге 5.1 представлен полный исходный код модуля ThrdU, который включает программную реализацию процесса обновления упомянутой выше строки редактирования в главной форме.

Листинг 5.1. Модуль ThrdU.PAS

```
unit ThrdU;

interface

uses
  Classes;

type
  TTestThread = class(TThread)
  private
    Answer: integer;
  protected
    procedure GiveAnswer;
    procedure Execute; override;
  end;

implementation

uses SysUtils, Main;

{ TTestThread }

procedure TTestThread.GiveAnswer;
begin
  MainForm.Edit1.Text := InttoStr(Answer);
```

```
end;

procedure TTestThread.Execute;
var
  I: Integer;
begin
  FreeOnTerminate := True;
  for I := 1 to    2000000 do
  begin
    if Terminated then Break;
    Inc(Answer, Round(Abs(Sin(Sqrt(I)))));
    Synchronize(GiveAnswer);
  end;
end;

end.
```

Как уже было сказано, с помощью метода `Synchronize()` можно выполнять методы в контексте основного потока, но до сих пор речь о них шла как о таинственном "черном ящике". Было известно, *что* он делает, но неизвестно *как*. Настало время приоткрыть завесу этой тайны.

При первом создании вторичного (или дополнительного) потока в приложении библиотека VCL создает и далее поддерживает скрытое *окно потока* (thread window) в контексте своего основного потока. Единственная цель этого окна заключается в организации последовательности вызовов процедур, выполненных с помощью метода `Synchronize()`.

Метод `Synchronize()` сохраняет метод, заданный параметром `Method`, в закрытом поле `FMethod` и посылает определенное библиотекой VCL сообщение `CM_EXECPROC` в окно потока, передавая в качестве параметра `lParam` данного сообщения значение `Self` (в этом случае `Self` указывает на объект `TThread`). Когда процедура окна потока получает сообщение `CM_EXECPROC`, она вызывает метод, заданный в поле `FMethod` с помощью экземпляра объекта `TThread`, переданного параметром `lParam`. Напомним, что, поскольку это окно было создано в контексте основного потока, процедура окна для него также выполняется основным потоком. Следовательно, метод, указанный в поле `FMethod`, также вызывается основным потоком.

Графическая иллюстрация того, что происходит внутри метода `Synchronize()`, представлена на рис. 5.2.

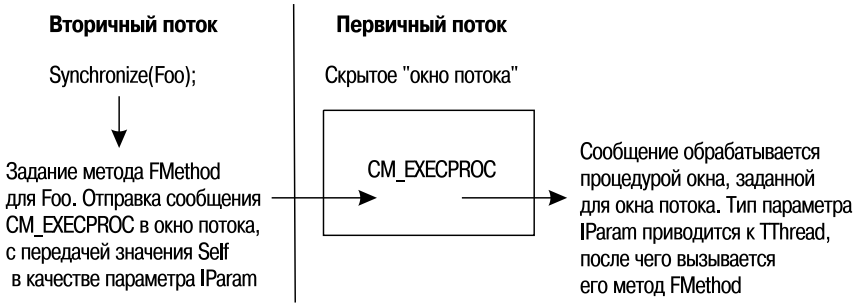

Рис. 5.2. *Схема работы метода* `Synchronize()`

Использование сообщений для синхронизации

В качестве альтернативы методу `TThread.Synchronize()` существует другой способ синхронизации, который заключается в использовании сообщений, передаваемых между потоками. В этом случае для отправки сообщений в окна, действующие в контексте другого потока, используется функция `SendMessage()` или `PostMessage()`. Например, приведенный ниже код можно было бы применить для вывода текста в строке редактирования, расположенной в другом потоке:

```
var
  S: string;
begin
  S := 'hello from threadland';
  SendMessage(SomeEdit.Handle, WM_SETTEXT, 0, Integer(PChar(S)));
end;
```

Демонстрационное приложение

Чтобы полностью проиллюстрировать работу нескольких потоков в среде Delphi, создадим проект `EZThrd`. Поместим в его главную форму окно примечания, как показано на рис. 5.3.

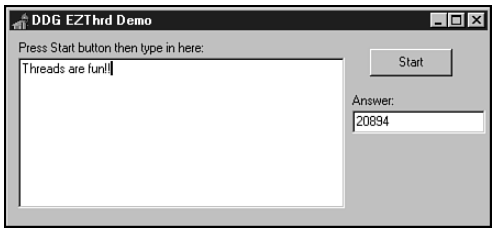

Рис. 5.3. *Главная форма демонстрационного приложения* `EZThrd`

Исходный код главного модуля приложения `EZThrd` приведен в листинге 5.2.

Листинг 5.2. Модуль `MAIN.PAS` приложения `EZThrd`

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ThrdU;

type
  TMainForm = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Memo1: TMemo;
    Label1: TLabel;
```

```
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.Button1Click(Sender: TObject);
var
  NewThread: TTestThread;
begin
  NewThread := TTestThread.Create(False);
end;

end.
```

Обратите внимание, что после щелчка на кнопке и активизации вторичного потока по-прежнему можно вводить информацию в окно примечания, как если бы вторичного потока не было совсем. По завершении вычислений результат будет отображен в строке редактирования.

Приоритеты и расписание

Как уже говорилось, операционная система отвечает за выделение каждому потоку нескольких циклов процессора, в течение которых они могут работать. Количество времени, выделяемое отдельному потоку, зависит от его приоритета. Приоритет отдельного потока определяет комбинация приоритета процесса, создавшего поток (называемого *приоритетом класса* (priority class)), и приоритета относительно самого потока (называемого *относительным приоритетом* (relative priority)).

Приоритет класса процесса

Приоритет класса процесса описывает приоритет определенного процесса, выполняющегося в системе. Система Win32 поддерживает четыре различных приоритета класса: `Idle` (ожидание), `Normal` (нормальный), `High` (высокий) и `Realtime` (реального времени). По умолчанию любому процессу присваивается приоритет `Normal`. Каждый из перечисленных приоритетов имеет соответствующий флаг, определенный в модуле `Windows`. С помощью операции `OR` (или) любой из этих флагов можно логически сложить с параметром `dwCreationFlags` функции `CreateProcess()`, что позволит при порождении процесса установить ему необходимый приоритет. Кроме того, такие флаги можно использовать и для динамической настройки приоритета класса данного процесса, как это будет показано ниже. Все классы можно со-

поставить уровням приоритета, которые выражаются числовыми значениями, находящимися в диапазоне от 4 до 24 включительно.

НА ЗАМЕТКУ

В среде Windows NT модификация приоритета класса процесса требует наличия у этого процесса специальных привилегий. Определенные приоритеты класса можно присвоить процессам с помощью стандартных параметров, но они могут быть отключены системными администраторами — в частности, на интенсивно загруженных серверах Windows NT/2000.

В табл. 5.1 приведены все существующие приоритеты класса, соответствующие им флаги и числовые значения.

Таблица 5.1. Приоритеты класса процесса

Класс	*Флаг*	*Значение*
`Idle`	`IDLE_PRIORITY_CLASS`	`$40`
`Below Normal*`	`BELOW_NORMAL_PRIORITY_CLASS`	`$4000`
`Normal`	`NORMAL_PRIORITY_CLASS`	`$20`
`Above Normal*`	`ABOVE_NORMAL_PRIORITY_CLASS`	`$8000`
`High`	`HIGH_PRIORITY_CLASS`	`$80`
`Realtime`	`REALTIME_PRIORITY_CLASS`	`$100`

** Эти значения доступны только в Windows 2000, а соответствующие константы для флагов в модуле* `Windpws.pas` *Delphi 6 отсутствуют.*

Для динамического считывания и установки приоритета класса данного процесса в API Win32 предусмотрены функции `GetPriorityClass()` и `SetPriorityClass()` соответственно. Эти функции определены следующим образом:

```
function GetPriorityClass(hProcess: THandle): DWORD; stdcall;

function SetPriorityClass(hProcess: THandle;
                          dwPriorityClass: DWORD): BOOL; stdcall;
```

В обеих функциях параметр `hProcess` представляет собой дескриптор процесса. Чаще всего эти функции используются для доступа к значению приоритета класса собственного процесса. В подобном случае можно обратиться и к функции интерфейса API Win32 `GetCurrentProcess()`, которая определена следующим образом:

```
function GetCurrentProcess: THandle; stdcall;
```

Значение, возвращаемое этими функциями, представляет собой псевдодескриптор текущего процесса. Приставка *псевдо* означает, что ни одна из таких функций не создает новый дескриптор, а возвращаемое значение не должно закрываться с помощью функции `CloseHandle()`. Это значение просто является дескриптором, который можно использовать для ссылки на существующий дескриптор.

Чтобы установить приоритет класса приложения равным `High`, используйте следующий код:

```
if not SetPriorityClass(GetCurrentProcess,
                        HIGH_PRIORITY_CLASS) then
  ShowMessage('Error setting priority class.');
```

СОВЕТ

В большинстве случаев следует избегать установки приоритета класса любого процесса равным `Realtime`. Поскольку большинство потоков операционных систем обладает приоритетом класса, который ниже, чем `Realtime`, то этот поток будет получать больше процессорного времени, чем сама операционная система, а это может вызвать некоторые неожиданные проблемы.
Вследствие неоправданной установки приоритета класса равным `High` могут возникнуть проблемы, если потоки процесса не тратят большую часть времени на простои или на ожидание внешних событий (например ввода-вывода). Один высокоприоритетный поток будет отбирать все процессорное время у низкоприоритетных потоков и процессов до тех пор, пока он или не заблокирует какое-нибудь событие, или не перейдет в состояние ожидания, или не займется обработкой сообщений. Неправильное назначение приоритетов легко может перечеркнуть все преимущества от использования многозадачности.

Относительный приоритет

Второй составляющей общего приоритета потока является *относительный приоритет* отдельного потока. Следует подчеркнуть, что приоритет класса связан с процессом, а относительный приоритет — с отдельными потоками внутри процесса. Потоку можно назначить один из семи возможных относительных приоритетов: `Idle` (ожидание), `Lowest` (низший), `Below Normal` (ниже нормального), `Normal` (нормальный), `Above Normal` (выше нормального), `Highest` (высший) или `Time Critical` (критический по времени).

Класс `TThread` имеет открытое свойство `Priority` перечислимого типа `TThreadPriority`. Для каждого относительного приоритета в этом типе существует свой элемент:

```
type
  TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal,
                     tpHigher, tpHighest, tpTimeCritical);
```

Приоритет любого объекта `TThread` можно получить или установить просто прочитав или записав его в свойство `Priority`. С помощью следующей строки кода присваивается высший приоритет экземпляру потомка класса `TThread`, который носит имя `MyThread`:

```
MyThread.Priority := tpHighest;
```

Подобно приоритетам классов, каждому относительному приоритету соответствует определенное числовое значение. Разница лишь в том, что значение относительного приоритета имеет знак, который при определении общего приоритета потока внутри системы влияет на результат суммирования приоритета класса процесса и относительного приоритета. Поэтому относительный приоритет иногда называется *дельта-приоритетом*. Общий приоритет потока может выражаться значением, находящимся в диапазоне от `1` до `31` (где число `1` означает самый низкий приоритет). В мо-

дуле Windows определены константы, представляющие знаковые значения для каждого приоритета. В табл. 5.2 показано соответствие констант API и элементов перечисления в типе TThreadPriority.

Таблица 5.2. Относительные приоритеты потока

TThreadPriority	*Константа*	*Значение*
tpIdle	THREAD_PRIORITY_IDLE	-15*
tpLowest	THREAD_PRIORITY_LOWEST	-2
tpBelow Normal	THREAD_PRIORITY_BELOW_NORMAL	-1
tpNormal	THREAD_PRIORITY_NORMAL	0
tpAbove Normal	THREAD_PRIORITY_ABOVE_NORMAL	1
tpHighest	THREAD_PRIORITY_HIGHEST	2
tpTimeCritical	THREAD_PRIORITY_TIME_CRITICAL	15*

Помеченные звездочкой (*) значения относительных приоритетов tpIdle и tpTimeCritical, в отличие от других, не добавляются к приоритету класса для определения общего приоритета потока. Общий приоритет потока, относительный приоритет которого равен tpIdle, независимо от приоритета класса устанавливается равным 1. Исключением из этого правила является приоритет Realtime, который, объединяясь с относительным приоритетом tpIdle, имеет значение общего приоритета, равное 16. Общий приоритет любого потока с относительным приоритетом tpTimeCritical, независимо от его приоритета класса равен 15. Исключением из этого правила является приоритет класса Realtime, который при объединении с относительным приоритетом tpTimeCritical имеет значение общего приоритета, равное 31.

Приостановка и возобновление потока

Давайте вспомним, что говорилось в этой главе ранее о конструкторе Create() класса TThread. Как уже было сказано, поток может быть создан в приостановленном состоянии, и для того чтобы он начал выполняться, необходимо вызвать метод Resume(). Логично предположить, что поток может быть приостановлен и возобновлен динамически. Эта задача решается с помощью методов Suspend() и Resume().

Хронометраж потока

Когда-то, во времена 16-разрядных приложений для Windows 3.*x*, было принято создавать оболочки для некоторых участков кода, которые с помощью функций GetTickCount() и timeGetTime() позволяли определить, сколько времени отнимает выполнение определенного вычисления. Например:

```
var
  StartTime, Total: Longint;
begin
  StartTime := GetTickCount;
```

```
{ Здесь выполняются вычисления }
Total := GetTickCount - StartTime;
```

Но в многопоточной среде это сделать намного труднее, поскольку выполнение приложения может быть приостановлено операционной системой ради предоставления ресурсов процессора другим процессам. Следовательно, любой хронометраж, если он опирается на системное время, не в состоянии дать истинную картину затрат времени, требуемых для выполнения вычислений в отдельном потоке.

Во избежание подобных проблем в системе Win32 (вариант Windows NT/2000) предусмотрена функция `GetThreadTimes()`, благодаря которой можно получить довольно подробную информацию о времени выполнения потока. Объявление этой функции имеет следующий вид:

```
function GetThreadTimes(hThread: THandle; var lpCreationTime,
                        lpExitTime, lpKernelTime,
                        lpUserTime: TFileTime): BOOL; stdcall;
```

Параметр `hThread` представляет собой дескриптор потока, для которого необходимо получить информацию о времени выполнения. Другие параметры передаются по ссылке и заполняются функцией. Рассмотрим каждый из них:

- `lpCreationTime`. Время создания потока.
- `lpExitTime`. Время окончания работы потока. Если поток продолжает выполняется, это значение не определено.
- `lpKernelTime`. Время, затраченное потоком на выполнение кода операционной системы.
- `lpUserTime`. Время, затраченное потоком на выполнение кода приложения.

Этих четыре параметра имеют тип `TFileTime`, который определен в модуле `Windows` следующим образом:

```
type
  TFileTime = record
    dwLowDateTime: DWORD;
    dwHighDateTime: DWORD;
  end;
```

Определение данного типа несколько необычно, но это часть интерфейса API Win32, и с этим придется мириться. Элементы записи `dwLowDateTime` и `dwHighDateTime` объединяются в учетверенное слово и образуют 64-разрядное значение, представляющее количество 100-наносекундных интервалов, прошедших с 1 января 1601 года. Это означает, что если бы потребовалось смоделировать движение английского флота, который нанес поражение испанской армаде в 1588 году, то тип `TFileTime` был бы совершенно неприемлем для отслеживания значений времени... Впрочем, мы несколько отвлеклись.

СОВЕТ

Поскольку тип `TFileTime` является 64-разрядным, то для выполнения арифметических действий над значениями этого типа можно использовать операцию приведения

типа `TFileTime` к типу `Int64`. Следующий пример демонстрирует возможный способ быстрого сравнения значений типа `TFileTime`:

```
if Int64(UserTime) > Int64(KernelTime) then Beep;
```

Для упрощения работы со значениями `TFileTime` в среде Delphi существует ряд функций, позволяющих выполнять преобразования между типами `TFileTime` и `TDateTime` в прямом и обратном направлениях:

```
function FileTimeToDateTime(FileTime: TFileTime): TDateTime;
var
  SysTime: TSystemTime;
begin
  if not FileTimeToSystemTime(FileTime, SysTime) then
    raise EConvertError.CreateFmt('FileTimeToSystemTime failed. '
    + 'Error code %d', [GetLastError]);
    with SysTime do
      Result := EncodeDate(wYear, wMonth, wDay) +
      EncodeTime(wHour, wMinute, wSecond, wMilliseconds)
end;

function DateTimeToFileTime(DateTime: TDateTime): TFileTime;
var
  SysTime: TSystemTime;
begin
  with SysTime do begin
    DecodeDate(DateTime, wYear, wMonth, wDay);
    DecodeTime(DateTime, wHour, wMinute, wSecond, wMilliseconds);
    wDayOfWeek := DayOfWeek(DateTime);
  end;
  if not SystemTimeToFileTime(SysTime, Result) then
    raise EConvertError.CreateFmt('SystemTimeToFileTime failed. '
    + 'Error code %d', [GetLastError]);
end;
```

СОВЕТ

Не забывайте, что функция `GetThreadTimes()` реализована только в Windows NT/2000. При ее вызове в среде Windows 95/98 всегда возвращается значение `False`. К сожалению, в Windows 95/98 не предусмотрен механизм получения информации о времени выполнения потоков.

Управление несколькими потоками

Как уже отмечалось, несмотря на то что с помощью потоков можно решить множество задач программирования, они приносят с собой новые типы проблем, с которыми приходится встречаться в создаваемых приложениях. Чаще всего эти новые проблемы связаны с доступом со стороны нескольких потоков к таким глобальным ресурсам, как глобальные переменные или дескрипторы. Кроме того, проблемы могут возникнуть в том случае, когда, например, нужно обеспечить постоянное возникновение некоторого события в одном потоке перед или после некоторого другого события во втором пото-

ке. В настоящем разделе обсудим, как решить эти вопросы, используя средства, предоставленные в Delphi для хранения локальных переменных потоков, а также средства API, предназначенные для синхронизации потоков.

Хранение локальных данных потоков

Поскольку каждый поток представляет собой отдельный и независимый путь выполнения программного кода внутри процесса, было бы логично предположить, что на определенном этапе потребуется какое-либо средство хранения данных, связанных с каждым потоком. Существует три метода хранения данных, уникальных для каждого потока. Первый, и самый простой, состоит в использовании локальных переменных (в стеке). Поскольку каждый поток получает собственный стек, при выполнении одной процедуры или функции он будет иметь и собственную копию локальных переменных. Второй метод заключается в сохранении локальной информации в объекте, производном от класса `TThread`. И, наконец, можно применить зарезервированное слово Object Pascal `threadvar`, чтобы воспользоваться преимуществами хранения локальной информации потока на уровне операционной системы.

Хранение данных в объекте TThread

Если сравнивать два последних варианта хранения данных потоков, то, бесспорно, следует остановить выбор на способе хранения данных в объекте, производном от класса `TThread`, поскольку он проще и эффективнее метода с использованием зарезервированного слова `threadvar` (речь о нем пойдет позже). Для объявления локальных данных потока этим способом достаточно добавить их в определение класса, производного от `TThread`:

```
type
  TMyThread = class(TThread)
  private
    FLocalInt: Integer;
    FLocalStr: String;
    .
    .
    .
  end;
```

СОВЕТ

Доступ к полю любого объекта осуществляется почти в 10 раз быстрее, чем доступ к переменной `threadvar`, поэтому данные потоков следует сохранять в потомке класса `TThread`, если, конечно, это возможно. Данные, которые существуют только в течение продолжительности жизни отдельной процедуры или функции, лучше сохранять в локальных переменных, поскольку доступ к ним осуществляется быстрее, чем к полям объекта `TThread`.

Ключевое слово threadvar и API хранения данных потока

Как уже говорилось, каждому потоку для хранения локальных переменных предоставляется собственный стек, в то время как глобальные данные должны совместно использоваться всеми потоками внутри приложения. Допустим, что существует процедура, которая устанавливает или отображает значение глобальной переменной, причем она построена так, что при передаче ей текстовой строки происходит установка, а при передаче пустой строки — отображение этой глобальной переменной. Такая процедура может иметь следующий вид:

```
var
  GlobalStr: String;
procedure SetShowStr(const S: String);
begin
  if S = '' then
    MessageBox(0, PChar(GlobalStr), 'The string is...', MB_OK)
  else
    GlobalStr := S;
end;
```

Если эта процедура вызывается в контексте только одного потока, никаких проблем не возникнет. Первый раз ее можно будет вызвать для установки значения переменной `GlobalStr`, а второй — для отображения этого значения. Давайте разберемся, что же произойдет, если два или больше потоков вызовут эту процедуру в произвольный момент времени. В таком случае возможна ситуация, когда один поток вызовет данную процедуру для установки строки, после чего время процессора будет отдано другому потоку, который также вызовет эту процедуру для установки своей строки. И к тому времени, когда операционная система передаст процессор обратно первому потоку, значение переменной `GlobalStr` будет безнадежно утрачено.

Для ситуаций, подобных описанной выше, в Win32 предусмотрено средство, известное под названием *хранение локальных данных потоков* (thread-local storage), благодаря которому можно создавать отдельные копии глобальных переменных для каждого выполняющегося потока. Delphi великолепно инкапсулирует это средство с помощью зарезервированного слова `threadvar`. Необходимо лишь объявить глобальные переменные, которые предполагается использовать раздельно для каждого потока, внутри раздела `threadvar` (а не `var`) — и делу конец. Переопределение переменной `GlobalStr` проще простого:

```
threadvar
  GlobalStr: String;
```

Модуль, код которого представлен в листинге 5.3, иллюстрирует именно такую проблему. Это главный модуль приложения Delphi, в форме которого содержится всего лишь одна кнопка. По щелчку на этой кнопке вызывается процедура установки, а затем и отображения значения переменной `GlobalStr`. Затем создается другой поток, где устанавливается и отображается значение его собственной переменной `GlobalStr`. После создания вторичного потока первичный вновь вызывает процедуру `SetShowStr` для отображения переменной `GlobalStr`.

Пробуйте запустить это приложение, объявив GlobalStr в разделе var, а не в threadvar. Результат выполнения будет существенно отличаться от предыдущего.

Листинг 5.3. Модуль MAIN.PAS — демонстрация хранения локальных переменных потоков

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

{ ЗАМЕЧАНИЕ: перенесите переменную GlobalStr из блока var в блок
threadvar и обратите внимание на различия в результатах работы
приложения. }
var
//threadvar
  GlobalStr: string;

type
  TTLSThread = class(TThread)
  private
    FNewStr: String;
  protected
    procedure Execute; override;
  public
    constructor Create(const ANewStr: String);
  end;

procedure SetShowStr(const S: String);
```

```
begin
  if S = '' then
    MessageBox(0, PChar(GlobalStr), 'The string is...', MB_OK)
  else
    GlobalStr := S;
end;

constructor TTLSThread.Create(const ANewStr: String);
begin
  FNewStr := ANewStr;
  inherited Create(False);
end;

procedure TTLSThread.Execute;
begin
  FreeOnTerminate := True;
  SetShowStr(FNewStr);
  SetShowStr('');
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  SetShowStr('Hello world');
  SetShowStr('');
  TTLSThread.Create('Dilbert');
  Sleep(100);
  SetShowStr('');
end;

end.
```

НА ЗАМЕТКУ

В этой демонстрационной программе после создания вторичного потока вызывается функция API Win32 `Sleep()`, которая объявляется следующим образом:

```
procedure Sleep(dwMilliseconds: DWORD); stdcall;
```

Функция `Sleep()` (объявленная как процедура) сообщает операционной системе о том, что текущий поток не нуждается в дополнительных циклах процессора в течение миллисекунд, заданных параметром `dwMilliseconds`. Эта процедура введена в данный код для создания эффекта имитации условий, когда система работает в режиме большей многозадачности, а также из-за необходимости внесения в приложение "случайности" выполнения различных потоков.

Иногда в качестве параметра `dwMilliseconds` передается нулевое значение. И хотя в этом случае текущий поток не застрахован от запуска в любой момент времени, тем не менее такой ход заставляет операционную систему передать процессорные циклы одному из ожидающих потоков с равным или высшим приоритетом.

В задачах, связанных с хронометражом, процедуру `Sleep()` необходимо использовать осторожно, поскольку она поможет справиться с отдельной проблемой на одной машине, но проблемы хронометража, которые не решены в общем виде, могут проявиться на дру-

гом компьютере, особенно если эта машина работает значительно быстрее или медленнее, чем первая, или обладает иным количеством процессоров.

Синхронизация потоков

При работе с несколькими потоками зачастую приходится синхронизировать доступ потоков к определенным данным или ресурсам. Предположим, что есть приложение, в котором один поток используется для считывания файла в память, а другой — для подсчета количества символов в этом файле. Само собой разумеется, что, до тех пор пока файл не загрузится в память полностью, подсчитать количество его символов невозможно. Но, поскольку каждая операция выполняется в своем собственном потоке, операционная система вольна обращаться с ними как с двумя совершенно независимыми задачами. Чтобы справиться с этой проблемой, необходимо синхронизировать эти два потока таким образом, чтобы поток, подсчитывающий символы, не начинал работать до тех пор, пока не завершится поток, загружающий файл.

Существует два типа проблем, связанных с синхронизацией потоков, и в Win32 предусмотрены различные пути их решения. В этом разделе находятся примеры методов синхронизации потоков с помощью *критических секций* (critical sections), *мьютексов* (mutexes), *семафоров* (semaphores) и событий.

В чем же заключается суть проблемы синхронизации потоков? Предположим, что существует массив целых чисел, который необходимо инициализировать возрастающими значениями. Сначала элементы массива инициализируются значениями от `1` до `128`, а затем им присваиваются значения от `128` до `255`. После этого результат работы потока отображается в окне списка. Эту задачу можно решить, дважды выполнив инициализацию в двух отдельных потоках. Попытка реализовать этот подход показана в коде модуля, представленного в листинге 5.4.

Листинг 5.4. Попытка инициализации одного массива в двух потоках

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;
```

```
  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  // возвращает глобальную переменную
  Inc(NextNumber);       // инкремент глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  OnTerminate := MainForm.ThreadsDone;
  for i := 1 to MaxSize do begin
    GlobalArray[i] := GetNextNumber;  // запись элемента массива
    Sleep(5);                         // смена потока
  end;
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then       // если оба потока завершены
    for i := 1 to MaxSize do
      { заполнить список элементами массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  TFooThread.Create(False);  // создать потоки
  TFooThread.Create(False);
end;
```

```
end.
```

Поскольку оба потока выполняются одновременно, при инициализации массива происходит искажение последовательности возрастания значений его элементов. Чтобы убедиться в этом, взгляните на результат работы данной программы, показанной на рис. 5.4.

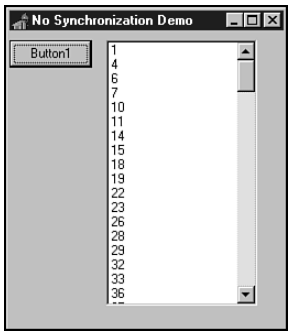

Рис. 5.4. *Результат несинхронизированной инициализации массива*

Решением этой проблемы является синхронизация двух потоков во время их обращения к глобальному массиву, что воспрепятствует их выполнению в одно и то же время. К этому можно подойти по-разному.

Критические секции

Критические секции представляют собой один из самых простых способов синхронизации потоков. *Критическая секция* (critical section) — это участок кода, который в каждый момент времени может выполняться только одним из потоков. Если код, используемый для инициализации массива, поместить в критическую секцию, то другие потоки не смогут использовать этот участок кода до тех пор, пока первый поток не завершит его выполнение.

Прежде чем использовать критическую секцию, ее необходимо инициализировать с помощью функции интерфейса API Win32 `InitializeCriticalSection()`, которая определяется следующим образом:

```
procedure InitializeCriticalSection(var lpCriticalSection:
                                    TRTLCriticalSection); stdcall;
```

Параметр `lpCriticalSection` представляет собой запись типа `TRTLCriticalSection`, которая передается по ссылке. Точное определение записи `TRTLCriticalSection` не имеет большого значения, поскольку в ее содержимое вряд ли понадобится когда-либо заглядывать. Необходимо лишь передать неинициализированную запись в параметр `lpCriticalSection`, и она будет тут же заполнена процедурой.

НА ЗАМЕТКУ

Microsoft преднамеренно не документирует структуру записи `TRTLCriticalSection`, поскольку ее содержимое варьируется при переходе от одной аппаратной платформы к другой, а также потому, что некорректное обращение к содержимому этой структуры спо-

собно нарушить работу процесса. В системах, построенных на платформе Intel, структура критического раздела содержит счетчик, поле с дескриптором текущего потока и (возможно) дескриптор системного события. В системах, построенных на платформе Alpha, счетчик заменен структурой данных Alpha-CPU, называемой *взаимоблокировкой* (spinlock), которая эффективнее решения Intel.

После заполнения записи в программе можно создать критическую секцию, поместив блок кода между вызовами функций API `EnterCriticalSection()` и `LeaveCriticalSection()`. Эти процедуры определяются следующим образом:

```
procedure EnterCriticalSection(var lpCriticalSection:
                               TRTLCriticalSection); stdcall;
procedure LeaveCriticalSection(var lpCriticalSection:
                               TRTLCriticalSection); stdcall;
```

Нетрудно догадаться, что параметр `lpCriticalSection`, который передается этим процедурам, является не чем иным, как записью, созданной процедурой `InitializeCriticalSection()`.

По окончании работы с записью `TRTLCriticalSection` необходимо освободить ее, вызвав функцию API `DeleteCriticalSection()`, которая определяется следующим образом:

```
procedure DeleteCriticalSection(var lpCriticalSection:
                                TRTLCriticalSection); stdcall;
```

Листинг 5.5 демонстрирует способ синхронизации потоков, инициализирующих массив с помощью критических секций.

Листинг 5.5. Использование критических секций

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
```

```
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  CS: TRTLCriticalSection;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  // возвращает глобальную переменную
  inc(NextNumber);       // инкремент глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  OnTerminate := MainForm.ThreadsDone;
  EnterCriticalSection(CS);          // начало критической секции
  for i := 1 to MaxSize do begin
    GlobalArray[i] := GetNextNumber; // запись элемента массива
    Sleep(5);                        // смена потока
  end;
  LeaveCriticalSection(CS);          // конец критической секции
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  inc(DoneFlags);
  if DoneFlags = 2 then begin    // если оба потока завершены
    for i := 1 to MaxSize do
      { заполнить список элементами массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    DeleteCriticalSection(CS);
  end;
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  InitializeCriticalSection(CS);
  TFooThread.Create(False);  // создать потоки
  TFooThread.Create(False);
end;
```

```
end.
```

После того как первый поток вызовет процедуру `EnterCriticalSection()`, всем другим потокам вход в этот блок кода будет запрещен. Следующий поток, который дойдет до этой строки кода, будет остановлен, т.е. перейдет в состояние ожидания до тех пор, пока первый поток не вызовет процедуру `LeaveCriticalSection()`. В этот момент система возобновит второй поток и разрешит ему пройти через критическую секцию. На рис. 5.5 показан результат работы этого приложения, когда потоки уже синхронизированы.

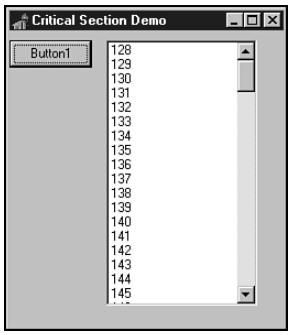

Рис. 5.5. *Результат синхронизированной инициализации массива*

Мьютексы

По принципу своего действия *мьютексы* (**MUT**ual **EX**clusions – взаимоисключения) очень похожи на критические секции, за исключением двух моментов. Во-первых, мьютексы можно использовать для синхронизации потоков, переступая через границы процессов. Во-вторых, мьютексу можно присвоить имя, ссылаясь на которое можно создавать дополнительные дескрипторы существующих объектов мьютексов.

СОВЕТ

Помимо семантических особенностей, самое большое различие между критическими секциями и такими объектами событий, как мьютексы, заключается в производительности. Критические секции очень эффективны: если нет конфликтов потоков, то на вход или выход из критической секции уходит всего 10–15 системных тактов процессора. Но если для данной критической секции возникает конфликт потоков, то система создает объект события (возможно, мьютекс).
В "стоимость" использования таких объектов событий, как мьютексы, входит обязательное обращение к подпрограммам ядра, что требует переключения контекста процесса и смены уровня цикла, а на это уходит от 400 до 600 системных тактов процессора. Причем без этих дополнительных затрат нельзя обойтись даже в том случае, если в приложении в данный момент вообще нет вторичных потоков или если нет других потоков, которые оспаривают право на защищаемые ресурсы.

Функция, используемая для создания мьютекса, называется `CreateMutex()`, а ее объявление выглядит следующим образом:

```
function CreateMutex(lpMutexAttributes: PSecurityAttributes;
            bInitialOwner: BOOL; lpName: PChar): THandle; stdcall;
```

Параметр `lpMutexAttributes` — это указатель на запись типа `TSecurityAttributes`. Обычно в качестве такого параметра передается значение `nil`, что приводит к использованию атрибутов защиты, установленных по умолчанию.

Параметр `bInitialOwner` определяет, следует ли считать поток, создающий мьютекс, его владельцем. Если данный параметр равен `False`, то мьютекс владельца не имеет.

Параметр `lpName` — это имя мьютекса. Если присваивать мьютексу имя не нужно, то установите данный параметр равным `nil`. Если же значение этого параметра отлично от `nil`, то функция выполнит в системе поиск мьютекса с таким же именем. При успешном завершении поиска функция вернет дескриптор найденного мьютекса, в противном случае — дескриптор нового мьютекса.

По завершении использования мьютекса необходимо закрыть его с помощью функции API `CloseHandle()`.

В листинге 5.6 продемонстрирован еще один способ синхронизации потоков, инициализирующих массив, но на этот раз с использованием мьютексов.

Листинг 5.6. Использование мьютексов для синхронизации потоков

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
```

```
  MaxSize =128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  hMutex: THandle = 0;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  // возвращает глобальную переменную
  Inc(NextNumber);       // инкремент глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  FreeOnTerminate := True;
  OnTerminate := MainForm.ThreadsDone;
  if WaitForSingleObject(hMutex, INFINITE) = WAIT_OBJECT_0 then
  begin
    for i := 1 to MaxSize do begin
      GlobalArray[i] := GetNextNumber;  // запись элемента массива
      Sleep(5);                         // смена потока
    end;
  end;
  ReleaseMutex(hMutex);
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then begin    // если оба потока завершены
    for i := 1 to MaxSize do
      { заполнить список элементами массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    CloseHandle(hMutex);
  end;
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  hMutex := CreateMutex(nil, False, nil);
  TFooThread.Create(False);  // создать потоки
  TFooThread.Create(False);
end;

end.
```

Нетрудно заметить, что в этом случае для управления входом потоков в синхронизируемый блок используется функция WaitForSingleObject(), объявленная следующим образом:

```
function WaitForSingleObject(hHandle: THandle;
                             dwMilliseconds: DWORD): DWORD; stdcall;
```

Данная функция предназначена для перевода текущего потока в состояние ожидания, пока объект API, заданный параметром hHandle, не станет доступным. При этом состояние ожидания может продлиться вплоть до истечения интервала времени, заданного в миллисекундах параметром dwMilliseconds. Термин *доступен* (signaled) для различных объектов понимается по-разному. Мьютекс становится доступным, если он больше не принадлежит потоку, в то время как, например, процесс становится доступным после его завершения. Помимо реального периода времени, параметр dwMilliseconds может также иметь нулевое значение, которое указывает на необходимость проверки состояния объекта и немедленный возврат. Возможно и значение INFINITE, указывающее, что ожидать следует до тех пор, пока объект не станет доступным. Значения, которые может возвращать эта функция, перечислены в табл. 5.3.

Таблица 5.3. Константы типа ожидания, используемые функцией API Win32 WaitForSingleObject()

Значение	*Описание*
WAIT_ABANDONED	Объект является объектом мьютекса, но поток, владевший этим мьютексом, был завершен до его освобождения. Такой мьютекс считается покинутым (abandoned), право собственности на его объект передается вызывающему потоку, а сам мьютекс становится недоступным
WAIT_OBJECT_0	Состояние объекта определяется как доступное
WAIT_TIMEOUT	Установленный интервал времени ожидания истек, и состояние объекта определяется как недоступное

Повторим еще раз: если мьютекс не принадлежит какому-либо потоку, он находится в доступном состоянии. Первый же поток, вызвавший функцию WaitForSingleObject() с запросом на данный мьютекс, получит право собственности на него, а состояние самого объекта мьютекса станет недоступным. Когда поток вызывает функцию ReleaseMutex(), передавая в качестве параметра дескриптор принадлежащего ему мьютекса, право собственности на этот мьютекс у данного потока отбирается, а мьютекс вновь становится доступным.

НА ЗАМЕТКУ

Помимо функции WaitForSingleObject(), в интерфейсе API Win32 есть также функции WaitForMultipleObjects() и MsgWaitForMultipleObjects(), которые позволяют продлить ожидание до тех пор, пока один или несколько объектов не станут доступными. Эти функции описаны в интерактивной справочной системе интерфейса API Win32.

Семафоры

Существует еще один метод синхронизации потоков, в котором используются объекты *семафоров* (semaphore) API. В семафорах применен принцип действия мьютексов, но с одной существенной деталью. В них заложена возможность подсчета ресурсов, что позволяет заранее определенному количеству потоков одновременно войти в синхронизируемый блок кода. Для создания семафора используется функция API `CreateSemaphore()`, которая объявляется следующим образом:

```
function CreateSemaphore(lpSemaphoreAttributes:
                         PSecurityAttributes; lInitialCount,
                         lMaximumCount: Longint;
                         lpName: PChar): THandle; stdcall;
```

Как и у функции `CreateMutex()`, первым параметром, передаваемым функции `CreateSemaphore()`, является указатель на запись `TSecurityAttributes`, причем значение `Nil` соответствует согласию на использование стандартных атрибутов защиты.

Параметр `lInitialCount` представляет собой начальное значение счетчика объекта семафора. Это число может находиться в диапазоне от `0` до значения `lMaximumCount`. Семафор доступен, если значение данного параметра больше нуля. Когда поток вызывает функцию `WaitForSingleObject()` или любую другую, ей подобную, значение счетчика семафора уменьшается на единицу. И наоборот, при вызове потоком функции `ReleaseSemaphore()` значение счетчика семафора увеличивается на единицу.

С помощью параметра `lMaximumCount` задается максимальное значение счетчика семафора. Если семафор используется для контроля за доступом к некоторым ресурсам, то это число должно соответствовать общему количеству этих ресурсов.

Параметр `lpName` содержит имя семафора. Поведение этого параметра аналогично поведению одноименного параметра функции `CreateMutex()`.

В листинге 5.7 продемонстрировано использование семафоров для синхронизации потоков при инициализации массива.

Листинг 5.7. Использование семафоров для синхронизации потоков

```
unit Main;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
```

```
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize =128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  hSem: THandle = 0;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  // возвращает глобальную переменную
  Inc(NextNumber);       // инкремент глобальной переменной
end;

procedure TFooThread.Execute;
var
  i: Integer;
  WaitReturn: DWORD;
begin
  OnTerminate := MainForm.ThreadsDone;
  WaitReturn := WaitForSingleObject(hSem, INFINITE);
  if WaitReturn = WAIT_OBJECT_0 then begin
    for i := 1 to MaxSize do begin
      GlobalArray[i] := GetNextNumber;  // запись элемента массива
      Sleep(5);                         // смена потока
    end;
  end;
  ReleaseSemaphore(hSem, 1, nil);
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then begin   // если оба потока завершены
    for i := 1 to MaxSize do
      { заполнить список элементами массива }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    CloseHandle(hSem);
  end;
```

```
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  hSem := CreateSemaphore(nil, 1, 1, nil);
  TFooThread.Create(False);  // создать потоки
  TFooThread.Create(False);
end;

end.
```

Поскольку проход через блок кода синхронизации разрешен только одному потоку, максимальное значение счетчика семафора равно 1.

Функция ReleaseSemaphore() используется для увеличения значения счетчика семафора. Обратите внимание, эта функция имеет больше параметров, чем ее "коллега" ReleaseMutex(). Объявление функции ReleaseSemaphore() выглядит следующим образом:

```
function ReleaseSemaphore(hSemaphore: THandle;
                          lReleaseCount: Longint;
                          lpPreviousCount: Pointer): BOOL; stdcall;
```

С помощью параметра lReleaseCount можно задать число, на которое будет уменьшено значение счетчика семафора. При этом старое значение счетчика будет сохранено в переменной типа Longint, на которую указывает параметр lpPreviousCount, если его значение не равно Nil. Скрытый смысл данной возможности заключается в том, что семафор никогда не принадлежит ни одному отдельному потоку. Предположим, что максимальное значение счетчика семафора было равно 10 и десять потоков вызвали функцию WaitForSingleObject(). В результате счетчик потоков сбрасывается до нуля и тем самым семафор переводится в недоступное состояние. После этого достаточно одному из потоков вызвать функцию ReleaseSemaphore() и в качестве параметра lReleaseCount передать число 10, как семафор не просто будет снова пропускать потоки, т.е. станет доступным, но и увеличит значение своего счетчика до прежнего числа — до 10. Такое мощное средство может привести к ошибкам, которые будет трудно обнаружить, поэтому его следует использовать с большой осторожностью.

Для освобождения дескриптора семафора, выделенного ему с помощью функции CreateSemaphore(), не забудьте вызвать функцию CloseHandle().

Пример многопоточного приложения

Для демонстрации использования объектов TThread в настоящем разделе приводится пример реального приложения, предназначенного для поиска файлов в специализированном потоке. Имя проекта — DelSrch, оно образовано от слов *Delphi Search*, означающих поиск файлов Delphi. Главная форма этой утилиты показана на рис. 5.6.

Приложение работает следующим образом. Пользователь выбирает путь, где следует проводить поиск, и указывает маску, чтобы уточнить тип искомых файлов. Кроме того, в соответствующую строку редактирования пользователь вводит текстовую строку для поиска. В форме также имеются флажки, с помощью которых можно указать специальные условия поиска. По щелчку на кнопке Search создается поток поис-

ка, и в объект потомка класса TThread передается необходимая для поиска информация: строка, путь и маска файла. Когда поток обнаруживает в определенных файлах искомую строку, в окно списка добавляется соответствующая информация. Наконец, если пользователь дважды щелкнет на имени файла в окне списка, ему предоставляется возможность просмотреть этот файл с помощью текстового редактора или другой связанной с ним программы.

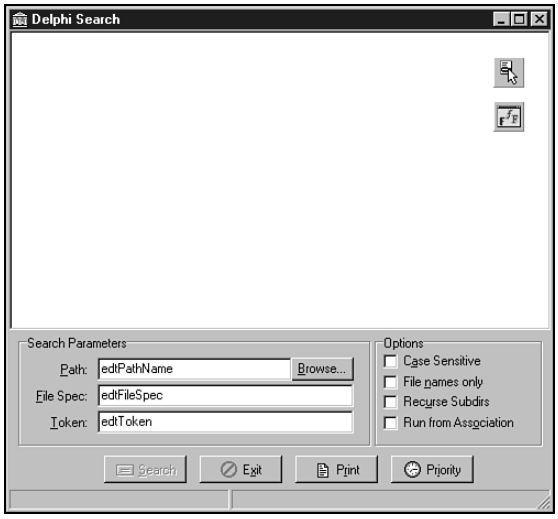

Рис. 5.6. *Главная форма проекта* `DelSrch`

Несмотря на то, что здесь приведено полнофункциональное приложение, внимание преимущественно будет уделено реализации основных функций поиска и их связи с многопоточностью.

Пользовательский интерфейс

Главный модуль этого приложения называется `Main.pas`, его исходный код представлен в листинге 5.8. Основная задача данного модуля — обеспечить функционирование главной формы и ее пользовательского интерфейса. В частности, в модуле выполняются действия, необходимые для заполнения окна списка, вызова соответствующей программы просмотра указанного пользователем файла, создания потока поиска, вывода на печать содержимого списка, а также чтения и записи параметров пользовательского интерфейса в файл инициализации (`.INI`).

Листинг 5.8. Модуль `Main.pas` проекта `DelSrch`

```
unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
```

```
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls, Menus,
  SrchIni, SrchU, ComCtrls, AppEvnts;

type
  TMainForm = class(TForm)
    lbFiles: TListBox;
    StatusBar: TStatusBar;
    pnlControls: TPanel;
    PopupMenu: TPopupMenu;
    FontDialog: TFontDialog;
    pnlOptions: TPanel;
    gbParams: TGroupBox;
    LFileSpec: TLabel;
    LToken: TLabel;
    lPathName: TLabel;
    edtFileSpec: TEdit;
    edtToken: TEdit;
    btnPath: TButton;
    edtPathName: TEdit;
    gbOptions: TGroupBox;
    cbCaseSensitive: TCheckBox;
    cbFileNamesOnly: TCheckBox;
    cbRecurse: TCheckBox;
    cbRunFromAss: TCheckBox;
    pnlButtons: TPanel;
    btnSearch: TBitBtn;
    btnClose: TBitBtn;
    btnPrint: TBitBtn;
    btnPriority: TBitBtn;
    Font1: TMenuItem;
    Clear1: TMenuItem;
    Print1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    ApplicationEvents: TApplicationEvents;
    procedure btnSearchClick(Sender: TObject);
    procedure btnPathClick(Sender: TObject);
    procedure lbFilesDrawItem(Control: TWinControl;
                              Index: Integer; Rect: TRect;
                              State: TOwnerDrawState);
    procedure Font1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnPrintClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure lbFilesDblClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure btnPriorityClick(Sender: TObject);
    procedure edtTokenChange(Sender: TObject);
    procedure Clear1Click(Sender: TObject);
    procedure ApplicationEventsHint(Sender: TObject);
  private
    procedure ReadIni;
```

```
    procedure WriteIni;
  public
    Running: Boolean;
    SearchPri: Integer;
    SearchThread: TSearchThread;
    procedure EnableSearchControls(Enable: Boolean);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses Printers, ShellAPI, StrUtils, FileCtrl, PriU;

procedure PrintStrings(Strings: TStrings);
{ Эта процедура выводит на печать все строки, переданные в
параметре Strings }
var
  Prn: TextFile;
  I: Integer;
begin
  if Strings.Count = 0 then // Есть ли строки для печати?
    raise Exception.Create('No text to print!');
  AssignPrn(Prn);           // назначить Prn принтеру
  try
    Rewrite(Prn);           // открыть принтер
    try
      for I := 0 to Strings.Count - 1 do  // цикл по всем строкам
        WriteLn(Prn, Strings.Strings[I]); // вывод на принтер
    finally
      CloseFile(Prn);                     // закрыть принтер
    end;
  except
    on EInOutError do
      MessageDlg('Error Printing text.', mtError, [mbOk], 0);
  end;
end;

procedure TMainForm.EnableSearchControls(Enable: Boolean);
{ Разрешить или запретить отдельные элементы управления, чтобы
нельзя было изменить параметры во время выполнения поиска. }
begin
  btnSearch.Enabled := Enable;          // разрешить (Enable)
  cbRecurse.Enabled := Enable;          // запретить (Disable)
  cbFileNamesOnly.Enabled := Enable;
  cbCaseSensitive.Enabled := Enable;
  btnPath.Enabled := Enable;
  edtPathName.Enabled := Enable;
  edtFileSpec.Enabled := Enable;
  edtToken.Enabled := Enable;
```

```
  Running := not Enable;        // установить флаг Running
  edtTokenChange(nil);
  with btnClose do begin
    if Enable then begin  // установить свойства кнопки Close/Stop
      Caption := '&Close';
      Hint := 'Close Application';
    end
    else begin
      Caption := '&Stop';
      Hint := 'Stop Searching';
    end;
  end;
end;

procedure TMainForm.btnSearchClick(Sender: TObject);
{ Вызывается при щелчке на кнопке Search. Создает поток поиска }
begin
  EnableSearchControls(False);   // отключить элементы управления
  lbFiles.Clear;                 // очистить список
  { запуск потока }
  SearchThread := TSearchThread.Create(cbCaseSensitive.Checked,
       cbFileNamesOnly.Checked, cbRecurse.Checked, edtToken.Text,
       edtPathName.Text, edtFileSpec.Text);
end;

procedure TMainForm.edtTokenChange(Sender: TObject);
begin
  btnSearch.Enabled := not Running and (edtToken.Text <> '');
end;

procedure TMainForm.btnPathClick(Sender: TObject);
{ Вызывается при щелчке на кнопке Path. Позволяет выбрать новый
путь }
var
  ShowDir: string;
begin
  ShowDir := edtPathName.Text;
  if SelectDirectory('Choose a search path...', '', ShowDir) then
    edtPathName.Text := ShowDir;
end;

procedure TMainForm.lbFilesDrawItem(Control: TWinControl;
             Index: Integer; Rect: TRect; State: TOwnerDrawState);
{ Вызывается, если пользователь использует список. }
var
  CurStr: string;
begin
  with lbFiles do begin
    CurStr := Items.Strings[Index];
    Canvas.FillRect(Rect);               // очистка прямоугольника
    if not cbFileNamesOnly.Checked then  // если не только имя
                                         // файла...
      { Если текущая строка является именем файла... }
```

```
      if (Pos('File ', CurStr) = 1) and
         (CurStr[Length(CurStr)] = ':') then
        with Canvas.Font do begin
          Style := [fsUnderline]; // шрифт с подчеркиванием
          Color := clRed;         // цвет красный
        end
      else
        Rect.Left := Rect.Left + 15;     // иначе - отступ
    DrawText(Canvas.Handle, PChar(CurStr), Length(CurStr),
             Rect, DT_SINGLELINE);
  end;
end;

procedure TMainForm.Font1Click(Sender: TObject);
{ Позволяет подобрать для списка новый шрифт. }
begin
  { Выбор нового шрифта списка. }
  if FontDialog.Execute then
    lbFiles.Font := FontDialog.Font;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
{ Обработчик события OnDestroy для формы. }
begin
  WriteIni;
end;

procedure TMainForm.FormCreate(Sender: TObject);
{ Обработчик события OnCreate для формы. }
begin
  ReadIni;                          // чтение INI-файла
end;

procedure TMainForm.btnPrintClick(Sender: TObject);
{ Вызывается при щелчке на кнопке Print. }
begin
  if MessageDlg('Send search results to printer?', mtConfirmation,
                [mbYes, mbNo], 0) = mrYes then
    PrintStrings(lbFiles.Items);
end;

procedure TMainForm.btnCloseClick(Sender: TObject);
{ Вызывается для остановки потока или завершения приложения. }
begin
  // если поток выполняется, завершить его
  if Running then SearchThread.Terminate
  // в противном случае завершить приложение
  else Close;
end;

procedure TMainForm.lbFilesDblClick(Sender: TObject);
{ Вызывается при двойном щелчке в окне списка. Запускает программу
просмотра для выделенного файла. }
```

```
var
  ProgramStr, FileStr: string;
  RetVal: THandle;
begin
  { Если пользователь щелкнул на файле.. }
  if (Pos('File ', lbFiles.Items[lbFiles.ItemIndex]) = 1) then
  begin // Загрузить текстовый редактор, указанный в INI-файле.
        // По умолчанию загружается Notepad.
    ProgramStr := SrchIniFile.ReadString('Defaults', 'Editor',
                                         'notepad');
    // Получить выбранный файл
    FileStr := lbFiles.Items[lbFiles.ItemIndex];
    // Удалить префикс
    FileStr := Copy(FileStr, 6, Length(FileStr) - 5);
    // Удалить ":"
    if FileStr[Length(FileStr)] = ':' then DecStrLen(FileStr, 1);
    if cbRunFromAss.Checked then
      { Загрузить файл в ассоциированное с ним приложение }
      RetVal := ShellExecute(Handle, 'open', PChar(FileStr), nil,
                             nil, SW_SHOWNORMAL)
    else { В противном случае - в текстовый редактор }
      RetVal := ShellExecute(Handle, 'open', PChar(ProgramStr),
                             PChar(FileStr), nil, SW_SHOWNORMAL);
    { Проверка ошибок }
    if RetVal < 32 then RaiseLastWin32Error;
  end;
end;

procedure TMainForm.FormResize(Sender: TObject);
{ Обработчик события OnResize. Центрирует элементы управления в
форме. }
begin
  { Делим строку состояния на две панели в отношении 1/3 - 2/3. }
  with StatusBar do begin
    Panels[0].Width := Width div 3;
    Panels[1].Width := Width * 2 div 3;
  end;
end;

procedure TMainForm.btnPriorityClick(Sender: TObject);
{ Отображает форму приоритета потока. }
begin
  ThreadPriWin.Show;
end;

procedure TMainForm.ReadIni;
{ Читает из реестра значения по умолчанию. }
begin
  with SrchIniFile do begin
    edtPathName.Text := ReadString('Defaults', 'LastPath', 'C:\');
    edtFileSpec.Text := ReadString('Defaults', 'LastFileSpec',
                                   '*.*');
    edtToken.Text := ReadString('Defaults', 'LastToken', '');
```

```
    cbFileNamesOnly.Checked := ReadBool('Defaults', 'FNamesOnly',
                                        False);
    cbCaseSensitive.Checked := ReadBool('Defaults', 'CaseSens',
                                        False);
    cbRecurse.Checked := ReadBool('Defaults', 'Recurse', False);
    cbRunFromAss.Checked := ReadBool('Defaults', 'RunFromAss',
                                     False);
    Left := ReadInteger('Position', 'Left', Left);
    Top := ReadInteger('Position', 'Top', Top);
    Width := ReadInteger('Position', 'Width', Width);
    Height := ReadInteger('Position', 'Height', Height);
  end;
end;

procedure TMainForm.WriteIni;
{ Записывает текущие параметры назад в реестр. }
begin
  with SrchIniFile do begin
    WriteString('Defaults', 'LastPath', edtPathName.Text);
    WriteString('Defaults', 'LastFileSpec', edtFileSpec.Text);
    WriteString('Defaults', 'LastToken', edtToken.Text);
    WriteBool('Defaults', 'CaseSens', cbCaseSensitive.Checked);
    WriteBool('Defaults', 'FNamesOnly', cbFileNamesOnly.Checked);
    WriteBool('Defaults', 'Recurse', cbRecurse.Checked);
    WriteBool('Defaults', 'RunFromAss', cbRunFromAss.Checked);
    WriteInteger('Position', 'Left', Left);
    WriteInteger('Position', 'Top', Top);
    WriteInteger('Position', 'Width', Width);
    WriteInteger('Position', 'Height', Height);
  end;
end;

procedure TMainForm.Clear1Click(Sender: TObject);
begin
  lbFiles.Items.Clear;
end;

procedure TMainForm.ApplicationEventsHint(Sender: TObject);
{ Обработчик события OnHint объекта Application }
begin
  { Отобразить подсказки на панели состояния окна приложения }
  StatusBar.Panels[0].Text := Application.Hint;
end;

end.
```

Некоторые фрагменты этого модуля заслуживают особого внимания. Прежде всего стоит остановиться на маленькой процедуре `PrintStrings()`, применяющейся для отправки содержимого объекта `TStrings` на принтер. Для этого процедура использует преимущества стандартной процедуры Delphi `AssignPrn()`, в которой принтеру назначается переменная типа `TextFile`, благодаря чему любой текст, запи-

сываемый в эту переменную, автоматически выводится на принтер. Завершив вывод на печать, следует обязательно разорвать подключение к принтеру с помощью процедуры `CloseFile()`.

Интересно также использование процедуры API Win32 `ShellExecute()` для запуска программы просмотра файла, отображаемого в окне списка. Эта процедура позволяет вызывать не только выполняемые программы, но и приложения, связанные с определенными расширениями файлов. Например, если с помощью процедуры `ShellExecute()` попытаться вызвать файл с расширением `.pas`, то для его просмотра будет автоматически загружена интегрированная среда разработки Delphi.

СОВЕТ

Если процедура `ShellExecute()` возвращает значение, указывающее на наличие ошибки, приложение вызывает функцию `RaiseLastWin32Error()`, которая расположена в модуле `SysUtils`. Данная функция возвращает более подробную информацию об ошибке. Она формирует эту информацию в строку и вызывает функцию API `GetLastError()` и функцию Delphi `SysErrorMessage()`. Если необходимо, чтобы пользователи получали подробные сообщения об ошибках, связанных со сбоями в работе интерфейса API, используйте в своих приложениях функцию `RaiseLastWin32Error()`.

Поток поиска

Реализация механизма поиска содержится в модуле `SrchU.pas`, который представлен в листинге 5.9. Этот модуль выполняет массу интересных вещей, включая копирование целого файла в строку, рекурсивный проход каталогов и передачу информации в главную форму.

Листинг 5.9. Модуль `SrchU.pas`

```
unit SrchU;

interface

uses Classes, StdCtrls;

type
  TSearchThread = class(TThread)
  private
    LB: TListbox;
    CaseSens: Boolean;
    FileNames: Boolean;
    Recurse: Boolean;
    SearchStr: string;
    SearchPath: string;
    FileSpec: string;
    AddStr: string;
    FSearchFile: string;
    procedure AddToList;
    procedure DoSearch(const Path: string);
```

```
    procedure FindAllFiles(const Path: string);
    procedure FixControls;
    procedure ScanForStr(const FName: string;
                         var FileStr: string);
    procedure SearchFile(const FName: string);
    procedure SetSearchFile;
  protected
    procedure Execute; override;
  public
    constructor Create(CaseS, FName, Rec: Boolean;
                       const Str, SPath, FSpec: string);
    destructor Destroy; override;
  end;

implementation

uses SysUtils, StrUtils, Windows, Forms, Main;

constructor TSearchThread.Create(CaseS, FName, Rec: Boolean;
                                 const Str, SPath, FSpec: string);
begin
  CaseSens := CaseS;
  FileNames := FName;
  Recurse := Rec;
  SearchStr := Str;
  SearchPath := AddBackSlash(SPath);
  FileSpec := FSpec;
  inherited Create(False);
end;

destructor TSearchThread.Destroy;
begin
  FSearchFile := '';
  Synchronize(SetSearchFile);
  Synchronize(FixControls);
  inherited Destroy;
end;

procedure TSearchThread.Execute;
begin
  FreeOnTerminate := True;     // освободить все поля
  LB := MainForm.lbFiles;
  Priority := TThreadPriority(MainForm.SearchPri);
  if not CaseSens then SearchStr := UpperCase(SearchStr);
  FindAllFiles(SearchPath);     // обработать текущий каталог
  if Recurse then DoSearch(SearchPath); // Если каталог,
end;                                    // то рекурсия.

procedure TSearchThread.FixControls;
{ Разрешает работу элементов управления в основной форме. Следует
вызывать через Synchronize }
begin
  MainForm.EnableSearchControls(True);
```

```
end;

procedure TSearchThread.SetSearchFile;
{ Обновляет имя файла в строке состояния. Следует вызывать через
Synchronize }
begin
  MainForm.StatusBar.Panels[1].Text := FSearchFile;
end;

procedure TSearchThread.AddToList;
{ Добавляет строку в основной список. Следует вызывать через Syn-
chronize }
begin
  LB.Items.Add(AddStr);
end;

procedure TSearchThread.ScanForStr(const FName: string;
                                   var FileStr: string);
{ Ищет подстроку SearchStr в строке FileStr файла FName. }
var
  Marker: string[1];
  FoundOnce: Boolean;
  FindPos: integer;
begin
  FindPos := Pos(SearchStr, FileStr);
  FoundOnce:= False;
  while (FindPos <> 0) and not Terminated do begin
    if not FoundOnce then begin
      // Использовать ":", если пользователь не выбрал
      // флажок "File Names Only" (Только имена файлов).
      if FileNames then Marker := ''
      else Marker := ':';
      { Добавить файл в список }
      AddStr := Format('File %s%s', [FName, Marker]);
      Synchronize(AddToList);
      FoundOnce := True;
    end;
    // Если нужны только имена, то не искать ту же самую строку в
    // том же самом файле }
    if FileNames then Exit;
    { Если нужны только имена, то добавить строку }
    AddStr := GetCurLine(FileStr, FindPos);
    Synchronize(AddToList);
    FileStr := Copy(FileStr, FindPos + Length(SearchStr),
                    Length(FileStr));
    FindPos := Pos(SearchStr, FileStr);
  end;
end;

procedure TSearchThread.SearchFile(const FName: string);
{ Ищет строку SearchStr в файле FName. }
var
  DataFile: THandle;
```

```
  FileSize: Integer;
  SearchString: string;
begin
  FSearchFile := FName;
  Synchronize(SetSearchFile);
  try
    DataFile := FileOpen(FName, fmOpenRead or fmShareDenyWrite);
    if DataFile = 0 then raise Exception.Create('');
    try
      { установить длину искомой строки }
      FileSize := GetFileSize(DataFile, nil);
      SetLength(SearchString, FileSize);
      { копировать содержимое файла в строку }
      FileRead(DataFile, Pointer(SearchString)^, FileSize);
    finally
      CloseHandle(DataFile);
    end;
    if not CaseSens then SearchString := UpperCase(SearchString);
    ScanForStr(FName, SearchString);
  except
    on Exception do begin
      AddStr := Format('Error reading file: %s', [FName]);
      Synchronize(AddToList);
    end;
  end;
end;

procedure TSearchThread.FindAllFiles(const Path: string);
{ Ищет в подкаталогах пути файлы, отвечающие спецификации }
var
  SR: TSearchRec;
begin
  { Поиск первого файла, соответствующего спецификации }
  if FindFirst(Path + FileSpec, faArchive, SR) = 0 then
  try
    repeat
      SearchFile(Path + SR.Name);            // обработка файла
    until (FindNext(SR) <> 0) or Terminated; // найти след. файл
  finally
    SysUtils.FindClose(SR);                  // освободить память
  end;
end;

procedure TSearchThread.DoSearch(const Path: string);
{ Рекурсивно обрабатывает дерево подкаталогов, начиная с пути Path.
}
var
  SR: TSearchRec;
begin
  { Просмотр каталогов }
  if FindFirst(Path + '*.*', faDirectory, SR) = 0 then
  try
    repeat
```

```
      { Если это каталог и нет '.' или '..' то... }
      if ((SR.Attr and faDirectory) <> 0) and (SR.Name[1] <> '.')
      and not Terminated then begin
        FindAllFiles(Path + SR.Name + '\');  // обработка каталога
        DoSearch(Path + SR.Name + '\');      // рекурсия
      end;
    until (FindNext(SR) <> 0) or Terminated; // Найти следующий
  finally                                    // каталог.
    SysUtils.FindClose(SR);                  // освободить память
  end;
end;

end.
```

При создании этого потока сначала вызывается метод `FindAllFiles()`, который использует функции `FindFirst()` и `FindNext()` для поиска в текущем каталоге всех файлов, соответствующих маске, заданной пользователем. Если пользователь установил флажок Recurse subdirs (Включая вложенные папки), то для исследования дерева подкаталогов вызывается метод `DoSearch()`. Данный метод пользуется услугами методов `FindFirst()` и `FindNext()`, но на этот раз — для обнаружения каталогов, причем весь фокус заключается в том, что для операций внутри дерева подкаталогов метод `DoSearch()` выполняет рекурсию, т.е. вызывает сам себя. При обнаружении каждого каталога вызывается процедура `FindAllFiles()`, которая обрабатывает все файлы, отвечающие условиям поиска.

СОВЕТ

Алгоритм рекурсии, используемый процедурой `DoSearch()`, является стандартным методом прохода дерева каталогов. Как известно, рекурсивные алгоритмы представляют определенную трудность для отладки, поэтому опытные программисты стараются использовать проверенные, заведомо работающие алгоритмы. Учитывая вышесказанное, рекомендуем сохранять этот метод, чтобы впоследствии можно было применить его в других приложениях.

Обратите внимание: при поиске заданной лексемы внутри файла используется объект `TMemMapFile`, который в системе Win32 инкапсулирует файл, отображенный в память. Более подробная информация об этом объекте приведена в предыдущем издании, в главе 12 — "Работа с файлами", но вполне достаточно знать, что он предоставляет собой простой способ отображения содержимого любого файла в память. Полный алгоритм работает следующим образом.

1. При обнаружении методом `FindAllFiles()` файла, отвечающего заданной спецификации, вызывается метод `SearchFile()` и его содержимое копируется в строку.
2. Для каждого файла-строки вызывается метод `ScanForStr()`, который осуществляет в строке поиск вхождения искомой подстроки.
3. Если вхождение подстроки обнаружено, то в список, отображаемый на форме, добавляется имя файла и/или строка текста. Строка текста добавляется только

в том случае, если пользователь не устанавливал флажок File Names Only (Только имена файлов).

Обратите внимание, все методы объекта `TSearchThread` периодически проверяют состояние флагов `StopIt` (который устанавливается при остановке потока) и `Terminated` (который устанавливается при завершении работы с объектом `TThread`).

СОВЕТ

Помните, что все методы объекта `TThread`, которые используют пользовательский интерфейс приложения, должны вызываться с помощью метода `Synchronize()`. Альтернативный способ модификации пользовательского интерфейса заключается в посылке сообщений.

Настройка приоритета

Проект `DelSrch` позволяет динамически настраивать приоритет потока поиска. Используемая для этого форма показана на рис. 5.7, а реализующий ее модуль `PriU.pas` представлен в листинге 5.10.

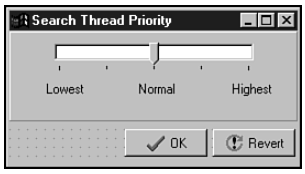

Рис. 5.7. *Форма установки приоритета проекта* `DelSrch`

Листинг 5.10. Модуль `PriU.pas`

```
unit PriU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, Buttons, ExtCtrls;

type
  TThreadPriWin = class(TForm)
    tbrPriTrackBar: TTrackBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    btnOK: TBitBtn;
    btnRevert: TBitBtn;
    Panel1: TPanel;
    procedure tbrPriTrackBarChange(Sender: TObject);
    procedure btnRevertClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure FormShow(Sender: TObject);
```

```
    procedure btnOKClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Закрытые объявления }
    OldPriVal: Integer;
  public
    { Открытые объявления }
  end;

var
  ThreadPriWin: TThreadPriWin;

implementation

{$R *.DFM}

uses Main, SrchU;

procedure TThreadPriWin.tbrPriTrackBarChange(Sender: TObject);
begin
  with MainForm do begin
    SearchPri := tbrPriTrackBar.Position;
    if Running then SearchThread.Priority :=
                      TThreadPriority(tbrPriTrackBar.Position);
  end;
end;

procedure TThreadPriWin.btnRevertClick(Sender: TObject);
begin
  tbrPriTrackBar.Position := OldPriVal;
end;

procedure TThreadPriWin.FormClose(Sender: TObject;
var
  Action: TCloseAction);
begin
  Action := caHide;
end;

procedure TThreadPriWin.FormShow(Sender: TObject);
begin
  OldPriVal:= tbrPriTrackBar.Position;
end;

procedure TThreadPriWin.btnOKClick(Sender: TObject);
begin
  Close;
end;

procedure TThreadPriWin.FormCreate(Sender: TObject);
begin
  tbrPriTrackBarChange(Sender); // инициализация приоритета потока
end;
```

```
end.
```

Этот модуль довольно прост. Он устанавливает значение переменной `SearchPri`, которое соответствует позиции на линейке с ползунком, находящейся на главной форме проекта. Если поток уже запущен, то его приоритет устанавливается тем же способом. Но поскольку объект `TThreadPriority` имеет перечислимый тип, то простая операция приведения типа преобразует значения от `1` до `5` (возвращаемые линейкой с ползунком) в элементы перечисления объекта `TThreadPriority`.

Многопоточный доступ к базе данных

Несмотря на то что создание приложений баз данных обсуждается позднее, рассмотрим в этом разделе некоторые вопросы, касающиеся использования многопоточности в контексте доступа к информации баз данных. Если читатель не знаком с основами программирования подобных приложений в среде Delphi, то, прежде чем приступить к изучению настоящего раздела, стоит просмотреть главу, посвященную данной теме.

Пожалуй, наиболее важным требованием к разработчикам приложений баз данных в системе Win32 является умение обеспечивать выполнение сложных запросов и хранимых процедур в фоновых потоках. К счастью, этот тип функций поддерживается 32-разрядным процессором баз данных *Borland* (BDE — Borland Database Engine) и довольно легко реализуется в Delphi.

К запуску в фоновом потоке запроса, выполняемого, например, с помощью компонента `TQuery`, предъявляются только два требования.

- Каждый поток запроса должен располагаться в собственном сеансе. Чтобы обеспечить объект `TQuery` собственным сеансом, следует разместить в форме компонент `TSession` и присвоить его имя свойству `SessionName` объекта `TQuery`. Т.е. для каждого сеанса объекта `TQuery` необходим отдельный экземпляр компонента `TDatabase`.
- Компонент `TQuery` не должен быть связан с какими бы то ни было компонентами `TDataSource` в тот момент, когда запрос открывается из вторичного потока. Если же запрос присоединен к компоненту `TDataSource`, то это должно быть реализовано в контексте основного (первичного) потока. Компонент `TDataSource` используется только для подключения наборов данных к элементам управления пользовательского интерфейса, а управление пользовательским интерфейсом должно осуществляться только в основном потоке.

Для иллюстрации методов реализации фоновых запросов в собственных потоках подготовлен демонстрационный проект `BDEThrd`, главная форма которого показана на рис. 5.8. С помощью этой формы можно задать псевдоним базы данных, имя пользователя, пароль для входа в конкретную базу данных, а также ввести сам запрос. При щелчке на кнопке Go! запускается вторичный поток процесса обработки запроса, а результаты его работы отображаются в дочерней форме проекта.

Дочерняя форма `TQueryForm` показана на рис. 5.9. Обратите внимание: эта форма содержит по одному экземпляру каждого из компонентов — `TQuery`, `TDatabase`,

TSession, TDataSource и TDBGrid. Следовательно, каждый экземпляр TQueryForm имеет собственные экземпляры этих компонентов.

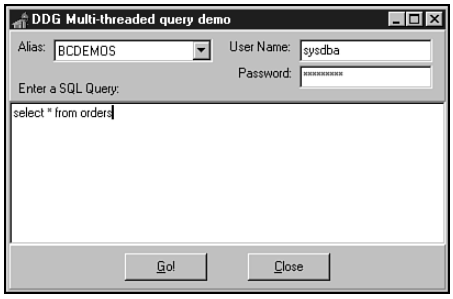

Рис. 5.8. *Главная форма демонстрационного проекта BDEThrd*

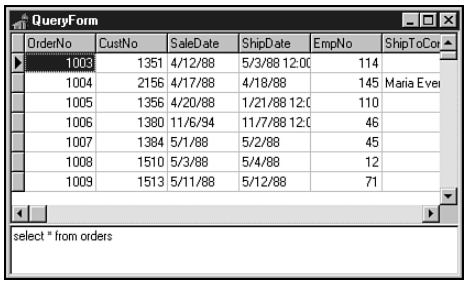

Рис. 5.9. *Дочерняя форма запроса демонстрационного проекта BDEThrd*

Листинг 5.11 содержит код главного модуля приложения Main.pas.

Листинг 5.11. Главный модуль Main.pas проекта BDEThrd

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, StdCtrls, ExtCtrls;

type
  TMainForm = class(TForm)
    pnlBottom: TPanel;
    pnlButtons: TPanel;
    GoButton: TButton;
    Button1: TButton;
    memQuery: TMemo;
    pnlTop: TPanel;
    Label1: TLabel;
    AliasCombo: TComboBox;
    Label3: TLabel;
    UserNameEd: TEdit;
    Label4: TLabel;
    PasswordEd: TEdit;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure GoButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;
```

```
var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses QryU, DB, DBTables;

var
  FQueryNum: Integer = 0;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.GoButtonClick(Sender: TObject);
begin
  Inc(FQueryNum);   // уникальный номер запроса
  { Формирование нового запроса }
  NewQuery(FQueryNum, memQuery.Lines, AliasCombo.Text,
           UserNameEd.Text, PasswordEd.Text);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Заполнение раскрывающегося списка псевдонимами BDE }
  Session.GetAliasNames(AliasCombo.Items);
end;

end.
```

Как можно заметить, этот модуль невелик по размеру. В обработчике события главной формы `OnCreate` комбинированный список (компонент `TComboBox`) `AliasCombo` заполняется псевдонимами BDE с помощью метода `GetAliasNames()` класса `TSession`. Обратите внимание: когда обработчик щелчка на кнопке Go! выполняет новый запрос (вызов процедуры `NewQuery()`, расположенной во втором модуле `QryU.pas`), этой процедуре при каждом щелчке на кнопке передается новый уникальный номер запроса `FQueryNum`. Этот номер используется для создания уникальных имен сеанса и базы данных для каждого потока запроса.

В листинге 5.12 содержится код модуля `QryU.pas`.

Листинг 5.12. Модуль `QryU.pas`

```
unit QryU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Grids, DBGrids, DB, DBTables, StdCtrls;

type
  TQueryForm = class(TForm)
    Query: TQuery;
    DataSource: TDataSource;
    Session: TSession;
    Database: TDatabase;
    dbgQueryGrid: TDBGrid;
    memSQL: TMemo;
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

procedure NewQuery(QryNum: integer; Qry: TStrings; const Alias,
                   UserName, Password: string);

implementation

{$R *.DFM}

type
  TDBQueryThread = class(TThread)
  private
    FQuery: TQuery;
    FDataSource: TDataSource;
    FQueryException: Exception;
    procedure HookUpUI;
    procedure QueryError;
  protected
    procedure Execute; override;
  public
    constructor Create(Q: TQuery; D: TDataSource); virtual;
  end;

constructor TDBQueryThread.Create(Q: TQuery; D: TDataSource);
begin
  inherited Create(True);        // создать приостановленный поток
  FQuery := Q;                   // установка параметров
  FDataSource := D;
  FreeOnTerminate := True;
  Resume;                        // Да здравствует поток!
end;
```

```
procedure TDBQueryThread.Execute;
begin
  try
    FQuery.Open;              // открыть запрос
    Synchronize(HookUpUI);    // обновить UI из основного потока
  except
    FQueryException := ExceptObject as Exception;
    Synchronize(QueryError); // Предупреждение из основного потока
  end;
end;

procedure TDBQueryThread.HookUpUI;
begin
  FDataSource.DataSet := FQuery;
end;

procedure TDBQueryThread.QueryError;
begin
  Application.ShowException(FQueryException);
end;

procedure NewQuery(QryNum: integer; Qry: TStrings; const Alias,
                   UserName, Password: string);
begin
  { Создать новую форму Query, чтобы показать результаты запроса }
  with TQueryForm.Create(Application) do begin
    { Установить уникальное имя сеанса }
    Session.SessionName := Format('Sess%d', [QryNum]);
    with Database do begin
      { Установить уникальное имя базы данных }
      DatabaseName := Format('DB%d', [QryNum]);
      { Установить параметр псевдонима }
      AliasName := Alias;
      { Подключить базу данных к сеансу }
      SessionName := Session.SessionName;
      { Имя пользователя и пароль }
      Params.Values['USER NAME'] := UserName;
      Params.Values['PASSWORD'] := Password;
    end;
    with Query do begin
      { Подключить запрос к базе данных и сеансу}
      DatabaseName := Database.DatabaseName;
      SessionName := Session.SessionName;
      { Установить строку запроса }
      SQL.Assign(Qry);
    end;
    { Отобразить строку запроса в окне SQL Memo }
    memSQL.Lines.Assign(Qry);
    { Отобразить форму запроса }
    Show;
    { Открыть запрос в его собственном потоке }
    TDBQueryThread.Create(Query, DataSource);
```

```
  end;
end;

procedure TQueryForm.FormClose(Sender: TObject;
                               var Action: TCloseAction);
begin
  Action := caFree;
end;

end.
```

В процедуре `NewQuery()` создается новый экземпляр дочерней формы `TQueryForm`, устанавливаются свойства каждого из ее компонентов доступа к данным и присваиваются уникальные имена ее компонентам `TDatabase` и `TSession`. Свойство запроса `SQL` заполняется значениями строк, переданных в качестве параметра `Qry`, затем запускается поток запроса.

Код класса `TDBQueryThread` не требуется большого объема. Конструктор просто устанавливает несколько переменных экземпляра, а сам запрос открывается в методе `Execute()`, откуда с помощью метода `Synchronize()` вызывается процедура `HookupUI()` для привязки запроса к источнику данных. Следует также обратить внимание на блок `try..except` внутри процедуры `Execute()`, которая использует метод `Synchronize()` для отображения сообщений об исключениях в контексте основного потока.

Многопоточная графика

Ранее уже отмечалось, что компоненты библиотеки VCL не предназначены для одновременной работы с несколькими потоками, однако это не совсем так. Подпрограммы библиотеки VCL способны поддерживать многопоточный режим для отдельных графических объектов. Благодаря новым методам `Lock()` и `Unlock()` класса `TCanvas` был создан целый модуль `Graphics`, который обеспечивает поддержку многопоточности. Он включает в себя такие классы, как `TCanvas`, `TPen`, `TBrush`, `TFont`, `TBitmap`, `TMetafile`, `TPicture` и `TIcon`.

НА ЗАМЕТКУ

Любое окно графического интерфейса Windows (не только высокого уровня, но и обычные кнопки), обладает двумя обязательными элементами: границей (border) и поверхностью (canvas). Окна высокого уровня могут иметь заголовок, содержащий название окна и кнопки управления. Таким образом, вся поверхность любого окна, за исключением границ и заголовка, представляет собой объект `Canvas`, на котором прорисовано все содержимое окна. В переводе с английского canvas означает “холст” или “канва”, а поскольку поверхность создаваемой формы в окне конструктора форм покрыта маркерами разметки и напоминает ткань, ее называли “тканью”.

Программная реализация методов `Lock()` имеет элементы, похожие и на критические секции, и на функцию API `EnterCriticalSection()` (описанную в этой главе ранее). Они предназначены для защиты доступа к объекту `Canvas` или графическим объектам. После того как конкретный поток вызовет метод `Lock()`, ему предос-

тавляется право монополного владения объектом Canvas или графическими объектами. Другие потоки, ожидающие выполнения кода, расположенного за обращением к методу Lock(), будут переведены в состояние ожидания. Оно продлится до тех пор, пока поток, владеющий критическим разделом, не вызовет метод Unlock(). Чтобы освободить критический раздел и разрешить следующему ожидающему потоку (если таковой имеется) выполнить защищенную часть кода, метод Unlock() вызывает, в свою очередь, функцию LeaveCriticalSection(). На примере следующих строк показано, как использовать эти методы для управления доступом к объекту Canvas:

```
Form.Canvas.Lock;
// здесь находится код, манипулирующий объектом Canvas
Form.Canvas.Unlock;
```

В листинге 5.13 представлен модуль Main проекта MTGraph — приложения, в котором демонстрируется доступ нескольких потоков к поверхности формы (объекту Canvas).

Листинг 5.13. Модуль Main.pas проекта MTGraph

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Menus;

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    Options1: TMenuItem;
    AddThread: TMenuItem;
    RemoveThread: TMenuItem;
    ColorDialog1: TColorDialog;
    Add10: TMenuItem;
    RemoveAll: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure AddThreadClick(Sender: TObject);
    procedure RemoveThreadClick(Sender: TObject);
    procedure Add10Click(Sender: TObject);
    procedure RemoveAllClick(Sender: TObject);
  private
    ThreadList: TList;
  public
    { Открытые объявления }
  end;

  TDrawThread = class(TThread)
  private
    FColor: TColor;
    FForm: TForm;
```

```
  public
    constructor Create(AForm: TForm; AColor: TColor);
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
{ TDrawThread }

constructor TDrawThread.Create(AForm: TForm; AColor: TColor);
begin
  FColor := AColor;
  FForm := AForm;
  inherited Create(False);
end;

procedure TDrawThread.Execute;
var
  P1, P2: TPoint;

  procedure GetRandCoords;
  var
    MaxX, MaxY: Integer;
  begin
    // Инициализировать точки P1 и P2 случайными координатами
    // в пределах границ формы
    MaxX := FForm.ClientWidth;
    MaxY := FForm.ClientHeight;
    P1.x := Random(MaxX);
    P2.x := Random(MaxX);
    P1.y := Random(MaxY);
    P2.y := Random(MaxY);
  end;

begin
  FreeOnTerminate := True;
  // Поток выполняется, пока он или приложение не будут завершены
  while not (Terminated or Application.Terminated) do begin
    GetRandCoords;           // инициализировать P1 и P2
    with FForm.Canvas do begin
      Lock;                  // Блокировать объект Canvas
      // только один поток может выполнять следующий код:
      Pen.Color := FColor;   // установить цвет пера
      MoveTo(P1.X, P1.Y);    // переход к точке P1
      LineTo(P2.X, P2.Y);    // нарисовать линию до точки P2
      // после выполнения следующий строки другому потоку будут
      // разрешено войти в вышеупомянутый блок кода
      Unlock;                // Разблокировать объект Canvas
    end;
```

```
  end;
end;

{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
begin
  ThreadList:= TList.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  RemoveAllClick(nil);
  ThreadList.Free;
end;

procedure TMainForm.AddThreadClick(Sender: TObject);
begin
  // Добавить новый поток в список и разрешить пользователю
  // выбрать цвет
  if ColorDialog1.Execute then
     ThreadList.Add(TDrawThread.Create(Self, ColorDialog1.Color));
end;

procedure TMainForm.RemoveThreadClick(Sender: TObject);
begin
  // завершить последний поток в списке и удалить его
  TDrawThread(ThreadList[ThreadList.Count - 1]).Terminate;
  ThreadList.Delete(ThreadList.Count - 1);
end;

procedure TMainForm.Add10Click(Sender: TObject);
var
  i: Integer;
begin
  // создать 10 потоков, каждый с произвольным цветом
  for i := 1 to 10 do
      ThreadList.Add(TDrawThread.Create(Self, Random(MaxInt)));
end;

procedure TMainForm.RemoveAllClick(Sender: TObject);
var
  i: Integer;
begin
  Cursor := crHourGlass;
  try
    for i := ThreadList.Count - 1 downto 0 do begin
      TDrawThread(ThreadList[i]).Terminate; // завершить поток
      TDrawThread(ThreadList[i]).WaitFor;   // Убедиться, что
    end;                                    // поток завершился.
    ThreadList.Clear;
  finally
    Cursor:= crDefault;
```

```
  end;
end;

initialization
  Randomize;  // инициализация генератора случайных чисел
end.
```

В этом приложении предусмотрено главное меню, содержащее четыре пункта (рис. 5.10). Пункт Add Thread (Добавить поток) создает новый экземпляр класса `TDrawThread`, который рисует прямые линии, случайным образом расположенные на главной форме. Этот пункт можно выбирать снова и снова, вливая тем самым свежие струи в коктейль потоков, уже получивших доступ к поверхности главной формы. При выборе следующего пункта, Remove Thread (Удалить поток), удаляется поток, который был добавлен последним. С помощью третьей команды, Add 10 (Добавить 10), создается десять новых экземпляров `TDrawThread`. И, наконец, четвертая команда, Remove All (Удалить все), завершает и удаляет все экземпляры `TDrawThread`. На рис. 5.10 также показан результат работы десяти потоков, которые одновременно рисуют на поверхности формы.

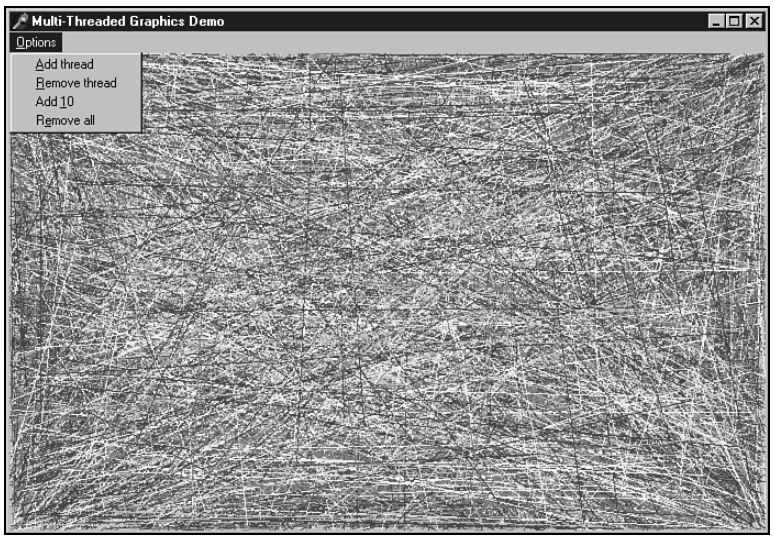

Рис. 5.10. *Главная форма проекта `MTGraph`*

Правило блокировки объекта `Canvas` гласит, что каждый пользователь будет блокировать его перед началом рисования и отменять блокировку по окончании. Таким образом потоки не будут мешать друг другу. Обратите внимание: все события `OnPaint` и вызовы метода `Paint()`, инициированные VCL, автоматически блокируют и отменяют блокировку объекта `Canvas` автоматически, поэтому обычный код Delphi может сосуществовать с графическими операциями новых фоновых потоков.

Используя это приложение в качестве примера, рассмотрим сценарий возможного конфликта потоков, возникающего при невыполнении блокировки объекта `Canvas`. Если поток 1 устанавливает красный цвет пера и рисует прямую, а поток 2 устанавливает

синий цвет пера и рисует окружность, причем эти потоки не блокируют объект Canvas перед началом выполнения своих операций, то возможен следующий сценарий конфликта потоков. Поток 1 установит красный цвет пера. *Планировщик операционной системы* (OS scheduler) передаст управление потоку 2. Поток 2 установит синий цвет пера и нарисует окружность. Эстафета выполнения кода снова переходит к потоку 1. Поток 1 рисует прямую. Но эта прямая оказывается не красной, а синей, поскольку потоку 2 удалось "втиснуться" со своей установкой синего цвета между операциями потока 1.

Заметим, что для возникновения подобных проблем достаточно иметь хотя бы один поток-нарушитель. Если поток 1 будет исправно блокировать объект Canvas, а поток 2 — нет, то только что описанного сценария избежать не удастся. Для предотвращения конфликта потоков необходимо, чтобы оба потока блокировали объект Canvas на время выполнения операций над ним.

Внеприоритетный поток

Внеприоритетный поток (fiber) — это своего рода поток, расписанием периодов выполнения которого приходится управлять самостоятельно. Подобно обычным потокам, внеприоритетный поток обладает информацией о состоянии и контексте исполнения форм, своим собственным стеком и доступом к регистрам процессора. Но, в отличие от обычного потока, расписание периодов выполнения внеприоритетного потока не контролируется операционной системой. Вся ответственность за переключение процесса выполнения между несколькими внеприоритетными потоками лежит на разработчике. В процессе разработки приложения применение внеприоритетных потоков редко оказывается предпочтительнее многопоточной архитектуры. Исключение составляют те редкие случаи, когда можно получать определенные преимущества от использования нескольких независимых стеков или необходим прямой доступ к регистрам процессора в обход всех этих сложностей, связанных с синхронизацией обычных потоков.

НА ЗАМЕТКУ

Внеприоритетные потоки допустимы для Windows NT 3.51 SP3 и выше, Windows 2000, Windows XP, Windows 98 и Windows ME.

Внеприоритетные потоки разработаны так, чтобы они выполнялись внутри контекста обычного потока, поэтому один обычный поток может содержать несколько внеприоритетных. Прежде чем можно будет использовать внеприоритетные потоки внутри обычного, его тоже необходимо преобразовать во внеприоритетный поток с помощью функции API ConvertThreadToFiber(). Эта функция определена в модуле Windows следующим образом:

```
function ConvertThreadToFiber(lpParameter:
                              Pointer): BOOL; stdcall;
```

Единый параметр (lone parameter), lpParameter, типа Pointer позволяет передать 32-битовый указатель на данные, специфические для конкретного внеприоритетного потока. Этот подход аналогичен применяемому при передаче данных обычному потоку с помощью функций BeginThread() и CreateThread(). В модуле Windows тип возвращаемого значения определен неправильно. Там он указан как BOOL,

но фактически возвращается указатель на объект внеприоритетного потока. Поэтому, прежде чем использовать возвращаемое значение, необходимо осуществить его приведение к необходимому типу.

Как только поток будет преобразован во внеприоритетный, появится возможность создать другие внеприоритетные потоки и расписание периодов их выполнения. Чтобы создать дополнительный внеприоритетный поток, применяется функция API `CreateFiber()`, которая определена в модуле `Windows` следующим образом:

```
function CreateFiber(dwStackSize: DWORD;
                     lpStartAddress: TFNFiberStartRoutine;
                     lpParameter: Pointer): BOOL; stdcall;
```

Параметр `dwStackSize` определяет исходный размер (в байтах) стека внеприоритетного потока. Нулевое значение этого параметра равнозначно требованию установить размер стека, принятый по умолчанию. Параметр `lpStartAddress` определяет адрес процедуры, которую внеприоритетный поток должен выполнить при запуске. `lpParameter` определяет все 32-битовые данные, специфические для внеприоритетного потока, которые необходимо ему передавать. Тип значения, возвращаемого этой функцией, как и у `ConvertThreadToFiber()`, указан неправильно. В действительности, возвращается указатель на созданный объект внеприоритетного потока, и перед его применением необходимо осуществить приведение типа.

Создав несколько внеприоритетных потоков, можно переключаться между ними используя функцию API `SwitchToFiber()`. Эта функция определена в модуле `Windows` таким образом:

```
function SwitchToFiber(lpFiber: Pointer): BOOL; stdcall;
```

Вызов этого метода с указателем на объект внеприоритетного потока в параметре `lpFiber` — вот и все, что необходимо сделать для перехода из контекста исполнения одного внеприоритетного потока к другому. Операционная система сама отработает все внутренние задачи, связанные с переключением контекста исполнения (такие, как изменение регистраторов процессора и указателя вершины стека). Возвращаемое значение этой функции опять неправильно определено как `BOOL`, но на самом деле эта функция должна была бы быть определена как процедура, поскольку она не возвращает никакого значения. Следовательно, от этой функции не стоит ожидать никакого допустимого возвращаемого значения.

Когда приходит время завершать работу внеприоритетного потока, достаточно передать указатель на его объект функции API `DeleteFiber()`:

```
function DeleteFiber(lpFiber: Pointer): BOOL; stdcall;
```

Между прочим, подобно остальным, возвращаемое значение и этой функции определено неправильно, она тоже должна была быть процедурой, потому что не возвращает никакого допустимого значения.

СОВЕТ

Вызов функции `DeleteFiber()` для выполняющегося в настоящий момент внеприоритетного потока приведет к вызову функции `ExitThread()`, которая завершает весь поток. Поэтому, если не планируется завершение главного потока, используйте функ-

цию `DeleteFiber()` лишь для ожидающих (приостановленных) внеприоритетных потоков.

Четыре вышеописанные функции позволяют выполнить бо́льшую часть действий, необходимых для работы с внеприоритетными потоками. Кроме того, в *файлах заголовков Win32* (Win32 header files) определены некоторые дополнительные вспомогательные функции и типы, отсутствующие в Delphi, но очень полезные и удобные. В листинге 5.14 представлен код модуля `Fiber`, который содержит дополнительные определения, отсутствующие в модуле `Windows`.

Листинг 5.14. Модуль `Fiber.pas`

```
unit Fiber;

interface

uses Windows;

// Определение типов для внеприоритетного потока и процедуры
// start из файла заголовка winbase.h:
type
  PFIBER_START_ROUTINE = procedure (lpFiberParameter: Pointer);
                                                      stdcall;
  LPFIBER_START_ROUTINE = PFIBER_START_ROUTINE;
  TFiberFunc = PFIBER_START_ROUTINE;

function GetCurrentFiber: Pointer;
function GetFiberData: Pointer;

implementation

// Специфические для процессоров X86 встраиваемые функции
// из файла заголовка winnt.h:

function GetCurrentFiber: Pointer;
asm
  mov eax, fs:[$10]
end;

function GetFiberData: Pointer;
asm
  mov eax, fs:[$10]
  mov eax, [eax]
end;

end.
```

В данном примере продемонстрировано практическое применение внеприоритетных потоков. Здесь создается несколько внеприоритетных потоков и осуществляется переключение между ними, что позволяет одновременно выполнять несколько действий. Основная форма этого приложения представлена на рис. 5.11.

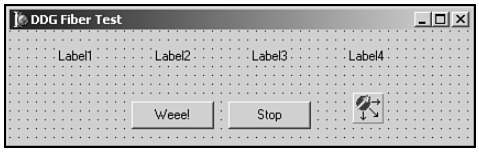

Рис. 5.11. *Главная форма проекта* `FibTest`

Главный модуль этой формы представлен в листинге 5.15.

Листинг 5.15. Модуль `FibMain.pas` проекта `FibTest`

```
unit FibMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, AppEvnts;

type
  TForm1 = class(TForm)
    BtnWee: TButton;
    BtnStop: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    AppEvents: TApplicationEvents;
    procedure BtnWeeClick(Sender: TObject);
    procedure AppEventsMessage(var Msg: tagMSG;
                               var Handled: Boolean);
    procedure BtnStopClick(Sender: TObject);
  private
    { Закрытые объявления }
    FThreadID: LongWord;
    FThreadHandle: Integer;
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

uses Fibers;

{$R *.dfm}

const
  DDG_THREADMSG = WM_USER;
```

```
var
  FFibers: array[0..3] of Pointer;
  StopIt: Boolean;

procedure FiberFunc(Param: Pointer); stdcall;
var
  J, FibNum, NextNum: Integer;
  I: Cardinal;
  Fiber: Pointer;
begin
  try
    I := 0;
    FibNum := 1;            // подавить предупреждение компилятора
    // сохранить, на будущее, указатель на внеприоритетный поток
    Fiber := GetCurrentFiber;
    // выяснить место текущего внеприоритетного потока в массиве
    // и сохранить, на будущее
    for J := Low(FFibers) to High(FFibers) do
      if FFibers[J] = Fiber then begin
        FibNum := J;
        Break;
      end;
    while not StopIt do begin
      // Послать число основному потоку, чтобы вывести на экран
      // очередную сотню циклов
      if I mod 100 = 0 then
        PostMessage(Application.Handle, DDG_THREADMSG,
                    Integer(GetFiberData), I);
      // переключать внеприоритетные потоки через каждые
      // 1000 циклов
      if I mod 1000 = 0 then begin
        if FibNum = High(FFibers) then NextNum := Low(FFibers)
        else NextNum := FibNum + 1;
        SwitchToFiber(FFibers[NextNum]);
      end;
      Inc(I);
    end;
  except
    // удавить все необработанные исключения
  end;
end;

function ThreadFunc(Param: Pointer): Integer;
var
  I: Integer;
begin
  Result := 0;
  // Преобразовать этот поток во внеприоритетный
  FFibers[0] := Pointer(ConvertThreadToFiber(Pointer(1)));
  // создать другие внеприоритетные потоки
  FFibers[1] := Pointer(CreateFiber(0, @FiberFunc, Pointer(2)));
  FFibers[2] := Pointer(CreateFiber(0, @FiberFunc, Pointer(3)));
  FFibers[3] := Pointer(CreateFiber(0, @FiberFunc, Pointer(4)));
```

```
  // вступить в игру
  FiberFunc(Pointer(1));
  // когда все закончится, удалить все внеприоритетные потоки
  // удалить текущий внеприоритетного поток, вызвав ExitThread
  for I := High(FFibers) downto Low(FFibers) do
      DeleteFiber(FFibers[I]);
end;

procedure TForm1.BtnWeeClick(Sender: TObject);
begin
  BtnWee.Enabled := False; // двойной щелчек приведет к блокировке
  FThreadHandle := BeginThread(nil, 0, @ThreadFunc, nil, 0,
                               FThreadID);
end;

procedure TForm1.AppEventsMessage(var Msg: tagMSG;
                                  var Handled: Boolean);
begin
  if Msg.message = DDG_THREADMSG then begin
    // wParam укажет, какой из внеприоритетных потоков послал
    // сообщение, а следовательно, которую из надписей необходимо
    // изменить
    case Msg.wParam of
      1: Label1.Caption := IntToStr(Msg.lParam);
      2: Label2.Caption := IntToStr(Msg.lParam);
      3: Label3.Caption := IntToStr(Msg.lParam);
      4: Label4.Caption := IntToStr(Msg.lParam);
    end;
    Handled := True;
  end;
end;

procedure TForm1.BtnStopClick(Sender: TObject);
begin
  StopIt := True;
end;

end.
```

Наиболее интересные события этого примера происходят в функции `ThreadFunc()`, которая является функцией вторичного потока, созданного при щелчке на кнопке. Здесь осуществляется вызов функции `ConvertThreadToFiber()`, преобразующей обычный поток во внеприоритетный, затем, несколько раз подряд, происходит вызов функции `CreateFiber()`, что позволяет создать три дополнительных внеприоритетных потока. Когда все внеприоритетные потоки будут подготовлены, выполняется функция `FiberFunc()`, которая представляет собой не более чем бесконечный цикл `For`, от `0` до бесконечности посылающий сообщения через каждые 100 циклов, чтобы отобразить очередное значение в надписях формы, и через каждые 1000 циклов, чтобы переключиться на следующий внеприоритетный поток.

Приложение использует простую и надежную методику связи вторичных потоков с основным, основанную на передаче сообщений дескриптору окна `Application`.

Каждому внеприоритетному потоку присвоен номер от 1 и 4. По этому номеру обработчик сообщения в основном потоке определяет, какой именно из внеприоритетных потоков послал сообщение.

Рис. 5.12 демонстрируют приложение FibTest в действии. Тот факт, что числа в надписях формы близки по значению но не одинаковы, доказывает, что каждый из внеприоритетных потоков при выполнении использует свой собственный стек.

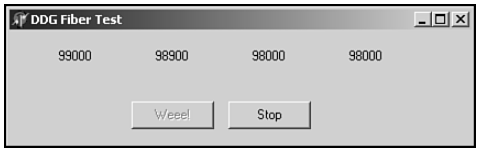

Рис. 5.12. *Проекта FibTest в действии*

Резюме

В этой главе рассматривались потоки и правила работы с ними в среде Delphi, а также описывались методы синхронизации нескольких потоков и поддержки связи между вторичными потоками и главным потоком приложения Delphi. Кроме того, здесь были продемонстрированы примеры использования потоков в контексте реальных приложений поиска файлов и работы с базой данных и дано общее представление о возможности рисования в форме с одновременным использованием нескольких потоков. И, в заключение, было показано все изящество внеприоритетных потоков, которые позволяют самостоятельно управлять периодами их выполнения. В главе 6, “Динамически компонуемые библиотеки”, содержится подробная информация о создании и применении DLL в Delphi.

ГЛАВА

6

Динамически компонуемые библиотеки

В ЭТОЙ ГЛАВЕ...

Эта глава посвящена *динамически компонуемым библиотекам* Win32, или DLL (dynamic link libraries). Они являются основным компонентом любого приложения Windows. В настоящей главе рассматриваются некоторые аспекты создания и использования библиотек DLL. Вначале содержатся обзор принципов работы DLL и обсуждение способов их создания и применения. Затем рассматриваются различные методы их загрузки и привязки к тем процедурам и функциям, которые они экспортируют. Кроме того, в этой главе уделяется внимание *функциям обратного вызова* (callback functions) и совместному использованию библиотек DLL различными вызывающими процессами.

Что такое библиотека DLL?

Динамически компонуемые библиотеки — это программные модули, содержащие код, данные или ресурсы, которые могут совместно использоваться несколькими приложениями Windows. Одно из основных назначений библиотек DLL — позволить приложениям загружать участки кода во время выполнения (динамически), а не компоновать их в само приложение в процессе компиляции (статически). Как следствие, один и тот же код, содержащийся в библиотеке DLL, смогут одновременно использовать несколько приложений. Так, файлы библиотек `Kernel32.dll`, `User32.dll` и `GDI32.dll` являются теми тремя китами, на которые опирается система Win32. Файл `Kernel32.dll` (ядро), например, отвечает за управление памятью, процессами и потоками. Файл `User32.dll` содержит функции пользовательского интерфейса, необходимые для создания окон и обработки сообщений Win32. И, наконец, на файл `GDI32.dll` возложена работа с графикой. Существуют и другие системные библиотеками DLL, например `AdvAPI32.dll` и `ComDlg32.dll`, которые предназначены для обеспечения работы с системным реестром и диалоговыми окнами общего назначения.

Другое преимущество использования библиотек DLL заключается в том, что приложение становится модульным. Это упрощает процесс его обновления, поскольку при необходимости обновляется не все приложение полностью, а только определенные библиотеки. Типичным примером может служить среда операционной системы Windows. При каждой установке любого нового устройства достаточно установить новую библиотеку DLL, содержащую драйвер, с помощью которого это устройство может общаться с Windows. Преимущество модульности станет очевидным, если представить необходимость повторной инсталляции Windows при установке в систему каждого нового устройства.

С точки зрения файловой системы, библиотеки DLL практически ничем не отличаются от исполняемых файлов Windows (EXE). Разница состоит лишь в том, что файл библиотеки DLL не является *независимым* исполняемым файлом, хотя может содержать исполняемый код. Чаще всего файлы библиотек DLL имеют расширение `.dll`. Но могут встречаться и другие: `.drv` — для драйверов устройств, `.sys` — для системных файлов, `.fon` — для файлов ресурсов шрифтов, которые не содержат исполняемого кода.

НА ЗАМЕТКУ

В Delphi используются специальные библиотеки, называемые *пакетами*. Они применяются не только в среде Delphi, но и в среде Borland C++ Builder. Более подробная информация о пакетах приведена в главе 14, “Пакеты”.

Библиотеки DLL способны использовать свой код совместно с другими приложениями благодаря процессу, называемому *динамической компоновкой* (dynamic linking), который рассматривается в этой главе далее. Как правило, когда какое-либо приложение использует библиотеку DLL, система Win32 гарантирует, что в памяти будет размещена только одна копия этой библиотеки. Для этого применяется *файл, отображенный в память* (memory-mapped file). Суть этого метода заключается в том, что библиотека DLL сначала загружается в глобальную *распределяемую память* (heap) системы Win32, а затем отображается на адресное пространство вызывающего процесса. В системе Win32 каждому процессу выделяется собственное непрерывное 32-разрядное адресное пространство. Поэтому, когда одна и та же библиотека DLL загружается сразу несколькими процессами, каждый из них получает собственный *образ*[1] (image) данной библиотеки. Следовательно, процессы не используют одновременно один и тот же физический код, данные или ресурсы, как это было в 16-разрядной Windows. В системе Win32 работа организована так, что библиотека DLL становится как бы реальным кодом, принадлежащим вызывающему процессу. Более подробная информация о работе системы Win32 приведена в предыдущем издании *Delphi 5 Руководство разработчика*, в главе 3 – "Win32 API".

Вышесказанное вовсе не означает, что, когда несколько процессов загружает одну и ту же библиотеку DLL, физическая память расходуется на хранение всех необходимых ее копий. Образ DLL размещается в адресном пространстве процессов как *отображение* системной распределяемой памяти в адресное пространство каждого процесса, использующего эту DLL, – по крайней мере, в идеале.

Установка предпочтительного базового адреса библиотеки DLL

Совместный доступ нескольких процессов к коду библиотеки DLL возможен только в том случае, если эту библиотеку можно загрузить в адресное пространство процессов всех заинтересованных клиентов по некоторому *предпочтительному базовому адресу* (preferred base address) библиотеки DLL. Если предпочтительный базовый адрес, с учетом размера библиотеки DLL, перекрывается каким-либо другим объектом, уже размещенным в памяти процесса, то загрузчик Win32 должен будет изменить расположение всего образа библиотеки DLL, используя другой базовый адрес. В этом случае ни один из перемещенных образов DLL не может использоваться никакими другими процессами в системе, т.е. каждый перемещенный экземпляр библиотеки DLL занимает участок собственной физической памяти и собственную часть пространства файла подкачки.

Важно установить базовый адрес каждой библиотеки DLL равным такому значению, которое не конфликтует и не перекрывается другими адресными диапазонами, используемыми в создаваемом приложении. Для этого служит директива компилятора `$IMAGEBASE`.

Если библиотека DLL предназначена для использования несколькими приложениями, то необходимо выбрать уникальный базовый адрес, который с минимальной вероятностью будет перекрываться адресами приложения в области младших значений диапазона виртуальных адресов или общих библиотек DLL (например пакетов VCL) в области старших адресов диапазона. По умолчанию базовый адрес всех исполняемых файлов (с расширениями `.EXE` и `.DLL`) равен `$400000`, а это означает, что если не изменить базовый адрес конкретной библиотеки DLL, то конфликт с базовым адресом

[1] *Копию, экземпляр.* – Прим. ред.

ее главного файла EXE будет неизбежен и она никогда не будет совместно использоваться всеми заинтересованными процессами.
В применении базового адреса есть еще один положительный момент. Поскольку библиотека DLL не требует изменения расположения или внесения исправлений (что обычно и происходит) и сохраняется на локальном диске, страницы ее памяти отображаются прямо в файл DLL на диске. Код DLL не занимает никакого пространства в системном *файле подкачки* (page file) (он же файл подкачки страниц (swap file), он же файл страничного обмена). Вот почему общее количество и размер выгружаемых системой на диск страниц может намного превышать размеры системного файла подкачки и объем RAM.
Более подробная информация об использовании директивы `$IMAGEBASE` приведена в разделе "Image Base Address" интерактивной справочной системы Delphi 6.

Прежде чем продолжить обсуждение данной темы, необходимо уточнить некоторые термины.

- *Приложение* (application). Программа Windows, размещенная в файле с расширением `.exe`.
- *Исполняемый* файл (executable). Файл, содержащий исполняемый код. Исполняемые файлы имеют расширения `.dll` и `.exe`.
- *Экземпляр* (instance). Когда речь идет о приложениях или библиотеках DLL, термин *экземпляр* означает отдельный *процесс* выполнения исполняемого файла. Каждому экземпляру соответствует *дескриптор экземпляра* (instance handle), который назначается системой Win32. Например, когда приложение запускается дважды, в памяти образуется два экземпляра данного приложения и, следовательно, два дескриптора экземпляра. Но при загрузке библиотеки DLL создается только один экземпляр этой библиотеки и, соответственно, один дескриптор экземпляра. Используемый здесь термин *экземпляр* не следует путать с экземпляром класса.
- *Модуль* (module). В 32-разрядной Windows (в отличие от 16-разрядной) термины *модуль* и *экземпляр* могут использоваться как синонимы[2]. В 16-разрядной Windows для управления модулями существует специальная база данных, в которой для каждого модуля хранится свой дескриптор. В 32-разрядной Windows каждый экземпляр приложения получает собственное адресное пространство, следовательно, нет нужды в образовании отдельного идентификатора модуля. Но корпорация *Microsoft* в своей документации по-прежнему использует этот термин. Поэтому необходимо просто иметь в виду, что модуль и экземпляр означают одно и то же.
- *Задача* (task). Система Windows является многозадачной средой (или средой с переключением задач). Она должна обладать способностью распределять системные ресурсы и время между различными экземплярами приложений, работающих под ее управлением. Для этого система поддерживает базу данных задач, в которой хранятся дескрипторы экземпляров и другая информация, позволяющая системе переключаться между задачами. Таким образом, задача — это элемент, для которого Windows выделяет ресурсы и периоды времени выполнения.

[2] *Речь идет о загружаемом модуле (module), а не о модуле программы (unit).* – Прим. ред.

Статическая компановка против динамической

Статической компоновкой (static linking) называется метод, с помощью которого компилятор Delphi реализует в исполняемом коде вызов функций и процедур. Код функций может храниться в файле приложения с расширением `.dpr` или в любом другом модуле, но при компоновке приложений эти функции и процедуры становятся частью исполняемого файла. Другими словами, каждая функция будет занимать на диске определенное место в составе исполняемого файла (`.exe`) программы.

Расположение функции заранее определено относительно места, занимаемого в памяти программой. Любые обращения к этой функции вызывают передачу управления непосредственно по адресу начала расположения функции. Далее эта функция выполняется и по завершении возвращает управление в то место, откуда она была вызвана. Относительный адрес функции вычисляется во время процесса компоновки.

Это весьма упрощенное описание действительно сложного процесса, используемого компилятором Delphi для статической компоновки. Но эта книга не о тонкостях работы компилятора и не о внутренней структуре выполнения программ. Для эффективного использования DLL в приложении такие подробности не так уж и важны.

НА ЗАМЕТКУ

В Delphi реализован *интеллектуальный компоновщик* (smart linker), автоматически удаляющий функции, процедуры, переменные и типизированные константы, на которые нет ссылок в окончательном варианте проекта. Следовательно, расположенные в больших модулях функции, не используемые в данном проекте, никогда не становятся частью создаваемого исполняемого файла.

Предположим, существуют два приложения, которые используют одну и ту же функцию, находящуюся в некотором модуле. В обоих приложениях имя данного модуля, безусловно, будет указано в разделе `uses`. Если одновременно запустить оба приложения в среде Windows, то код этой функции будет загружен в память дважды. В случае запуска третьего приложения в памяти появится и третий экземпляр этой функции, общий расход памяти также будет втрое бо́льшим. Приведенный выше пример иллюстрирует одну из главных причин использования динамической компоновки. Если такую функцию расположить в библиотеке DLL, то при ее загрузке в память одним приложением все остальные приложения, которым потребуется ссылка на нее, смогут использовать ее код с помощью отображения образа данной библиотеки DLL на адресное пространство их собственных процессов. В результате, данная функция будет существовать в памяти только в одном экземпляре — теоретически.

При *динамической компоновке* (dynamic linking) связь между вызовом функции и исполняемым кодом устанавливается во время его выполнения с помощью внешней ссылки на конкретную функцию библиотеки DLL. Подобные ссылки могут быть объявлены в самом приложении, но обычно они размещаются в отдельном модуле `import`. В этом модуле объявляются импортируемые функции и процедуры, а также определяются различные типы данных, используемые функциями библиотек DLL.

Предположим, например, что библиотека DLL по имени `MaxLib.dll` содержит следующую функцию:

```
function Max(I1, I2: integer): integer;
```

Эта функция возвращает большее из двух переданных ей целых чисел. Типичный модуль import для данной функции будет выглядеть таким образом:

```
unit MaxUnit;
interface
  function Max(I1, I2: integer): integer;
implementation
  function Max; external 'MAXLIB';
end.
```

Можно заметить, что, хотя эта функция внешне напоминает типичный модуль, в ней нет определения функции Max(). Ключевое слово external просто указывает на то, что данная функция находится в файле, имя которого приведено сразу за этим ключевым словом. Для использования указанного модуля приложению достаточно иметь в списке раздела uses его имя, т.е. MaxUnit. При выполнении приложения библиотека DLL загружается в память автоматически, и любые обращения к функции Max() связываются с функцией Max(), находящейся в этой библиотеке.

Этот пример иллюстрирует один из двух способов загрузки библиотек DLL, называемый *неявной загрузкой* (implicit loading). Он требует от Windows автоматически загружать библиотеку DLL при загрузке приложения. Второй способ называется *явной загрузкой* (explicitly load) DLL и будет рассматриваться в настоящей главе далее.

Зачем нужны библиотеки DLL?

Существует несколько причин использования библиотек DLL, часть из которых уже упоминалась выше. Как правило, такие библиотеки применяются либо для совместного использования кода и системных ресурсов, либо для сокрытия реализации программного кода или системных функций низкого уровня, либо для реализации пользовательских элементов управления. Эти темы и рассматриваются в последующих разделах.

Совместное использование кода, ресурсов и данных несколькими приложениями

Ранее уже упоминалось, что самой распространенной причиной создания библиотек DLL является совместное использование кода. В отличие от модулей, которые обеспечивают возможность совместно использовать исходный код в различных приложениях Delphi, библиотеки DLL позволяют совместно использовать один и тот же исполняемый код любым приложениям Windows, способным вызывать эти функции из библиотек DLL.

Кроме того, такие библиотеки позволяют совместно использовать растровые изображения, шрифты, пиктограммы и других ресурсы, которые обычно входят в состав файла ресурсов и непосредственно связываются с создаваемым приложением. Если эти ресурсы разместить в библиотеке DLL, то многие приложения смогут воспользоваться ими, не затрачивая дополнительной памяти, необходимой для загрузки дополнительных экземпляров таких ресурсов.

В 16-разрядной Windows библиотеки DLL использовали собственный сегмент данных, поэтому все приложения, которые обращались к определенной библиотеке DLL, получали доступ к одним и тем же глобальным данным и статическим переменным. В системе Win32 дело обстоит иначе. Поскольку образ библиотеки DLL отображается на адресное пространство каждого процесса, все данные, определенные функциями в DLL, принадлежат отдельному процессу. При этом стоит подчеркнуть одну деталь: несмотря на то что данные библиотек DLL не используются совместно различными *процессами*, они могут совместно использоваться несколькими *потоками* внутри одного и того же процесса. А поскольку потоки выполняются независимо друг от друга, необходимо предпринимать меры предосторожности, позволяющие избежать конфликтов при доступе к глобальным данным библиотек DLL.

Указанное вовсе не означает, что нет способов заставить несколько процессов совместно использовать данные, доступные через библиотеки DLL. Один из таких способов заключается в создании внутри библиотеки DLL общей области памяти (с помощью отображенного в память файла). В этом случае каждое приложение, использующее такую библиотеку, получает возможность прочитать данные, хранящиеся в общей области памяти. Более подробная информация по этой теме приведена в настоящей главе далее.

Сокрытие реализации

Иногда возникает потребность в сокрытии деталей реализации программ, доступных из библиотеки DLL. Для этого существует множество причин. Библиотеки DLL позволяют сделать функции доступными пользователям, не раскрывая их исходный код. Следует только подготовить модуль интерфейса, позволяющий другим пользователям получить доступ к создаваемой библиотеке. В принципе, подобная возможность уже давно реализована механизмом *откомпилированных модулей Delphi* (DCU — Delphi Compiled Unit), но необходимо заметить, что файл DCU может использоваться лишь другими приложениями Delphi, причем созданными компилятором той же самой версии. А вот формат библиотек DLL не зависит от использованного языка программирования, поэтому созданные в Delphi библиотеки можно применять в приложениях, написанных на C++, Visual Basic или любом другом языке, который поддерживает работу с файлами DLL.

Модуль `Windows` является модулем интерфейса с библиотекой DLL Win32. В комплект поставки Delphi 6 включены исходные файлы модуля API Win32. Среди них есть файл `Windows.pas`, содержащий исходный код модуля `Windows`. В разделе `interface` файла `Windows.pas` нетрудно отыскать определения функций, подобных следующему:

```
function ClientToScreen(Hwnd: HWND;
                        var lpPoint: TPoint): BOOL; stdcall;
```

В разделе `implementation` (реализации) присутствует соответствующая связь с библиотекой DLL:

```
function ClientToScreen; external user32 name 'ClientToScreen';
```

Эта строка означает, что функция `ClientToScreen()` находится в динамически связываемой библиотеке `User32.dll` и ее имя `ClientToScreen`.

Создание и использование библиотек DLL

В этом разделе демонстрируется процесс создания реальной библиотеки DLL в среде Delphi. Рассмотрим, как создается модуль интерфейса, позволяющий сделать библиотеку DLL доступной другим программам. Но прежде чем перейти к более сложным вопросам использования библиотек в Delphi, давайте познакомимся с методами размещения в библиотеке DLL форм Delphi.

Подсчет пенсов (пример простой DLL)

В данном примере проиллюстрировано размещение в библиотеке DLL функции, которую очень любят предлагать студентам профессора компьютерных наук. Эта функция рассчитывает минимальное количество монет достоинством в 1 пенс, 5 пенсов (nickel), 10 пенсов (dime) и 25 пенсов (quarter), чтобы получить заданную сумму.

Простейшая DLL

Создаваемая библиотека содержит метод `PenniesToCoins()`. В листинге 6.1 приведен законченный проект библиотеки DLL.

Листинг 6.1. `PenniesLib.dpr` — DLL для пересчета пенсов в другие монеты

```
library PenniesLib;
{$DEFINE PENNIESLIB}
uses
  SysUtils, Classes, PenniesInt;

function PenniesToCoins(TotPennies: word;
                        CoinsRec: PCoinsRec): word; StdCall;

begin
  Result := TotPennies;  // Присвоение значения Result
  { Вычисление количества монет по 25, 10, 5, и 1 пенсу }
  with CoinsRec^ do begin
    Quarters   := TotPennies div 25;
    TotPennies := TotPennies - Quarters * 25;
    Dimes      := TotPennies div 10;
    TotPennies := TotPennies - Dimes * 10;
    Nickels    := TotPennies div 5;
    TotPennies := TotPennies - Nickels * 5;
    Pennies    := TotPennies;
  end;
end;

{ Экспорт функции по имени }
```

```
exports
  PenniesToCoins;
end.
```

Обратите внимание, что в этой библиотеке используется модуль PenniesInt, который более подробно рассматривается чуть ниже.

Раздел exports указывает, какие функции и процедуры данной библиотеки DLL экспортируются и станут доступными вызывающим приложениям.

Определение модуля интерфейса

Модуль интерфейса позволяет пользователям созданной DLL статически импортировать функции данной библиотеки в свои приложения, разместив имя импортируемого модуля в разделе uses. Модули интерфейса позволяют разработчику библиотеки DLL определять общие структуры, используемые как самой библиотекой, так и вызывающим приложением. Продемонстрируем эти возможности на примере модуля интерфейса для библиотеки PenniesLib.dll. В листинге 6.2 содержится исходный код раздела interface модуля PenniesInt.pas.

Листинг 6.2. PenniesInt.pas — модуль интерфейса PenniesLib.dll

```
unit PenniesInt;
{ Процедуры интерфейса для PENNIES.DLL }

interface
type
  { Эта запись будет содержать перечень монет после выполнения
    преобразований. }
  PCoinsRec = ^TCoinsRec;
  TCoinsRec = record
    Quarters,
    Dimes,
    Nickels,
    Pennies: word;
  end;

{$IFNDEF PENNIESLIB}
{ Объявление функции с использованием ключевого слова export }

function PenniesToCoins(TotPennies: word;
                        CoinsRec: PCoinsRec): word; StdCall;
{$ENDIF}

implementation

{$IFNDEF PENNIESLIB}
{ Определение импортируемой функции }
function PenniesToCoins;
                  external 'PENNIESLIB.DLL' name 'PenniesToCoins';
{$ENDIF}
```

```
end.
```

В разделе type этого проекта объявлена запись TCoinsRec, а также указатель на нее. Такая запись будет содержать сведения о количестве монет каждого достоинства, необходимых для вычисления суммы в пенсах, переданной функции PenniesToCoins(). Этой функции передается два параметра: общая сумма денег в пенсах и указатель на переменную TCoinsRec. Результатом выполнения функции является та же самая денежная сумма, которая была ей передана в качестве параметра, но уже в виде некоторого набора монет.

В модуле PenniesInt.pas объявляется функция, которую библиотека PenniesLib.dll экспортирует в своем разделе interface. Определение функции находится в разделе implementation (реализации). Это определение означает, что функция является внешней и расположена в файле DLL PenniesLib.dll. Связь с функцией DLL устанавливается по имени функции. Обратите внимание, что здесь была использована директива $IFNDEF PENNIESLIB, применяемая для выполнения условной компиляции объявления функции PenniesToCoins(). Это сделано потому, что нет необходимости компоновать данное объявление при компиляции модуля интерфейса библиотеки. Указанное позволяет совместно использовать объявления типа из модуля интерфейса как другим библиотекам, так и любым другим приложениями, которые будут использовать эту библиотеку. Любые изменения в структуру, используемую как библиотекой, так и приложениями, нужно вносить только в модуле интерфейса.

СОВЕТ

Чтобы определить директиву условной компиляции в масштабе приложения, укажите ее во вкладке Directories/Conditionals диалогового окна Options проекта. После этого следует обязательно заново создать данный проект, чтобы вступили в силу изменения, связанные с условными директивами, поскольку в логике команды Make не предусмотрено переоценки условных определений.

НА ЗАМЕТКУ

Следующее определение показывает один из двух возможных способов импорта функции DLL:

```
function PenniesToCoins; external 'PENNIESLIB.DLL' index 1;
```

Этот метод называется *импортом по порядковому номеру* (importing by ordinal). Импортировать функции DLL можно также и другим способом, который имеет название *импорт по имени* (importing by name):

```
function PenniesToCoins; external 'PENNIESLIB.DLL' name 'PenniesToCoins';
```

Нетрудно догадаться, что метод определения связываемой функции по имени использует ее имя, указанное после ключевого слова name.

При использовании импорта по порядковому номеру уменьшается время загрузки библиотеки DLL, поскольку в этом случае отпадает необходимость поиска имени функции в таблице имен. Тем не менее, в Win32 предпочтение отдается импорту по

имени. В таком случае приложению удается избежать нежелательной жесткой привязки к конкретному месторасположению точек входа в функции DLL, которое может измениться

при обновлении библиотеки. При импорте по порядковому номеру программа привязана к определенному месту в библиотеке DLL, а при импорте по имени — к имени функции, независимо от ее расположения в библиотеке.

Если рассматриваемая выше библиотека была бы реальной DLL, которую предполагалось использовать и в дальнейшем, то следовало бы предоставить пользователям оба файла – и `PenniesLib.dll`, и `PenniesInt.pas`. Тогда они смогли бы работать с данной библиотекой, определяя в модуле `PenniesInt.pas` типы и функции, необходимые для `PenniesLib.dll`. Кроме того, программисты, работающие на других языках (например C++), могли бы преобразовать модуль `PenniesInt.pas` в формат своего языка, что позволило бы им использовать данную библиотеку DLL в своей среде разработки. Проект, работающий с библиотекой `PenniesLib.dll`, можно найти на прилагаемом компакт-диске (см. также `www.williamspublishing.com`).

Отображение модальных форм из DLL

В настоящем разделе рассматривается разработка модальной формы, доступной в библиотеке DLL. Одна из причин, по которым имеет смысл размещать используемые формы в библиотеке DLL, заключается в возможности применения таких форм в других приложениях Windows и даже в другой среде разработки (например в C++ или в Visual Basic).

Для этого, прежде всего, необходимо удалить будущую форму DLL из списка автоматически создаваемых форм.

Ранее уже была создана подходящая форма, в главном окне которой содержится компонент `TCalendar`. Вызывающее приложение будет обращаться к функции DLL, вызывающей данную форму. Когда пользователь выберет в календаре день, в вызывающее приложение возвращается соответствующая дата.

В листинге 6.3 приведен исходный код файла проекта DLL `CalendarLib.dpr`, а в листинге 6.4 – исходный код модуля самой формы DLL `DllFrm.pas`. Оба эти файла служат иллюстрацией метода инкапсуляции формы в библиотеке DLL.

Листинг 6.3. Исходный код проекта `CalendarLib.dpr`

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
    procedure calDllCalendarDblClick(Sender: TObject);
  end;

{ Объявление экспортируемой функции }
```

```
function ShowCalendar(AHandle: THandle;
                      ACaption: String): TDateTime; StdCall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle;
                      ACaption: String): TDateTime;
var
  DLLForm: TDllForm;
begin
  // Копирование дескриптора приложения в объект Tapplication
  // библиотеки DLL
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  try
    DLLForm.Caption := ACaption;
    DLLForm.ShowModal;
    // Передача даты обратно в Result
    Result := DLLForm.calDLLCalendar.CalendarDate;
  finally
    DLLForm.Free;
  end;
end;

procedure TDLLForm.calDllCalendarDblClick(Sender: TObject);
begin
  Close;
end;

end.
```

Главная форма в этой библиотеке DLL входит в состав экспортируемой функции. Обратите внимание, что объявление `DLLForm` было удалено из раздела `interface` и перенесено в код самой функции.

В рассматриваемой функции DLL свойству `Application.Handle` был присвоен параметр `AHandle`. Напомним, что все проекты Delphi, в том числе и проекты DLL, содержат глобальный объект `Application`. В функции библиотеки DLL этот объект отличается от объекта `Application`, который существует в вызывающем приложении. Чтобы форма DLL правильно работала в качестве модальной формы вызывающего приложения, необходимо присвоить дескриптор такого вызывающего приложения свойству `Application.Handle` функции библиотеки DLL, что и было продемонстрировано. Без этого поведение формы DLL было бы некорректным, особенно в случае ее минимизации. Кроме того, следует удостовериться, что в качестве владельца формы DLL не было передано значение `nil`.

После создания формы DLL ее свойству `Caption` присваивается строка `ACaption`. Затем форма отображается в модальном режиме. Когда форма закрывается, дата, выбранная пользователем в компоненте `TCalendar`, передается обратно в вызывающую функцию. Форма закрывается по двойному щелчку на компоненте `TCalendar`.

СОВЕТ

Если создаваемая библиотека DLL экспортирует какие либо процедуры или функции, которые получают в качестве параметров или возвращают в виде результатов выполнения строки или динамические массивы, то первым элементом раздела `uses` этой библиотеки и проекта должен быть модуль `ShareMem` (меню View пункт Project Source). Это относится ко всем строкам, передаваемым или получаемым от библиотек DLL, даже если они входят в состав записей или классов. Модуль `ShareMem` представляет собой модуль интерфейса для библиотеки `Borlndmm.dll` (диспетчера распределяемой памяти), который необходимо использовать вместе с этой библиотекой. Чтобы обойтись без модуля `Borlndmn.dll`, передавайте строковую информацию с помощью параметра типа `PChar` или `ShortString`.

Использование модуля `ShareMem` является единственным обязательным условием для организации передачи размещаемых в динамической памяти строк и динамических массивов из одного приложения в другое. При этом также передается и право собственности на память, занимаемую строками. После приведения внутренних строк к типу `Pchar` и последующей их передачи в другой модуль как параметра типа `PChar` право собственности на строковую память вызываемому модулю не передается, поэтому модуль интерфейса `ShareMem` не требуется.

Следует учесть, что модуль `ShareMem` применяется только к библиотекам DLL Delphi/C++Builder, которые передают строки или динамические массивы другим библиотекам Delphi/C++Builder или файлам EXE. Никогда не передавайте строки или динамические массивы Delphi (в качестве параметров или результатов выполнения экспортируемых функций библиотек DLL) в функции библиотек DLL или приложения, написанные на языке, отличном от Delphi, поскольку чужеродные приложения не смогут правильно освободить память после использования элементов среды Delphi.

Кроме того, модуль `ShareMem` не нужен приложениям, встроенными в пакеты. В этом случае диспетчер памяти распределяет память между членами пакета неявно.

Вот и все, что требуется знать для инкапсуляции модальной формы в функцию библиотеки DLL. В следующем разделе рассматривается помещение в библиотеку DLL немодальной формы.

Отображение немодальных форм из библиотек DLL

Для иллюстрации размещения в библиотеке DLL немодальных форм вновь воспользуемся формой с календарем из предыдущего раздела.

Для отображения из библиотеки немодальных форм такая DLL должна содержать две функции. На первую возлагается задача создания и отображения формы, а на вторую – ее освобождения. В листинге 6.4 представлен исходный код, иллюстрирующий вывод немодальной формы функциями DLL.

Листинг 6.4. Немодальная форма в библиотеке DLL

```
unit DLLFrm;
```

```
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
  end;

{ Объявление экспортируемых функций }
function ShowCalendar(AHandle: THandle;
                      ACaption: String): Longint; stdcall;
procedure CloseCalendar(AFormRef: Longint); stdcall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle;
                      ACaption: String): Longint;
var
  DLLForm: TDllForm;
begin
  // Копирование дескриптора приложения в объект TApplication
  // библиотеки DLL.
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  Result := Longint(DLLForm);
  DLLForm.Caption := ACaption;
  DLLForm.Show;
end;

procedure CloseCalendar(AFormRef: Longint);
begin
  if AFormRef > 0 then
    TDLLForm(AFormRef).Release;
end;

end.
```

Данный пример содержит два метода: `ShowCalendar()` и `CloseCalendar()`. Метод `ShowCalendar()` похож на одноименную функцию из примера с модальной формой тем, что дескриптор вызывающего приложения присваивается свойству `Handle` объекта приложения функции DLL, после чего создается сама форма. Но вместо вызова метода `ShowModal()` эта функция вызывает метод `Show()` и при этом не освобождает форму. Обратите внимание на то, что она возвращает значение типа `longint`, в роли которого выступает экземпляр `DLLForm`. Все дело в том, что ссылка на созданную форму должна быть сохранена, поэтому забота о сохранении ссылки на форму поручается вызывающему приложению. Подобная забота распространяется на все эк-

земпляры формы, связанные с любыми приложениями, вызывающими эту функцию библиотеки DLL и создающими собственные экземпляры данной формы.

В процедуре `CloseCalendar()` просто проверяется корректность ссылки на форму и вызывается ее метод `Release()`. Здесь вызывающее приложение должно передать обратно ту же самую ссылку, которая была ранее получена от функции `ShowCalendar()`.

При использовании такого подхода нужно иметь в виду, что библиотека DLL никогда не освобождает форму. Если предпринять подобную попытку (например попытаться вернуть значение `caFree` функции `CanClose()`), то обращение к функции `CloseCalendar()` приведет к возникновению аварийной ситуации.

Демонстрационные примеры модальной и немодальной форм находятся на прилагаемом компакт-диске (см. также `www.williamspublishing.com`).

Использование DLL в приложениях Delphi

Как уже отмечалось в этой главе, существует два способа загрузки или импортирования функций библиотек DLL — явный и неявный. В настоящем разделе оба способа будут продемонстрированы на примере созданных ранее библиотек DLL.

Первая библиотека DLL, созданная в данной главе, содержала модуль интерфейса (`PenniesInt.pas`). Воспользуемся этим модулем для иллюстрации неявного связывания с библиотекой DLL. Главная форма демонстрационного проекта содержит компоненты `TMaskEdit`, `TButton` и девять экземпляров компонента `TLabel`.

В данном приложении пользователь вводит сумму денег в пенсах. По щелчку на кнопке программа отображает введенную сумму в виде набора монет разного достоинства. Эта информация поступает от экспортируемой функции `PenniesToCoins()` библиотеки `PenniesLib.dll`.

Главная форма приложения определена в модуле `MainFrm.pas`, код которого приведен в листинге 6.5.

Листинг 6.5. Главная форма проекта

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Mask;

type

  TMainForm = class(TForm)
    lblTotal: TLabel;
    lblQlbl: TLabel;
    lblDlbl: TLabel;
    lblNlbl: TLabel;
```

```
    lblPlbl: TLabel;
    lblQuarters: TLabel;
    lblDimes: TLabel;
    lblNickels: TLabel;
    lblPennies: TLabel;
    btnMakeChange: TButton;
    meTotalPennies: TMaskEdit;
    procedure btnMakeChangeClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation
uses PenniesInt;  // Использование модуля интерфейса

{$R *.DFM}

procedure TMainForm.btnMakeChangeClick(Sender: TObject);
var
  CoinsRec: TCoinsRec;
  TotPennies: word;
begin
  { Вызов функции DLL для определения минимального количества
  монет, составляющих заданную сумму денег. }
  TotPennies := PenniesToCoins(StrToInt(meTotalPennies.Text),
                               @CoinsRec);
  with CoinsRec do begin
    { Теперь отобразить информацию о монетах }
    lblQuarters.Caption := IntToStr(Quarters);
    lblDimes.Caption    := IntToStr(Dimes);
    lblNickels.Caption  := IntToStr(Nickels);
    lblPennies.Caption  := IntToStr(Pennies);
  end
end;

end.
```

Обратите внимание: в модуле `MainFrm.pas` используется модуль `PenniesInt`. Напомним, что модуль `PenniesInt.pas` содержит внешние объявления для функций, помещенных в файл библиотеки `PenniesLib.dpr`. При запуске этого приложения система Win32 автоматически загружает библиотеку `PenniesLib.dpr` и отображает ее в адресное пространство процесса, вызывающего приложения.

Не обязательно использовать модуль `import`. Имя `PenniesInt` из раздела `uses` вполне можно удалить и вставить объявление `external` для функции `PenniesToCoins()` в раздел `implementation` модуля `MainFrm.pas`:

```
implementation

function PenniesToCoins(TotPennies: word;
 ChangeRec: PChangeRec): word; eStdCall external 'PENNIESLIB.DLL';
```

В модуле MainFrm.pas придется снова определить PChangeRec и TChangeRec (можно просто скомпилировать это приложение с указанием директивы компилятора PENNIESLIB). Такой метод может с успехом использоваться в тех случаях, когда необходим доступ лишь к некоторым функциям данной библиотеки. Но чаще всего оказывается, что требуются не только внешние объявления для функций библиотеки DLL, но и доступ к типам, определенным в ее разделе interface.

НА ЗАМЕТКУ

При использовании DLL стороннего производителя, модуля ее интерфейса на языке Pascal может и не оказаться. Обычно, вместо него поставляется библиотека импортируемых функций C/C++. В таком случае придется перевести эту библиотеку в эквивалентный модуль интерфейса Delphi.

Явная загрузка библиотек DLL

Несмотря на удобство метода неявной загрузки библиотек DLL, он не всегда бывает желательным. Предположим, что существует некоторая библиотека DLL, содержащая множество функций. Вполне вероятно, что в обычном режиме работы приложение никогда не вызовет ни одной из функций этой библиотеки. Получается, что при каждом запуске такого приложения загрузка данной библиотеки приводит к напрасным затратам памяти, особенно при использовании этим приложением сразу нескольких библиотек DLL. Другим примером может служить применение компонентов библиотек DLL для заполнения объектов, имеющих очень большой размер и содержащих списки весьма специализированных функций, предоставляемых на выбор для использования в конкретных ситуациях. (Например, списки доступных драйверов принтеров или подпрограмм преобразования формата файлов.) В такой ситуации имело бы смысл загружать каждую библиотеку DLL только по конкретному требованию, исходящему от приложения. Этот метод и называется *явной загрузкой* (explicitly loading) библиотек DLL.

Для иллюстрации явной загрузки вернемся к примеру библиотеки DLL с модальной формой. В листинге 6.6 содержится код главной формы приложения, демонстрирующего явную загрузку этой DLL. Данный файл можно найти на прилагаемом компакт-диске (см. также www.williamspublishing.com).

Листинг 6.6. Главная форма приложения Calendar DLL

```
unit MainFfm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  { Сначала определяется процедурный тип данных, который должен
  отражать процедуру, экспортируемую из библиотеки DLL. }
  TShowCalendar = function (AHandle: THandle;
                            ACaption: String): TDateTime; StdCall;
```

```
  { Создать новый класс исключения для отображения неудачной
  загрузки DLL. }
  EDLLLoadError = class(Exception);

  TMainForm = class(TForm)
    lblDate: TLabel;
    btnGetCalendar: TButton;
    procedure btnGetCalendarClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnGetCalendarClick(Sender: TObject);
var
  LibHandle   : THandle;
  ShowCalendar: TShowCalendar;
begin

  { Попытка загрузить DLL }
  LibHandle := LoadLibrary('CALENDARLIB.DLL');
  try
    { Если попытка не удалась, LibHandle содержит ноль.
      В этом случае передается исключение. }
    if LibHandle = 0 then
      raise EDLLLoadError.Create('Unable to Load DLL');
    { Если выполнение программы дойдет до этого места, значит, DLL
      загрузилась удачно. Теперь устанавливаем связь с
      экспортируемой функцией DLL, чтобы ее можно было вызывать. }
    @ShowCalendar := GetProcAddress(LibHandle, 'ShowCalendar');
    { Если функция импортируется успешно, устанавливаем свойство
      lblDate.Caption для отображения даты, возвращаемой этой
      функцией. В противном случае передается исключение. }
    if not (@ShowCalendar = nil) then
      lblDate.Caption := DateToStr(ShowCalendar
                                   (Application.Handle, Caption))
    else
      RaiseLastWin32Error;
  finally
    FreeLibrary(LibHandle); // Выгрузить DLL.
  end;
end;

end.
```

В начале этого модуля определяется процедурный тип данных TShowCalendar, который отображает определение функции, вызываемой из библиотеки Calendar-Lib.dll. Затем определяется специальное исключение, которое передается при возникновении проблем с загрузкой библиотеки DLL. Обратите внимание: в обработчике

событий btnGetCalendarClick() используются три функции API Win32: LoadLibrary(), FreeLibrary() и GetProcAddress().

Функция LoadLibrary() определена следующим образом:

```
function LoadLibrary(lpLibFileName: PChar): HMODULE; stdcall;
```

Эта функция загружает экземпляр библиотеки DLL, указанной переменной lpLibFileName, и отображает ее в адресное пространство вызывающего процесса. Если работа функции завершается успешно, она возвращает дескриптор экземпляра DLL, а в случае неудачи – нулевое значение, которое и является сигналом к передаче исключения. Более подробная информация о работе функции LoadLibrary() и возвращаемых ей кодах ошибок приведена в интерактивной справочной системе.

Функция FreeLibrary() определена следующим образом:

```
function FreeLibrary(hLibModule: HMODULE): BOOL; stdcall;
```

Эта функция уменьшает счетчик экземпляров библиотеки, заданной значением переменной hLibModule. Если счетчик экземпляров примет нулевое значение, то данная библиотека будет удалена из памяти. Счетчик экземпляров отслеживает количество задач, использующих данную библиотеку DLL.

Функция GetProcAddress() определена следующим образом:

```
function GetProcAddress(hModule: HMODULE;
                        lpProcName: LPCSTR): FARPROC; stdcall
```

Эта функция возвращает адрес функции внутри библиотечного модуля, заданного значением первого параметра hModule. Этот параметр имеет тип THandle, а его значение можно получить, обратившись к функции LoadLibrary(). В случае неуспешного завершения работы функции GetProcAddress() последняя возвращает значение nil. Чтобы получить более подробную информацию об ошибках, необходимо обратиться к функции API GetLastError().

В обработчике события Button1.OnClick функция LoadLibrary() вызывается для загрузки библиотеки CALDLL. При неудачном завершении загрузки передается исключение. В случае успешной загрузки будет выполнено обращение к функции окна GetProcAddress(), которая вернет адрес функции ShowCalendar(). Наличие символа оператора адреса (@) перед переменной процедурного типа ShowCalendar не позволяет компилятору выдать сообщение об ошибке несовпадения типов. Теперь, полученный адрес функции ShowCalendar() можно использовать в виде переменной типа TShowCalendar. И, наконец, внутри блока finally вызывается функция FreeLibrary(), гарантирующая, что загруженная библиотека будет удалена из памяти.

Таким образом, библиотека загружается и освобождается при каждом вызове данной функции. Если за все время работы приложения функция вызывается только один раз, то становится ясно, что явная загрузка может сэкономить ресурсы памяти, которых всегда так не хватает. Но если бы эта функция вызывалась достаточно часто, то повторение операций загрузки и выгрузки библиотеки DLL привело бы к существенной дополнительной нагрузке на систему.

Функция входа/выхода динамически компонуемых библиотек

В случае необходимости создаваемую DLL можно обеспечить и дополнительным кодом для входа и выхода, который может применяться при операциях инициализации и завершения работы. Эти операции могут выполняться во время инициализации и завершения потока либо процесса.

Функции инициализации и завершения процессов и потоков

К типичным операциям инициализации относятся: регистрация классов Windows, инициализация глобальных переменных и функции входа/выхода. Все указанные операции выполняются в методе входа в библиотеку DLL, который реализуется в виде функции `DLLEntryPoint`. На самом деле эта функция представлена блоком `begin..end` файла проекта DLL. Именно здесь целесообразно было бы поместить процедуру обработки входа/выхода. Такой процедуре должен передаваться один параметр типа `DWord`.

Глобальная переменная `DLLProc` представляет собой процедурный указатель, которому можно присвоить адрес процедуры входа/выхода. Сначала эта переменная имеет значение `nil`, сохраняемое до тех пор, пока в нее не будет помещен адрес некоторой процедуры. После установки адреса процедуры входа/выхода в программе появляется возможность реагировать на события, перечисленные в табл. 6.1.

Таблица 6.1. События входа/выхода библиотеки DLL

Событие	*Назначение*
`DLL_PROCESS_ATTACH`	Библиотека DLL присоединяется к адресному пространству текущего процесса при запуске процесса или в результате вызова функции `LoadLibrary()`. При обработке этого события в библиотеке DLL инициализируются экземпляры всех типов данных
`DLL_PROCESS_DETACH`	Библиотека DLL отсоединяется от адресного пространства вызывающего процесса. Эта ситуация возникает в результате полного завершения процесса или при обращении к функции `FreeLibrary()`. При обработке этого события в библиотеке DLL могут быть удалены экземпляры данных любых типов
`DLL_THREAD_ATTACH`	Это событие происходит, когда текущий процесс создает новый поток. В подобных случаях система вызывает функцию входа для любой библиотеки DLL, присоединенной к процессу. Такой вызов выполняется в контексте нового потока и может быть использован для размещения в памяти любых данных этого потока

Окончание табл. 6.1.

Событие	*Назначение*
DLL_THREAD_DETACH	Это событие происходит, когда работа потока завершается. В процессе обработки такого события в библиотеке DLL могут освобождаться любые инициализированные данные конкретного потока

НА ЗАМЕТКУ

В потоках, работа которых завершается ненормально (например, при вызове функции `TerminateThread()`), передача события `DLL_THREAD_DETACH` не гарантируется.

Пример функции входа/выхода

Листинг 6.7 демонстрирует назначение переменной `DLLProc` процедуры входа/выхода DLL.

Листинг 6.7. Исходный код файла `DllEntry.dpr`

```
library DLLEntryLib;
uses
  SysUtils, Windows, Dialogs, Classes;

procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: ShowMessage('Attaching to process');
    DLL_PROCESS_DETACH: ShowMessage('Detaching from process');
    DLL_THREAD_ATTACH:  MessageBeep(0);
    DLL_THREAD_DETACH:  MessageBeep(0);
  end;
end;

begin
  { Сначала назначить процедуру переменной DLLProc }
  DllProc := @DLLEntryPoint;
  { Теперь вызвать процедуру и продемонстрировать, что эта DLL
    присоединена к процессу. }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

Процедура входа/выхода назначается переменной `DLLProc` библиотеки DLL в блоке `begin..end` файла проекта самой библиотеки. Эта процедура, `DLLEntryPoint()`, проверяет значение параметра типа `word`, чтобы определить, какое событие явилось причиной ее вызова. Возможные события перечислены в табл. 6.1. Для большей наглядности загрузка, выгрузка этой библиотеки DLL, а также каждое событие в данном проекте будут сопровождаться выводом соответствующего окна сообще-

ния. При создании и удалении потока в вызывающем приложении одновременно с сообщением будет подан звуковой сигнал.

Для иллюстрации использования этой библиотеки DLL рассмотрим программный код, представленный в листинге 6.8.

Листинг 6.8. Пример кода процедуры входа/выхода библиотеки DLL

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, Gauges;

type

  { Определение потомка класса TThread }
  TTestThread = class(TThread)
    procedure Execute; override;
    procedure SetCaptionData;
  end;

  TMainForm = class(TForm)
    btnLoadLib: TButton;
    btnFreeLib: TButton;
    btnCreateThread: TButton;
    btnFreeThread: TButton;
    lblCount: TLabel;
    procedure btnLoadLibClick(Sender: TObject);
    procedure btnFreeLibClick(Sender: TObject);
    procedure btnCreateThreadClick(Sender: TObject);
    procedure btnFreeThreadClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    LibHandle   : THandle;
    TestThread  : TTestThread;
    Counter     : Integer;
    GoThread    : Boolean;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TTestThread.Execute;
begin
  while MainForm.GoThread do begin
    Synchronize(SetCaptionData);
```

```
    Inc(MainForm.Counter);
  end;
end;

procedure TTestThread.SetCaptionData;
begin
  MainForm.lblCount.Caption := IntToStr(MainForm.Counter);
end;

procedure TMainForm.btnLoadLibClick(Sender: TObject);
{ Эта процедура загружает библиотеку DllEntryLib.DLL }
begin
  if LibHandle = 0 then begin
    LibHandle := LoadLibrary('DLLENTRYLIB.DLL');
    if LibHandle = 0 then
      raise Exception.Create('Unable to Load DLL');
  end
  else
    MessageDlg('Library already loaded', mtWarning, [mbok], 0);
end;

procedure TMainForm.btnFreeLibClick(Sender: TObject);
{ Эта процедура освобождает библиотеку }
begin
  if not (LibHandle = 0) then begin
    FreeLibrary(LibHandle);
    LibHandle := 0;
  end;
end;

procedure TMainForm.btnCreateThreadClick(Sender: TObject);
{ Эта процедура создает экземпляр класса TThread. Если DLL
  загружена, то будет подан звуковой сигнал. }
begin
  if TestThread = nil then begin
    GoThread   := True;
    TestThread := TTestThread.Create(False);
  end;
end;

procedure TMainForm.btnFreeThreadClick(Sender: TObject);
{ При освобождении экземпляра TThread будет подан звуковой сигнал,
  если DLL загружена.}
begin
  if not (TestThread = nil) then begin
    GoThread   := False;
    TestThread.Free;
    TestThread := nil;
    Counter    := 0;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
```

```
begin
  LibHandle  := 0;
  TestThread := nil;
end;

end.
```

Данный проект состоит из главной формы с четырьмя кнопками (компонент TButton). По щелчку на кнопке BtnLoadLib загружается библиотека DllEntryLib.dll. Кнопка BtnFreeLib предназначена для удаления этой библиотеки из процесса. С помощью кнопки BtnCreateThread создается объект, класс которого является производным от класса TThread. Этот объект используется для создания нового потока. По щелчку на кнопке BtnFreeThread объект TThread удаляется. Метка lblCount используется только для демонстрации работы потока.

Обработчик события btnLoadLibClick() вызывает функцию LoadLibrary() для загрузки библиотеки DllEntryLib.dll. В результате, данная библиотека загружается и отображается в адресное пространство процесса. Кроме того, в библиотеке DLL выполняется подпрограмма инициализации, содержащаяся в блоке begin..end, которая запускает процедуру входа/выхода для этой библиотеки:

```
begin
  { Сначала назначить процедуру переменной DLLProc }
  DllProc := @DLLEntryPoint;
  { Теперь вызвать процедуру и продемонстрировать, что эта DLL
    присоединена к процессу. }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

На протяжении всей работы процесса раздел инициализации будет вызван только один раз. Если другой процесс загрузит ту же самую библиотеку DLL, то этот раздел будет вызван снова, но уже в контексте нового процесса, поскольку процессы не используют экземпляры библиотек DLL совместно.

Обработчик события btnFreeLibClick() выгружает библиотеку DLL с помощью вызова функции FreeLibrary(). В этом случае вызывается процедура DLLEntryProc(), на которую указывает переменная DLLProc; в качестве параметра ей передается значение DLL_PROCESS_DETACH.

Обработчик события btnCreateThreadClick() создает объект, являющийся потомком класса TThread. При этом вызывается процедура DLLEntryProc() и ей в качестве параметра передается значение DLL_THREAD_ATTACH. Обработчик событий btnFreeThreadClick() вызывает ту же процедуру DLLEntryProc(), но на этот раз с параметром, равным DLL_THREAD_DETACH. Несмотря на то что в данном примере при возникновении событий выводится всего лишь окно сообщений, эти события можно использовать для любых операций инициализации либо завершения любого процесса или потока, которые могут потребоваться в приложении. Ниже рассматривается применение этого метода для установки совместно используемых глобальных данных DLL. Приведенный выше пример содержится в проекте DLLEntryTest.dpr на прилагаемом компакт-диске (см. также www.williamspublishing.com).

Исключения в DLL

В этом разделе обсуждаются проблемы, возникающие в библиотеках DLL и Win32, а также применение исключений для их разрешения.

Перехват исключений в 16-разрядной Delphi

В среде 16-разрядной Delphi 1 объекты исключений имели формат, специфический именно для этой среды разработки. Поэтому при передаче исключений библиотекой DLL их приходилось перехватывать до выхода из подпрограмм DLL, поскольку, освобождая при проходе стек вызова модулей, они приводили к возникновению ошибки и аварийному завершению приложения. Во избежание таких неприятностей каждую точку входа в библиотеку DLL следовало заключать в собственный блок обработки исключения, как показано в следующем примере:

```
procedure SomeDLLProc;
begin
  try
    { Здесь находится содержимое }
  except
    on Exception do
    { Не позволять исключению покинуть этот блок, обязательно
      обработать его здесь и не передавать повторно. }
  end;
end;
```

В Delphi 2 такого уже не было. Исключения Delphi 6 полностью соответствуют исключениям Win32. Исключения библиотек DLL больше не зависят ни от языка Delphi, ни от компилятора, а являются, скорее, компонентом операционной системы Win32.

Впрочем, для этого необходимо удостовериться, что модуль `SysUtils` включен в раздел `uses` исходного файла библиотеки DLL. В противном случае Delphi отключит поддержку исключений внутри библиотек DLL.

СОВЕТ

Большинство приложений Win32 неспособно самостоятельно обрабатывать исключения, поэтому ни то, что исключения Delphi совместимы с исключениями Win32, ни то, что исключение можно передать из библиотеки DLL в главную программу, не спасет, вероятно, положение от аварийного завершения.
Если главное приложение создано в Delphi или C++Builder, то проблем не должно быть, но если это недоработанный код C и C++, то исключение сможет доставить много неприятностей.
Следовательно, чтобы сделать библиотеку DLL полностью отказоустойчивой, необходимо использовать старый, как у 16-разрядной Windows, метод защиты DLL по точкам входа с помощью блока `try..except`. Это позволит гарантированно перехватывать и обрабатывать исключения внутри самой DLL и не полагаться на возможности ее пользователей.

НА ЗАМЕТКУ

Когда приложение, написанное на языке, отличном от Delphi, использует библиотеку DLL, созданную в Delphi, оно не сможет работать с классами исключений Delphi. Но и такая ситуация может быть обработана как системное исключение Win32, представленное кодом `$0EEDFACE`. При этом адресом исключения будет первый элемент массива `ExceptionInformation` структуры `EXCEPTION_RECORD` системы Win32. Второй элемент такого массива содержит ссылку на объект исключения Delphi. Более подробная информация по данной теме приведена в интерактивной справочной системе Delphi.

Исключения и директива SafeCall

Функции, отмеченные директивой `safecall`, используются в модели COM и при обработке исключений. Она гарантируют, что любое исключение будет передано вызывающей функции. Функция с директивой `SafeCall` преобразует исключение в возвращаемое значение `HResult`. Кроме того, применение ключевого слова `SafeCall` подразумевает соблюдение соглашения о вызовах `StdCall`. Следовательно, функция с директивой `SafeCall` должна быть объявлена так:

```
function Foo(i: integer): string; Safecall;
```

На самом деле компилятор воспринимает ее как:

```
function Foo(i: integer): string; HResult; StdCall;
```

Затем компилятор неявно поместит все содержимое этой функции в блок `try..except`, который будет перехватывать все исключения. Блок `except` содержит обращение к функции `SafecallExceptionHandler()` для преобразования исключения в значение `HResult`. Это напоминает метод перехвата исключения и передачи кода ошибки, принятый в 16-разрядной Delphi.

Функции обратного вызова

Функция обратного вызова (callback function) — это функция приложения, вызываемая библиотеками DLL Win32 или другими библиотеками DLL. Фактически в системе Windows присутствует несколько функций API, которые используют функции обратного вызова. При вызове таких функций им передается адрес той функции приложения, которую Windows может вызвать. Если не совсем понятно, какое отношение это имеет к библиотекам DLL, то напомним, что реально функции API Win32 экспортируются из системных библиотек DLL. По сути, при передаче адреса функции обратного вызова в функцию API Win32 происходит передача этой функции в библиотеку DLL.

Одной из таких функций является функция интерфейса API `EnumWindows()`, которая регистрирует все главные окна. Она передает функциям обратного вызова приложений дескрипторы каждого окна, объединенные в перечисление. Требуется лишь передать функции `EnumWindows()` адрес функции обратного вызова, предварительно определив ее следующим образом:

```
function EnumWindowsProc(Hw: HWnd; lp: lParam): Boolean; stdcall;
```

Пример использования функции EnumWindows() приводится в проекте CallBack.dpr, код которого показан в листинге 6.9.

Листинг 6.9. MainForm.pas — пример функции обратного вызова

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls;

type

  { Определение записи/класса для хранения имени окна и имени
    класса для каждого окна. Экземпляры этого класса помещаются в
    список ListBox1 }
  TWindowInfo = class
    WindowName,          // Имя окна
    WindowClass: String; // Имя класса окна
  end;

  TMainForm = class(TForm)
    lbWinInfo: TListBox;
    btnGetWinInfo: TButton;
    hdWinInfo: THeaderControl;
    procedure btnGetWinInfoClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure lbWinInfoDrawItem(Control: TWinControl;
              Index: Integer; Rect: TRect; State: TOwnerDrawState);
    procedure hdWinInfoSectionResize(HeaderControl:
                      THeaderControl; Section: THeaderSection);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
function EnumWindowsProc(Hw: HWnd;
                         AMainForm: TMainForm): Boolean; stdcall;
{ Эта процедура вызывается библиотекой User32.DLL, поскольку она
  регистрирует активные окна в системе. }
var
  WinName, CName: array[0..144] of char;
  WindowInfo: TWindowInfo;
begin
  { Возвращаемое значение по умолчанию равно true, что означает
    незаконченную регистрацию окон. }
  Result := True;
```

```
  GetWindowText(Hw, WinName, 144); // Получить текст текущего окна
  GetClassName(Hw, CName, 144);    // Получить имя класса окна
  { Создается экземпляр класса TWindowInfo, поля которого
    заполняются значениями имен окон и имен классов окон. Затем
    этот объект добавляется в массив Objects списка ListBox1.
    Позднее эти значения будут отображены в окне списка. }
  WindowInfo := TWindowInfo.Create;
  with WindowInfo do begin
    SetLength(WindowName, strlen(WinName));
    SetLength(WindowClass, StrLen(CName));
    WindowName := StrPas(WinName);
    WindowClass := StrPas(CName);
  end;
  { Добавление в массив Objects }
  MainForm.lbWinInfo.Items.AddObject('', WindowInfo);
end;

procedure TMainForm.btnGetWinInfoClick(Sender: TObject);
begin
  { Регистрация всех отображаемых окон верхнего уровня. Передача
    адреса функции обратного вызова EnumWindowsProc, которая будет
    вызвана для каждого окна. }
  EnumWindows(@EnumWindowsProc, 0);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  { Освобождение всех экземпляров класса TWindowInfo }
  for i := 0 to lbWinInfo.Items.Count - 1 do
    TWindowInfo(lbWinInfo.Items.Objects[i]).Free
end;

procedure TMainForm.lbWinInfoDrawItem(Control: TWinControl;
             Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
  { Сначала очищаем прямоугольник, предназначенный для вывода
    информации. }
  lbWinInfo.Canvas.FillRect(Rect);
  { Теперь выводим строки записи TWindowInfo, хранящейся в списке
    под номером Index. Позиции вывода каждой строки определяются
    разделами HeaderControl1. }
  with TWindowInfo(lbWinInfo.Items.Objects[Index]) do begin
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowName),
             Length(WindowName), Rect,dt_Left or dt_VCenter);
    { Смещаем прямоугольник рисования, используя размер разделов
      HeaderControl1 для определения позиции вывода следующей
      строки. }
    Rect.Left := Rect.Left + hdWinInfo.Sections[0].Width;
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowClass),
             Length(WindowClass), Rect, dt_Left or dt_VCenter);
  end;
```

```
end;

procedure TMainForm.hdWinInfoSectionResize(HeaderControl:
                          THeaderControl; Section: THeaderSection);
begin
  lbWinInfo.Invalidate; // Перерисовка списка ListBox1.
end;

end.
```

В этом приложении функция `EnumWindows()` используется для получения всех имен окон верхнего уровня и названий их классов, помещаемых затем в нестандартный список объектов в главной форме. Главная форма использует этот нестандартный список для вывода имени окна и имени класса окна в виде отдельных столбцов. Но прежде чем разбираться в том, как создаются нестандартные списки со столбцами, рассмотрим методы использования функции обратного вызова.

Использование функции обратного вызова

В листинге 6.9 приведено определение процедуры `EnumWindowsProc()`, которой в качестве первого параметра передается дескриптор окна. Вторым параметром указываются определяемые пользователем данные, что позволяет передавать любые значения, размер которых эквивалентен размеру целочисленного типа данных.

`EnumWindowsProc()` — это процедура обратного вызова, адрес которой передается функции интерфейса API Win32 `EnumWindows()`, поэтому она должна быть объявлена с директивой `StdCall`, указывающей на использование соглашения о вызове, принятого в системе Win32. При передаче адреса этой процедуры в функцию `EnumWindows()` предполагается, что она будет вызвана для каждого существующего в данный момент окна верхнего уровня, дескриптор которого будет указан в качестве первого параметра. Этот дескриптор окна используется для получения имени окна и имени его класса. Затем создается экземпляр класса `TWindowInfo`, и его поля заполняются полученной информацией. Созданный экземпляр добавляется в массив `lbWinInfo.Objects`. Информация, содержащаяся в объектах этого массива, будет выведена в виде списка, включающего несколько столбцов данных.

Напомним, что в обработчике события `OnDestroy` главной формы обязательно должны освобождаться все созданные экземпляры класса `TWindowInfo`.

В обработчике события `btnGetWinInfoClick()` выполняется вызов процедуры `EnumWindows()` с передачей ей в качестве первого параметра адреса процедуры `EnumWindowsProc()`.

Запустив это приложение и щелкнув на кнопке в его форме, можно увидеть полученную от каждого окна информацию, представленную в виде списка.

Отображение нестандартного списка

Имена окон и имена классов всех окон верхнего уровня выводятся в виде отдельных столбцов в объекте под именем `lbWinInfo`. Это — экземпляр класса `TListBox`, в котором свойству `Style` присвоено значение `lbOwnerDraw`. При выборе данного стиля событие `TListBox.OnDrawItem` генерируется всякий раз, когда в компоненте `TListBox` необхо-

димо отобразить очередной элемент данных. При этом вся ответственность за отображение данных возлагается на программиста, что позволяет ему самостоятельно выбирать способ их представления.

В листинге 6.9 обработчик событий `lbWinInfoDrawItem()` содержит весь код, осуществляющий вывод элементов списка. В данном случае выводятся строки, содержащиеся в экземплярах класса `TWindowInfo`, помещенных в массив `lbWinInfo.Objects`. Эти значения были получены от функции обратного вызова `EnumWindowsProc()`. Чтобы разобраться в работе подпрограммы обработки данного события, проанализируйте комментарии, помещенные в ее код.

Обращение к функциям обратного вызова из библиотеки DLL

Аналогично тому, как адреса функции обратного вызова передаются функциям библиотек DLL, можно организовать и вызов этих функций из программ библиотек DLL. В настоящем разделе будет показано, как создать библиотеку DLL, в которой экспортируемой функции в качестве параметра передается адрес процедуры обратного вызова. Получив адрес процедуры обратного вызова, функция библиотеки DLL способна осуществить ее вызов. В листинге 6.10 содержится исходный код подобной библиотеки DLL.

Листинг 6.10. Исходный код `StrSrchLib.dll` — обращение к функции обратного вызова из библиотеки DLL

```
library StrSrchLib;

uses
  Wintypes, WinProcs, SysUtils, Dialogs;

type
 { Объявление типа функции обратного вызова }
 TFoundStrProc = procedure(StrPos: PChar); StdCall;

function SearchStr(ASrcStr, ASearchStr: PChar;
                   AProc: TFarProc): Integer; StdCall;
{ Эта функция выполняет поиск подстроки ASearchStr в строке
ASrcStr. В случае обнаружения искомой подстроки ASearchStr
вызывается процедура обратного вызова, заданная параметром Aproc
(если он был передан). Если в качестве значения этого параметра
пользователь передал значение nil, то эта процедура вызываться не
будет. }
var
  FindStr: PChar;
begin
  FindStr := ASrcStr;
  FindStr := StrPos(FindStr, ASearchStr);
  while FindStr <> nil do begin
    if AProc <> nil then
```

```
      TFoundStrProc(AProc)(FindStr);
    FindStr := FindStr + 1;
    FindStr := StrPos(FindStr, ASearchStr);
  end;
end;

exports
  SearchStr;
begin

end.
```

В этой библиотеке DLL также определяется процедурный тип `TFoundStrProc`, предназначенный для функций обратного вызова. Он используется для выполнения приведения типа функции обратного вызова при осуществлении обращения к ней.

Собственно вызов функции обратного вызова осуществляется в экспортируемой процедуре `SearchStr()`. Работа этой процедуры поясняется в комментариях.

Пример использования этой библиотеки DLL приведен в проекте `CallBack-Demo.dpr`. Исходный код главной формы этого демонстрационного проекта можно увидеть в листинге 6.11.

Листинг 6.11. Главная форма проекта DLL Callback

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    btnCallDLLFunc: TButton;
    edtSearchStr: TEdit;
    lblSrchWrd: TLabel;
    memStr: TMemo;
    procedure btnCallDLLFuncClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;
  Count: Integer;

implementation

{$R *.DFM}

{ Определение экспортируемой процедуры библиотеки DLL }
function SearchStr(ASrcStr, ASearchStr: PChar;
     AProc: TFarProc): Integer; StdCall external 'STRSRCHLIB.DLL';
```

```
{ Определяя процедуру обратного вызова, не забудьте директиву
StdCall }
procedure StrPosProc(AStrPsn: PChar); StdCall;
begin
  inc(Count); // Увеличить значение счетчика Count.
end;

procedure TMainForm.btnCallDLLFuncClick(Sender: TObject);
var
  S: String;
  S2: String;
begin
  Count := 0; // Инициализировать счетчик Count нулевым значением
  { Длина текста, в котором осуществляется поиск подстроки. }
  SetLength(S, memStr.GetTextLen);
  { Теперь скопировать этот текст в переменную S }
  memStr.GetTextBuf(PChar(S), memStr.GetTextLen);
  { Скопировать текст Edit1 в строковую переменную, чтобы ее можно
было передать функции библиотеки DLL }
  S2 := edtSearchStr.Text;
  { Вызов функции библиотеки DLL }
  SearchStr(PChar(S), PChar(S2), @StrPosProc);
  { Вывод сведений о том, сколько раз заданное слово встречается в
    строке. Результат содержится в счетчике Count, который
    используется в функции обратного вызова }
  ShowMessage(Format('%s %s %d %s',
              [edtSearchStr.Text, 'occurs', Count, 'times.']));
end;

end.
```

В этом приложении используется элемент управления класса `TMemo` под именем `memStr`. Свойство `EdtSearchStr.Text` компонента класса `TEdit` содержит строку, которую необходимо отыскать в содержимом компонента `memStr`. Содержимое объекта `memStr` передается функции `SearchStr()` библиотеки DLL в качестве исходной строки, а значение свойства `edtSearchStr.Text` — в качестве искомой строки.

Функция `StrPosProc()` — это используемая функция обратного вызова. Она увеличивает на единицу значение глобальной переменной `Count`, предназначенной для хранения количества вхождений искомой строки в текст, помещенный в компонент `memStr`.

Совместное использование DLL несколькими процессами

В среде 16-разрядной Windows управление памятью библиотек DLL происходило совсем не так, как в среде 32-разрядной. Одна из самых характерных особенностей работы 16-разрядных библиотек DLL заключалась в совместном использовании гло-

бальной памяти различными приложениями. Иными словами, если в функции 16-разрядной DLL объявляется глобальная переменная, то к ней получит доступ любое приложение, использующее эту DLL. Изменения, внесенные в эту переменную одним приложением, будут видны всем остальным приложениям.

В определенных случаях такое поведение могло быть опасным, поскольку одно приложение способно перезаписать данные, от которых зависит работа другого приложения. Иногда разработчики специально использовали такую особенность системы.

В среде Win32 совместного использования глобальных данных библиотек DLL больше не существует. Поскольку каждый процесс приложения отображает библиотеку DLL в свое собственное адресное пространство, данные библиотеки также отображаются в это адресное пространство. В результате каждое приложение получает собственный экземпляр данных библиотеки DLL. Поэтому изменения, внесенные в глобальные данные библиотеки DLL одним приложением, не будут видны другому.

Если планируется работа с 16-разрядным приложением, которое опирается на общедоступность глобальных данных DLL, то можно (как и прежде) предоставить такому приложению возможность совместного использования данных DLL с другими приложениями. Этот процесс не является автоматическим, и для хранения общих данных используются отображенные в память файлы, более подробная информация о которых приведена в предыдущем издании *Delphi 5 Руководство разработчика*, в главе 12 — "Работа с файлами". Здесь же воспользуемся ими лишь для иллюстрации данного метода.

Создание DLL с совместно используемой памятью

В листинге 6.12 содержится исходный код файла проекта библиотеки DLL, позволяющей приложениям, обращающимся к ее функциям, совместно использовать глобальные данные. Эти глобальные данные хранятся в переменной под именем `GlobalData`.

Листинг 6.12. `ShareLib` — совместное использование глобальных данных

```
library ShareLib;

uses
  ShareMem, Windows, SysUtils, Classes;

const
  cMMFileName: PChar = 'SharedMapData';

{$I DLLDATA.INC}

var
  GlobalData : PGlobalDLLData;
  MapHandle  : THandle;

{ GetDLLData - экспортируемая функция DLL }
procedure GetDLLData(var AGlobalData: PGlobalDLLData); StdCall;
begin
```

```
  { Указатель AGlobalData указывает на тот же адрес памяти, на
    который ссылается и указатель GlobalData.}
  AGlobalData := GlobalData;
end;

procedure OpenSharedData;
var
   Size: Integer;
begin
  { Определение размера отображаемых данных. }
  Size := SizeOf(TGlobalDLLData);

  { Теперь получаем объект отображенного в память файла. Обратите
внимание: первый параметр передает значение $FFFFFFFF, или Dword (-
1), чтобы пространство выделялось из системного файла подкачки.
Необходимо, чтобы имя отображенного в память объекта передавалось в
качестве последнего параметра. }

  MapHandle := CreateFileMapping(DWord(-1), nil, PAGE_READWRITE,
                                 0, Size, cMMFileName);

  if MapHandle = 0 then
    RaiseLastWin32Error;
  { Отображение данных в адресное пространство вызывающего
    процесса и получение указателя на начало этого адреса. }
  GlobalData := MapViewOfFile(MapHandle, FILE_MAP_ALL_ACCESS,
                              0, 0, Size);
  { Инициализация этих данных }
  GlobalData^.S := 'ShareLib';
  GlobalData^.I := 1;
  if GlobalData = nil then begin
    CloseHandle(MapHandle);
    RaiseLastWin32Error;
  end;
end;

procedure CloseSharedData;
{ Эта процедура закрывает отображенный в память файл и освобождает
  его дескриптор. }
begin
  UnmapViewOfFile(GlobalData);
  CloseHandle(MapHandle);
end;

procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: OpenSharedData;
    DLL_PROCESS_DETACH: CloseSharedData;
  end;
end;

exports
```

```
  GetDLLData;

begin
  { Сначала назначить процедуру переменной DLLProc }
  DllProc := @DLLEntryPoint;
  { Теперь вызвать процедуру, чтобы продемонстрировать факт
    присоединения DLL к процессу. }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

Переменная `GlobalData` имеет тип `PGlobalDLLData`, который определен в *подключаемом файле* (include file) `DllData.inc`. Этот файл содержит следующее определение типа (обратите внимание: подключение файла осуществляется с помощью директивы подключения `$I`):

```
type

  PGlobalDLLData = ^TGlobalDLLData;
  TGlobalDLLData = record
    S: String[50];
    I: Integer;
  end;
```

В этой библиотеке DLL используется тот же процесс, который рассматривался в настоящей главе выше при обсуждении добавления кода инициализации/завершения библиотеки DLL в виде процедуры входа/выхода. Упомянутая процедура называется `DLLEntryPoint()`, как показано в листинге. Когда процесс загружает эту DLL, вызывается метод `OpenSharedData()`, а когда процесс отгружает ее — метод `CloseSharedData()`.

Файлы, отображенные в память, позволяют резервировать некоторую область адресного пространства в системе Win32, для которой назначаются страницы физической памяти. Этот процесс напоминает выделение памяти с возможностью ссылки на нее с помощью указателя. Но в случае отображенных в память файлов в это адресное пространство можно отобразить любой дисковый файл, а затем с помощью указателя ссылаться на адресное пространство внутри файла так же, как и на любую другую область памяти.

Для работы с отображенным в память файлом необходимо сначала получить дескриптор существующего на диске файла, для которого будет создан объект отображения в память. Затем этот объект связывается с указанным файлом. Ранее в настоящей главе обсуждалось, как система позволяет нескольким приложениям совместно использовать библиотеки DLL. Вначале библиотека загружается в память, а затем каждому приложению предоставляется его собственный образ этой DLL. В результате создается впечатление, что каждое приложение загрузило отдельный экземпляр библиотеки. Но в действительности в памяти находится только один экземпляр файла DLL. Эффект множественности достигается за счет использования механизма отображенных файлов в память. Тот же процесс можно использовать и для предоставления доступа к файлам данных. Для этого достаточно вызвать необходимые функции API Win32, которые обеспечивают создание и доступ к файлам, отображенным в память.

Теперь рассмотрим следующий сценарий. Предположим, что некоторое приложение (назовем его App1) создает отображенный в память файл, который связывается с дисковым файлом MyFile.dat. Приложение App1 может теперь считывать и записывать данные в этот файл. Если во время работы приложения App1 другое приложение, App2, также будет выполнять отображение в тот же файл, то изменения, внесенные App1, будут видны App2. На самом деле все обстоит несколько сложнее: для немедленного переноса на диск внесенных в файл изменений необходимо установить определенные флаги, а также выполнить некоторую другую подготовительную работу. Но в данный момент все эти подробности не имеют значения. Достаточно сказать, что изменения будут восприняты обоими приложениями, поскольку такое возможно.

Одним из вариантов использования отображенных в память файлов является создание механизма отображения файлов на основе файла подкачки (страничного обмена) Win32, а не обычного файла данных. Это означает, что вместо отображения в память файла, реально существующего на диске, есть возможность зарезервировать некоторую область памяти, куда затем можно ссылаться как на дисковый файл. Подобный подход избавит от необходимости создания и уничтожения временного файла, если необходимо всего лишь создать некоторое адресное пространство, доступное нескольким процессам. Система Win32 автоматически управляет своим файлом подкачки, поэтому, когда память больше не требуется, она освобождается системой.

Выше был предложен сценарий, иллюстрирующий, как два приложения могут получить доступ к одним и тем же файловым данным с помощью файла, отображенного в память. Тот же сценарий подходит и для совместного использования данных приложений и DLL. Действительно, если библиотека DLL, загружаемая некоторым приложением, создает отображенный в память файл, то после загрузки этой библиотеки другим приложением она будет продолжать использовать все тот же отображенный в память файл. При этом будут существовать два образа библиотеки DLL (по одному для каждого вызывающего приложения), использующих один и тот же экземпляр файла, отображенного в память. Функции библиотеки DLL могут обеспечить доступ к данным отображенного в память файла любому вызывающему их приложению. Когда одно приложение внесет в эти данные изменения, они будут видны другому приложению, поскольку оба приложения обращаются к одним и тем же данным, но отображенным в память двумя различными экземплярами библиотеки. Описанный метод используется в примере, представленном в листинге 6.12.

В этом примере создание отображенного в память файла возложено на процедуру OpenSharedData(). Для создания исходного объекта отображения файла в ней используется функция CreateFileMapping(), которая передает созданный объект функции MapViewOfFile(). Эта функция отображает образ файла в адресное пространство вызывающего процесса и возвращает адрес начала выделенного адресного пространства. Теперь вспомним, что это адресное пространство принадлежит вызывающему процессу. Для двух различных приложений, использующих данную библиотеку DLL, эти пространства адресов могут быть различными, несмотря на то, что данные, на которые они ссылаются, одни и те же.

НА ЗАМЕТКУ

Первым параметром, передаваемым функции `CreateFileMapping()`, является дескриптор отображаемого в память файла. Но если отображение выполняется для адресного пространства системного файла подкачки, то передается значение `$FFFFFFFF`, что равносильно значению `DWord (-1)` — как в данном примере. В качестве второго параметра функции `CreateFileMapping()` нужно передать имя объекта отображения файла в память. Это имя система будет использовать для ссылки на соответствующее файловое отображение. Если несколько процессов одновременно создадут отображение в память одноименного файла, то объекты отображения будут ссылаться на одну и ту же область системной памяти.

После обращения к функции `MapViewOfFile()` переменная `GlobalData` будет указывать на адресное пространство файла, отображенного в память. В результате выполнения экспортируемой функции `GetDLLData()` параметр `AGlobalData` будет указывать на ту же область памяти, что и указатель `GlobalData`. Параметр `AGlobalData` передается из вызывающего приложения, следовательно, вызывающее приложение получит возможность считывать и записывать общие данные.

Процедура `CloseSharedData()` закрывает отображение общего файла для данного вызывающего процесса и освобождает его объект отображения файла. Это никак не влияет на другие объекты отображения файла и на отображение глобального файла со стороны других приложений.

Применение DLL с совместно используемой памятью

Для иллюстрации применения DLL с совместно используемой памятью было создано два работающих с ней приложения. Первое приложение (проект `App1.dpr`) позволяет модифицировать глобальные данные библиотеки DLL. Второе приложение (проект `App2.dpr`) обращается к этим данным и периодически обновляет два компонента `TLabel`, используя для этого компонент `TTimer`. При запуске обоих приложений можно увидеть последствия совместного доступа к данным библиотеки DLL, т.е. окно приложения `App2` будет отображать изменения, внесенные в окне приложения `App1`.

В листинге 6.13 содержится исходный код проекта `App1`.

Листинг 6.13. Главная форма приложения `App1.dpr`

```
unit MainFrmA1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Mask;

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
```

```
    edtGlobDataStr: TEdit;
    btnGetDllData: TButton;
    meGlobDataInt: TMaskEdit;
    procedure btnGetDllDataClick(Sender: TObject);
    procedure edtGlobDataStrChange(Sender: TObject);
    procedure meGlobDataIntChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;

var
  MainForm: TMainForm;

{ Определение экспортируемой процедуры библиотеки DLL }
procedure GetDLLData(var AGlobalData: PGlobalDLLData);
                     StdCall External 'SHARELIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnGetDllDataClick(Sender: TObject);
begin
  { Получить указатель на данные библиотеки DLL }
  GetDLLData(GlobalData);
  { Обновление элементов управления для отражения
    значений полей GlobalData. }
  edtGlobDataStr.Text := GlobalData^.S;
  meGlobDataInt.Text  := IntToStr(GlobalData^.I);
end;

procedure TMainForm.edtGlobDataStrChange(Sender: TObject);
begin
  { Обновление данных библиотеки DLL в соответствии с внесенными
    изменениями. }
  GlobalData^.S := edtGlobDataStr.Text;
end;

procedure TMainForm.meGlobDataIntChange(Sender: TObject);
begin
  { Обновление данных библиотеки DLL в соответствии с внесенными
    изменениями. }
  if meGlobDataInt.Text = EmptyStr then
    meGlobDataInt.Text := '0';
  GlobalData^.I := StrToInt(meGlobDataInt.Text);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  btnGetDllDataClick(nil);
end;

end.
```

В состав этого приложения также входит подключаемый файл DllData.inc, в котором определен тип данных TGlobalDLLData и указатель на эти данные. Обработчик события btnGetDllDataClick() с помощью вызова функции GetDLLData() получает указатель на глобальные данные библиотеки DLL, доступ к которым возможен с помощью отображенного в память файла, принадлежащего библиотеке. Затем, используя значение этого указателя GlobalData, выполняется обновление элементов управления формы. Обработчики события OnChange для полей ввода изменяют значение области памяти, на которую указывает переменная GlobalData. Поскольку указатель GlobalData ссылается на глобальные данные библиотеки DLL, модификации подвергаются те данные, на которые ссылается отображенный в память файл, созданный библиотекой DLL.

В листинге 6.14 представлен исходный код главной формы приложения App2.dpr.

Листинг 6.14. Главная форма приложения App2.dpr

```
unit MainFrmA2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls;

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    lblGlobDataStr: TLabel;
    tmTimer: TTimer;
    lblGlobDataInt: TLabel;
    procedure tmTimerTimer(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;

{ Определение экспортируемой процедуры библиотеки DLL }
procedure GetDLLData(var AGlobalData:
                PGlobalDLLData); StdCall External 'SHARELIB.DLL';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.tmTimerTimer(Sender: TObject);
begin
  GetDllData(GlobalData);  // Получить доступ к данным
  { Представление содержимого полей GlobalData. }
  lblGlobDataStr.Caption := GlobalData^.S;
```

```
  lblGlobDataInt.Caption := IntToStr(GlobalData^.I);
end;

end.
```

Эта форма содержит два компонента TLabel, которые обновляются во время обработки события OnTimer объекта tmTimer. Если пользователь изменит значения глобальных данных библиотеки DLL в приложения App1, приложение App2 отобразит эти изменения.

Оба приложения можно запустить и поработать с ними. Они содержаться на прилагаемом компакт-диске (см. также www.williamspublishing.com).

Экспорт объектов из библиотек DLL

К объекту и его методам можно получить доступ даже в том случае, если этот объект содержится внутри библиотеки DLL. Однако к определению объекта, расположенного внутри библиотеки DLL, предъявляются определенные требования, а на его использование накладываются некоторые ограничения. Представленный здесь подход может быть полезен в весьма специфических ситуациях. Как правило, такого же эффекта можно достичь за счет применения пакетов или интерфейсов.

Ниже приведен список условий и ограничений, накладываемых на экспорт объекта из DLL:

- Вызывающее приложение может использовать лишь те методы объекта, которые были объявлены как виртуальные.
- Экземпляры объектов должны создаваться только внутри библиотеки DLL.
- Экспортируемый объект должен быть определен как в библиотеке DLL, так и в вызывающем приложении, причем объявление методов должно выполняться в одном и том же порядке.
- Нельзя создать объект класса, производного от содержащегося внутри библиотеки DLL.

Здесь перечислены лишь основные ограничения, но возможны и некоторые другие.

Для иллюстрации этой технологии подготовлен простой, но достаточно наглядный пример экспортируемого объекта. Данный объект содержит функцию, возвращающую заданную строку, написанную прописными или строчными буквами — в зависимости от значения другого параметра, определяющего необходимый регистр. Определение этого класса приведено в листинге 6.15.

Листинг 6.15. Объект, предназначенный для экспорта из DLL

```
type

  TConvertType = (ctUpper, ctLower);

  TStringConvert = class(TObject)
{$IFDEF STRINGCONVERTLIB}
  private
```

```
    FPrepend: String;
    FAppend : String;
{$ENDIF}
  public
    function ConvertString(AConvertType: TConvertType;
                           AString: String): String;
   virtual; stdcall; {$IFNDEF STRINGCONVERTLIB} abstract; {$ENDIF}
{$IFDEF STRINGCONVERTLIB}
    constructor Create(APrepend, AAppend: String);
    destructor Destroy; override;
{$ENDIF}
  end;

{ У любого приложения, использующего этот класс, идентификатор
STRINGCONVERTLIB не определен (поскольку у них есть свой
идентификатор), а значит, для них определение этого класса будет
выглядеть следующим образом:

  TStringConvert = class(TObject)
  public
    function ConvertString(AConvertType: TConvertType;
      AString: String): String; virtual; stdcall; abstract;
  end;
}
```

В листинге 6.15 фактически содержится исходный код подключаемого файла `StrConvert.inc`. Размещение определения класса этого объекта в подключаемом файле вызвано необходимостью соблюдения третьего пункта приведенного выше списка требований, согласно которому класс объекта должен быть одинаково определен как в библиотеке DLL, так и в вызывающем приложении. Когда вызывающее приложение и библиотека DLL получают определение класса из одного и того же подключаемого файла, совпадение определения гарантировано. Если в определение объекта необходимо внести изменения, то достаточно, отредактировав один подключаемый файл, повторно скомпилировать оба проекта, а не вводить эти изменения дважды — сначала в вызывающее приложение, а затем в библиотеку DLL, что может привести к возникновению ошибок.

Рассмотрим следующее определение метода `ConvertSring()`:

```
function ConvertString(AConvertType: TConvertType;
                       AString: String): String; virtual; stdcall;
```

Причина объявления этого метода виртуальным (с помощью ключевого слова `virtual`) заключается не в необходимости создавать производные классы, в которых можно было бы впоследствии переопределять метод `ConvertString()`. Он объявлен виртуальным, чтобы создать точку входа в *таблицу виртуальных методов* (VMT — Virtual Method Table). Не вдаваясь пока в подробности, скажем, что таблица VMT представляет собой участок памяти, содержащий указатели на виртуальные методы объекта. С помощью таблицы VMT вызывающее приложение может получить указатель на определенный метод конкретного объекта. Для методов, которые не объявлены виртуальными, таблица VMT записей не содержит, а вызывающее приложение не сможет по-

лучить на них указатель. Как видите, все это делается для получения вызывающим приложением указателя на функцию. Но поскольку этот указатель зависит от типа метода, определяемого в объекте, Delphi автоматически, неявным образом, обработает любые ссылки на адреса, передавая методу дополнительный параметр `self`.

Обратите внимание на идентификатор условного определения `STRINGCONVERTLIB`. При экспорте объекта переопределение в вызывающем приложении необходимо лишь тем методам, к которым нужно обеспечить внешний доступ из DLL. Кроме того, данные методы можно определить как абстрактные, чтобы избежать сообщения об ошибке во время компиляции. Это вполне допустимо, поскольку во время выполнения программы эти методы будут реализованы в коде DLL. Приведенный в листинге комментарий показывает, как будет выглядеть определение класса объекта `TStringConvert` со стороны приложения.

Листинг 6.16 демонстрирует реализацию объекта `TStringConvert`.

Листинг 6.16. Реализация объекта `TStringConvert`

```
unit StringConvertImp;
{$DEFINE STRINGCONVERTLIB}

interface
uses SysUtils;
{$I StrConvert.inc}

function InitStrConvert(APrepend,
                        AAppend: String): TStringConvert; stdcall;

implementation

constructor TStringConvert.Create(APrepend, AAppend: String);
begin
  inherited Create;
  FPrepend := APrepend;
  FAppend  := AAppend;
end;

destructor TStringConvert.Destroy;
begin
  inherited Destroy;
end;

function TStringConvert.ConvertString(AConvertType: TConvertType;
                                      AString: String): String;
begin
  case AConvertType of
    ctUpper: Result := Format('%s%s%s',
                          [FPrepend, UpperCase(AString), FAppend]);
    ctLower: Result := Format('%s%s%s',
                          [FPrepend, LowerCase(AString), FAppend]);
  end;
end;

function InitStrConvert(APrepend,
```

```
                          AAppend: String): TStringConvert;
begin
  Result := TStringConvert.Create(APrepend, AAppend);
end;

end.
```

Согласно предъявляемым к экспортируемым объектам требованиям, такой объект должен создаваться в библиотеке DLL. Это условие реализуется в стандартной экспортируемой функции `InitStrConvert()`, которой передаются два параметра, предназначенных для конструктора. Мы добавили это для иллюстрации способа передачи информации конструктору объекта через функцию интерфейса.

Обратите внимание: в этом модуле объявляется условная директива `STRINGCONVERTLIB`. Остальная часть модуля не нуждается в дополнительных разъяснениях. Файл проекта библиотеки DLL представлен в листинге 6.17.

Листинг 6.17. Файл проекта библиотеки `StringConvertLib.dll`

```
library StringConvertLib;
uses
  ShareMem, SysUtils, Classes,
  StringConvertImp in 'StringConvertImp.pas';

exports
  InitStrConvert;
end.
```

В этой библиотеке нет ничего нового. Обратите внимание на использование модуля `ShareMem`. Его имя должно быть объявлено первым в файле проекта библиотеки, а также в файле проекта вызывающего приложения. Этот момент очень важен и о нем не следует забывать.

В листинге 6.18 показан пример использования экспортируемого объекта для преобразования произвольной строки в строку из прописных или строчных букв. Проект этого примера называется `StrConvertTest.dpr`. Его можно найти на прилагаемом компакт-диске (см. также `www.williamspublishing.com`).

Листинг 6.18. Проект, использующий объект преобразования строк

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

{$I strconvert.inc}

type
  TMainForm = class(TForm)
    btnUpper: TButton;
```

```
    edtConvertStr: TEdit;
    btnLower: TButton;
    procedure btnUpperClick(Sender: TObject);
    procedure btnLowerClick(Sender: TObject);
  private
  public
  end;

var
  MainForm: TMainForm;

function InitStrConvert(APrepend,
                        AAppend: String): TStringConvert; stdcall;
                        external 'STRINGCONVERTLIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnUpperClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('Upper ', ' end');
  try
    ConvStr := edtConvertStr.Text;
    if ConvStr <> EmptyStr then
      edtConvertStr.Text := FStrConvert.ConvertString(ctUpper,
                                                      ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

procedure TMainForm.btnLowerClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('Lower ', ' end');
  try
    ConvStr := edtConvertStr.Text;
    if ConvStr <> EmptyStr then
      edtConvertStr.Text := FStrConvert.ConvertString(ctLower,
                                                      ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

end.
```

Резюме

Разработка библиотек DLL — неотъемлемая часть создания приложений Windows, допускающих многократное использование кода. В этой главе рассказывалось о создании и использовании библиотек DLL в приложениях Delphi, а также о различных методах их загрузки. Кроме того, были отмечены некоторые специальные соглашения, которые необходимо учитывать при использовании библиотек DLL в среде Delphi, а также показано, как сделать данные в библиотеках DLL доступными различным приложениям.

Вооружившись этими знаниями, можно самостоятельно создавать библиотеки DLL в среде Delphi и без труда использовать их в своих приложениях. Дополнительные сведения о работе с библиотеками DLL будут приведены и в других главах настоящей книги.

ЧАСТЬ

III

Разработка баз данных

В ЭТОЙ ЧАСТИ...

ГЛАВА

7

Архитектура баз данных в Delphi

В ЭТОЙ ГЛАВЕ...

В настоящей главе рассматриваются вопросы доступа к файлам внешних баз данных из приложений Delphi. Вниманию новичков в этой области программирования будет предложено немного теории, которая, надеемся, поможет им найти свой путь к созданию высококачественных приложений для работы с базами данных. Для тех, кому подобные приложения хорошо известны, предлагается ознакомиться с полезной информацией о программировании баз данных именно в Delphi. Для доступа к данным Delphi 6 обладает несколькими механизмами. В этой главе представлен их краткий обзор, а более подробное описание содержится в последующих главах. Здесь обсуждается также архитектура, на базе которой построены все механизмы доступа к данным в Delphi 6.

Типы баз данных

Приведенное ниже описание компонентов взято из раздела "*Using Databases*" (Использование баз данных) интерактивной справочной системы Delphi, поскольку авторы полагают, что никто кроме *Borland* не в состоянии лучше описать типы баз данных, поддерживаемых архитектурой Delphi. Ссылки, упоминаемые в этом разделе, также относятся к интерактивной справочной системе.

- Вкладка BDE палитры компонентов (Component Palette) содержит компоненты, используемые *механизмом баз данных компании Borland* (BDE — Borland Database Engine). В BDE определено большинство интерфейсов API для взаимодействия с базами данных. Из всех механизмов доступа к данным, BDE обладает наиболее широким диапазоном функций, а также предоставляет наибольший комплект разнообразных вспомогательных утилит. Это наилучший способ взаимодействия с данными в таблицах Paradox и dBASE. В то же время установка такого механизма достаточно сложна. Более подробная информация о применении компонентов BDE приведена в разделе "*Using the Borland Database Engine*" (Использование механизма баз данных *Borland*) интерактивной справочной системы Delphi.
- Вкладка ADO палитры компонентов содержит компоненты, используемые *объектами данных ActiveX* (ADO — ActiveX Data Objects), которые обеспечивают доступ к информации баз данных с помощью OLEDB. ADO является стандартом *Microsoft*. Широкий диапазон драйверов ADO позволяет установить соединение с различными серверами баз данных. Использование компонентов на базе ADO позволяет интегрировать создаваемое приложение в ADO-ориентированную среду (например применять ADO-ориентированные серверы приложений). Более подробная информация об использовании компонентов ADO приведена в разделе "*Working with ADO Components*" (Работа с компонентами ADO) интерактивной справочной системы Delphi.
- Вкладка dbExpress палитры компонентов содержит компоненты, которые использует dbExpress для доступа к информации баз данных. dbExpress — это облегченный набор драйверов, обеспечивающий наиболее быстрый доступ к информации баз данных. Кроме того, компоненты dbExpress обеспечивают возможность разработки межплатформенных приложений, поскольку они доступны также и в Linux. Но компоненты баз данных dbExpress обладают наименьшим количеством функций для манипулирования данными. Более подроб-

ная информация об использовании компонентов dbExpress приведена в разделе "*Using Unidirectional Datasets*" (Использование односторонних наборов данных) интерактивной справочной системы Delphi.

- Вкладка InterBase палитры компонентов содержит компоненты, способные обращаться к базам данных InterBase непосредственно, минуя уровень специального механизма. Более подробная информация об использовании компонентов InterBase приведена в разделе "*Getting Started with InterBase Express*" (Приступим к InterBase Express) интерактивной справочной системы Delphi.

Архитектура баз данных

Архитектура баз данных Delphi состоит из компонентов, которые отвечают за представление и инкапсуляцию информации баз данных. Рис. 7.1 демонстрирует эти взаимоотношения в таком виде, как они представлены в интерактивной справочной системе Delphi 6, в разделе "*Database Architecture*" (Архитектура баз данных).

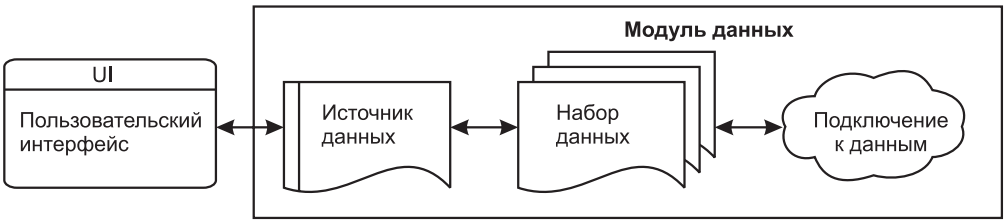

Рис. *7.1. Архитектура баз данных Delphi*

На рис. 7.1 весьма упрощенно показана архитектура применения базы данных. То есть пользовательский интерфейс взаимодействует с данными через источник данных, который подключен к набору данных, который, в свою очередь, инкапсулирует сами данные. В предыдущем разделе обсуждались различные типы баз данных, с которыми Delphi может работать. Различные типы хранилищ данных требуют различных типов наборов данных. Набор данных, изображенный на рис. 7.1, представляет собой абстрактный тип набора данных, от которого происходят другие, конкретные типы наборов, способные обеспечить доступ к различным типам данных.

Подключение к серверам баз данных

Естественно, разработчиком баз данных в первую очередь захочется узнать, как с помощью Delphi установить соединение с собственноручно созданной базой данных. В этом разделе рассматривается некоторые способы, позволяющие Delphi подключиться к с серверам баз данных.

Способы подключения к базе данных

Наборы данных должны подключаться к серверам баз данных. Обычно для этого используют компонент соединения (подключения к базам данных) `Connection`. Компо-

ненты класса Connection инкапсулируют методы подключения к серверу баз данных и выступают в роли единой точки подключения для всех наборов данных в приложении.

Компоненты Connection инкапсулированы в компоненте класса TCustomConnection, от которого происходят классы всех остальных компонентов, специализированных для различных типов хранилищ данных. Для каждого типа хранилища данных поддерживаются следующие типы компонентов доступа к данным:

- Компонент TDatabase предназначен для подключения наборов данных BDE. К таким наборам данных относятся TTable, TQuery и TStoreproc. Подключение базы данных BDE описано в главе 28 предыдущего издания, *Delphi 5 Руководство разработчика*, содержащегося на прилагаемом CD.
- Компонент TADOConnection предназначен для подключения таких баз данных ADO, как Microsoft Access и Microsoft SQL. Поддерживаются следующие наборы данных: TADODataset, TADOTable, TADOQuery и TADOStoredProc. Подключение баз данных ADO описано в главе 9, "Применение dbGo for ADO при разработке баз данных".
- Компонент TSQLConnection предназначен для подключения наборов данных, ориентированных на dbExpress. Наборы данных dbExpress представляют собой специальные облегченные односторонние наборы данных. К таким наборам данных относятся TSQLDataset, TSQLTable, TSQLQuery и TSQLStoredProc. Более подробная информация о dbExpress приведена в главе 8, "Применение dbExpress при разработке баз данных".
- Компонент TIBDatabase предназначен для подключения наборов данных, ориентированных на InterBase Express. К таким наборам данных относятся: TIBDataSet, TIBTable, TIBQuery и TIBStoredProc. В этой книге InterBase Express специально не рассматривается, поскольку большая часть его функциональных возможностей совпадает с методами других соединений.

Каждый из этих наборов данных обладает общими функциональными возможностями, содержащимися в классе TCustomConnection. К этим функциональным возможностям относятся методы, свойства и события, перечисленные ниже.

- Подключение к хранилищу данных и отключение.
- Вход (login) и поддержка установленного защищенного соединения.
- Манипулирование набором данных.

Подключение к базе данных

Несмотря на то, что большинство методов одинаковы для всех компонентов подключения к базам данных (соединений), существует и ряд отличий. Это обусловлено различиями между хранилищами данных, обслуживаемых такими компонентами соединений. Следовательно, компонент TADOConnection будет функционировать немного иначе, чем компонент соединения TDatabase. Более подробно методы соединения для TSQLConnection и TADOConnection рассматриваются в главах 8, "Применение dbExpress при разработке баз данных", и 9, "Применение dbGo for ADO при разработке баз данных", соответственно, а подключение к BDE с помощью набора данных рассматривается в главе 28, "Создание локальных приложений баз дан-

ных", предыдущего издания, *Delphi 5 Руководство разработчика*, содержащегося на прилагаемом CD.

Работа с наборами данных

Набор данных (dataset) — это коллекция строк и столбцов данных. Каждый *столбец* (column) содержит данные одинакового типа, а каждая *строка* (row) представляет собой набор данных каждого из типов данных столбцов. Столбец иногда называют *полем* (field), а строку — *записью* (record). Библиотека VCL инкапсулирует набор данных в абстрактном классе по имени `TDataSet`, который обладает множеством свойств и методов, необходимых для манипулирования и перемещения по набору данных. От этого базового класса происходят все остальные классы, предназначенные для работы с различными типами наборов данных.

Чтобы облегчить понимание терминологии, ниже приводится список чаще всего встречающихся терминов из области баз данных, которые используются в этой и других главах.

- *Набор данных* (dataset) — это набор отдельных записей данных. Каждая запись содержит несколько полей. Каждое поле может содержать различные типы данных (целые числа, строковые значения, десятичные числа, графику и т.д.). Наборы данных представлены в библиотеке VCL абстрактным классом `TDataSet`.
- *Таблица* (table) — это специальный тип набора данных. Как правило, она представляет собой файл, содержащий записи и физически хранящийся где-нибудь на диске. В библиотеке VCL таблицы инкапсулируют классы `TTable`, `TADOTable`, `TSQLTable` и `TIBTable`.
- *Запрос* (query) — это специальный тип набора данных. Запрос можно рассматривать как команду, которую должен выполнить сервер баз данных. Вследствие таких команд может быть возвращен *результат* (result). Если возвращается набор данных, то он оформляется в виде виртуальной (созданной в памяти) таблицы, называемой *результатом запроса* (resultset). Результат запроса представляет собой специальный набор данных, который инкапсулирован в классах `TQuery`, `TADOQuery`, `TSQLQuery` и `TIBQuery`.

НА ЗАМЕТКУ

Как уже говорилось, настоящая книга подразумевает наличие у читателя определенных знаний в области баз данных. Эта глава не является учебником для начинающих программистов баз данных, и мы полагаем, что термины вышеприведенного списка читателю уже знакомы. В противном случае, если такие термины, как *база данных*, *таблица* и *индекс*, понятны не до конца, то прежде чем приступать к изучению настоящей главы, рекомендуем обратиться к литературе, посвященной основам работы с базами данных.

Как открыть и закрыть набор данных

Прежде чем выполнять манипуляции с набором данных, его необходимо открыть. Для этого достаточно вызвать метод `Open()`, как показано в приведенном ниже фрагменте кода:

```
Table1.Open;
```

Это эквивалентно присвоению свойству `Active` набора данных значения `True`:

```
Table1.Active := True;
```

Второй способ является менее накладным, поскольку метод `Open()` в конечном счете также присваивает свойству `Active` значение `True`. Однако потери при этом настолько незначительны, что об этом не стоит и беспокоиться.

Открытым набором данных можно свободно манипулировать. По завершении использования набора данных его необходимо закрыть, вызвав для этого метод `Close()`:

```
Table1.Close;
```

Альтернативный способ закрыть набор данных заключается в присвоении его свойству `Active` значения `False`:

```
Table1.Active := False;
```

СОВЕТ

При взаимодействии с SQL-сервером соединение с базой данных должно быть установлено при первом открытии набора данных в этой базе. При закрытии последнего набора данных в базе установленное с ней соединение будет закрыто. Открытие и закрытие подобных соединений создает значительную нагрузку на систему. Если открытие и закрытие соединений с базой данных SQL-сервера приходится выполнять слишком часто, то благоразумней воспользоваться компонентом `TDatabase`, который позволит избежать этих проблем. Более подробная информация о компоненте `TDatabase` приведена в следующей главе.

Листинг 7.1 во всех подробностях демонстрирует то, как необходимо открывать и закрывать различные типы наборов данных.

Листинг 7.1. Как открыть и закрыть набор данных

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, FMTBcd, DBXpress, IBDatabase, ADODB,
  DBTables, DB, SqlExpr, IBCustomDataSet, IBQuery, IBTable,
  StdCtrls;

type
  TForm1 = class(TForm)
    SQLDataSet1: TSQLDataSet;
```

```
    SQLTable1: TSQLTable;
    SQLQuery1: TSQLQuery;

    ADOTable1: TADOTable;
    ADODataSet1: TADODataSet;
    ADOQuery1: TADOQuery;

    IBTable1: TIBTable;
    IBQuery1: TIBQuery;
    IBDataSet1: TIBDataSet;

    Table1: TTable;
    Query1: TQuery;

    SQLConnection1: TSQLConnection;
    Database1: TDatabase;
    ADOConnection1: TADOConnection;
    IBDatabase1: TIBDatabase;
    Button1: TButton;
    Label1: TLabel;
    Button2: TButton;
    IBTransaction1: TIBTransaction;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure Button2Click(Sender: TObject);
  private
    { Закрытые объявления }
    procedure OpenDatasets;
    procedure CloseDatasets;
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  IBDatabase1.Connected    := True;
  ADOConnection1.Connected := True;
  Database1.Connected      := True;
  SQLConnection1.Connected := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDatasets;
```

```
end;

procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  CloseDatasets;
  IBDatabase1.Connected    := false;
  ADOConnection1.Connected := false;
  Database1.Connected      := false;
  SQLConnection1.Connected := false;
end;

procedure TForm1.CloseDatasets;
begin

  // Отключиться от набора данных dbExpress
  SQLDataSet1.Close;  // или .Active := false;
  SQLTable1.Close;    // или .Active := false;
  SQLQuery1.Close;    // или .Active := false;

  // Отключиться от набора данных ADO
  ADOTable1.Close;    // или .Active := false;
  ADODataSet1.Close;  // или .Active := false;
  ADOQuery1.Close;    // или .Active := false;

  // Отключиться от набора данных Interbase Express
  IBTable1.Close;    // или .Active := false;
  IBQuery1.Close;    // или .Active := false;
  IBDataSet1.Close;  // или .Active := false;

  // Отключиться от набора данных BDE
  Table1.Close;    // или .Active := false;
  Query1.Close;    // или .Active := false;

  Label1.Caption := 'Datasets are closed.'
end;

procedure TForm1.OpenDatasets;
begin

  // Подключиться к набору данных dbExpress
  SQLDataSet1.Open;  // или .Active := true;
  SQLTable1.Open;    // или .Active := true;
  SQLQuery1.Open;    // или .Active := true;

  // Подключиться к набору данных ADO
  ADOTable1.Open;    // или .Active := true;
  ADODataSet1.Open;  // или .Active := true;
  ADOQuery1.Open;    // или .Active := true;

  // Подключиться к набору данных Interbase Express
  IBTable1.Open;    // или .Active := true;
  IBQuery1.Open;    // или .Active := true;
```

```
  IBDataSet1.Open;  // или .Active := true;

  // Подключиться к набору данных BDE
  Table1.Open;    // или .Active := true;
  Query1.Open;    // или .Active := true;

  Label1.Caption := 'Datasets are open.';
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  CloseDatasets;
end;

end.
```

Этот пример содержится на прилагаемом CD. При установке соединений с базами данных могут возникнуть некоторые проблемы, поскольку пример был создан для конкретной машины. Чтобы он стал работоспособен на другом компьютере, необходимо установить параметры соединения, соответствующие конкретной машине. Однако целью настоящего примера была демонстрация способов подключения к различным типам наборов данных.

Навигация по набору данных

Класс `TDataSet` содержит несколько простых методов для работы с записями. Методы `First()` и `Last()` перемещают указатель текущей записи к первой и последней записям в наборе данных соответственно, а методы `Next()` и `Prior()` — на одну запись вперед или назад. Методу `MoveBy()` передается параметр типа `Integer`, в котором указывается, на какое количество записей следует переместить указатель вперед или назад.

Свойства BOF, EOF и циклы

Свойства `BOF` и `EOF` класса `TDataSet` имеют тип `Boolean` и показывают, является ли текущая запись первой или последней в наборе данных. Например, необходимо выполнить выборку каждой записи набора данных, вплоть до последней. Эту простую задачу можно решить с помощью цикла `while`, в котором выборка записей будет продолжаться до тех пор, пока свойство `EOF` не примет значение `True`, как показано в приведенном ниже фрагменте кода:

```
Table1.First;                  // Переход к началу набора данных
while not Table1.EOF do begin // Перебор всех записей в таблице
  { Выполнение обработки текущей записи }
  Table1.Next;                 // Переход к следующей записи
end;
```

СОВЕТ

Вызывайте метод `Next()` только внутри цикла `while-not-EOF`, иначе приложение может попасть в бесконечный цикл.

Избегайте использования цикла `repeat..until` для выполнения действий над набором данных. Следующий код на первый взгляд выглядит вполне нормально, но если попытаться выполнить его с пустым набором данных, то может возникнуть проблема, поскольку процедура `DoSomeStuff()` будет выполняться по крайней мере один раз, независимо от того, содержит набор данных записи или нет.

```
repeat
  DoSomeStuff;
  Table1.Next;
until Table1.EOF;
```

Поскольку цикл `while-not-EOF` выполняет проверку условия вначале, то при использовании этой конструкции описанной выше проблемы не возникает.

Листинг 7.2 содержит пример, иллюстрирующий возможности навигации по различным типам наборов данных.

Листинг 7.2. Навигация по различным типам наборов данных

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, FMTBcd, DBXpress, IBDatabase, ADODB,
  DBTables, DB, SqlExpr, IBCustomDataSet, IBQuery, IBTable,
  StdCtrls, Grids, DBGrids, ExtCtrls;

type
  TForm1 = class(TForm)
    SQLTable1: TSQLTable;
    ADOTable1: TADOTable;
    IBTable1: TIBTable;
    Table1: TTable;

    SQLConnection1: TSQLConnection;
    Database1: TDatabase;
    ADOConnection1: TADOConnection;
    IBDatabase1: TIBDatabase;
    Button1: TButton;
    Label1: TLabel;
    Button2: TButton;
    IBTransaction1: TIBTransaction;
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    RadioGroup1: TRadioGroup;
    btnFirst: TButton;
    btnLast: TButton;
    btnNext: TButton;
    btnPrior: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
```

```
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure Button2Click(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
    procedure btnFirstClick(Sender: TObject);
    procedure btnLastClick(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnPriorClick(Sender: TObject);
    procedure DataSource1DataChange(Sender: TObject;
                                    Field: TField);
  private
    { Закрытые объявления }
    procedure OpenDatasets;
    procedure CloseDatasets;
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  IBDatabase1.Connected    := True;
  ADOConnection1.Connected := True;
  Database1.Connected      := True;
  SQLConnection1.Connected := True;

  Datasource1.DataSet := IBTable1;
  OpenDatasets;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDatasets;
end;

procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  CloseDatasets;
  IBDatabase1.Connected    := false;
  ADOConnection1.Connected := false;
  Database1.Connected      := false;
  SQLConnection1.Connected := false;
end;

procedure TForm1.CloseDatasets;
begin
```

```
  // Отключиться от набора данных dbExpress dataset
  SQLTable1.Close;    // или .Active := false;

  // Отключиться от набора данных ADO dataset
  ADOTable1.Close;    // или .Active := false;

  // Отключиться от набора данных Interbase Express dataset
  IBTable1.Close;    // или .Active := false;

  // Отключиться от набора данных BDE datasets
  Table1.Close;    // или .Active := false;

  Label1.Caption := 'Datasets are closed.'
end;

procedure TForm1.OpenDatasets;
begin

  // Подключиться к набору данных dbExpress
  SQLTable1.Open;    // или .Active := true;

  // Подключиться к набору данных ADO
  ADOTable1.Open;    // или .Active := true;

  // Подключиться к набору данных Interbase Express
  IBTable1.Open;    // или .Active := true;

  // Подключиться к набору данных BDE
  Table1.Open;    // или .Active := true;

  Label1.Caption := 'Datasets are open.';
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  CloseDatasets;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0: Datasource1.DataSet := IBTable1;
    1: Datasource1.DataSet := Table1;
    2: Datasource1.DataSet := ADOTable1;
  end; // case
end;

procedure TForm1.btnFirstClick(Sender: TObject);
begin
  DataSource1.DataSet.First;
end;
```

```
procedure TForm1.btnLastClick(Sender: TObject);
begin
  DataSource1.DataSet.Last;
end;

procedure TForm1.btnNextClick(Sender: TObject);
begin
  DataSource1.DataSet.Next;
end;

procedure TForm1.btnPriorClick(Sender: TObject);
begin
  DataSource1.DataSet.Prior;
end;

procedure TForm1.DataSource1DataChange(Sender: TObject;
                                       Field: TField);
begin
  btnLast.Enabled := not DataSource1.DataSet.Eof;
  btnNext.Enabled := not DataSource1.DataSet.Eof;
  btnFirst.Enabled := not DataSource1.DataSet.Bof;
  btnPrior.Enabled := not DataSource1.DataSet.Bof;
end;

end.
```

В этом примере переключатель `TRadioGroup` используется для того, чтобы позволить пользователю выбирать один из трех типов баз данных. Кроме того, обработчик события `OnDataChange` демонстрирует возможность применения свойств `BOF` и `EOF` для того, чтобы в зависимости от их состояния разрешать или запрещать использование кнопок. Обратите внимание: для навигации по набору данных применяются одинаковые методы, независимо от конкретного типа набора данных.

НА ЗАМЕТКУ

Имейте в виду: в данном примере компонент dbExpress не применяется. Это связано с тем, что dbExpress предназначен для *односторонних наборов данных* (unidirectional dataset), по которым можно перемещаться только в одном направлении и получать доступ только для чтения. Фактически, если попытаться подключать к набору данных dbExpress двунаправленный компонент навигации (типа `TDBGrid`), то произойдет ошибка. Для управления односторонними наборами данных необходим специальный подход. Более подробная информация по этой теме приведена в главе 8, “Применение dbExpress при разработке баз данных”.

Манипулирование наборами данных

Приложение базы данных бесполезно, если его данными невозможно управлять. К счастью, наборы данных содержат методы, которые позволяют сделать это. Наборы данных дают возможность добавлять, редактировать и удалять записи из исходной таблицы. Для этого применяются следующие методы: `Insert()` (добавить), `Edit()` (редактировать), и `Delete()` (удалить).

Листинг 7.3 содержит пример приложения, использующего эти методы.

Листинг 7.3. `MainFrm.pas` — пример манипулирования данными

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Mask, DBCtrls, DB, Grids,
  DBGrids, ADODB;

type
  TMainForm = class(TForm)
    ADOConnection1: TADOConnection;
    adodsCustomer: TADODataSet;
    dtsrcCustomer: TDataSource;
    DBGrid1: TDBGrid;
    adodsCustomerCustNo: TAutoIncField;
    adodsCustomerCompany: TWideStringField;
    adodsCustomerAddress1: TWideStringField;
    adodsCustomerAddress2: TWideStringField;
    adodsCustomerCity: TWideStringField;
    adodsCustomerStateAbbr: TWideStringField;
    adodsCustomerZip: TWideStringField;
    adodsCustomerCountry: TWideStringField;
    adodsCustomerPhone: TWideStringField;
    adodsCustomerFax: TWideStringField;
    adodsCustomerContact: TWideStringField;
    Label1: TLabel;
    dbedtCompany: TDBEdit;
    Label2: TLabel;
    dbedtAddress1: TDBEdit;
    Label3: TLabel;
    dbedtAddress2: TDBEdit;
    Label4: TLabel;
    dbedtCity: TDBEdit;
    Label5: TLabel;
    dbedtState: TDBEdit;
    Label6: TLabel;
    dbedtZip: TDBEdit;
    Label7: TLabel;
    dbedtPhone: TDBEdit;
    Label8: TLabel;
    dbedtFax: TDBEdit;
    Label9: TLabel;
    dbedtContact: TDBEdit;
    btnAdd: TButton;
    btnEdit: TButton;
    btnSave: TButton;
    btnCancel: TButton;
```

```
    Label10: TLabel;
    dbedtCountry: TDBEdit;
    btnDelete: TButton;
    procedure btnAddClick(Sender: TObject);
    procedure btnEditClick(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure btnCancelClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure btnDeleteClick(Sender: TObject);
  private
    { Закрытые объявления }
    procedure SetButtons;
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.dfm}

procedure TMainForm.btnAddClick(Sender: TObject);
begin
  adodsCustomer.Insert;
  SetButtons;
end;

procedure TMainForm.btnEditClick(Sender: TObject);
begin
  adodsCustomer.Edit;
  SetButtons;
end;

procedure TMainForm.btnSaveClick(Sender: TObject);
begin
  adodsCustomer.Post;
  SetButtons;
end;

procedure TMainForm.btnCancelClick(Sender: TObject);
begin
  adodsCustomer.Cancel;
  SetButtons;
end;

procedure TMainForm.SetButtons;
begin
  btnAdd.Enabled  := adodsCustomer.State = dsBrowse;
  btnEdit.Enabled := adodsCustomer.State = dsBrowse;
```

```
  btnSave.Enabled := (adodsCustomer.State = dsInsert) or
                     (adodsCustomer.State = dsEdit);
  btnCancel.Enabled := (adodsCustomer.State = dsInsert) or
                       (adodsCustomer.State = dsEdit);
  btnDelete.Enabled := adodsCustomer.State = dsBrowse;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  adodsCustomer.Open;
  SetButtons;
end;

procedure TMainForm.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  adodsCustomer.Close;
  ADOConnection1.Connected := False;
end;

procedure TMainForm.btnDeleteClick(Sender: TObject);
begin
  adodsCustomer.Delete;
end;

end.
```

Рис. 7.2 иллюстрирует пример приложения, манипулирующего данными.

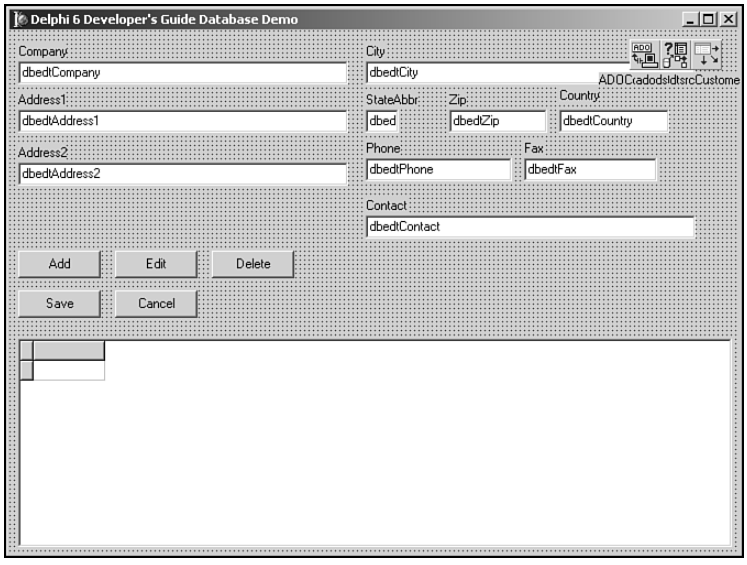

Рис. 7.2. *Главная форма приложения, манипулирующего данными*

Это приложение управляет данными в самой простой форме. Ниже приведен список использованных методов.

- `Insert()` добавляет новую запись.
- `Edit()` изменяет активную запись.
- `Post()` сохраняет в таблицу новую запись или изменения, внесенные в уже существующую.
- `Cancel()` отменяет любые изменения, сделанные в записи.
- `Delete()` удаляет текущую запись из таблицы.

Состояния набора данных

Листинг 7.3 демонстрирует также применение свойства `TDataSet.State`, которое позволяет исследовать состояние набора данных, чтобы получить возможность программно установить соответствующее состояние кнопок формы (разрешена/запрещена). Это дает возможность отключить кнопку Add (Добавить), когда набор данных находится в состоянии `Insert` (вставка) или `Edit` (редактирование). Другие состояния представлены в табл. 7.1.

Таблица 7.1. Возможные значения свойства `TDataset.State`

Значение	*Описание*
`dsBrowse`	Состояние по умолчанию. Набор данных находится в режиме просмотра (`Browse`)
`dsCalcFields`	Вызван обработчик события `OnCalcFields`, и значение полей записи в настоящий момент пересчитывается
`dsEdit`	Набор данных находится в режиме редактирования (`Edit`). Это означает, что был вызван метод `Edit()`, но отредактированная запись еще не была внесена в таблицу
`dsInactive`	Набор данных закрыт
`dsInsert`	Набор данных находится в режиме вставки (`Insert`). Обычно это означает, что был вызван метод `Insert()`, но добавляемая запись еще не была внесена в таблицу
`dsSetKey`	Набор данных находится в режиме установки ключа (`SetKey`). Был вызван метод `SetKey()`, но метод `GotoKey()` еще не вызывался
`dsNewValue`	Набор данных находится в переходном состоянии, когда осуществляется доступ к свойству `NewValue`
`dsOldValue`	Набор данных находится в переходном состоянии, когда осуществляется доступ к свойству `OldValue`
`dsCurValue`	Набор данных находится в переходном состоянии, когда осуществляется доступ к свойству `CurValue`

Окончание табл. 7.1.

Значение	*Описание*
dsFilter	В настоящее время в наборе данных выполняется фильтрация записей, выборка значений или другая операция с использованием фильтра
dsBlockRead	Набор данных буферизируется, поэтому при установке этого значения перемещение курсора не приводит к обновлению данных в элементах управления и генерации событий
dsInternalCalc	В настоящее время вычисляется значение поля, у которого свойство FieldKind имеет значение fkInternalCalc
dsOpening	Набор данных находится в состоянии открытия, которое в настоящей момент еще не завершено. Это состояние устанавливается только тогда, когда набор данных открывается для асинхронной выборки данных

Работа с полями

Delphi позволяет получить доступ к полям любого набора данных с помощью класса TField и его потомков. Эти классы дают возможность не только возвратить или установить значение выбранного поля текущей записи набора данных, но и изменить поведение поля модифицировав его свойства. Кроме того, можно модифицировать набор данных в целом, изменяя визуальный порядок расположения полей, удаляя поля или же создавая новые вычисляемые (calculated) или подстановочные поля (lookup fields).

Значения полей

Получить доступ к значениям полей в Delphi очень просто. Класс TDataSet обладает стандартным массивом свойств по имени FieldValues[], который возвращает значение определенного поля как значение типа Variant. Поскольку массив свойств FieldValues[] стандартный, то для доступа к определенному свойству не нужно специально указывать имя массива, достаточно имени свойства. Например, в следующем фрагменте кода значение поля CustName таблицы Table1 присваивается переменной S типа String:

```
S := Table1['CustName'];
```

Так же просто присвоить значение целого типа поля CustNo переменной I:

```
I := Table1['CustNo'];
```

Основным выводом из сказанного выше является возможность сохранения значений нескольких полей в массиве типа Variant. Единственным осложнением является то, что индекс массива типа Variant должен начинаться с нуля, а его содержимое представляет собой переменные varVariant. Приведенный ниже фрагмент кода демонстрирует эту возможность:

```
const
  AStr = 'The %s is of the %s category and its length is %f in.';
var
  VarArr: Variant;
  F: Double;
begin
  VarArr := VarArrayCreate([0, 2], varVariant);
  { Предполагаем, что объект Table1 связан с таблицей Biolife }
  VarArr := Table1['Common_Name;Category;Length_In'];
  F := VarArr[2];
  ShowMessage(Format(AStr, [VarArr[0], VarArr[1], F]));
end;
```

Для доступа к отдельным объекта класса `TField`, ассоциированным с набором данных, можно использовать также массив свойств `TDataset.Fields[]` или функцию `FieldsByName()`, поскольку класс `TField` содержит информацию обо всех полях.

Массив `Fields[]` (начинающийся с нулевого элемента) является массивом объектов класса `TField`. Так, элемент `Fields[0]` возвращает объект `TField`, представляющий первое логическое поле записи. Функции `FieldsByName()` передается строковый параметр, который соответствует некоторому имени поля в таблице. Таким образом, вызов этой функции в виде `FieldsByName('OrderNo')` возвратит компонент `TField`, соответствующий полю `OrderNo` в текущей записи набора данных.

Создав объект класса `TField`, можно получить доступ и на чтение и на запись ко всем его полям, используя свойства, представленные в табл. 7.2.

Таблица 7.2. Свойства для доступа к значениям полей объекта `TField`

Свойство	*Возвращаемый тип*
`AsBoolean`	`Boolean`
`AsFloat`	`Double`
`AsInteger`	`Longint`
`AsString`	`String`
`AsDateTime`	`TDateTime`
`Value`	`Variant`

Если первое поле в текущем наборе данных — строковое, то сохранить его значение в переменной `S` типа `String` можно таким образом:

```
S := Table1.Fields[0].AsString;
```

В следующем фрагменте кода целой переменной `I` присваивается значение поля `'OrderNo'` в текущей записи таблицы:

```
I := Table1.FieldsByName('OrderNo').AsInteger;
```

Типы данных полей

Если необходимо узнать тип поля, используйте свойство DataType объекта класса TField, которое отображает тип данных таблицы базы данных (независимо от типа Object Pascal). Свойство DataType имеет тип TFieldType, определенный следующим образом:

```
type
  TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger,
                ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD,
                ftDate, ftTime, ftDateTime, ftBytes, ftVarBytes,
                ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
                ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor,
                ftFixedChar, ftWideString, ftLargeint, ftADT,
                ftArray, ftReference, ftDataSet, ftOraBlob,
                ftOraClob, ftVariant, ftInterface, ftIDispatch,
                ftGuid);
```

Для работы с упоминаемыми выше типами данных существуют специальные классы, производные от класса TField. Они рассматриваются в настоящей главе далее.

Имена и номера полей

Для поиска имени определенного поля используйте свойство FieldName класса TField. Например, в следующем фрагменте кода имя первого поля в текущей таблице записывается в переменную S типа String:

```
var
  S: String;
begin
  S := Table1.Fields[0].FieldName;
end;
```

Аналогично, зная имя поля, можно получить его номер, используя свойство FieldNo. В следующем фрагменте кода номер поля OrderNo помещается в переменную I типа Integer:

```
var
  I: integer;
begin
  I := Table1.FieldsByName('OrderNo').FieldNo;
end;
```

НА ЗАМЕТКУ

Для определения количества полей, содержащихся в наборе данных, используйте свойство FieldList объекта TDataSet. Свойство FieldList представляет собой линейное представление всех вложенных полей в таблице, имеющих *абстрактный тип данных* (ADT — Abstract Data Type).
Для совместимости с прежними версиями в текущей версии Delphi поддерживается также свойство FieldCount, но оно не поддерживает поля типа ADT, которые просто пропускаются.

Манипулирование данными полей

Ниже приведена последовательность действий, которые следует предпринять для изменения одного или нескольких полей в текущей записи.

1. Чтобы перейти в режим редактирования (Edit), вызовите метод `Edit()` текущего набора данных.
2. Присвойте полям новые значения.
3. Внесите изменения в набор данных, вызвав метод `Post()` или просто перейдя на новую запись. В этом случае изменения будут внесены автоматически.

Например, типичная процедура изменения записи выглядит следующим образом:

```
Table1.Edit;
Table1['Age'] := 23;
Table1.Post;
```

СОВЕТ

Иногда приходится работать с наборами данных, доступными только для чтения. Примером таких данных может служить информация, расположенная на компакт-диске или полученная в ответ на запрос, результат которого не допускает редактирования. Прежде чем предпринимать попытки редактирования данных, следует убедиться, что текущий набор не содержит данных, доступных только для чтения. Для этого достаточно проверить значение свойства `CanModify`. Если возвращаемое значение свойства `CanModify` равно `True`, то редактирование набора данных разрешено.

Редактор полей

В Delphi существует утилита Fields Editor (Редактор полей), предоставляющая возможность более гибкого управления полями набора данных. Этот инструмент можно использовать для просмотра отдельного набора данных в окне конструктора форм, для чего достаточно дважды щелкнуть на компоненте `TTable`, `TQuery` или `TStoredProc` либо выбрать пункт Fields Editor в контекстном меню для набора данных. В окне редактора полей можно выбрать поля набора данных, с которыми необходимо работать, или же создать новые вычисляемые либо подстановочные поля. Для этого можно воспользоваться контекстным меню редактора. На рис. 7.3 показано окно редактора полей с раскрытым контекстным меню.

Для демонстрации использования редактора полей откройте новый проект и поместите в главную форму компонент `TTable`. Установите свойство `DatabaseName` объекта `Table1` равным `DBDEMOS` (это псевдоним демонстрационной таблицы, которая входит в комплект Delphi), а свойство `TableName` — равным `ORDERS.DB`. Поместите в форму компоненты `TDataSource` и `TDBGrid`, чтобы отобразить результат. Свяжите объект `DataSource1` с объектом `Table1`, а объект `DBGrid1` — с объектом `DataSource1`. Теперь установите свойство `Active` объекта `Table1` в состояние `True` и вы увидите данные таблицы на экране.

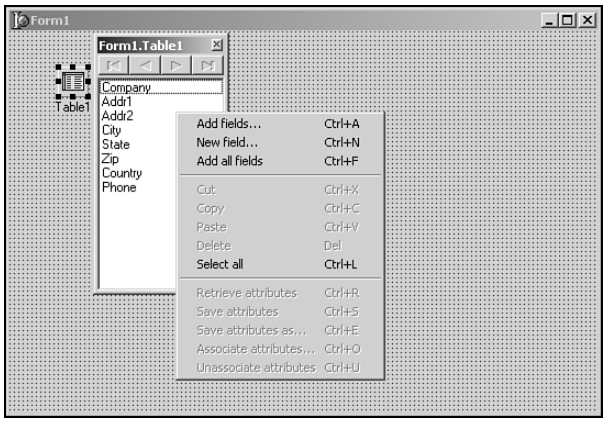

Рис. 7.3. *Контекстное меню окна редактора полей (Fields Editor)*

Добавление полей

Откройте окно редактора полей (см. рис. 7.3), дважды щелкнув на объекте `Table1`. Предположим, что требуется ограничить представление таблицы несколькими полями. В контекстном меню окна редактора полей выберите команду Add Fields (Добавить поля). На экране раскроется диалоговое окно Add Fields. Выделите в списке Available fields (Доступные поля) поля `OrderNo`, `CustNo` и `ItemsTotal`, а затем щелкните на кнопке OK. Эти три выделенных поля появятся в окне редактора полей и в сетке с данными.

Для представления полей набора данных, выделенных в окне редактора полей, Delphi создает объекты классов, производных от класса `TField`. Например, для выбранных только что трех полей таблицы `ORDERS.DB` Delphi помещает в исходный код формы следующие объявления:

```
Table1OrderNo: TFloatField;
Table1CustNo: TFloatField;
Table1ItemsTotal: TCurrencyField;
```

Заметьте, что имя объекта поля — это объединение имени `TTable` и имени поля. Поскольку эти объекты полей создаются программно, можно получить доступ ко всем унаследованным ими от класса `TField` свойствам и методам непосредственно во время выполнения программы, а не только во время проектирования.

Потомки класса TField

Давайте отвлечемся на минуту от рассмотрения компонентов класса `TField`. Для каждого типа поля (типы данных полей описывались выше) существует один или несколько различных классов, производных от класса `TField`. Многие из этих типов полей также соответствуют типам данных, существующим в языке Object Pascal. В табл. 7.3 приведены сведения о различных классах иерархии `TField`, типах их базовых классов, типах данных их полей и эквивалентных типах данных Object Pascal.

Таблица 7.3. Потомки класса `TField` и типы данных их полей

Производный класс	*Базовый*	*Тип поля*	*Тип Object Pascal*
TStringField	TField	ftString	String
TWideStringField	TStringField	ftWideString	WideString
TGuidField	TStringField	ftGuid	TGUID
TNumericField	TField	*	*
TIntegerField	TNumericField	ftInteger	Integer
TSmallIntField	TIntegerField	ftSmallInt	SmallInt
TLargeintField	TNumericField	ftLargeint	Int64
TWordField	TIntegerField	ftWord	Word
TAutoIncField	TIntegerField	ftAutoInc	Integer
TFloatField	TNumericField	ftFloat	Double
TCurrencyField	TFloatField	ftCurrency	Currency
TBCDField	TNumericField	ftBCD	Double
TBooleanField	TField	ftBoolean	Boolean
TDateTimeField	TField	ftDateTime	TDateTime
TDateField	TDateTimeField	ftDate	TDateTime
TTimeField	TDateTimeField	ftTime	TDateTime
TBinaryField	TField	*	*
TBytesField	TBinaryField	ftBytes	Нет
TVarBytesField	TBytesField	ftVarBytes	Нет
TBlobField	TField	ftBlob	Нет
TMemoField	TBlobField	ftMemo	Нет
TGraphicField	TBlobField	ftGraphic	Нет
TObjectField	TField	*	*
TADTField	TObjectField	ftADT	Нет
TArrayField	TObjectField	ftArray	Нет
TDataSetField	TObjectField	ftDataSet	TDataSet
TReferenceField	TDataSetField	ftReference	
TVariantField	TField	ftVariant	OleVariant
TInterfaceField	TField	ftInterface	IUnknown
TIDispatchField	TInterfaceField	ftIDispatch	IDispatch
TAggregateField	TField	Нет	Нет

* *Означает абстрактный базовый класс в иерархии* `TField`

Как видно из табл. 7.3, типы полей `BLOB` и `Object` — это специальные типы, которые не имеют прямого аналога среди типов Object Pascal. Поля типа `BLOB` будут подробно рассматриваться в этой главе далее.

Поля и инспектор объектов

Если выделить поле в окне редактора полей, то в окне *инспектора объектов* (Object Inspector) можно будет получить доступ к свойствам и событиям, ассоциированным с данным экземпляром класса, производного от `TField`. Это позволяет модифицировать свойства полей (например, определять минимальное и максимальное значение, формат отображения, а также делать их доступными только для чтения). Назначение одних свойств (таких как `ReadOnly` — только для чтения) очевидно из их названия, а назначение других может быть не совсем понятно. Некоторые из "интуитивно непонятных" свойств рассматриваются в настоящей главе далее.

Открыв в окне инспектора объектов вкладку Events (События), можно увидеть, что с объектами поля ассоциированы и некоторые события. События `OnChange`, `OnGetText`, `OnSetText` и `OnValidate` подробно описаны в интерактивной справочной системе. Для получения справки по событию достаточно щелкнуть слева от его имени, а затем нажать клавишу <F1>. Из всех событий чаще всего используется, пожалуй, `OnChange`. Оно позволяет выполнять некоторые действия при каждом изменении содержимого поля (например, переходить на другую запись или добавлять новую).

Вычисляемые поля

Помимо прочего, в окне редактора полей к набору данных можно добавить вычисляемые поля. Допустим, к примеру, что в набор данных необходимо добавить поле, отображающее для каждой строки в таблице `ORDERS` объем оптовой продажи, составляющий 32% от общего объема. Выберите в контекстном меню окна редактора полей пункт New Field. На экране раскроется диалоговое окно New Field, показанное на рис. 7.4. В поле Name этого окна введите имя нового поля — **`WholesaleTotal`**. Тип этого поля — **`Currency`**, поэтому в раскрывающемся списке Type выберите именно это значение. В группе Field Type установите переключатель в положение Calculated (Вычисляемый) и щелкните на кнопке OK. Новое поле появится в сетке, но пока не будет содержать никаких данных.

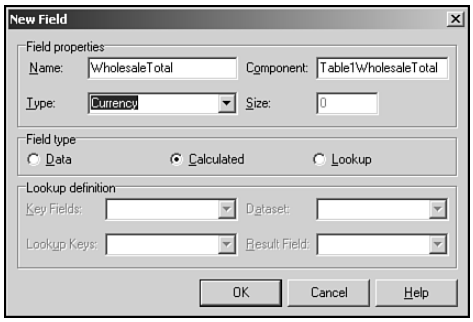

Рис. 7.4. *Добавление вычисляемого поля в диалоговым окне* New Field

Чтобы заполнить новое поле данными, следует назначить соответствующий метод событию `OnCalcFields` объекта `Table1`. В коде обработчика этого события полю `WholesaleTotal` нужно просто присвоить значение, равное 32% от существующего значения поля `SalesTotal`. Код метода обработки события `Table1.OnCalcFields` приведен ниже:

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
  DataSet['WholesaleTotal'] := DataSet['ItemsTotal'] * 0.68;
end;
```

На рис. 7.5 показано поле `WholesaleTotal`, расположенное в сетке и содержащее правильные данные.

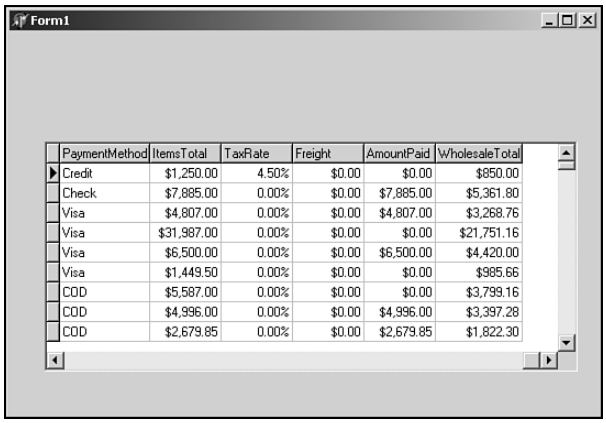

Рис. 7.5. *Вычисляемое поле добавлено к таблице*

Подстановочные поля

Подстановочные (lookup) поля позволяют создавать в наборе данных такие поля, значения которых будут выбираться из другого набора данных. Для иллюстрации сказанного добавим такое поле к текущему проекту. Вряд ли по номеру клиента (поле `CustNo` таблицы `ORDERS`) можно будет вспомнить его имя. Поэтому целесообразно добавить к таблице `Table1` подстановочное поле, связанное с таблицей `CUSTOMER`, из которой по номеру клиента будет выбираться его имя.

Вначале поместите в форму второй объект класса `TTable` и присвойте его свойству `DatabaseName` значение **`DBDEMOS`**, а свойству `TableName` — значение **`CUSTOMER.DB`**. Это будет объект `Table2`. Затем вновь откройте окно New Field, выбрав пункт New Field в контекстном меню окна редактора поля. Присвойте новому полю имя **`CustName`** и тип **`String`**. Размер поля установите равным 15 символам. Не забудьте установить переключатель в группе Field Type в положение Lookup. В списке Dataset этого диалогового окна выберите значение **`Table2`** — именно этот набор данных необходимо просматривать. В обоих списках (Key Fields и Lookup Keys) этого диалогового окна выберите значение **`CustNo`** — это то общее поле, по значению которого будет выполняться поиск. И, наконец, в списке Result выберите значение **`Contact`** — именно это поле необходимо отображать в создаваемом наборе данных. На рис. 7.6 показано диалоговое окно New Field в процессе создания подстановочного поля, а на рис. 7.7 — готовая форма с подстановочными данными.

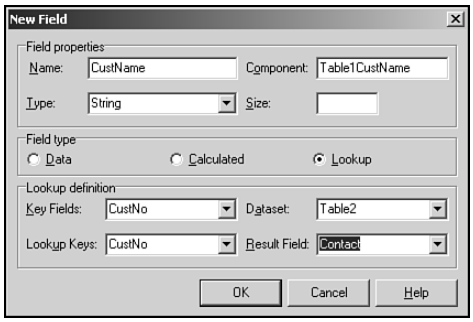

Рис. 7.6. *Добавление подстановочного поля в диалоговом окне* New Filed

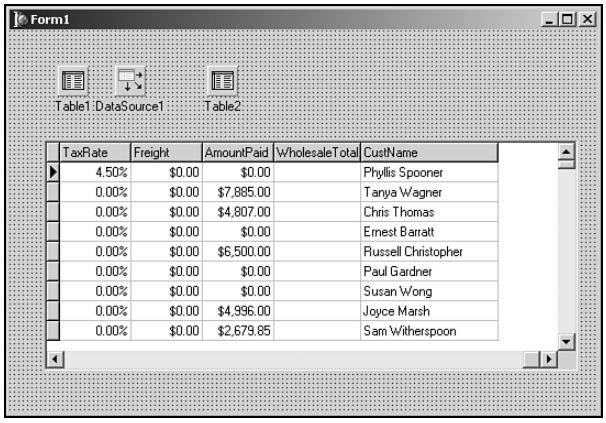

Рис. 7.7. *Форма, содержащая подстановочное поле*

Перетаскивание полей мышью

Другое, менее очевидное свойство окна редактора полей — возможность перетаскивать поля из списка полей в создаваемую форму. Это свойство можно продемонстрировать следующим образом: создайте новый проект, который будет содержать в главной форме только один объект `TTable`. Установите свойство `Table1.DatabaseName` равным **`DBDEMOS`**, а свойство `Table1.TableName` — равным **`BIOLIFE.DB`**. Откройте для этой таблицы окно редактора полей и добавьте все поля таблицы в список полей набора данных. Теперь появилась возможность перетащить одно или несколько полей из окна редактора полей в главную форму.

Отметим несколько наиболее впечатляющих особенностей этой процедуры. Во-первых, Delphi распознает тип помещаемого в форму поля и создает соответствующий элемент управления для отображения его данных (например, для строкового поля создается объект класса `TDBEdit`, а для графического — объект класса `TDBImage`). Во-вторых, Delphi проверяет, существует ли объект класса `TDataSource`, связанный с этим набором данных. Если это так, то именно он устанавливается в соответствие новому полю, в противном случае объект создается заново. На рис. 7.8 показан результат перетаскивания с помощью мыши полей таблицы `BIOLIFE` в форму.

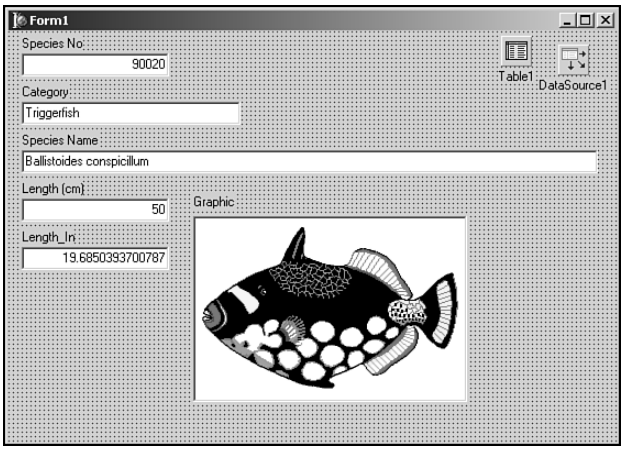

Рис. 7.8. *Перетаскивание полей в форму*

Работа с полями типа BLOB

Поля BLOB (Binary Large Object) разработаны для размещения в них данных неопределенного размера. Поля BLOB в одной записи набора данных могут содержать 3 байта данных, в то время как подобное же поле в другой записи может содержать 3 Кбайта данных. Эти поля наиболее удобны для хранения большого количества текста, графики либо таких непредсказуемых типов данных, как объекты OLE.

Класс TBlobField и типы полей

Как уже отмечалось в этой главе, в библиотеке VCL существует класс TBlobField, производный от класса TField и специально предназначенный для инкапсуляции полей BLOB. Класс TBlobField содержит свойство BlobType типа TBlobType, которое отображает, какой именно тип данных хранится в его поле BLOB. Тип TBlobType определен в модуле DB следующим образом:

```
TBlobType = ftBlob..ftOraClob;
```

Все возможные типы объектов и данных, которые может принимать поле BLOB, приведены в табл. 7.4.

Таблица 7.4. Типы полей класса TBlobField

Тип поля	*Тип данных*
ftBlob	Нетипизированные или определенные пользователем данные
ftMemo	Текст
ftGraphic	Растровая графика Windows
ftFmtMemo	Поле MEMO в формате Paradox
ftParadoxOle	Объект OLE Paradox

Окончание табл. 7.4.

Тип поля	*Тип данных*
ftDBaseOLE	Объект OLE dBASE
ftTypedBinary	Любые неструктурированные данные
ftCursor..ftDataSet	Недопустимые типы BLOB
ftOraBlob	Поля BLOB таблиц Oracle8
ftOraClob	Поля CLOB таблиц Oracle8

Как правило, основная часть работы по выборке и помещению данных в компонент класса TBlobField может быть выполнена при их загрузке или сохранении в файле либо с помощью компонента класса TBlobStream. Класс TBlobStream является специализированным, производным от класса TStream, который использует поле BLOB внутри физической таблицы как место размещения потока. Для демонстрации методов взаимодействия с компонентом класса TBlobField создадим демонстрационное приложение.

Пример использования поля BLOB

В описанном далее проекте создается приложение, которое позволяет пользователю сохранять звуковые файлы (.wav) в таблице базы данных, а затем воспроизводить их непосредственно из этой таблицы. Начните проект с создания главной формы с компонентами, показанными на рис. 7.9. Компонент TTable может описывать таблицу Wavez, расположенную в соответствии с псевдонимом DDGUtils, или вашу собственную таблицу с подобной структурой. Структура таблицы Wavez приведена ниже.

Имя поля	*Тип поля*	*Размер*
WaveTitle	Character	25
FileName	Character	25
Wave	BLOB	

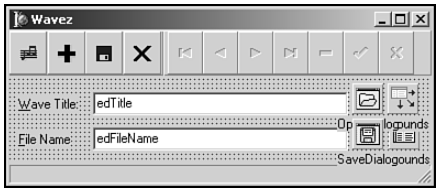

Рис. 7.9. *Главная форма для таблицы Wavez – примера использования поля BLOB*

Кнопка Add используется для загрузки звукового файла с диска и добавления его в таблицу. Подпрограмма обработки события OnClick кнопки Add имеет следующий вид:

```
procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
```

```
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;
```

В этой процедуре сначала предпринимается попытка выполнить метод OpenDialog. Если такая операция выполняется успешно, то источник данных tblSound переключается в режим Append, полю FileName присваивается значение и поле типа BLOB по имени Wave заполняется данными из файла, определенного параметром OpenDialog. Обратите внимание, насколько удобно использовать здесь метод LoadFromFile класса TBlobField и как просто выглядит код загрузки файла в поле BLOB.

Подобным же образом щелчок на кнопке Save приводит к сохранению во внешнем файле содержимого звукового файла, расположенного в поле Wave. Код процедуры для этой кнопки выглядит следующим образом:

```
procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do begin
    FileName := tblSounds['FileName'];    // Получить имя файла
    if Execute then                       // Диалог Execute
      tblSoundsWave.SaveToFile(FileName); // Сохранить blob в файл
  end;
end;
```

В данном методе используется еще меньше кода. Объект SaveDialog инициализируется значением поля FileName. Если процедура SaveDialog выполняется успешно, то для сохранения содержимого поля BLOB в файле вызывается метод SaveToFile объекта tblSoundsWave.

Обработчик кнопки Play считывает звуковые данные из поля BLOB и передает их функции API PlaySound() для воспроизведения. Код этого обработчика приведен ниже. Обратите внимание, что данный код немного сложнее приведенного выше кода для кнопки Save.

```
procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  // Создание потока BLOB
  B := TBlobStream.Create(tblSoundsWave, bmRead);
  Screen.Cursor := crHourGlass; // Отображение песочных часов
  try
    M := TMemoryStream.Create;  // Создание потока памяти
    try
      // Копирование из потока BLOB в поток памяти
      M.CopyFrom(B, B.Size);
      // Попытка воспроизвести звук. Передача исключения,
      // если что-нибудь будет не так, как надо.
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
```

```
  finally
    Screen.Cursor := crDefault;
    B.Free;                      // Освобождение памяти
  end;
end;
```

Сначала этот метод создает экземпляр класса TBlobStream по имени B, используя поле BLOB таблицы tblSoundsWave. Первый параметр, передаваемый методу TBlobStream.Create(), является объектом поля BLOB, а второй параметр указывает, как следует открыть поток. Обычно к потоку BLOB обращаются только для чтения, используя значение bmRead, а для записи используют значение bmReadWrite.

СОВЕТ

Чтобы открыть поток TBlobStream с параметром bmReadWrite используемый набор данных должен быть переведен в режим Edit, Insert или Append.

Затем создается экземпляр M класса потока TMemoryStream. В этот момент обычный указатель мыши изменяется на изображение песочных часов, что указывает пользователю на продолжительность выполняемой операции. Затем поток B копируется в поток M. Функции PlaySound(), применяемой для воспроизведения звукового файла, в качестве первого параметра должно быть передано имя файла или указатель на область памяти. Класс TBlobStream не предоставляет доступа к данным потока с помощью указателя, а класс TMemoryStream обеспечивает такую возможность с помощью свойства Memory. Воспользовавшись этим, можно вызвать функцию PlaySound() для воспроизведения данных, указатель на которые содержится в свойстве M.Memory. После завершения работы этой функции необходимо освободить потоки и восстановить вид указателя мыши. Полный код основного модуля данного проекта приведен в листинге 7.4.

Листинг 7.4. Основной модуль проекта Wavez

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, DBCtrls, DB, DBTables, StdCtrls, Mask,
  Buttons, ComCtrls;

type
  TMainForm = class(TForm)
    tblSounds: TTable;
    dsSounds: TDataSource;
    tblSoundsWaveTitle: TStringField;
    tblSoundsWave: TBlobField;
    edTitle: TDBEdit;
    edFileName: TDBEdit;
    Label1: TLabel;
    Label2: TLabel;
    OpenDialog: TOpenDialog;
```

```
    tblSoundsFileName: TStringField;
    SaveDialog: TSaveDialog;
    pnlToobar: TPanel;
    sbPlay: TSpeedButton;
    sbAdd: TSpeedButton;
    sbSave: TSpeedButton;
    sbExit: TSpeedButton;
    Bevel1: TBevel;
    dbnNavigator: TDBNavigator;
    stbStatus: TStatusBar;
    procedure sbPlayClick(Sender: TObject);
    procedure sbAddClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
  private
    procedure OnAppHint(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses MMSystem;

procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  // Создать поток blob
  B := TBlobStream.Create(tblSoundsWave, bmRead);
  Screen.Cursor := crHourGlass;      // песочные часы
  try
    M := TMemoryStream.Create;       // Создать поток в памяти
    try
      // Копирование из потока BLOB в поток памяти
      M.CopyFrom(B, B.Size);
      // Попытка воспроизвести звук. Передача исключения,
      // если что-нибудь будет не так, как надо.
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
  finally
    Screen.Cursor := crDefault;
    B.Free;                          // Освобождение памяти
  end;
```

```
end;

procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;

procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do begin
    FileName := tblSounds['FileName'];    // Получить имя файла
    if Execute then                       // Диалог Execute
      tblSoundsWave.SaveToFile(FileName); // Сохранить blob в файл
  end;
end;

procedure TMainForm.sbExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := OnAppHint;
  tblSounds.Open;
end;

procedure TMainForm.OnAppHint(Sender: TObject);
begin
  stbStatus.SimpleText := Application.Hint;
end;

procedure TMainForm.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  tblSounds.Close;
end;

end.
```

Фильтрация данных

Фильтры позволяют существенно упростить поиск и выборку записей в наборе данных, даже не прибегая ни к чему более сложному, чем код Object Pascal. Основным

преимуществом фильтров является то, что они не используют индекс и не нуждаются ни в каких иных подготовительных действиях над обрабатываемым набором данных. Обычно фильтрация выполняется несколько медленнее, чем поиск по индексу (речь о котором пойдет в этой главе далее), тем не менее они эффективно используются практически во всех типах приложений.

Использование класса TDataset для фильтрации

Одним из наиболее популярных механизмов фильтрации, используемых в Delphi, является ограничение представления данных из набора до нескольких определенных записей. Этот процесс выполняется следующим образом.

1. Назначьте событию `OnFilterRecord` набора данных процедуру. В эту процедуру следует поместить операторы, выполняющие отбор записей на основе значений одного или нескольких полей.
2. Установите свойство `Filtered` набора данных в состояние `True`.

На рис. 7.10 показана форма, содержащая объект `TDBGrid`, в котором отображены неотфильтрованные данные таблицы `CUSTOMER`.

На первом этапе создадим обработчик события `OnFilterRecord` для этой таблицы. В данном случае будут отбираться только те записи, в которых значение поле `Company` начинается с прописной буквы *S*. Код этой процедуры выглядит следующим образом:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
                                    var Accept: Boolean);
var
  FieldVal: String;
begin
  // Получить значения поля Company
  FieldVal := DataSet['Company'];
  // Принять запись, если поле начинается с буквы S
  Accept := FieldVal[1] = 'S';
end;
```

После выполнения второго этапа (установки свойства `Filtered` таблицы равным `True`) таблица примет вид, показанный на рис. 7.11. В сетке отображаются только те записи, которые удовлетворяют критерию фильтра.

НА ЗАМЕТКУ

Событие `OnFilterRecord` должно использоваться только в тех случаях, когда фильтр не может быть задан как значение свойства `Filter`. В последнем случае может быть достигнут значительный выигрыш в производительности. Например, при работе с базой данных SQL компонент `TTable` будет передавать в базу данных содержимое свойства `FILTER` как условие выражения `WHERE`, которое обычно обрабатывается значительно быстрее, — по сравнению с перебором всех записей, выполняемым в обработчике события `OnFilterRecord`.

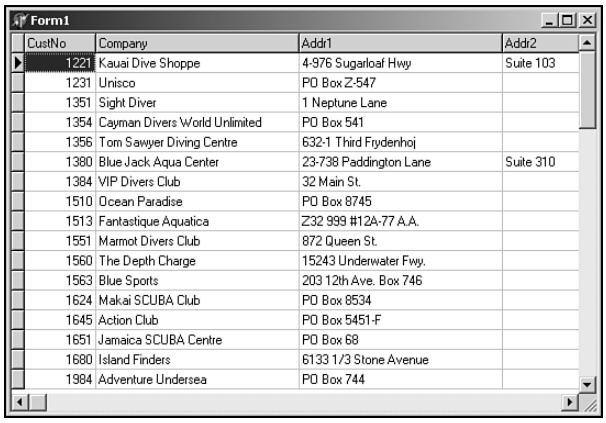

Рис. 7.10. *Неотфильтрованные данные таблицы CUSTOMER*

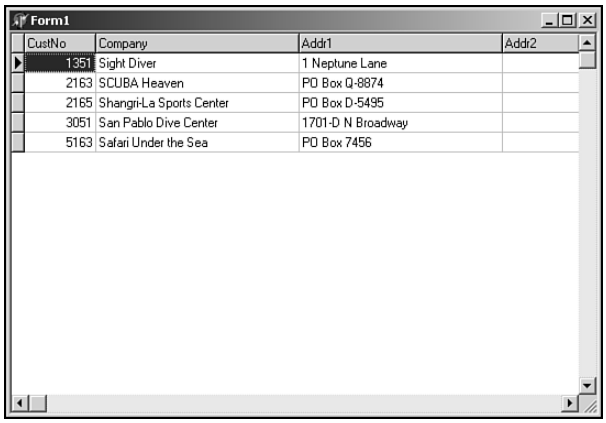

Рис. 7.11. *Отфильтрованные данные таблицы CUSTOMER*

Поиск в наборе данных

Наборы обеспечивают несколько вариантов поиска данных. Здесь рассматриваются методы поиска, не применяющиеся для SQL. Методы поиска SQL рассматриваются в главе 29, "Разработка приложений архитектуры клиент/сервер", предыдущего издания, *Delphi 5 Руководство разработчика*, содержащегося на прилагаемом CD.

Методы FindFirst и FindNext

Класс `TDataSet` обладает методами `FindFirst()`, `FindNext()`, `FindPrior()` и `FindLast()`, которые используют фильтр для поиска записей, отвечающих его критерию поиска. Все эти функции работают на неотфильтрованных наборах данных, вызывая обработчик события `OnFilterRecord`. На основании критерия поиска в обработчике события эти функции способны найти соответственно первую

(first), следующую (next), предыдущую (previous) или последнюю (last) запись. Эти функции не имеют параметров и возвращают значение Boolean, сигнализирующее о том, что текущая запись отвечает условию поиска.

Поиск записи с помощью метода Locate()

Фильтры эффективны не только при определении просматриваемого подмножества записей набора данных, они также могут использоваться для поиска записей в наборе данных по значению одного или нескольких полей. Для этого класс TDataSet содержит метод Locate(). Обратите внимание: функция Locate() выполняет поиск с помощью средств фильтрации и работает независимо от каких-либо индексов, созданных для набора данных. Метод Locate() определен следующим образом:

```
function Locate(const KeyFields: string; const KeyValues: Variant;
                Options: TLocateOptions): Boolean;
```

Первый параметр, KeyFields, содержит имя поля (или полей), по которому проводится поиск. Второй параметр, KeyValues, содержит искомое значение (или значения) поля. Третий параметр, Options, позволяет задать необходимый тип поиска. Этот параметр имеет тип TLocateOptions, который представляет собой множество, определенное в модуле DB следующим образом:

```
type
  TLocateOption = (loCaseInsensitive, loPartialKey);
  TLocateOptions = set of TLocateOption;
```

Если задан параметр loCaseInsensitive, то поиск будет выполняться без учета регистра. Если задан параметр loPartialKey, то значения, содержащиеся в KeyValues, будут считаться удовлетворяющими критерию, даже если они являются подстрокой искомого значения.

Метод Locate() возвращает значение True, если искомая запись найдена. Например, для поиска первого вхождения значения 1356 в поле CustNo набора данных Table1 можно использовать следующий оператор:

```
Table1.Locate('CustNo', 1356, []);
```

СОВЕТ

Везде, где только это возможно, используйте для поиска записей метод Locate(), поскольку он всегда пытается применить самый быстрый из возможных методов поиска элемента, в случае необходимости временно переключаясь на индексный метод поиска. Это сделает программу независимой от индексов. Кроме того, если выяснится, что индекс по некоторому из полей больше не потребуется или, напротив, что добавление индекса увеличит производительность приложения, то можно будет изменить только данные, не изменяя при этом код приложения.

Поиск с помощью методов класса TTable

В настоящем разделе речь пойдет об основных свойствах и методах компонента класса TTable, а также о способах их применения. В частности, будут рассмотрены вопросы поиска записей, отбора записей с использованием диапазонов, а также способы создания таблиц. Здесь описаны события компонента TTable.

Поиск записей

Библиотека VCL содержит несколько методов поиска записей в таблице. При работе с таблицами dBASE или Paradox Delphi предполагает, что все используемые для поиска поля индексированы. Поиск в неиндексированных полях таблиц SQL существенно менее эффективен, чем в индексированных.

В качестве примера рассмотрим таблицу, индексированную по первому полю с числовым типом и по второму полю с текстовым типом. В этом случае поиск записей можно будет осуществлять одним из двух способов: с помощью метода `FindKey()` или используя пару методов `SetKey()..GotoKey()`.

Метод FindKey()

Метод `FindKey()` класса `TTable` позволяет искать запись, по одному или нескольким ключевым полям при одном вызове функции. В качестве параметра методу `FindKey()` передается массив типа `array of const`, содержащий критерии поиска. Если поиск прошел успешно, то метод возвращает значение `True`. Например, следующий оператор осуществит переход к записи в наборе данных, где первое поле в индексе имеет значение `123`, а второе содержит строку `Hello`:

```
if not Table1.FindKey([123, 'Hello']) then MessageBeep(0);
```

Если поле не найдено, то метод `FindKey()` возвращает значение `False` и компьютер издает звуковой сигнал.

Методы SetKey()..GotoKey()

Вызов метода `SetKey()` класса `TTable` переводит таблицу в режим подготовки полей для заполнения значениями критерия поиска. Как только критерий поиска установлен, можно вызывать метод `GotoKey()` для осуществления поиска соответствующий записи в направлении сверху вниз. Используя методы `SetKey()..GotoKey()`, предыдущий пример можно было бы переписать следующим образом:

```
with Table1 do begin
  SetKey;
  Fields[0].AsInteger := 123;
  Fields[1].AsString := 'Hello';
  if not GotoKey then MessageBeep(0);
end;
```

Ближайшее соответствие

Аналогично работает метод `FindNearest()` или пара методов `SetKey()..GotoNearest()`, которыми можно пользоваться для поиска в таблице значения, наиболее близко соответствующего критерию поиска. Для поиска первой записи, где значение первого индексированного поля наиболее близко (больше или равно) числу `123`, используйте следующий код:

```
Table1.FindNearest([123]);
```

И снова в качестве аргумента методу `FindNearest()` передается массив типа `array of const`, содержащий значения полей, по которым выполняется поиск.

Методы `SetKey()..GotoNearest()` можно использовать для поиска следующим образом:

```
with Table1 do begin
  SetKey;
  Fields[0].AsInteger := 123;
  GotoNearest;
end;
```

Если поиск завершится успешно и свойство `KeyExclusive` объекта таблицы имеет значение `False`, то указатель текущей записи будет установлен на первую из записей, соответствующих критерию поиска. Если свойство `KeyExclusive` имеет значение `True`, то текущей станет та запись, которая следует за последней из всех записей, соответствующих условию поиска.

СОВЕТ

Если поиск выполняется по индексированному полю таблицы, то предпочтительнее использовать методы `FindKey()` и `FindNearest()`, а не `SetKey()..GotoX()` — это уменьшит объем программы и, следовательно, количество возможных ошибок.

Использование индексов

Во всех описанных выше методах подразумевалось, что поиск осуществляется с использованием первичного индекса таблицы. Но если поиск проводится по вторичному индексу, то необходимо поместить его имя в параметр `IndexName` объекта таблицы. Например, если в таблице существует вторичный индекс по полю `Company` называемый `ByCompany`, то поиск записи компании "Unisco" можно выполнить следующим образом:

```
with Table1 do begin
    IndexName := 'ByCompany';
    SetKey;
    FieldValues['Company'] := 'Unisco';
    GotoKey;
end;
```

НА ЗАМЕТКУ

Помните, что переключение индекса открытой таблицы создает заметную дополнительную нагрузку на систему. Поэтому присвоение свойству `IndexName` нового значения будет связано с ощутимой задержкой в работе программы — секунда и более.

Диапазоны (range) позволяют отфильтровывать таблицу так, что представлены будут лишь те записи, значения полей которых находится внутри определенных границ. Диапазоны обрабатываются подобно поиску по ключу, и, как и при поиске, существует несколько способов применения диапазона к выбранной таблице. Можно использовать метод `SetRange()` или группу методов `SetRangeStart()`, `SetRangeEnd()` и `ApplyRange()`.

СОВЕТ

При использовании таблиц dBASE или Paradox, диапазоны применимы только к индексированным полям. Применение диапазона к индексированному полю таблиц SQL способно ухудшить эффективность обработки.

Метод SetRange()

Как и метод FindKey() или FindNearest(), метод SetRange() позволяет выполнить достаточно сложные действия над таблицей с помощью единственного вызова этой функции. В качестве параметров методу SetRange() передаются два массива типа array of const. Первый представляет значения полей начала диапазона, а второй — значения полей конца диапазона. Приведенный ниже оператор, выполняет фильтрацию записей, у которых значение первого поля больше или равно 10, но меньше или равно 15:

```
Table1.SetRange([10], [15]);
```

Метод ApplyRange()

Использование метода установки диапазона ApplyRange() предполагает выполнение следующих действий.

1. Вызовите метод SetRangeStart(), а затем модифицируйте массив свойств Fields[] таблицы, установив в нем начальное значение ключевого поля (или полей).
2. Вызовите метод SetRangeEnd() и вновь модифицируйте массив свойств Fields[], установив конечное значение ключевого поля (или полей).
3. Вызовите метод ApplyRange() для установки нового фильтра диапазона.

Используя диапазоны, предыдущий пример можно переписать следующим образом:

```
with Table1 do begin
  SetRangeStart;
  Fields[0].AsInteger := 10;     // Диапазон начинается с 10
  SetRangeEnd;
  Fields[0].AsInteger := 15;     // Диапазон заканчивается на 15
  ApplyRange;
end;
```

СОВЕТ

Для фильтрации записей везде где только возможно используйте метод SetRange() — это уменьшит вероятность появления ошибок в программе.

Для удаления установленного с помощью методов ApplyRange() или SetRange() фильтра, и приведения таблицы в исходное состояние следует вызвать метод CancelRange() объекта TTable.

```
Table1.CancelRange;
```

Использование модулей данных

Модули данных (data modules) позволяют расположить все *правила* (rule) и *отношения* (relationship) базы данных в одном месте для того, чтобы обеспечить их совместное использование отдельными проектами, группами или организациями. В библиотеке VCL модули данных инкапсулирует класс TDataModule. Компонент этого класса

можно представить себе как скрытую форму, в которую помещаются все используемые в проекте компоненты доступа к данным. Создать экземпляр компонента TDataModule несложно — в главном меню File выберите пункт New, а затем в окне Object Repository — объект Data Module.

Самая очевидная причина, по которой предпочтительнее использовать компонент TDataModule, нежели помещать компоненты доступа к данным в настоящую форму, заключается в том, что это упрощает доступ к одним и тем же данным из нескольких форм и модулей проекта. В более сложных ситуациях может потребоваться упорядочить использование многочисленных компонентов TTable, TQuery и/или TStoredProc, установить постоянные отношения между компонентами и, возможно, правила, определенные на уровне поля (например, минимальные/максимальные значения или форматы отображения). В целом, этот ассортимент компонентов доступа к данным будет отображать бизнес-правила всего предприятия. Выполнив множество подобных действий, их вряд ли захочется повторять вновь и вновь для других приложений. Чтобы избежать этого, поместите созданный модуль данных в хранилище объектов (Object Repository). Это позволит использовать его многократно. При проведении работ группой разработчиков, хранилище объектов следует разместить на общедоступном сетевом диске — это обеспечит доступ к нему всем разработчикам группы.

В приведенном далее примере создается экземпляр простого модуля данных, благодаря которому из многих форм можно будет получить доступ к одним и тем же данным. В приложениях баз данных, рассматриваемых в последующих главах, в модулях данных будут устанавливаться более сложные отношения.

Пример применения поиска, фильтра и диапазона

Теперь рассмотрим демонстрационное приложение, в котором использованы важнейшие концепции, обсуждавшиеся в настоящей главе. В частности, в этом приложении продемонстрировано правильное использование методов фильтрации, поиска по индексу и установки диапазонов данных. Предлагаемый проект называется SRF и включает в себя несколько форм. В главной форме содержится сетка, предназначенная для просмотра данных таблицы, а остальные формы демонстрируют примеры вышеописанных концепций. Каждая из этих форм подробно рассматривается в настоящей главе.

Модуль данных

Хоть это и не совсем правильно, но, нарушая порядок построения приложения, рассмотрим сперва модуль данных. Этот модуль данных, по имени DM, содержит лишь компоненты классов TTable и TDataSource. Компонент TTable, по имени Table1, связан с таблицей CUSTOMERS.DB базы данных под псевдонимом DBDEMOS. Компонент класса TDataSource по имени DataSource1 связан с компонентом Table1. Все элементы управления для работы с данными в этом проекте будут использовать компонент DataSource1 в качестве источника данных. Модуль данных DM содержится в программном модуле DataMod.

Главная форма

Главная форма проекта SRF называется MainForm и представлена на рис. 7.12. Ее код содержится в модуле Main. Форма включает в себя элемент управления TDBGrid по имени DBGrid1, предназначенный для просмотра таблицы, а также переключатель, позволяющий воспользоваться различными индексами таблицы. Компонент DBGrid1, как уже упоминалось ранее, связан с источником данных DM.DataSource1.

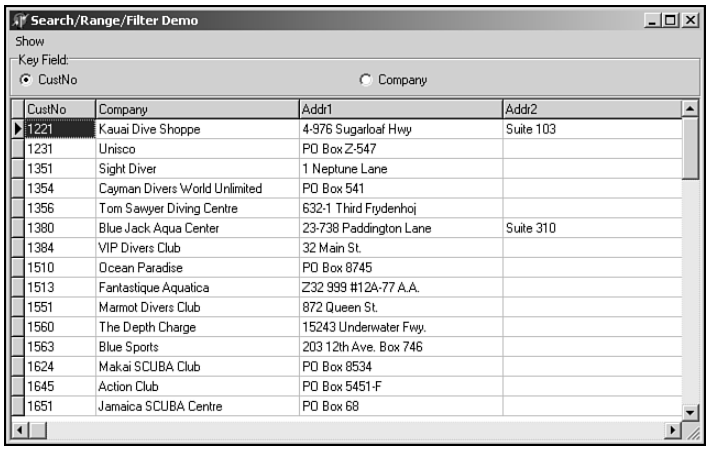

Рис. 7.12. *Главная форма проекта SRF*

НА ЗАМЕТКУ

Для того чтобы компонент DBGrid1 можно было связать с источником данных DM.DataSource1 в режиме разработки, модуль DataMod должен быть указан в разделе uses модуля Main. Простейший способ сделать это — отобразить модуль Main в окне редактора кода, а затем выбрать в меню File пункт Use Unit главного меню. На экране раскроется диалоговое окно со списком модулей проекта, в котором следует выбрать модуль DataMod. Эту же операцию необходимо повторить для каждого модуля, из которого будет осуществляться доступ к данным, содержащимся в модуле DM.

Переключатель RGKeyField используется для определения того, какой из двух индексов таблицы будет активен в текущий момент. Код обработчика события OnClick переключателя RGKeyField выглядит следующим образом:

```
procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';           // Первичный индекс
    1: DM.Table1.IndexName := 'ByCompany';  // Вторичный индекс
  end;
end;
```

Форма MainForm содержит также компонент TMainMenu по имени MainMenu1, позволяющий открывать и закрывать каждую из форм приложения. Определены

следующие пункты меню: Key Search, Range, Filter и Exit. Полный код модуля Main.pas приведен в листинге 7.5.

Листинг 7.5. Код модуля Main.pas

```
unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls, Grids, DBGrids,
  DB, DBTables, Buttons, Mask, DBCtrls, Menus, KeySrch, Rng, Fltr;

type
  TMainForm = class(TForm)
    DBGrid1: TDBGrid;
    RGKeyField: TRadioGroup;
    MainMenu1: TMainMenu;
    Forms1: TMenuItem;
    KeySearch1: TMenuItem;
    Range1: TMenuItem;
    Filter1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure RGKeyFieldClick(Sender: TObject);
    procedure KeySearch1Click(Sender: TObject);
    procedure Range1Click(Sender: TObject);
    procedure Filter1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

uses DataMod;

{$R *.DFM}

procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';           // Первичный индекс
    1: DM.Table1.IndexName := 'ByCompany';  // Вторичный индекс
  end;
end;
```

```
procedure TMainForm.KeySearch1Click(Sender: TObject);
begin
  KeySearch1.Checked := not KeySearch1.Checked;
  KeySearchForm.Visible := KeySearch1.Checked;
end;

procedure TMainForm.Range1Click(Sender: TObject);
begin
  Range1.Checked := not Range1.Checked;
  RangeForm.Visible := Range1.Checked;
end;

procedure TMainForm.Filter1Click(Sender: TObject);
begin
  Filter1.Checked := not Filter1.Checked;
  FilterForm.Visible := Filter1.Checked;
end;

procedure TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;

end.
```

НА ЗАМЕТКУ

Обратите внимание, что в коде модуля `Rng` (относящегося к этому проекту, но не приведенному здесь) содержится следующая строка:

```
DM.Table1.SetRange([StartEdit.Text], [EndEdit.Text]);
```

Может показаться странным тот факт, что методу `SetRange()` всегда передается строковое значение, несмотря на то что ключевое поле может быть числового или текстового типа. Delphi допускает это, поскольку методы `SetRange()`, `FindKey()` и `FindNearest()` автоматически выполняют преобразование из типа `String` в тип `Integer`, и наоборот.

Это означает, что в данной ситуации нет необходимости заботится о вызове функции `IntToStr()` или `StrToInt()` — все преобразования будут выполнены автоматически.

Форма поиска по ключевому значению

Код формы `KeySearchForm` содержится в модуле `KeySrch`. Она предназначена для поиска в таблице записи с конкретным значением. Поиск может выполняться одним из двух предлагаемых способов. Если переключатель установлен в положение Normal, пользователь может ввести в поле Search for искомое значение и, щелкнув на кнопке Exact или Nearest, выполнять поиск записи, точно отвечающей условию или ближайшей к нему. Если переключатель установлен в положение Incremental, то в таблице будет выполняться пошаговый поиск непосредственно в процессе ввода условия поиска в поле Search for. При этом обращение к таблице выполняется при каждом изменении значения в данном поле. Код модуля `KeySrch` приведен в листинге 7.6.

Листинг 7.6. Исходный код модуля `KeySrch.PAS`

```
unit KeySrch;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TKeySearchForm = class(TForm)
    Panel1: TPanel;
    Label3: TLabel;
    SearchEdit: TEdit;
    RBNormal: TRadioButton;
    Incremental: TRadioButton;
    Label6: TLabel;
    ExactButton: TButton;
    NearestButton: TButton;
    procedure ExactButtonClick(Sender: TObject);
    procedure NearestButtonClick(Sender: TObject);
    procedure RBNormalClick(Sender: TObject);
    procedure IncrementalClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
  private
    procedure NewSearch(Sender: TObject);
  end;

var
  KeySearchForm: TKeySearchForm;

implementation

uses DataMod, Main;

{$R *.DFM}

procedure TKeySearchForm.ExactButtonClick(Sender: TObject);
{ Попытка найти ту запись, где ключевое поле точно соответствует
значению поля SearchEdit. Заметьте, что Delphi обеспечивает
преобразование типов из строкового значения поля в числовое
значение ключевого поля. }
begin
if not DM.Table1.FindKey([SearchEdit.Text]) then
    MessageDlg(Format('Match for "%s" not found.',
               [SearchEdit.Text]),
               mtInformation, [mbOk], 0);
end;

procedure TKeySearchForm.NearestButtonClick(Sender: TObject);
```

```
{ Поиск ближайшего соответствия значению в поле SearchEdit.
Заметьте, что снова выполняется неявное преобразование типа.}
begin
  DM.Table1.FindNearest([SearchEdit.Text]);
end;

procedure TKeySearchForm.NewSearch(Sender: TObject);
{ Этот метод связывается с событием OnChange поля SearchEdit при
установке переключателя в положение Incremental. }
begin
  DM.Table1.FindNearest([SearchEdit.Text]); // Поиск текста
end;

procedure TKeySearchForm.RBNormalClick(Sender: TObject);
begin
  ExactButton.Enabled := True;   // Кнопка Search доступна
  NearestButton.Enabled := True;
  SearchEdit.OnChange := Nil;    // Отключение события OnChange
end;

procedure TKeySearchForm.IncrementalClick(Sender: TObject);
begin
  ExactButton.Enabled := False;      // Кнопка Search недоступна
  NearestButton.Enabled := False;
  SearchEdit.OnChange := NewSearch;  // Передача события OnChange
  NewSearch(Sender);           // Поиск текущего фрагмента текста
end;

procedure TKeySearchForm.FormClose(Sender: TObject;
                                   var Action: TCloseAction);
begin
  Action := caHide;
  MainForm.KeySearch1.Checked := False;
end;

end.
```

Текст приведенного модуля не должен вызывать вопросов. В нем методам `FindKey()` и `FindNearest()` вновь передается строковое значение в полной уверенности, что требуемое преобразование типов произойдет автоматически. Полагаем, вы оцените небольшой трюк, используемый для включения и отключения пошагового поиска "на лету". Для этого событию `OnChange` элемента управления `SearchEdit` присваивается либо необходимый метод, либо значение `Nil`. При назначении в качестве обработчика реального метода, событие `OnChange` будет передаваться всякий раз при изменении значения в текстовом поле. Вызов в этом обработчике метода `FindNearest()` позволяет организовать в таблице пошаговый поиск, выполняемый при любом изменении пользователем условия поиска.

Форма фильтра

Расположенная в модуле Fltr форма FilterForm предназначена для решения двух задач. Во-первых, она позволяет отфильтровать отображаемые в таблице данные по значению поля State, которое должно соответствовать заданному пользователем значению. Во-вторых, эта форма позволяет выбирать из таблицы те записи, где значение некоторого поля равно значению, введенному пользователем.

Для реализации фильтра требуется не так уж и много кода. Состояние флажка Filter on this State (объект cbFiltered) определяет значение свойства Filtered объекта DM.Table1. Это реализовано с помощью показанного далее оператора в обработчике события cbFiltered.OnClick:

```
DM.Table1.Filtered := cbFiltered.Checked;
```

Когда свойство DM.Table1.Filtered принимает значение True, таблица Table1 фильтрует записи с помощью приведенного ниже метода OnFilterRecord, который расположен в модуле DataMod.

```
procedure TDM.Table1FilterRecord(DataSet: TDataSet;
                                 var Accept: Boolean);
{ Запись выбирается на обработку, если значение поля State равно
значению, введенному в текстовое поле DBEdit1.Text. }
begin
  Accept := Table1State.Value = FilterForm.DBEdit1.Text;
end;
```

Для выполнения условного поиска необходимо использовать метод Locate() компонента TTable таким образом:

```
DM.Table1.Locate(CBField.Text, EValue.Text, LO);
```

Имя поля выбирается пользователем из раскрывающегося списка CBField. Содержимое списка создается обработчиком события OnCreate формы с помощью следующего кода, выполняющего просмотр полей таблицы Table1:

```
procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;
```

СОВЕТ

Приведенный код работает только в том случае, если модуль данных DM был создан раньше модуля формы. Любые попытки доступа к модулю данных до его создания приведут, скорее всего, к появлению ошибки типа Access Violation. Чтобы иметь уверенность, что модуль данных DM будет создан раньше, чем любая из дочерних форм, мы вручную отредактировали порядок создания форм в списке Auto-create forms, расположенном во вкладке Forms диалогового окна Project Options (это окно можно вызвать, выбрав в меню Project пункт Options).

Безусловно, главная форма будет создана прежде всего, но следует иметь гарантии, что сразу за ней последует модуль данных — прежде, чем будет создана любая другая форма приложения.

Полный код модуля `Fltr` приведен в листинге 7.7.

Листинг 7.7. Исходный код модуля `Fltr.pas`

```
unit Fltr;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons, Mask, DBCtrls, ExtCtrls;

type
  TFilterForm = class(TForm)
    Panel1: TPanel;
    Label4: TLabel;
    DBEdit1: TDBEdit;
    cbFiltered: TCheckBox;
    Label5: TLabel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    Panel2: TPanel;
    EValue: TEdit;
    LocateBtn: TButton;
    Label1: TLabel;
    Label2: TLabel;
    CBField: TComboBox;
    MatchGB: TGroupBox;
    RBExact: TRadioButton;
    RBClosest: TRadioButton;
    CBCaseSens: TCheckBox;
    procedure cbFilteredClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure LocateBtnClick(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure SpeedButton3Click(Sender: TObject);
    procedure SpeedButton4Click(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
  end;

var
  FilterForm: TFilterForm;
```

```
implementation

uses DB, DataMod, Main;

{$R *.DFM}

procedure TFilterForm.cbFilteredClick(Sender: TObject);
begin
  { Фильтровать таблицу, если установлен соответствующий флажок }
  DM.Table1.Filtered := cbFiltered.Checked;
end;

procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;

procedure TFilterForm.LocateBtnClick(Sender: TObject);
var
  LO: TLocateOptions;
begin
  LO := [];
  if not CBCaseSens.Checked then Include(LO, loCaseInsensitive);
  if RBClosest.Checked then Include(LO, loPartialKey);
  if not DM.Table1.Locate(CBField.Text, EValue.Text, LO) then
    MessageDlg('Unable to locate match', mtInformation,
               [mbOk], 0);
end;

procedure TFilterForm.SpeedButton1Click(Sender: TObject);
begin
  DM.Table1.FindFirst;
end;

procedure TFilterForm.SpeedButton2Click(Sender: TObject);
begin
  DM.Table1.FindNext;
end;

procedure TFilterForm.SpeedButton3Click(Sender: TObject);
begin
  DM.Table1.FindPrior;
end;

procedure TFilterForm.SpeedButton4Click(Sender: TObject);
begin
  DM.Table1.FindLast;
end;
```

```
procedure TFilterForm.FormClose(Sender: TObject;
                                var Action: TCloseAction);
begin
  Action := caHide;
  MainForm.Filter1.Checked := False;
end;

end.
```

Закладки

Закладки (bookmarks) позволяют сохранить положение в наборе данных, чтобы позднее можно было вновь вернуться к этому же месту. В Delphi работать с закладками очень просто, поскольку необходимо запомнить значение всего одного свойства.

Delphi представляет закладу как тип данных `TBookmarkStr`. Класс `TTable` обладает свойством `Bookmark` данного типа. При чтении значения из этого свойства будет возвращено имя закладки, а при его записи — переход к указанной закладке. Если какое-то место в наборе данных вызвало определенный интерес и к нему потребуется вернуться позднее, используйте следующий фрагмент кода:

```
var
  BM: TBookmarkStr;
begin
  BM := Table1.Bookmark;
```

Когда потребуется вернуться к помеченному месту в наборе данных, сделать это будет очень просто — достаточно присвоить свойству `Bookmark` значение, полученное ранее при чтении свойства `Bookmark`:

```
Table1.Bookmark := BM;
```

Тип `TBookmarkStr` определен как `AnsiString`, поэтому память для закладки выделяется автоматически (заботится о ее освобождении не нужно). Если существующую закладку необходимо удалить, то достаточно установить ее значение равным пустой строке:

```
BM := '';
```

Обратите внимание: тип `TBookmarkStr` определен как `AnsiString` лишь для удобства применения. Его можно рассматривать как "черный ящик", который не зависит от реализации, поскольку реальные данные закладки полностью определяются BDE и более низкими уровнями управления данными.

НА ЗАМЕТКУ

В 32-разрядной версии Delphi по-прежнему поддерживаются методы `GetBookmark()`, `GotoBookmark()` и `FreeBookmark()`, пришедшие из Delphi 1. Но использовать их имеет смысл только в том случае, если необходимо сохранить совместимость с 16-разрядными проектами.

Примеры использования закладок для работы с набором данных ADO находятся на прилагаемом CD в каталоге `\Bookmark`.

Резюме

Авторы полагают, что, прочитав эту главу, можно создавать в Delphi любые типы приложений баз данных. В настоящей главе был рассмотрен базовый компонент `TDataSet` и производные от него компоненты `TTable`, `TQuery` и `TStoredProc`. Обсуждалось также, как оперировать с объектами `TTable`, управлять полями и работать с текстовыми таблицами.

В следующих главах рассматриваются такие технологии разработки баз данных Delphi, как dbExpress и dbGo, а также более подробно освещаются вопросы взаимодействия приложений Delphi с данными ADO.

ГЛАВА

8

Применение dbExpress при разработке баз данных

В ЭТОЙ ГЛАВЕ...

dbExpress — это новая технология компании *Borland*, позволяющая упростить процесс разработки баз данных в Delphi 6.

Данная технология обладает тремя важными преимуществами. Во-первых, средства dbExpress гораздо проще с точки зрения установки по сравнению с их предшественником BDE. Во-вторых, технология dbExpress является межплатформенной. Это означает, что разработанные приложения могут использоваться и в среде Kylix на платформе Linux. В-третьих, для dbExpress можно создавать новые драйверы. При этом достаточно реализовать соответствующий интерфейс и разработать библиотеку, обладающую методами доступа к базе данных.

В основе структуры dbExpress лежат драйверы баз данных, каждый из которых реализует набор интерфейсов, осуществляющих доступ к соответствующему серверу данных. Эти драйверы взаимодействуют с приложениями через компоненты соединения DataCLX, наподобие того, как компоненты `TDatabase` взаимодействуют с драйверами BDE (но без излишних трудностей).

Применение dbExpress

DbExpress разработан как эффективное средство доступа к данным, несущее минимальные непроизводительные затраты. Для этого dbExpress использует *односторонние наборы данных* (unidirectional dataset).

Односторонние наборы данных

Суть односторонних наборов данных заключается в том, что они не буферизируют данные при навигации или модификации. В отличие от используемых в BDE двухсторонних наборов данных с буферизацией в оперативной памяти, односторонние наборы отличаются большей эффективностью, но обладают и некоторыми ограничениями:

- Односторонние наборы данных обладают всего лишь двумя навигационными методами: `First()` и `Next()`. Попытка вызова иных методов, например `Last()` или `Prior()`, приведет к исключению.
- Данные односторонних наборов нельзя редактировать, поскольку для них не выделяется буфер, пригодный для редактирования. Обратите внимание: если данные необходимо редактировать, то применяются *другие* компоненты (`TClientDataset` и `TSQLClientDataset`), рассматриваемые далее.
- Односторонние наборы данных не поддерживают фильтрацию, поскольку отсутствие буфера не позволяет создавать набор для нескольких записей.
- Односторонние наборы данных не поддерживают подстановочные поля.

DbExpress против Borland Database Engine (BDE)

По сравнению с BDE, dbExpress обладает некоторыми преимуществами. Рассмотрим их кратко.

В отличие от BDE, средства dbExpress не расходуют ресурсы сервера на хранение метаданных внешних запросов.

Кроме того, при работе с dbExpress расходуется меньше ресурсов клиента, так как при использовании односторонних курсоров отсутствует кэширование данных. При этом метаданные на стороне клиента также не кэшируются, поскольку доступ к ним осуществляется через процедуры интерфейса DLL.

В отличие от BDE, в dbExpress не создаются внутренние запросы для таких задач, как навигация и обращение к данным типа `BLOB`. Благодаря этой особенности средства dbExpress во время выполнения приложений работают намного эффективнее, поскольку на сервере обрабатываются лишь пользовательские запросы. Другими словами, dbExpress намного проще BDE.

Использование dbExpress при разработке межплатформенных приложений

Главным преимуществом dbExpress является возможность переноса приложений с платформы Windows (Delphi 6) на платформу Linux (Kylix). Разработанное в Delphi приложение, в котором используются компоненты CLX, может быть откомпилировано в Kylix и работать в среде Linux. Фактически средства dbExpress взаимодействуют с серверами, не зависящими от типа платформы (например MySQL или InterBase).

НА ЗАМЕТКУ

На момент написания этой книги в dbExpress была реализована поддержка MySQL версии 3.22. Тем не менее, после обновления соответствующей библиотеки DLL в Delphi 6 можно работать и с более поздней версией 3.23 На данный момент в *Borland* ведутся работы по обновлению этой библиотеки.

Компоненты dbExpress

Все компоненты dbExpress расположены во вкладке dbExpress палитры компонентов.

Компонент TSQLConnection

Тому, кто раньше разрабатывал приложения с использованием BDE, компонент класса `TSQLConnection` покажется очень похожим на компонент класса `TDatabase`. И в самом деле — оба эти компонента инкапсулируют методы подключения к базам данных. Именно через компонент `TSQLConnection` наборы данных dbExpress осуществляют доступ к серверу.

Объекты класса `TSQLConnection` используют два конфигурационных файла: `dbxdrivers.ini` и `dbxconnections.ini`. При установке эти файлы размещаются в каталоге `\Program Files\Common Files\Borland Shared\DbExpress`. Файл `dbxdrivers.ini` содержит перечень всех драйверов, поддерживаемых dbExpress, и их специфические параметры. Файл `dbxconnections.ini` содержит перечень так называемых *именованных соединений* (named connections), которые по сути аналогичны *псевдонимам* (alias) BDE. В этом же файле хранятся и параметры, специфические для каждого именованного соединения. Во время выполнения приложения установленный по умолчанию файл `dbxconnections.ini` можно не использовать. Для этого

свойству TSQLConnection.LoadParamsOnConnect необходимо присвоить значение True. Соответствующий пример будет рассмотрен немного позднее.

Для подключения к базе данных компонент TSQLConnection использует соответствующий драйвер dbExpress, указанный в файле dbxdrivers.ini.

Методы и свойства класса TSQLConnection достаточно полно описаны в интерактивной справочной системе. Как обычно, за более подробной информацией рекомендуем обратиться к интерактивной справочной системе, а в этой книге рассмотрим только процесс подключения к базе данных и создание нового соединения.

Подключение к базе данных

Для подключения к уже существующей базе данных достаточно просто разместить в форме компонент TSQLConnection и выбрать одно из значений в раскрывающемся списке свойства ConnectionName в инспекторе объектов. В этом списке должно быть, по крайней мере, четыре элемента: IBLocal, DB2Connection, MSConnection и Oracle. Если при установке Delphi не был установлен InterBase, то сделайте это прямо сейчас, поскольку он будет использоваться в примерах. Установив InterBase, выберите в списке свойства ConnectionName значение IBLocal. Это соединение соответствует локальной базе данных InterBase, которая устанавливается вместе с Delphi 6.

После выбора значения для свойства ConnectionName будут автоматически инициализированы такие свойства, как DriverName, GetDriverFunc, LibraryName, и VendorLib. Это стандартные значения, хранящиеся в файле dbxdrivers.ini. Остальные параметры драйвера можно просмотреть и модифицировать в редакторе свойства Params (рис. 8.1).

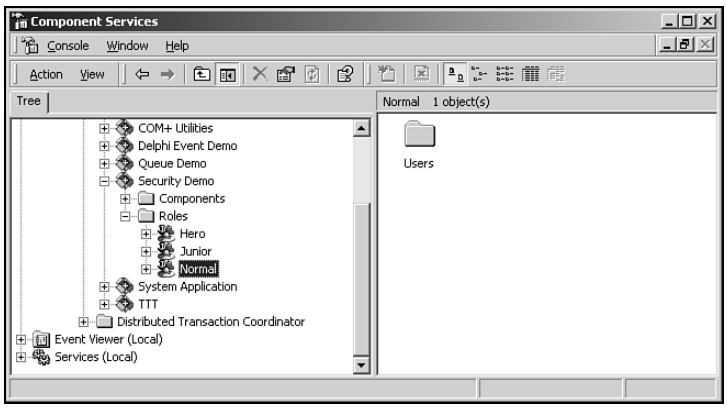

Рис. 8.1. *Редактор свойства TSQLConnection.Params*

НА ЗАМЕТКУ

По умолчанию в редакторе свойства Params для параметра Database указано значение database.gdb. На самом же деле базы данных с таким названием не существует, поэтому данное значение можно изменить на Employee.gdb. Эта база данных устанавливается вместе с InterBase и находится обычно в каталоге ...\Program Files\Borland\ InterBase6\examples\Database\Employee.gdb.

После того как для компонента класса TSQLConnection будет подключен к реальной базе данных, можно присвоить свойству Connected значение True. В результате на экране появится приглашение ввести имя пользователя и пароль. Введите в соответствующих полях значения "**sysdba**" и "**masterkey**". Подключение к базе данных будет завершено. Остальные свойства класса TSQLConnection подробно описаны в интерактивной справочной системе.

Создание нового соединения

Для обращения к базе данных можно создавать дополнительные именованные соединения. Они, например, могут использоваться в тех приложениях, которые подключаются к двум различным базам данных. Чтобы создать новое соединение, достаточно открыть редактор свойств подключения, дважды щелкнув на компоненте SQLConnection (рис. 8.2). Этот редактор можно вызвать иначе, щелкнув правой кнопкой мыши на компоненте класса TSQLConnection и выбрав из контекстного меню пункт Edit Connection Properties (Изменить свойства подключения).

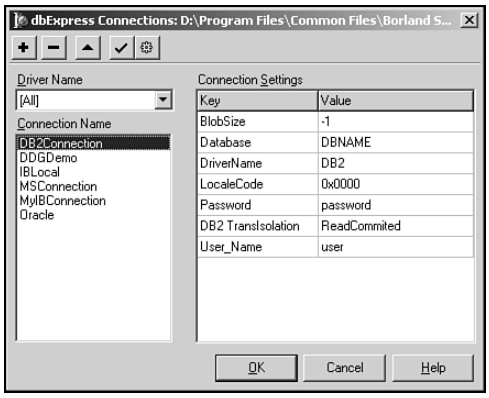

Рис. 8.2. *Компонент соединения TSQLConnection в окне редактора свойств*

На панели инструментов данного редактора свойств расположено пять кнопок. Рассмотрим кнопку Add (Добавить). Если щелкнуть на этой кнопке, то на экране появится диалоговое окно, в котором можно указать имя и драйвер для нового соединения. В раскрывающемся списке Driver Name можно выбрать один из четырех возможных драйверов. В данном случае выберем драйвер InterBase. В поле Connection Name (Имя соединения) можно ввести любую строку, например "**MyIBConnection**". После щелчка на кнопке OK параметры соединения, совпадающие со значениями свойства TSQLConnection.Params, будут отображены в таблице Connection Settings. Для нового соединения также необходимо изменить значение параметра Database, чтобы он указывал на реальную базу данных InterBase. Теперь можно закрыть редактор свойств соединения, присвоить свойству Connected значение True и ввести имя пользователя и пароль, в результате будет осуществлено подключение к базе данных.

Ввод имени пользователя и пароля при подключении к базе данных

Присвоив значение False свойству LoginPrompt, можно отключить запрос на ввод имени пользователя и пароля при подключении к базе данных. Однако при этом для параметров UserName и Password в редакторе свойства Params должны быть указаны корректные значения.

Для того чтобы изменить стандартный запрос на ввод имени пользователя и пароля, необходимо присвоить свойству LoginPrompt значение True, а в обработчик события OnLogin добавить примерно такой код:

```
procedure TMainForm.SQLConnection1Login(Database: TSQLConnection;
                                        LoginParams: TStrings);
var
  UserName: String;
  Password: String;
begin
  if InputQuery('Get UserName', 'Enter UserName', UserName) then
    if InputQuery('Get Password', 'Enter Password', Password) then
    begin
      LoginParams.Values['UserName'] := UserName;
      LoginParams.Values['Password'] := Password;
    end;
end;
```

В этом примере для ввода значений используется функция InputQuery(), но можно разработать и собственное диалоговое окно. Пример, демонстрирующий пользовательское диалоговое окно регистрации, находится на прилагаемом компакт-диске. Кроме того, в этом примере демонстрируется использование событий AfterConnect и AfterDisconnect.

Загрузка параметров подключения во время выполнения приложения

Параметры подключения, которые отображаются в редакторе свойства Params во время разработки, по умолчанию загружаются из файла dbxconnections.ini. Значения таких параметров можно извлекать и во время выполнения приложения. Это может понадобиться, например, в том случае, если используется файл dbxconnections.ini, отличный от того, который поставляется с Delphi. Конечно, при этом модифицированный конфигурационный файл должен распространяться вместе с приложением.

Для загрузки значений параметров подключения во время выполнения приложения необходимо присвоить значение True свойству LoadParamsOnConnect. При загрузке приложения компонент класса TSQLConnection определяет размещение файла dbxconnections.ini на основании содержимого ключа системного реестра Connection Registry File в ветви HKEY_CURRENT_USER\Software\Borland\DBExpress. Изменение значения этого ключа можно сделать в момент установки приложения.

Класс TSQLDataset

Класс TSQLDataset используется для извлечения данных с сервера при помощи одностороннего курсора. Такие наборы данных используются для отображения содержимого таблиц, а также результатов выполнения запросов или хранимых процедур. Кроме того, объекты данного класса способны выполнять хранимые процедуры.

Основными свойствами класса TSQLDataset являются CommandType и CommandText. Значение свойства CommandType определяет характер использования содержимого свойства CommandText. Возможные значения свойства CommandType перечислены в табл. 8.1, а также в интерактивной справочной системе Delphi.

Таблица 8.1. Возможные значения свойства CommandType (по материалам справочной системы Delphi)

CommandType	*Содержимое свойства* CommandText
ctQuery	Оператор SQL
ctStoredProc	Имя хранимой процедуры
ctTable	Имя таблицы, расположенной на сервере баз данных. Для извлечения всех записей и полей этой таблицы компонент TSQLDataset автоматически создает оператор SQL SELECT

Если свойство CommandType содержит значение ctQuery, то свойство CommandText содержит какой-либо оператор SQL. Например, это может быть оператор SELECT, предназначенный для возврата некоторого набора данных: "SELECT * FROM CUSTOMER".

Если свойство CommandType содержит значение ctTable, то свойство CommandText содержит имя таблицы, расположенной на сервере баз данных. В этом случае поле свойства CommandText в инспекторе объектов примет вид раскрывающего списка. В случае подключения к SQL-серверу все операторы SQL, необходимые для извлечения данных, создаются автоматически.

Если свойство CommandType содержит значение ctStoredProc, то свойство CommentText содержит имя хранимой процедуры. Для выполнения этой процедуры лучше использовать метод TSQLDataSet.ExecSQL(), а не присваивать значение True свойству Active. Кроме того, метод ExecSQL() может быть использован в том случае, если свойство CommandType имеет значение ctQuery и оператор SQL не возвращает никакого набора данных.

Извлечение данных из таблиц

Для того чтобы извлечь данные из таблицы при помощи компонента TSQLDataset, нужно просто присвоить значение ctTable свойству TSQLDataSet.CommandType. В результате поле свойства CommandText в инспекторе объектов примет вид раскрывающегося списка с перечнем имен доступных таблиц. Соответствующий пример можно найти на прилагаемом компакт-диске в каталоге TableData.

Отображение результатов запросов

Для извлечения данных с помощью оператора SQL SELECT следует просто присвоить значение ctQuery свойству TSQLDataSet.CommandType. В результате в поле свойства CommandText можно будет ввести запрос вида "Select * from Country". Соответствующий пример находится на прилагаемом компакт-диске в каталоге QueryData.

Отображение результатов, возвращаемых хранимыми процедурами

При помощи компонента TSQLDataset можно получать наборы данных, возвращаемые хранимыми процедурами InterBase. Например, следующим образом:

```
CREATE PROCEDURE SELECT_COUNTRIES RETURNS (
  RCOUNTRY VARCHAR(15),
  RCURRENCY VARCHAR(10)
) AS
BEGIN
  FOR SELECT
    COUNTRY, CURRENCY FROM COUNTRY
  INTO
    :rCOUNTRY, :rCURRENCY
  DO
    SUSPEND;
END
```

Для этого нужно присвоить свойству TSQLDataset.CommandType значение ctQuery, а свойству CommandText — значение "select * from SELECT_COUNTRIES". Обратите внимание: имя хранимой процедуры используется точно так же, как и имя таблицы.

Выполнение хранимых процедур

При помощи компонента TSQLDataset можно выполнять хранимые процедуры, не возвращающие результата. Для этого необходимо присвоить свойству TSQLDataSet.CommandType значение ctStoredProc. В результате поле свойства TSQLDataset.CommandText в инспекторе объектов примет вид раскрывающегося списка с перечнем доступных хранимых процедур. Теперь в этом списке можно выбрать одну из тех процедур, которые не возвращают результата. Соответствующий пример можно найти на прилагаемом компакт-диске в каталоге ExecSProc. В нем выполняется следующая хранимая процедура:

```
CREATE PROCEDURE ADD_COUNTRY (
  ICOUNTRY VARCHAR(15),
  ICURRENCY VARCHAR(10)
) AS
BEGIN
  INSERT INTO COUNTRY(COUNTRY, CURRENCY)
  VALUES (:iCOUNTRY, :iCURRENCY);
  SUSPEND;
END
```

Эта процедура просто добавляет запись в таблицу COUNTRY. Для ее выполнения вызывается метод TSQLDataset.ExecSQL():

```
procedure TForm1.btnAddCurrencyClick(Sender: TObject);
begin
  sqlDSAddCountry.ParamByName('ICountry').AsString :=
                                          edtCountry.Text;
  sqlDSAddCountry.ParamByName('ICURRENCY').AsString :=
                                          edtCurrency.Text;
  sqlDSAddCountry.ExecSQL(False);
end;
```

В первую очередь присваиваются значения параметрам хранимой процедуры, а затем выполняется сама процедура при помощи метода `ExecSQL()`. Обратите внимание на логический параметр метода `ExecSQL()` (по умолчанию имеет значение `True`). Он указывает на то, нуждаются параметры хранимой процедуры в предварительной подготовке или нет.

Представление метаданных

При помощи компонента `TSQLDataset` можно получить информацию о самой базе данных. Для этого используется процедура `TSQLDataset.SetSchemaInfo()`, в которой задается требуемый тип информации о структуре. Определение процедуры `SetSchemaInfo` имеет следующий вид:

```
procedure SetSchemaInfo( SchemaType: TSchemaType;
                         SchemaObjectName, SchemaPattern: string );
```

Параметр `SchemaType` определяет тип схемы запрашиваемой информации. Параметр `SchemaObjectName` содержит имя таблицы или процедуры, в случае запроса информации о столбце или индексе, а параметр `SchemaPattern` — маску шаблона SQL для фильтрации возвращаемого набора данных.

Табл. 8.2 взята из интерактивной справочной системы Delphi. В ней описаны все типы схем информации, которые можно получить при помощи процедуры `SetSchemaInfo()`.

Таблица 8.2. Возможные значения параметра `SchemaType` (по материалам справочной системы Delphi)

Значение `SchemaType`	*Описание*
`stNoSchema`	Информация о схеме не возвращается. Вместо метаданных с сервера возвращается набор данных, соответствующий указанному запросу или хранимой процедуре
`stables`	Возвращается информация обо всех таблицах данных на сервере, которые удовлетворяют критерию, указанному в свойстве соединения SQL `TableScope`
`stSysTables`	Возвращается информация обо всех системных таблицах на сервере баз данных. Не все серверы используют системные таблицы для хранения метаданных. Если послать запрос о списке системных таблиц на такой сервер, то будет возвращен пустой набор данных

Окончание табл. 8.2.

Значение `SchemaType`	*Описание*
`stProcedures`	Возвращает информацию обо всех хранимых процедурах
`stColumns`	Возвращается информация обо всех столбцах (полях) в указанной таблице
`stProcedureParams`	Возвращается информация обо всех параметрах указанной хранимой процедуры
`stIndexes`	Возвращается информация обо всех индексах для указанной таблицы

Пример использования метода `SetSchemaInfo()` находится на прилагаемом компакт-диске в каталоге `SchemaInfo`. Фрагмент программного кода из этого примера представлен в листинге 8.1.

Листинг 8.1. Пример использования метода `TSQLDataset.SetSchemaInfo()`

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  sqldsSchemaInfo.Close;
  cdsSchemaInfo.Close;

  case RadioGroup1.ItemIndex of
    0: sqldsSchemaInfo.SetSchemaInfo(stSysTables, '', '');
    1: sqldsSchemaInfo.SetSchemaInfo(stTables, '', '');
    2: sqldsSchemaInfo.SetSchemaInfo(stProcedures, '', '');
    3: sqldsSchemaInfo.SetSchemaInfo(stColumns, 'COUNTRY', '');
    4: sqldsSchemaInfo.SetSchemaInfo(stProcedureParams,
                                     'ADD_COUNTRY', '');
    5: sqldsSchemaInfo.SetSchemaInfo(stIndexes, 'COUNTRY', '');
  end; // case

  sqldsSchemaInfo.Open;
  cdsSchemaInfo.Open;
end;
```

В приведенном примере для выбора типа схемы информации используется компонент `TRadioGroup`. Затем данный тип используется при вызове процедуры `SetSchemaInfo()` в качестве значения параметра `SchemaType` перед тем, как открыть набор данных. Результат отображается при помощи сетки данных `TDBGrid`.

Компоненты совместимости с прежней версией

Во вкладке dbExpress палитры компонентов есть три компонента, аналогичных компонентам BDE: `TSQLTable`, `TSQLQuery` и `TSQLStoredProc`. Эти компоненты используются точно так же, как и эквивалентные им компоненты BDE, за исключением того, что они не могут использоваться при работе с двунаправленными наборами данных. В большинстве случаев для решения поставленных задач вполне достаточно компонента `TSQLDataset`.

Компонент TSQLMonitor

Компонент `TSQLMonitor` очень полезен при отладке приложений SQL. Он регистрирует все команды SQL, которые обрабатываются компонентом типа `TSQLConnection`, указанным в свойстве `TSQLMonitor.SQLConnection`.

Свойство `TSQLMonitor.Tracelist` содержит журнал команд, которыми обмениваются клиент и сервер баз данных. Такое свойство является объектом класса, производного от `TStrings`, поэтому содержащуюся в нем информацию можно сохранить в файле или просмотреть в поле типа `memo`.

НА ЗАМЕТКУ

Для автоматического сохранения содержимого свойства `TraceList` можно использовать свойства `FileName` и `AutoSave`.

Пример использования содержимого свойства `TraceList` в поле `memo` находится на прилагаемом компакт-диске в каталоге `SQLMon`. Результат трассировки команд SQL показан на рис. 8.3.

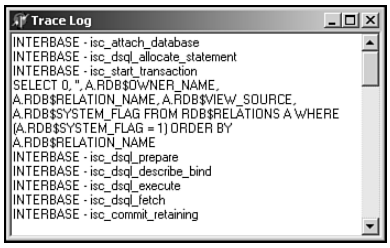

Рис. 8.3. *Результат работы компонента `TSQLMonitor`*

Разработка приложений dbExpress, позволяющих редактировать данные

До сих пор средства dbExpress рассматривались в контексте однонаправленных наборов данных, открытых только для чтения. Единственным исключением был пример использования компонента `TSQLDataset` для выполнения хранимой процедуры, добавляющей записи в таблицу. Еще одним способом редактирования данных является использование двунаправленных наборов данных для кэширования обновлений. Для реализации такого подхода применяется компонент `TSQLClientDataset`.

Компонент TSQLClientDataset

В компоненте TSQLClientDataset реализованы свойства компонентов TSQLDataset и TProvider. Внутренняя реализация класса TSQLDataset позволяет компонентам TSQLClientDataset осуществлять быстрый доступ к данным через средства dbExpress. Внутренняя реализация класса TSQLProvider позволяет осуществлять двунаправленную навигацию и редактирование данных.

Применение компонента TSQLClientDataset очень похоже на применение стандартного компонента TClientDataset. Этот вопрос рассматривается в главе 21, "Разработка приложений DataSnap".

Создание приложений с использованием компонента TSQLClientDataset особых затруднений не составляет. Для отображения данных в таких приложениях необходимы три компонента: TSQLConnection, TSQLClientDataset и TDatasource. Соответствующий пример находится на прилагаемом компакт-диске в каталоге Editable.

В свойстве TSQLClientDataset.DBConnection указывается ссылка на необходимый компонент SQLConnection. Свойства CommandType и CommandText используются точно так же, как было описано в разделе, посвященном компоненту TSQLDataset.

В представленном приложении можно перемещаться по набору данных в обоих направлениях, а также добавлять, редактировать и удалять записи. Однако если набор данных закрыть, то все изменения теряются, поскольку на самом деле все действия осуществляются с буфером набора данных, хранимым компонентом TSQLClientDataset в оперативной памяти. Другими словами, все изменения кэшируются, и для того чтобы сохранить их в базе данных на сервере, необходимо вызвать метод TSQLClientDataset.ApplyUpdates(). В примере, представленном на прилагаемом компакт-диске, вызов метода ApplyUpdates() происходит в обработчиках событий AfterDelete и AfterPost компонента TSQLClientDataset. В результате вызова этого метода записи на сервере обновляются одна за другой. Дополнительную информацию об использовании компонента TSQLClientDataset можно найти в главах 21, 32 и 34 книги *Delphi 5 Руководство разработчика*, находящейся на прилагаемом CD.

НА ЗАМЕТКУ

Класс TSQLClientDataset реализует лишь некоторые свойства и события компонентов TSQLDataSet и TProvider. Если нужно использовать те свойства, которые не реализованы в классе TSQLClientDataset, то лучше вместо него применить обычные компоненты TClientDataset и TDatasetProvider.

Распространение приложений dbExpress

Приложения dbExpress могут распространяться в виде полностью автономных исполняемых файлов или в качестве библиотеки DLL, укомплектованной необходимыми драйверами. Для компиляции автономного приложения следует добавить в раздел uses модули, перечисленные в табл. 8.3. Все эти модули описаны в интерактивной справочной системе Delphi.

Таблица 8.3. Модули, необходимые для автономного приложения dbExpress

Модуль	*Необходимо включить*
`dbExpInt`	Если приложение подключается к базе данных InterBase
`dbExpOra`	Если приложение подключается к базе данных Oracle
`dbExpDb2`	Если приложение подключается к базе данных DB2
`dbExpMy`	Если приложение подключается к базе данных MySQL
`Crtl`, `MidasLib`	Если приложение dbExpress использует такие клиентские наборы данных, как `TSQLClientDataSet`

Если приложение поставляется в комплекте с библиотеками, то список необходимых библиотек DLL и их назначение приведен в табл. 8.4.

Таблица 8.4. Библиотеки DLL, поставляемые вместе с приложениями dbExpress

Библиотека DLL	*Используется*
`dbexpint.dll`	Если приложение подключается к базе данных InterBase
`dbexpora.dll`	Если приложение подключается к базе данных Oracle
`dbexpdb2.dll`	Если приложение подключается к базе данных DB2
`dbexpmy.dll`	Если приложение подключается к базе данных MySQL
`Midas.dll`	Если приложение использует клиентские наборы данных

Резюме

Средства dbExpress позволяют разрабатывать простые и надежные приложения, значительно превосходящие производительность аналогичных приложений BDE. Используя механизмы кэширования, реализованные в классах `TSQLClientDataset` и `TClientDataset`, разработчики получают возможность создавать приложения, не зависящие от типа платформы.

ГЛАВА

9

Применение dbGo for ADO при разработке баз данных

В ЭТОЙ ГЛАВЕ...

Введение в dbGo

В настоящей главе обсуждаются вопросы программирования с применением *объектов данных ActiveX* (ADO — ActiveX Data Objects). Эта технология, разработанная корпорацией *Microsoft*, в Delphi реализована компонентами dbGo for ADO.

Такие компоненты расположены во вкладке ADO палитры компонентов и используются для доступа к данным средствами ADO.

Обзор стратегии *Microsoft* по универсальному доступу к данным

Предложенная корпорацией *Microsoft* стратегия *универсального доступа к данным* (Universal Data Access) заключается в использовании одной общей модели. При этом учитываются как реляционные, так и нереляционные базы данных. Такая идея была реализована с помощью *компонентов Microsoft для доступа к данным* (MDAC — Microsoft Data Access Components), которые уже установлены на всех системах Windows 2000 и могут быть загружены по адресу `http://www.microsoft.com/data/`.

Средства MDAC включают в себя три элемента: OLE DB (базовая упрощенная спецификация COM и интерфейс API для доступа к данным), ADO (ActiveX Data Objects — объекты данных ActiveX) и ODBC (Open Database Connectivity — открытое соединение с базами данных).

Краткий обзор OLE DB, ADO и ODBC

OLE DB — это интерфейс системного уровня, в котором для доступа к данным используется модель компонентных объектов COM. При этом учитываются как реляционные форматы баз данных, так и нереляционные. Вполне возможно создание кода, способного взаимодействовать непосредственно с OLE DB, но в случае с ADO это сделать значительно сложнее и, чаще всего, излишне.

Большинство провайдеров OLE DB, предоставляющих доступ к данным специфического провайдера (например Paradox, Oracle, Microsoft SQL Server, Microsoft Jet Engine и ODBC), реализованы также на интерфейсах OLE DB.

ADO — это интерфейс прикладного уровня, который используется разработчиками приложений для доступа к данным. В отличие от OLE DB, состоящего из более чем 60 различных интерфейсов, в ADO используются лишь несколько интерфейсов для взаимодействия с разработчиками. Фактически интерфейсы OLE DB также используются и в ADO в качестве базовой технологии доступа к данным.

Технология ODBC была предшественницей OLE DB и все еще часто используется разработчиками для доступа к реляционным и некоторым нереляционным базам данных. По сути, один из интерфейсов OLE DB использует средства ODBC.

Использование dbGo for ADO

dbGo for ADO — это название набора компонентов Delphi, инкапсулирующего интерфейсы ADO. Такие компоненты используются для разработки приложений баз данных, характерных для Delphi.

Следующие разделы посвящены именно вопросу использования компонентов dbGo for ADO. В настоящей главе будет рассматриваться главным образом доступ к базе данных Microsoft Access через источник данных (провайдер) ODBC.

Установка провайдера OLE DB для ODBC

Для подключения к базе данных необходимо сначала создать *имя источника данных* (DSN — Data Source Name) ODBC. Имена источников данных подобны псевдонимам BDE в том смысле, что они позволяют осуществлять подключение на системном уровне на основании указанной пользователем информации о базе данных. Для создания нового DSN используется специальная программа Administrator ODBC, которая входит в поставку Windows. В системе Windows 2000 эту программу можно найти в группе Administrative Tools. Внешний вид диалогового окна администратора ODBC представлен на рис. 9.1.

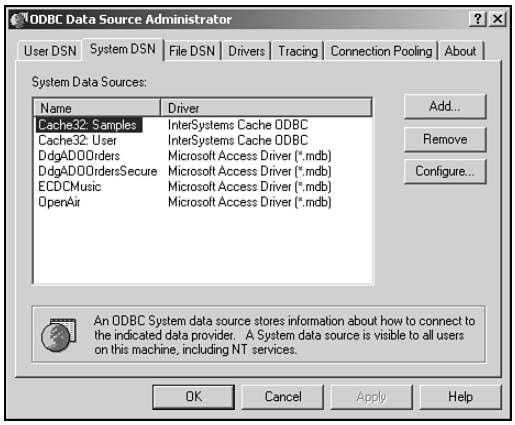

Рис. 9.1. *Администратор ODBC*

Существует три типа DSN:

- Пользовательский DSN (User DSN) — имена источников данных, использующихся только на локальном компьютере и лишь для текущего пользователя.
- Системные DSN (System DSN) — имена источников данных, которые могут использоваться всеми пользователями системы с соответствующими правами доступа, но только на локальном компьютере.
- Файловые DSN (File DSN) — имена источников данных, доступные всем пользователям, на компьютерах которых установлены соответствующие драйверы.

В этом примере создадим имя для системного источника данных. Запустите администратор ODBC, перейдите во вкладку System DSN (Системные DSN) и щелкните на кнопке Add (Добавить). В результате появится диалоговое окно Create New Data Source (Создать новый источник данных) (рис. 9.2).

В этом диалоговом окне представлен перечень доступных драйверов. В данном случае необходим драйвер Microsoft Access Driver (*.mdb). Выделите его и щелкните на кнопке Finish (Завершить). В результате на экране появится диалоговое окно ODBC Microsoft Access Setup (рис. 9.3).

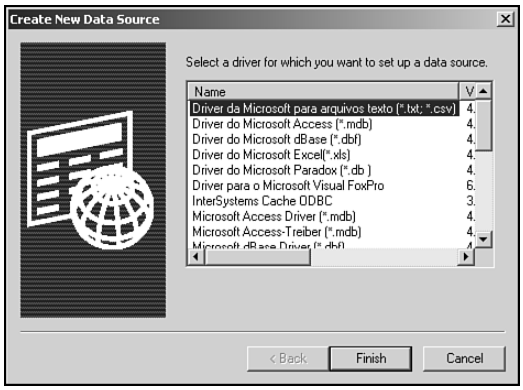

Рис. 9.2. *Диалоговое окно Create New Data Source*

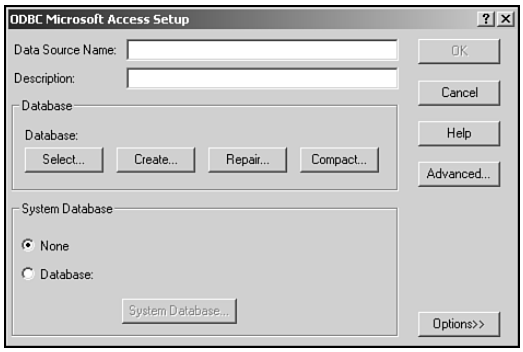

Рис. 9.3. *Диалоговое окно ODBC Microsoft Access Setup*

Здесь нужно указать имя источника данных, которое будет использоваться в приложении Delphi. Напомним, что DSN подобен псевдониму BDE. При желании можно ввести строку описания источника данных. Далее щелкните на кнопке Select (Выбор) и в диалоговом окне File Open (Открыть файл) укажите путь к файлу базы данных с расширением `*.mdb`. В данном случае – файл `ddgADO.mdb`, который должен быть установлен в каталоге `..\Delphi Developer's Guide\Data`. После щелчка на кнопке OK новый источник данных будет внесен в список System Data Sources. Для выхода из программы ODBC Administrator щелкните еще раз на кнопке OK.

База данных Access

Структура базы данных, для которой только что было создано имя источника данных, представлена на рис. 9.4.

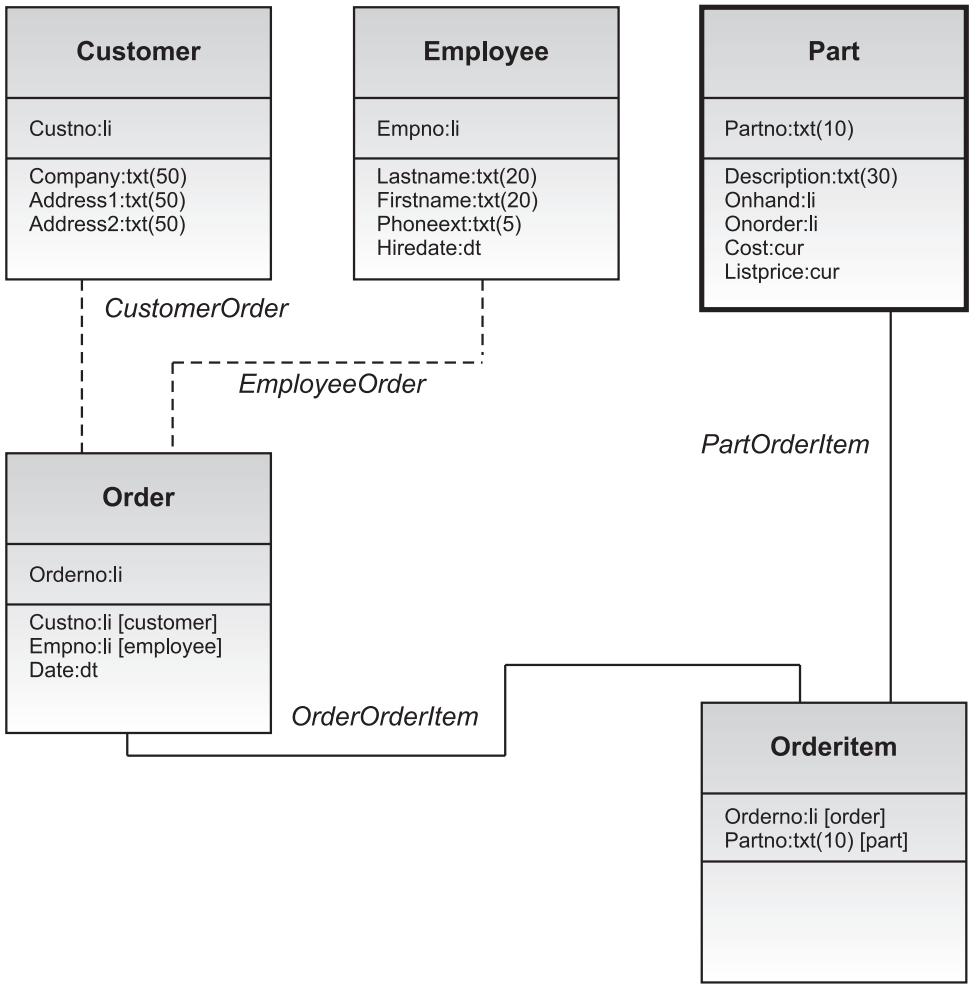

Рис. 9.4. *Структура используемой базы данных*

Это – простая база данных для хранения информации о заказах, которая содержит всего лишь несколько связанных таблиц. Такая простая база данных как нельзя лучше подходит для изучения возможностей компонентов dbGo for ADO.

Компоненты dbGo for ADO

Все компоненты dbGo for ADO находятся во выкладке ADO палитры компонентов.

Компонент TADOConnection

Класс `TADOConnection` инкапсулирует объект соединения ADO. Этот компонент используется другими компонентами для подключения к источникам данных ADO. Компонент `TADOConnection` аналогичен компоненту `TDatabase`, который использовался для подключения к базам данных через BDE. Так же, как и `TDatabase`, он поддерживает запрос на ввод имени пользователя и пароля, а также транзакции.

Подключение к базе данных

Создайте новое приложение и разместите в форме компонент `TADOConnection`. Для изменения значения свойства `TADOConnection.ConnectionString` щелкните на кнопке с многоточием в поле этого свойства в инспекторе объектов. В результате на экране появится редактор свойства `ConnectionString` (рис. 9.5).

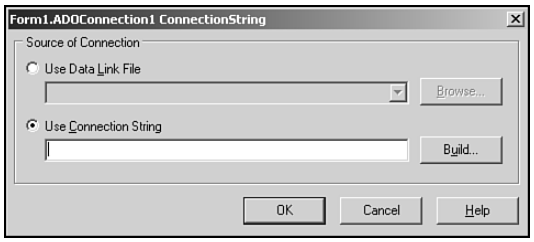

Рис. 9.5. *Редактор свойства `TADOConnection.ConnectionString`*

Свойство `ConnectionString` состоит из одного или нескольких параметров, необходимых ADO для подключения к базе данных. Набор этих параметров зависит от типа используемого провайдера OLE DB.

Источник соединения указывается в редакторе свойства `ConnectionString` либо в поле Data Link File (файл, содержащий строку подключения), либо непосредственно в строке подключения, которую в дальнейшем можно сохранить во внешнем файле. Имя источника данных уже существует, поэтому необходимо просто создать строку подключения, которая обращается к необходимому DSN. Щелкните на кнопке Build (Создать), в результате появится диалоговое окно Data Link Properties (Свойства связи с данными) (рис. 9.6).

Первая страница этого диалогового окна позволяет выбрать провайдер OLE DB. В данном случае необходимо выбрать Microsoft OLE DB Provider For ODBC Drivers (см. рис. 9.6) и щелкнуть на кнопке Next (Дальше). На второй странице (Connection Page) можно выбрать имя источника данных из раскрывающегося списка Data Source Name (рис. 9.7).

В данном случае база данных не защищена, поэтому можно щелкнуть на кнопке Text Connection и сформировать корректную строку подключения. Дважды щелкните на кнопке OK, чтобы вернуться в главную форму. Теперь строка подключения выглядит следующим образом:

```
Provider=MSDASQL.1;Persist Security Info=False;
Data Source=DdgADOOrders
```

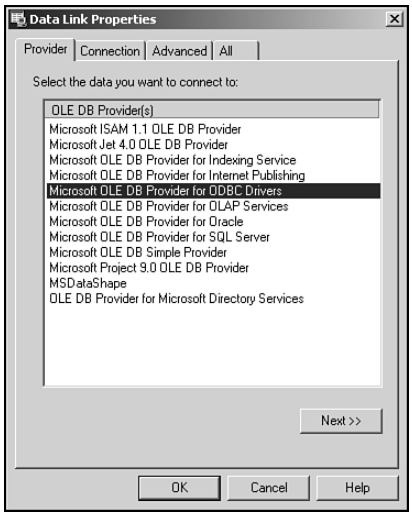

Рис. 9.6. *Диалоговое окно Data Link Properties*

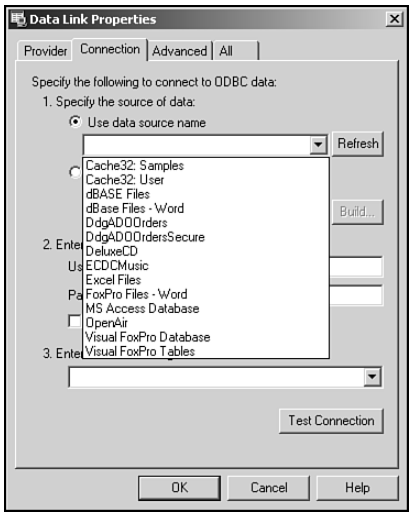

Рис. 9.7. *Выбор имени источника данных*

Если бы использовался другой провайдер OLE DB, то строка подключения выглядела бы совершенно иначе. Например, если выбрать провайдер Microsoft Jet 4.0 OLE DB Provider, то строка подключения будет иметь такой вид:

```
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source="C:\Program Files\Delphi '
↳ Developer's Guide\Data\ddgADO.mdb";
Persist Security Info=False
```

Теперь можно подключиться к базе данных, присвоив значение `True` свойству `TADOConnection.Connected`. На экране появится запрос на ввод имени пользователя и пароля. Не вводя никаких значений, просто щелкните на кнопке OK. В следующем разделе будет рассмотрен способ отключения запроса на ввод имени пользователя и пароля, а также пути замены данного стандартного диалогового окна своим собственным. Этот пример находится на прилагаемом компакт-диске в каталоге `ADOConnect`.

Ввод имени пользователя и пароля при подключении к базе данных

Чтобы отменить запрос на ввод имени пользователя и пароля при подключении к базе данных, достаточно присвоить свойству `TADOConnection.LoginPrompt` значение `False`. Но если для подключения к базе данных имя пользователя и пароль являются необходимым условием, то придется выполнить некоторые дополнительные действия.

СОВЕТ

Для проверки необходимости этого условия можно защитить базу данных паролем. В Microsoft Access эта операция выполняется в режиме монопольного доступа, который включается во вкладке Advanced (Дополнительно) диалогового окна Options (меню Tools пункт Options). Можно также просто использовать базу данных `ddgADOPW.mdb`, находящуюся на прилагаемом компакт-диске (пароль — `ddg` рис. 9.8).

Чтобы воспользоваться базой данных `ddgADOPW.mdb`, для нее необходимо создать новое имя источника данных, например `DdgADOOrdersSecure`.

Если необходимо отменить запрос при подключении к защищенной базе данных, то корректное имя пользователя и пароль должны указываться в свойстве `ConnectionString`. Это можно сделать вручную или при помощи редактора свойства `ConnectionString`. В случае использования редактора следует также установить флажок Allow Saving Password (Разрешить сохранение пароля) (рис. 9.8).

Теперь строка подключения выглядит следующим образом:

```
Provider=MSDASQL.1;Password=ddg;
Persist Security Info=True;
User ID=Admin;Data Source=DdgADOOrdersSecure
```

Обратите внимание! Здесь указаны и пароль, и имя пользователя (ID). Теперь можно присвоить свойству `Connected` компонента `TADOConnection` значение `True` (при этом свойство `LoginPrompt` должно иметь значение `False`).

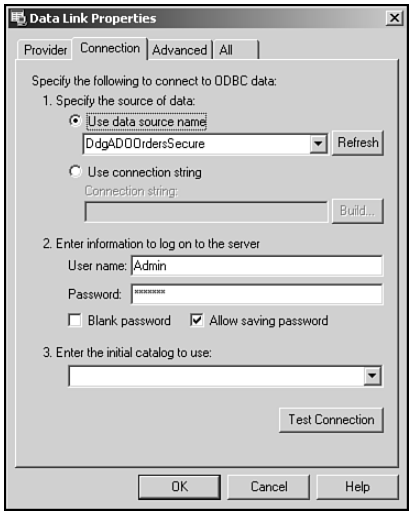

Рис. 9.8. *Добавление имени пользователя и пароля в строку подключения*

Теперь заменим стандартное диалоговое окно запроса при подключении к базе данных. Для этого нужно удалить имя пользователя и пароль из свойства `ConnectionString` и создать обработчик события `TADOConnection.OnWillConnect` так, как представлено в листинге 9.1.

Листинг 9.1. Обработчик события `OnWillConnect`

```
procedure TForm1.ADOConnection1WillConnect(Connection:
              TADOConnection;
              var ConnectionString, UserID, Password: WideString;
              var ConnectOptions: TConnectOption;
              var EventStatus: TEventStatus);
var
  vUserID,
  vPassword: String;
begin
  if InputQuery('Provide User name',
                'Enter User name', vUserID) then
    if InputQuery('Provide Password',
                  'Enter Password', vPassword) then
    begin
      UserID := vUserID;
      Password := vPassword;
    end;
end;
```

Это – упрощенный пример реализации ввода имени пользователя и пароля. В реальных приложениях реализация данного диалогового окна, скорее всего, окажется более сложной.

НА ЗАМЕТКУ

На первый взгляд может показаться, что реализация ввода имени пользователя и пароля должна выполняться так же, как и для компонента `TDatabase`, в обработчике события `TADOConnection.OnLogin`. Но вместо него используется событие `TADOConnection.OnWillConnnect`, которое является оболочкой стандартного события ADO. В данном случае событие `OnLogin` используется классом `TDispatchConnection`, предназначенным для поддержки многоуровневой архитектуры.

Компонент TADOCommand

Класс `TADOCommand` инкапсулирует объект ADO `Command`. Компоненты этого типа используются для выполнения операторов SQL `SELECT` и операторов DDL (Data Definition Language – язык определения данных), не возвращающих результатов (`INSERT`, `DELETE` и `UPDATE`). Пример использования такого компонента находится на прилагаемом компакт-диске в каталоге `ADOCommand`. Это – простой пример, демонстрирующий вставку и удаление записей в таблице `EMPLOYEE` при помощи операторов SQL `INSERT` и `DELETE`. В приведенном ниже примере оператор SQL, указанный в свойстве `TADOCommand.CommandText`, имеет следующий вид:

```
DELETE FROM EMPLOYEE WHERE
FirstName='Rob' AND LastName='Smith
```

Для добавления записи в базу свойство `CommandText` компонента `TADOCommand` содержит следующий оператор SQL:

```
INSERT INTO EMPLOYEE (
LastName,
```

```
FirstName,
PhoneExt,
HireDate)

VALUES(
'Smith',
'Rob',
'123',
'12/28/1998')
```

Чтобы выполнить оператор SQL, необходимо вызвать метод `TADOCommand.Execute()`.

Компонент TADODataset

Компонент типа `TADODataset` предназначен для извлечения данных из одной или нескольких таблиц. При помощи этого компонента можно также выполнять пользовательские хранимые процедуры и операторы SQL, не возвращающие результатов.

Подобно компоненту `TADOCommand`, компонент `TADODataset` может выполнять операторы `INSERT`, `DELETE` и `UPDATE`, но, кроме того, он может извлекать наборы данных при помощи оператора `SELECT`. Пример использования компонента `TADODataSet` находится на прилагаемом компакт-диске в каталоге `ADODataset`. В данном примере выполняется следующий оператор `SELECT`:

```
SELECT * FROM Customer
```

Этот оператор возвращает полный набор данных из таблицы `Customer`. Если необходимо отфильтровать записи, то можно воспользоваться директивой SQL `WHERE`.

В данном примере для демонстрации возможностей навигации и редактирования набора данных с помощью компонента `TADODataSet`, используется компонент `TDBNavigator`.

Компонент `TADODataSet` еще будет использоваться в настоящей главе при создании приложения, предназначенного для работы с базой данных заказов.

Компоненты для работы с наборами данных BDE

Вкладка ADO палитры компонентов содержит три компонента, которые предназначены исключительно для упрощения преобразования приложений BDE в приложения ADO: `TADOTable`, `TADOQuery` и `TADOStoredProc`. Для реализации функций этих компонентов при разработке приложений ADO вполне достаточно одного компонента `TADODataSet`. Тем не менее, при желании можно использовать и эти альтернативные компоненты, аналогичные их предшественникам BDE `TTable`, `TQuery` и `TStoredProc`.

Класс TADOTable

Класс `TADOTable` является прямым производным от класса `TCustomADODataSet` и позволяет работать с одной из таблиц базы данных. Его свойства очень похожи на свой-

ства компонента BDE TTable. Фактически TADOTable обладает дополнительным свойством TableName. Основным преимуществом табличного типа набора данных является то, что они поддерживают индексы. Индексы позволяют очень быстро осуществлять сортировку и поиск. Это особенно актуально для баз данных, отличных от SQL, например для Microsoft Access. Но при использовании баз данных SQL сортировку, фильтрацию и другие подобные операции лучше выполнять при помощи операторов языка SQL. Дополнительную информацию о наборах данных табличного типа можно найти в интерактивной справочной системе Delphi в разделе "*Overview of ADO components*".

СОВЕТ

Согласно справочной системе Delphi, одним из преимуществ наборов данных табличного типа является простота очистки таблиц. При этом в качестве примера используется вызов метода TCustomADODataSet.DeleteRecords(). Тем не менее, это не работает в случае использования объекта ADO RecordSet. Вызов метода DeleteRecords() приводит к возникновению исключения, хотя вызов такого метода, как TCustomADODataSet.Supports([coDelete]), возвращает значение True. Поэтому для очистки таблиц лучше использовать оператор вида DELETE FROM TableName или удалять каждую запись поочередно.

Пример использования компонента TADOTable с индексом находится на прилагаемом компакт-диске в каталоге ADOTableIndex. Кроме того, в приведенном ниже примере демонстрируется использование функции TADOTable.Locate() для поиска информации в таблице. Фрагмент исходного кода этого примера представлен в листинге 9.2.

Листинг 9.2. Использование компонента TADOTable

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
begin
  adotblCustomer.Open;
  for i := 0 to adotblCustomer.FieldCount - 1 do
    ListBox1.Items.Add(adotblCustomer.Fields[i].FieldName);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
  adotblCustomer.IndexFieldNames :=
     ListBox1.Items[ListBox1.ItemIndex];
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  adotblCustomer.Locate('Company', Edit1.Text, [loPartialKey]);
end;
```

В обработчике события FormCreate() открывается таблица и список TListBox заполняется названиями полей этой таблицы. Затем в обработчике события TList-

Box.OnClick в свойстве TADOTable.IndexFieldName указывается название поля, по которому необходимо отсортировать записи.

Наконец, в обработчике события Button1Click() выполняется поиск данных при помощи метода Locate().

Компонент TADOTable полезен для тех, кто привык работать с компонентом TTable. Но при работе с SQL-серверами более эффективно применение компонентов TADODataSet и TADOQuery.

Компонент TADOQuery

Класс TADOQuery также является производным от класса TCustomADODataSet и очень похож на класс TADODataSet. У него есть свойство SQL, которому можно присваивать операторы SQL. В компонентах TADODataSet для назначения операторов SQL используется свойство CommandText, когда свойство TADODataSet.CommandType имеет значение cmdText.

Не будем рассматривать компонент TADOQuery подробно, так как большинство его возможностей реализовано в классе TADODataSet.

Компонент TADOStoredProc

Компоненты типа TADOStoredProc предназначены для работы с хранимыми процедурами. Он полностью аналогичен компоненту TADOCommand, у которого свойство CommandType имеет значение cmdStoredProc. Этот компонент очень похож на компонент TStoredProc, который рассматривается в главе 29, "Разработка клиент-серверных приложений", предыдущего издания *Delphi 5 Руководство разработчика*, находящегося на прилагаемом CD.

Обработка транзакций

Технология ADO поддерживает обработку транзакций, что реализовано в классе TADOConnection. В качестве примера рассмотрим фрагмент исходного кода приложения базы данных (листинг 9.3).

Листинг 9.3. Обработка транзакций при помощи компонента TADOConnection

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  if TNewOrderForm.Execute then begin
    ADOConnection1.BeginTrans;
    try
      // Сначала создать запись Orders
      adodsOrders.Insert;
      adodsOrders.FieldByName('CustNo').Value :=
                adodsCustomer.FieldByName('CustNo').Value;
      adodsOrders.FieldByName('EmpNo').Value :=
                adodsEmployee.FieldByName('EmpNo').Value;
      adodsOrders.FieldByName('Date').Value := Date;
      ShowMessage(IntToStr(adodsOrders.
↳FieldByName('OrderNo').AsInteger));
      adodsOrders.Post;
```

```
      // Затем создать элементы Order Line.
      cdsPartList.First;
      while not cdsPartList.Eof do begin
        adocmdInsertOrderItem.Parameters.ParamByName
↳('iOrderNo').Value := adodsOrders.FieldByName('OrderNo').Value;
        adocmdInsertOrderItem.Parameters.ParamByName
↳('iPartNo').Value := cdsPartListPartNo.Value;
        adocmdInsertOrderItem.Execute;
        cdsPartList.Next;
      end;
      adodsOrderItemList.Requery([]);
      ADOConnection1.CommitTrans;
      cdsPartList.EmptyDataSet;
    except
      ADOConnection1.RollbackTrans;
      raise;
    end;
  end;
end;
```

Метод, представленный в листинге 9.3, отвечает за создание заказа. Данная транзакция состоит из двух частей. В первой создается запись в таблице `Order`, а во второй — добавляются элементы в таблицу `OrderItem`. В этом примере происходит обновление двух таблиц, из-за чего имеет смысл поместить их в единую транзакцию.

Ниже представлена структура такой транзакции:

```
  begin
    ADOConnection1.BeginTrans;
    try
      // Сначала создать запись Orders
      // Затем создать элементы Order Line.
      ADOConnection1.CommitTrans;
    except
      ADOConnection1.RollbackTrans;
      raise;
    end;
  end;
end;
```

Как видно из примера, транзакция реализована внутри блока `try..except`. Метод `ADOConnection1.BeginTrans()` начинает транзакцию, а метод `ADOConnection1.CommitTrans()` завершает ее (фиксирует). Если при фиксации транзакции возникает какой-либо сбой, то создается исключение и вызывается метод `ADOConnection1.RollbackTrans()`, который отменяет все внесенные в таблицу изменения.

Резюме

В настоящей главе были рассмотрены компоненты dbGo for ADO, которые для доступа к базам данных используют технологию Microsoft ADO.

ЧАСТЬ

IV

Компонент-ориентированная разработка

В ЭТОЙ ЧАСТИ...

ГЛАВА

10

Архитектура компонентов: VCL и CLX

В ЭТОЙ ГЛАВЕ...

Немногие, наверно, помнят первую *библиотеку объектов для Windows* (OWL — Object Windows Library), которой был укомплектован Turbo Pascal от *Borland*. Библиотека OWL существенно упростила программирование под Windows, поскольку объекты OWL автоматизировали и рационализировали много утомительных задач, которые требовали написания вручную большего количества кода при традиционном подходе. Ушли в небытие чудовищные операторы `case`, предназначенные для перехвата сообщений Windows и громадные блоки кода взаимодействия с классами Windows; все это библиотека OWL делает сама. С другой стороны, понадобилось изучить новую методологию — объектно-ориентированное программирование.

Затем, в Delphi 1, Borland представила *библиотеку визуальных компонентов* (VCL — Visual Component Library). В принципе, VCL была разработана на базе объектной модели, как и OWL, но радикально отличалась по реализации. В Delphi 6 библиотека VCL практически та же, что и во всех предыдущих версиях.

В Delphi 6 *Borland* снова представила новую технологию: *библиотеку межплатформенных компонентов* (CLX — Component Library for Cross-Platform). Согласно *Borland*, CLX — это "библиотека компонентов следующего поколения, или платформа для разработки приложений и компонентов многократного использования, совместимых как с Linux, так и с Windows".

Обе библиотеки, и VCL и CLX, разработаны специально для визуальной среды разработки Delphi. Вместо того чтобы создавать программный код окна или диалога и описывать его поведение, достаточно модифицировать поведение и характеристики готовых компонентов в процессе визуальной разработки приложения.

Уровень знаний, необходимый для работы с VCL и CLX, зависит от способа их применения. Программистов Delphi можно разделить на разработчиков приложений и создателей визуальных компонентов. Первые создают приложения непосредственно в визуальной среде Delphi (возможность, предоставляемая далеко не всеми средами программирования). С помощью VCL и CLX они разрабатывают графический интерфейс и некоторые другие элементы приложения, например, связывающие его с базами данных. Вторые создают новые компоненты, расширяя стандартные возможности VCL и CLX. Такие компоненты являются законченным коммерческим продуктом и известны под названием *продуктов стороннего производителя*.

Независимо от того, предполагается ли создавать приложения или компоненты Delphi, следует хорошо изучить библиотеки VCL и CLX. Разработчик приложений должен знать, какие методы, события и свойства доступны ему в каждом компоненте. Весьма полезно также разобраться в функционировании объектной модели VCL и CLX. Чаще всего проблемы разработчиков Delphi при использовании того или иного инструмента связаны с тем, что они не до конца понимают, как он работает. Разработчикам компонентов желательно глубже разбираться в специфике VCL и CLX — в их внутреннем устройстве, способах обработки сообщений и уведомлений, проблемах собственности компонентов, наследования, редакторах свойств и так далее.

Настоящая глава целиком посвящена библиотекам VCL и CLX. В ней обсуждается иерархия компонентов и поясняется назначение основных уровней в этой иерархии. Здесь же описаны наиболее общие свойства, методы и события, присущие компонентам разных уровней иерархии. И, наконец, в главе содержится описание *информации о типах времени выполнения* (RTTI — Runtime Type Information).

Немного подробнее о новой библиотеке CLX

CLX — это новая межплатформенная библиотека, которая фактически состоит из четырех частей. Их описание (из интерактивной справочной системы Delphi 6) приведено в табл. 10.1.

Таблица 10.1. Составные части CLX

Часть	*Описание*
VisualCLX	Базовые межплатформенные компоненты GUI и графики. Компоненты этой области для Linux и Windows могут отличаться друг от друга
DataCLX	Клиентские компоненты доступа к данным. Здесь расположен набор локальных, клиент/серверных и многоуровневых компонентов для клиентских наборов данных. Компоненты для Linux и Windows одинаковы
NetCLX	Компоненты для Internet, включая DSO Apache и CGI Web Broker. Компоненты для Linux и Windows одинаковы
RTL	Динамическая библиотека в составе `Classes.pas`. Компоненты для Linux и Windows одинаковы. В Linux этот файл называется `BaseRTL`

VisualCLX — потомок среды разработки Qt от *Trolltech*. Большинство людей произносят Qt как "куте" ("cute"), хотя *Trolltech* неоднократно объявляла, что это название следует произносить как "кю-ти" ("kyu-tee"). В настоящее время данная среда разработки совместима и с Linux и с Windows. В этой главе обсуждается только VisualCLX, а иные элементы CLX рассматриваются в других главах.

Что такое компонент?

Компоненты (component) — это строительные блоки приложения, используя которые разработчик создает пользовательский интерфейс и включает в приложение некоторые невизуальные элементы. Для разработчиков приложений компонент — это нечто, взятое из палитры компонентов и помещенное в форму. После помещения компонента в форму можно манипулировать его свойствами и добавлять обработчики событий, чтобы придать ему специфический вид или задать определенное поведение. С точки зрения разработчика компонентов — это объект, реализованный в виде кода, написанного на языке Object Pascal. Такие объекты могут, например, инкапсулировать поведение системных элементов (элементов управления Windows). Другие объекты могут представлять собой абсолютно новый визуальный или даже невизуальный элемент (в таком случае поведение компонента полностью определяется кодом, написанным разработчиком).

По своей сложности компоненты могут значительно отличаться друг от друга. Существуют как совсем простые компоненты, так и достаточно сложные. На степень сложности компонента нет никаких ограничений. Можно использовать простой ком-

понент — метку `TLabel` — или компонент, инкапсулирующий функциональные возможности целой электронной таблицы.

Для работы с VCL и CLX необходимо знать типы существующих компонентов, а также понимать иерархию компонентов и назначение каждого уровня этой иерархии. Такая информация приведена в следующем разделе.

Иерархия компонентов

Ниже (рис. 10.1 и 10.2) представлены, соответственно, иерархии классов библиотек VCL и CLX. Можно заметить, что они очень похожи друг на друга.

Существует два типа компонентов: визуальный и невизуальный.

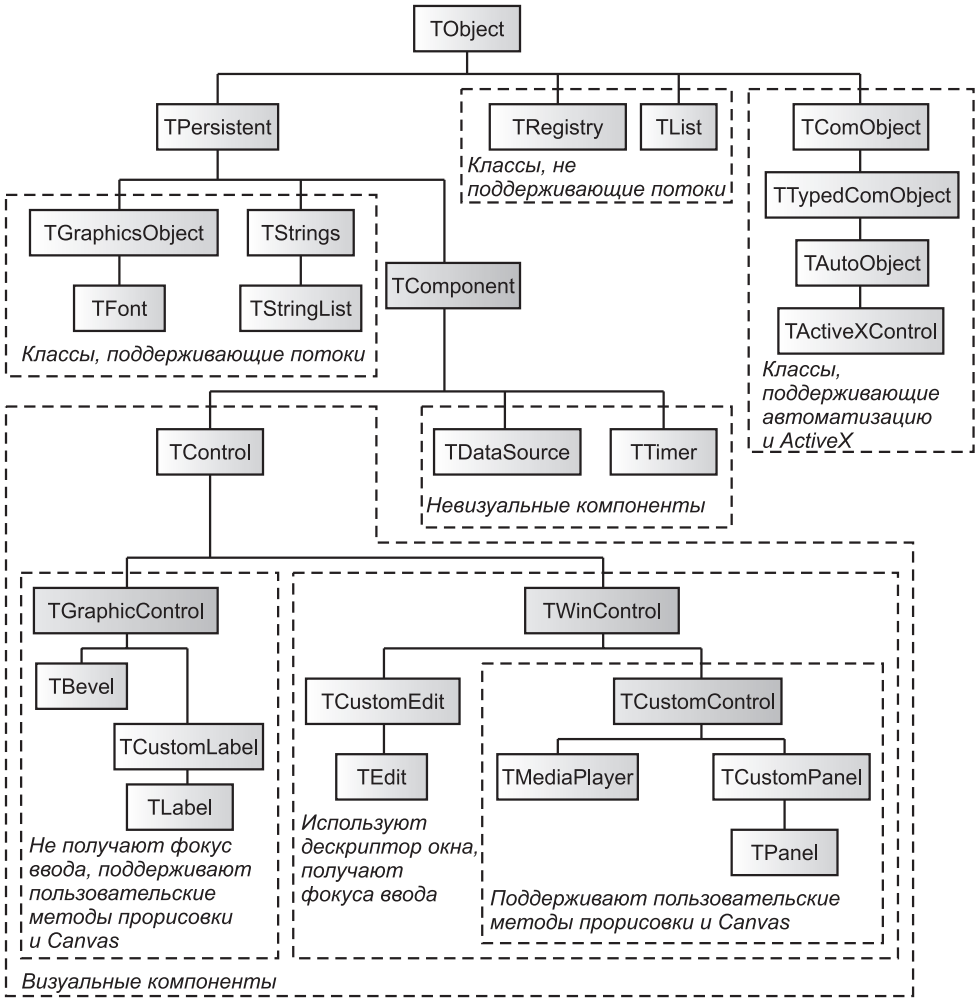

Рис. 10.1. *Иерархия классов VCL*

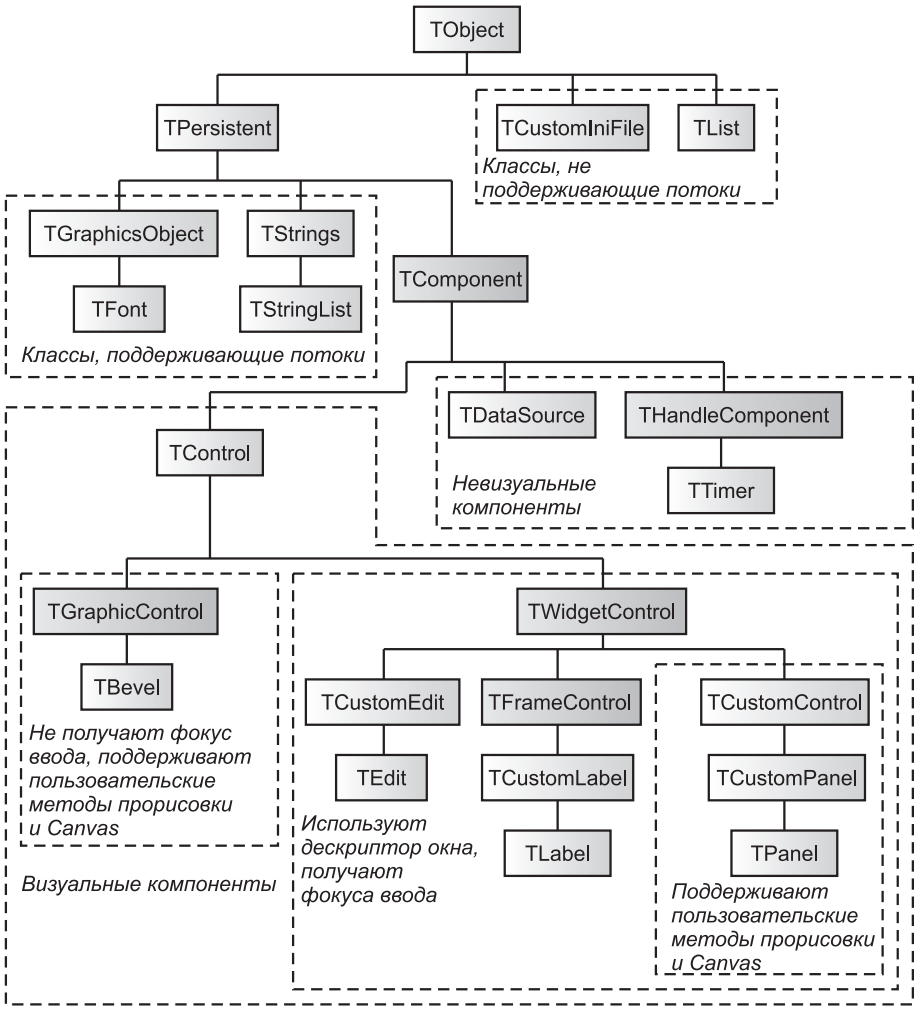

Рис. 10.2. *Иерархия классов CLX*

Невизуальные компоненты

Как следует из названия, невизуальные компоненты не отражаются в форме выполняющегося приложения. Данные компоненты обладают собственным поведением, которое разработчик может изменить, модифицировав с помощью инспектора объектов его свойства или создав обработчики его событий. В качестве примеров можно привести классы `TOpenDialog`, `TTable` и `TTimer`. Как можно увидеть на рис. 10.1 и 10.2, классы всех этих невизуальных компонентов происходят непосредственно от класса `TComponent`.

Визуальные компоненты

Визуальные компоненты (visual components), как и следует из их названия, являются теми компонентами, которые пользователь видит на экране. Визуальные компоненты можно увидеть, они обладают собственным поведением, но не все из них способны взаимодействовать с пользователем. Классы этих компонентов происходят непосредственно от класса `TControl`. Фактически класс `TControl` был задуман как базовый для подобных компонентов. Он обладает такими свойствами и методами, которые необходимы для обеспечения представления элемента управления на экране (например `Top`, `Left`, `Color` и т.д.).

НА ЗАМЕТКУ

Иногда термины *компонент* и *элемент управления* (control) употребляются в одном и том же контексте, хотя их значения не всегда совпадают. Понятие "элемент управления" относится к отдельным элементам графического пользовательского интерфейса. В Delphi элементы управления всегда являются компонентами, поскольку происходят от класса `TComponent`. Компоненты — это объекты, которые могут отображаться в палитре компонентов и изменяться в процессе разработки формы. Компоненты всегда происходят от класса `TComponent`, но не всегда являются элементами управления, т.е. элементами графического интерфейса.

Визуальные компоненты бывают двух разновидностей: способные получать фокус и не способные.

Визуальные компоненты, способные получать фокус

Некоторые типы элементов управления способны получать фокус. Указанное означает, что пользователь может взаимодействовать с такими элементами управления. Эти типы элементов управления являются потомками класса `TWinControl` (в библиотеке VCL) или `TWidgetControl` (в библиотеке CLX). Потомки класса `TWinControl` являются оболочками стандартных элементов управления Windows, а потомки класса `TWidgetControl` — оболочками экранных объектов Qt. Эти элементы управления обладают следующими характеристиками:

- Они способны получать фокус и реагировать на события клавиатуры.
- Пользователь может взаимодействовать с ними.
- Они способны быть контейнерами (родительскими компонентами) других элементов управления.
- Они обладают собственным дескриптором (в библиотеке VCL) или указателем (`widget` в библиотеке CLX).

НА ЗАМЕТКУ

Оба класса — и `TWinControl`, и `TWidgetControl` — обладают свойством `Handle` (дескриптор). Дескрипторы класса `TWinControl` являются стандартными дескрипторами Windows для элементов управления, а дескрипторы класса `TWidgetControl` — указателями на объект Qt (`widget`). Оба дескриптора расположены в свойстве `Handle` для совместимости с прежними версиями и обеспечения возможности взаимной компиляции между CLX и VCL.

Более подробная информация о классах `TWinControl` и `TWidgetControl` приведена в главах 11–14, в разделах, посвященных созданию компонентов для VCL и CLX.

Дескрипторы

Дескриптор (handle) представляет собой 32-разрядное число, указывающее на определенный экземпляр объекта в системе Win32 (в данном случае имеются в виду объекты Win32, а не Delphi). В Win32 существуют различные типы объектов: объекты ядра, пользовательские объекты и объекты GDI. Термин *объекты ядра* (kernel object) применяют к событиям, объектам отображения файлов и процессам. К *пользовательским объектам* (user object) относят объекты окон, например поля редактирования, раскрывающиеся списки и кнопки. К *объектам GDI* относят растровые изображения, кисти, шрифты и так далее.

В среде Win32 каждое окно имеет свой уникальный дескриптор, используемый в качестве параметра в некоторых функциях интерфейса API. Delphi инкапсулирует большинство функций интерфейса API Win32 и обеспечивает работу с дескрипторами. Если необходимо использовать те функции API Windows, для выполнения которых требуется дескриптор, воспользуйтесь потомками классов `TWinControl` и `TCustomControl`, так как оба эти компонента обладают свойством `Handle`.

Визуальные компоненты, не способные получать фокус

Некоторые элементы управления хоть и видны на экране, но не обладают возможностью взаимодействия с пользователем, аналогичной элементам управления Windows. Такие элементы предназначены только для показа, поэтому их называют еще *графическими элементами* (graphical control). Классы графических элементов происходят непосредственно от класса `TGraphicControl` (см. рис. 10.1 и 10.2).

В отличие от оконных, графические элементы управления не получают *фокус ввода* (input focus). Они очень полезны, когда нужно отобразить что-либо на экране, но не хочется расходовать слишком много системных ресурсов, как у оконных элементов управления. Графические элементы не используют ресурсы Windows, поэтому им и не требуется дескриптор окна (или устройство CLX). Поскольку дескриптор окна отсутствует, элемент не может получать фокус. К графическим элементам относятся `TLabel` и `TShape`; они не могут выступать в качестве контейнеров, поскольку не способны содержать дочерние элементы управления. К графическим элементам управления относятся также `TImage`, `TBevel` и `TPaintBox`.

Структура компонентов

Как уже было сказано ранее, компоненты представляют собой классы Object Pascal, инкапсулирующие функции и поведение элементов, добавляемых разработчиком в приложение для придания ему необходимого поведения и свойств. Все компоненты имеют определенную структуру, которая обсуждается далее в этой главе.

НА ЗАМЕТКУ

Класс (class) и *компонент* (component) — это не одно и то же! Компонент — это визуальное представление класса для использования в интегрированной среде разработки Delphi, а собственно класс — это структура Object Pascal, подробно описанная в главе 2, "Язык программирования Object Pascal".

Свойства

Более подробная информация о свойствах приведена в главе 2, "Язык программирования Object Pascal". Свойства предоставляют пользователю[1] интерфейс к внутренним полям компонентов: используя свойства, пользователь компонента способен считывать и модифицировать значения, хранимые полем. Как правило, пользователь не имеет прямого доступа к этим полям, поскольку в разделе определения класса компонента они объявлены как `private` (закрытые).

Свойства: доступ к полям компонента

Свойства обеспечивают доступ к данным, хранящимся в полях компонента, либо непосредственно, либо с помощью специальных *методов доступа* (access method или accessor). Рассмотрим следующее определение свойства:

```
TCustomEdit = class(TWinControl)
private
  FMaxLength: Integer;
protected
  procedure SetMaxLength(Value: Integer);
...
published
 property MaxLength: Integer read FMaxLength
                            write SetMaxLength default 0;
...
end;
```

Свойство `MaxLength` предоставляет доступ к содержимому поля `FMaxLength`. Определение свойства состоит из его имени, типа, объявлений `read` и `write` и необязательного значения `default`. Ключевое слово `read` определяет метод, позволяющий *получить* значение поля. Свойство `MaxLength` возвращает значение поля `FMaxLength` непосредственно. Ключевое слово `write` определяет метод, позволяющий *присвоить* полю необходимое значение. В данном случае в свойстве `MaxLength` для этого используется метод `SetMaxLength()`. Конечно, свойство может содержать и метод чтения значения поля. В таком случае свойство `MaxLength` могло бы быть объявлено следующим образом:

```
property MaxLength: Integer read GetMaxLength
                           write SetMaxLength default 0;
```

Метод чтения значения поля `GetMaxLength()` можно было бы объявить так:

---

[1] *Здесь и далее под пользователем авторы подразумевают не только человека, но и процедуру, функцию или участок кода, использующий компонент или объект.* – Прим. ред.

```
function GetMaxLength: Integer;
```

Методы доступа к свойствам

Методы доступа обладают одним параметром того же типа, что и у самого свойства. Метод доступа для записи (`write`) присваивает значение своего параметра соответствующему полю объекта. Промежуточный уровень в виде метода для присвоения значений полю предназначен как для защиты поля от недопустимых данных, так и для реализации, при необходимости, различных побочных эффектов. Например, рассмотрим реализацию метода `SetMaxLength()`:

```
procedure TCustomEdit.SetMaxLength(Value: Integer);
begin
  if FMaxLength <> Value then begin
    FMaxLength := Value;
    if HandleAllocated then SendMessage(Handle,
                                        EM_LIMITTEXT, Value, 0);
  end;
end;
```

Этот метод сначала проверяет, не пытается ли пользователь присвоить полю прежнее значение. Если нет, то внутреннему полю `FMaxLength` присваивается необходимое значение, а затем вызывается функция `SendMessage()` для передачи сообщения Windows `EM_LIMITTEXT` в окно элемента `TCustomEdit`. Это сообщение оповещает о размере текста, вводимого в поле редактирования. Вызов `SendMessage()` в методе доступа `write` свойства называется *побочным эффектом* (side effect) присвоения значения свойству.

Побочные эффекты — это любые действия, вызванные присвоением свойству значения. При присвоении значения свойству `MaxValue` побочный эффект заключается в том, что элементу управления "поле редактирования" передается максимальный размер вводимого поля. Естественно, побочные эффекты могут быть значительно сложнее.

Основным преимуществом использования методов для доступа к значениям внутренних полей компонента является то, что разработчик может изменять внутреннюю реализацию компонента без необходимости вносить изменения в код, использующий этот компонент.

Метод доступа на чтение способен приводить возвращаемое значение к типу, отличному от типа реального значения поля, хранящегося в компоненте.

Другая немаловажная причина использования свойств — это возможность их редактирования в процессе разработки. Свойство, объявленное в разделе `published` (публикуемые) компонента, отображается также и в окне инспектора объектов, а пользователь компонента может модифицировать его значение.

Более подробная информация о свойствах, их создании и методах доступа к ним приведена в главах 11, "Разработка компонентов VCL", и 13, "Разработка компонентов CLX", для библиотек VCL и CLX соответственно.

Типы свойств

К свойствам применимы стандартные правила, используемые в отношении типов Object Pascal. Важным моментом является то, что тип свойства определяет, каким обра-

зом это свойство будет редактироваться в окне инспектора объектов. Типы свойств перечислены в табл. 10.2. Более подробная информация по этой теме приведена в интерактивной справочной системе Delphi в разделе "Properties".

Таблица 10.2. Типы свойств

Тип свойства	*Интерпретация инспектором объектов*
Простой (simple)	Числовые, символьные и строковые свойства отображаются в окне Object Inspector как числа, символы и строки соответственно. Пользователь может вводить и редактировать их значения непосредственно
Перечислимый (enumerated)	Свойства перечислимых типов (включая `Boolean`) выводятся в том же виде, что и в исходном коде. Пользователь может циклически перебирать их значения, дважды щелкнув в столбце Value, либо выбирать их в раскрывающемся списке
Множество (set)	Свойства этого типа отображаются в окне инспектора объектов именно в виде множества. Редактируя его, пользователь рассматривает каждый элемент как логический: если элемент присутствует, то он соответствует `True`, а если нет — то `False`
Объект (object)	Свойства, которые сами являются объектами, иногда имеют свои собственные редакторы свойств. Впрочем, если у свойства, являющегося объектом, в свою очередь, есть свойства, объявленные как `published`, окно инспектора объектов позволяет расширить за их счет список свойств исходного объекта, а затем редактировать их в обычном порядке. Свойства-объекты должны происходить от класса `TPersistent`
Массив (array)	Свойства, имеющие тип массива, обязаны иметь собственные редакторы. В окне Object Inspector нет встроенных возможностей для редактирования таких свойств

Методы

Поскольку компоненты — это всего лишь объекты, у них могут быть свои методы. Методы объектов уже рассматривались в главе 2, "Язык программирования Object Pascal", поэтому не будем останавливаться на них. Ниже, в разделе "Иерархия визуальных компонентов", находится информация о некоторых основных методах компонентов различных уровней иерархии.

События

События (events) возникают в результате выполнения каких-либо действий, обычно системных (таких, как щелчок мышью или нажатие клавиши на клавиатуре). Компоненты содержат специальные свойства, называемые событиями, и пользователи компонента могут обладать кодом (называемым *обработчиком события* (event handler)), который выполняется при наступлении того или иного события.

Назначение кода событию во время разработки

Если посмотреть на страницу событий компонента TEdit, то можно увидеть события OnChange (при изменении), OnClick (при щелчке) и OnDblClick (при двойном щелчке). Для разработчика компонента событие — это просто указатель на соответствующий метод. Когда пользователи компонентов назначают событию некоторый код, то тем самым они создают обработчик события. Например, если дважды щелкнуть на некотором событии компонента во вкладке событий окна инспектора объектов, то интегрированная среда разработки Delphi создаст заготовку метода, в которую можно будет добавить собственный программный код, как это сделано для события OnClick компонента TButton в следующем примере:

```
TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
end;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Здесь находится код обработчика события }
end;
```

Подобный фрагмент кода интегрированная среда разработки Delphi создает самостоятельно.

Динамическое назначение кода события

Тот факт, что события — это указатели на методы, наиболее наглядно проявляется при динамическом назначении обработчика событию. Например, чтобы назначить собственный обработчик событию OnClick компонента TButton, следует вначале объявить и определить метод, который будет отвечать за это событие. Такой метод должен принадлежать форме, владеющей компонентом TButton, как показано в следующем фрагменте кода:

```
TForm1 = class(TForm)
  Button1: TButton;
...
private
  MyOnClickEvent(Sender: TObject); // Объявление метода
end;
...
{ Определение метода }
procedure TForm1.MyOnClickEvent(Sender: TObject);
begin
  { Собственно код находится здесь }
end;
```

Определенный пользователем метод MyOnClickEvent() должен быть назначен обработчиком события Button1.OnClick. Следующая строка показывает, как назначить этот метод событию Button1.OnClick динамически (подобное назначение обычно выполняется в обработчике события OnCreate формы):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
  Button1.OnClick := MyOnClickEvent;
end;
```

Подобная технология позволяет использовать различные обработчики событий, в зависимости от выполнения определенных условий в программе. Кроме того, можно вообще отключить обработчик события, присвоив событию значение `nil`:

```
Button1.OnClick := nil;
```

Динамическое назначение обработчика мало отличается от назначения обработчика события в окне инспектора объектов, за исключением того факта, что в последнем случае Delphi создает объявление метода. Нельзя назначить обработчику события абсолютно произвольный метод. Поскольку свойства событий — это указатели на определенные методы, они обладают специфическими признаками, зависящими от типа события. Например, типом метода для обработки события `OnMouseDown` является `TMouseEvent`, определенный как:

```
TMouseEvent = procedure (Sender: TObject; Button: TMouseButton;
                         Shift: TShiftState;
                         X, Y: Integer) of object;
```

Следовательно, методы, назначаемые обработчику события, должны удовлетворять некоторым условиям, зависящим от типа события. Обработчики должны иметь определенное количество параметров заданных типов (с определенным порядком их следования).

Как уже отмечалось, события являются свойствами. Подобно свойствам данных, события ссылаются на закрытые поля данных компонента. Эти поля всегда имеют процедурный тип, похожий на `TMouseEvent`. Обратите внимание на следующий фрагмент кода:

```
TControl = class(TComponent)
private
  FOnMouseDown: TMouseEvent;
protected
  property OnMouseDown: TMouseEvent read FOnMouseDown
                                    write FOnMouseDown;
public
end;
```

Вспомните: свойства обеспечивают доступ к полям данных компонента. Как видите, событие, будучи свойством, так же предоставляет доступ к закрытому (`private`) полю процедурного типа.

Более подробная информация о создании событий и их обработчиках приведена в главах 11, “Разработка компонентов VCL”, и 13, “Разработка компонентов CLX”.

Работа с потоками данных

Важной характеристикой компонентов является их способность работать с потоками (streaming), которая позволяет хранить компонент и значения относящихся к нему свойств в файле. При этом Delphi берет работу с потоками данных на себя, однако разработчикам компонентов иногда могут понадобиться возможности работы с потоками данных, выходящие за рамки автоматически предоставляемых Delphi. Фактически создаваемый Delphi файл `.DFM` — не более чем файл ресурсов, содержащий

информацию о потоках данных в форме и ее компонентах в виде ресурса RCDATA. Механизм работы с потоками данных в Delphi подробно изложен в главе 12, "Создание расширенного компонента VCL".

Отношения владения

Компоненты могут владеть другими компонентами. *Владелец* (owner) компонента определяется его свойством `Owner`. При закрытии компонента-владельца принадлежащие ему компоненты также закрываются, а занимаемая ими память освобождаются. В качестве примера можно привести форму, которая обычно владеет всеми расположенными на ней компонентами. При помещении некоторого компонента в форму в окне конструктора форм она автоматически становится владельцем этого компонента. Если компонент создается динамически, то в его конструктор (метод `Create`) потребуется передать параметр, указывающий владельца компонента. Это значение и будет присвоено свойству `Owner` вновь созданного компонента. В следующей строке кода показано, как передать неявную переменную `self` в конструктор `TButton.Create()`, в результате чего форма становится владельцем создаваемого компонента:

```
MyButton := TButton.Create(self);
```

Когда форма закрывается, экземпляр класса `TButton`, на который ссылается компонент `MyButton`, также освобождается. Это один из краеугольных камней фундамента VCL — форма определяет компоненты, которые подлежат уничтожению при закрытии, исходя из свойства `Components`, представляющего собой массив.

Можно также создать "ничейный" компонент, без владельца, передав в метод компонента `Create()` параметр `nil`. Но в этом случае о последующем удалении такого компонента придется позаботится самостоятельно. Этот подход можно проиллюстрировать следующим кодом:

```
MyTable := TTable.Create(nil)
try
  { Здесь выполняются действия с компонентом MyTable }
finally
  MyTable.Free;
end;
```

Здесь можно использовать конструкцию `try..finally`, обеспечивающую освобождение ресурсов при возникновении исключения. Однако желательно избегать использования описанной технологией "ничейных" компонентов, за исключением тех редких случаев, когда компоненту просто невозможно назначить владельца.

Другим свойством, связанным с владением, является свойство `Components`. Это свойство представляет собой массив, содержащий список всех компонентов, принадлежащих данному компоненту-владельцу. Приведенный ниже код выводит информацию о классах всех компонентов, перечисленных в свойстве `Components`:

```
var
  i: integer;
begin
  for i := 0 to ComponentCount - 1 do
    ShowMessage(Components[i].ClassName);
end;
```

Данный фрагмент лишь иллюстрирует принципы работы с компонентами, владеющими другими компонентами. На практике обычно используются более сложные операции с ними.

Отношения наследования

Не путайте понятия владелец и *родитель* (parent) компонента — это совершенно разные отношения. Некоторые компоненты могут быть родительскими для других компонентов. Обычно родительскими компонентами являются оконные компоненты, например потомки класса `TWinControl`. Родительские компоненты отвечают за функционирование методов отображения дочерних компонентов, а также за корректность этого отображения. Родительский компонент задается значением свойства `Parent`.

Родительский компонент не обязательно должен быть владельцем дочернего. Наличие разных родителя и владельца — совершенно нормальная ситуация для компонента.

Иерархия визуальных компонентов

Как уже было сказано в главе 2, "Язык программирования Object Pascal", абстрактный класс `TObject` является базовым классом, от которого произошли все остальные классы Delphi (см. рис. 10.1 и 10.2).

Разработчикам компонентов вовсе не обязательно создавать свои компоненты как непосредственные потомки класса `TObject`. Библиотека VCL предоставляет широкий выбор классов-потомков класса `TObject`, и создаваемые компоненты могут быть производными от них. Эти уже существующие классы обеспечивают большинство функциональных возможностей, которые могут понадобиться новым компонентам. Лишь при создании классов, не являющихся компонентами, имеет смысл делать их потомками класса `TObject`.

Методы `Create()` и `Destroy()` класса `TObject` предназначены для выделения и освобождения памяти для экземпляра объекта. Конструктор `TObject.Create()` возвращает указатель на созданный объект. Класс `TObject` содержит несколько полезных функций, позволяющих получить информацию об объекте.

Библиотека VCL использует в основном внутренние вызовы методов класса `TObject`, что позволяет получить необходимую информацию о типе класса, его имени, базовых классах (предках) для экземпляра любого класса, поскольку все они являются потомками `TObject`.

СОВЕТ

Используйте метод `TObject.Free()` вместо метода `TObject.Destroy()`. Метод `Free` вызывает метод `Destroy()`, но перед этим проверяет, имеет ли указатель объекта значение `nil`, что позволяет избежать передачи исключения при попытке уничтожить несуществующий объект.

Класс TPersistent

Класс TPersistent происходит непосредственно от класса TObject. Особенностью класса TPersistent является то, что экземпляры происходящих от него объектов могут читать и записывать свои свойства в поток. Поскольку все компоненты являются потомками класса TPersistent, то все они обладают этой способностью. Класс TPersistent определяет не специальные свойства или события, а только некоторые методы, полезные как пользователям, так и разработчикам компонентов.

Методы класса TPersistent

В табл. 10.3 приведен список наиболее интересных методов, определенных в классе TPersistent.

Таблица 10.3. Методы класса TPersistent

Метод	*Назначение*
Assign()	Этот открытый (public) метод позволяет компоненту присваивать себе данные, связанные с другим компонентом
AssignTo()	Это защищенный (protected) метод, в который производные от TPersistent классы должны помещать реализацию собственного метода AssignTo(), объявленного в VCL. При вызове данного метода непосредственно из класса TPersistent передается исключение. С помощью такого метода компонент может присвоить свои данные другому экземпляру класса — операция, обратная выполняемой методом Assign()
DefineProperties()	Этот защищенный метод позволяет разработчику компонентов определить, каким образом компонент хранит значения дополнительных или закрытых свойств. Обычно он используется для хранения в компонентах данных нестандартного формата, например бинарных

Более подробная информация о работе компонентов с потоками приведена в главе 12, "Работа с файлами", предыдущего издания — *Delphi 5 Руководство разработчика*, находящегося на прилагаемом CD. А пока достаточно знать, что с помощью потока компоненты могут быть сохранены и прочитаны из файла на диске.

Класс TComponent

Этот класс происходит непосредственно от класса TPersistent. Отличительными особенностями класса TComponent являются возможность редактирования его свойств в окне инспектора объектов и способность объектов данного класса владеть другими компонентами.

Невизуальные компоненты также происходят от класса TComponent, наследуя возможность их редактирования в процессе разработки. Компонент TTimer — пре-

красный пример такого невизуального потомка класса TComponent. Не являясь визуальным элементом управления, он, тем не менее, доступен в палитре компонентов.

Класс TComponent определяет наиболее важные свойства и методы, описанные в следующих разделах.

Свойства класса TComponent

Свойства класса TComponent и описание их назначения приведены в табл. 10.4.

Таблица 10.4. Специальные свойства класса TComponent

Свойство	*Назначение*
Owner	Указывает владельца компонента
ComponentCount	Определяет количество компонентов, принадлежащих данному компоненту
ComponentIndex	Текущая позиция в списке компонентов, принадлежащих данному компоненту. Первый компонент списка имеет номер 0
Components	Массив, содержащий список компонентов, принадлежащих данному компоненту
ComponentState	Это свойство содержит информацию о текущем состоянии компонента типа TComponentState. Дополнительная информация о классе TComponentState содержится в интерактивной справочной системе Delphi и в главе 11, "Разработка компонентов VCL"
ComponentStyle	Управляет поведением компонента. Этому свойству могут быть присвоены два значения — csInheritable и csCheckPropAvail. Их назначение объясняется в интерактивной справочной системе Delphi
Name	Содержит имя компонента
Tag	Целочисленное свойство без определенного значения. Предназначено для пользователей компонентов, а не их разработчиков. Благодаря целочисленному формату свойство Tag может быть использовано для хранения указателя на структуру данных или экземпляр объекта
DesignInfo	Используется конструктором форм. Значение этого свойства изменять не следует

Методы класса TComponent

Класс TComponent определяет несколько методов, связанных с его способностью владеть другими компонентами и возможностью редактирования в конструкторе форм.

Класс TComponent определяет конструктор компонентов Create(). Этот конструктор создает экземпляр компонента и назначает ему владельца на основе передаваемого ему параметра. В отличие от конструктора TObject.Create(), конструктор класса TComponent.Create() является виртуальным. Потомки класса TComponent, включающие реализацию этого конструктора, обязаны заявить о конструкторе Cre-

ate(), используя директиву override (переопределение). Хотя можно использовать и другие конструкторы для класса компонента, однако лишь конструктор VCL TComponent.Create() позволяет создать экземпляр класса во время разработки или загрузить компонент из потока во время выполнения программы.

Деструктор TComponent.Destroy() уничтожает компонент вместе со всеми ресурсами, выделенными этим компонентом.

Метод TComponent.Destroying() приводит сам компонент и принадлежащие ему компоненты в состояние, предшествующее их уничтожению. Метод TComponent.DestroyComponents() уничтожает все компоненты, принадлежащие данному компоненту. Но эти методы редко приходится использовать самостоятельно.

Метод TComponent.FindComponent() удобно использовать для получения указателя на компонент с известным именем. Предположим, что главная форма имеет компонент TEdit по имени Edit1. Тогда для получения указателя на экземпляр компонента достаточно выполнить следующий код:

```
EditInstance := FindComponent.('Edit1');
```

В этом примере экземпляр компонента EditInstance должен иметь тип TEdit. Метод FindComponent возвратит значение nil, если объект с указанным именем не существует.

Метод TComponent.GetParentComponent() позволяет найти экземпляр предка компонента. Данный метод возвращает значение nil в случае отсутствия такого компонента.

Метод TComponent.HasParent() возвращает логическое значение, определяющее, есть ли у компонента родитель. Запомните, что этот метод не проверяет наличие владельца у заданного компонента.

Метод TComponent.InsertComponent() позволяет добавить компонент, переходящий во владение вызывающего компонента. Метод TComponent.RemoveComponent() удаляет все компоненты, принадлежащие тому компоненту, который вызвал этот метод. В обычных случаях такие методы не используются, потому что они автоматически вызываются конструктором компонента Create() и его деструктором Destroy().

Класс TControl

Класс TControl определяет свойства, методы и события, общие для большинства визуальных компонентов. Например, класс TControl позволяет визуальным компонентам отображаться на экране. Класс TControl содержит такие позиционные свойства, как Top и Left, свойства размеров Width и Height, значения которых определяют размеры элемента по горизонтали и вертикали. Имеются и некоторые другие свойства: ClientRect, ClientWidth и ClientHeight.

Класс TControl содержит свойства, отвечающие за внешний вид и доступ к компоненту: Visible, Enabled и Color. В свойстве Font даже можно задать шрифт, используемый для текста, помещаемого в компонент TControl. Этот текст выводится с помощью свойств Text и Caption.

В классе TControl впервые появляются некоторые стандартные события: события мыши — OnClick, OnDblClick, OnMouseDown, OnMouseMove и OnMouseUp, а также события перетаскивания с помощью мыши — OnDragOver, OnDragDrop и OnEndDrag.

Сам по себе класс TControl не очень полезен на своем уровне иерархии. Прямые потомки этого класса никогда не создаются.

Компонент TControl может иметь родительский компонент. Он обязательно должен принадлежать классу TWinControl (в VCL) или TWidgetControl (в CLX). Родительские элементы управления должны быть *оконными* (windowed) элементами управления. Для этого в класс TControl введено свойство Parent.

Большинство элементов управления Delphi являются производными от прямых потомков класса TControl: классов TWinControl и TWidgetControl.

Классы TWinControl и TWidgetControl

Стандартные элементы управления происходят от класса TWinControl (в VCL) и от класса TWidgetControl (в CLX). Объекты пользовательского интерфейса программ Windows — это и есть стандартные элементы управления. Поля редактирования (edit control), списки (list box), раскрывающиеся списки (list box) и кнопки (button) — вот примеры подобных элементов. Поскольку Delphi сам инкапсулирует поведение стандартных элементов управления, а не использует для этого функции интерфейса API на уровне Windows или Qt, он обладает возможностью непосредственно управлять этими элементами.

Три основные особенности класса TWinControl: объекты этого класса имеют дескриптор Windows, могут получить фокус ввода и являться родительским элементом для других компонентов. Элементы управления CLX не имеют дескриптора окна; вместо него используется указатель на объект Qt, который выполняет ту же самую роль. Свойства, методы и события обеспечивают обработку изменения фокуса, событий клавиатуры, отображение элементов и другие необходимые функции.

Обычно разработчикам приложений достаточно просто знать, как использовать потомки классов TWinControl и TWidgetControl, а разработчикам компонентов, несомненно, придется изучить их намного глубже.

Свойства классов TWinControl и TWidgetControl

В классах TWinControl и TWidgetControl определено несколько свойств, предназначенных для изменения фокуса и внешнего вида элемента. Далее речь пойдет только о классе TWinControl, но все это будет справедливо и для класса TWidgetControl.

Свойство TWinControl.Brush используется для отображения элемента управления на экране. (Более подробная информация по этой теме приведена в главе 8, "GDI, шрифты и графика", предыдущего издания — *Delphi 5 Руководство разработчика*, находящегося на прилагаемом CD.)

Свойство-массив TWinControl.Controls предоставляет доступ к списку всех элементов управления, для которых вызывающий компонент TWinControl является родительским.

Свойство TWinControl.ControlCount содержит количество дочерних элементов управления.

Свойство TWinControl.Ctl3D определяет использование трехмерной стилизации при отображении элемента.

Свойство TWinControl.Handle содержит дескриптор объекта Windows, инкапсулированный классом TWinControl. Именно этот дескриптор следует использовать в качестве параметра функций API Win32, требующих указания дескриптора окна.

Свойство TWinControl.HelpContext содержит контекстный справочный номер, соответствующий определенному справочному окну в файле справки. Это свойство

применяется для поддержки функций контекстно-зависимой справки для отдельных элементов управления.

Свойство `TWinControl.Showing` содержит информацию о видимости элемента, а свойство `TWinControl.TabStop` — логическое значение, определяющее поведение элемента при использовании клавиши <Tab>. Свойство `TWinControl.TabOrder` показывает, где в списке элементов родительского компонента, изменяющих фокус при нажатии клавиши <Tab>, находится указанный элемент (если он там есть).

Методы класса TWinControl

Компонент `TWinControl` так же предлагает несколько методов создания и позиционирования окна, управления фокусом и диспетчеризации событий. Подробное обсуждение этих методов явно не укладывается в рамки одной главы, однако все они исчерпывающе описаны в интерактивной справочной системе Delphi. Далее рассмотрим наиболее интересные методы.

Методы создания окна, его восстановления и уничтожения в основном применяются разработчиками компонентов и рассматриваются в главе 11, "Разработка компонентов VCL". Это методы класса `TWinControl` библиотеки VCL — `CreateParams()`, `CreateWnd()`, `CreateWindowHandle()`, `DestroyWnd()`, `DestroyWindowHandle()` и `RecreateWnd()`, а также методы класса `TWidgetControl` библиотеки CLX — `CreateWidget()`, `DestroyWidget()`, `CreateHandle()` и `DestroyHandle()`.

К методам управления фокусом, позиционированием и выравнивания окна относятся: `CanFocus()`, `Focused()`, `AlignControls()`, `EnableAlign()`, `DisableAlign()` и `ReAlign()`.

События класса TWinControl

Класс `TWinControl` предлагает новые события для работы с клавиатурой и управления фокусом. События клавиатуры — это `OnKeyDown`, `OnKeyPress` и `OnKeyUp`. К событиям управления фокусом относятся `OnEnter` и `OnExit`. Все они описаны в интерактивной справочной системе Delphi.

Класс TGraphicControl

В отличие от класса `TWinControl`, объекты класса `TGraphicControl` не имеют дескриптора окна и поэтому не могут получить фокус. Они также не могут быть родительскими для других элементов управления. Класс `TGraphicControl` используется, когда необходимо отобразить в форме какую-то информацию, не предоставляя пользователю доступ к этому элементу. Преимуществом элементов `TGraphicControl` является то, что они не нуждаются в дескрипторе, использующем системные ресурсы Windows. К тому же отсутствие дескриптора говорит о том, что элемент `TGraphicControl` не участвует в непрерывном процессе перерисовки окон. Таким образом, отображение элемента типа `TGraphicControl` происходит значительно быстрее, чем его эквивалента из класса `TWinControl`.

Объект класса `TGraphicControl` способен реагировать на события мыши. В действительности сообщения мыши обрабатываются родительским компонентом класса `TGraphicControl`, а затем передаются его дочерним элементам.

Класс `TGraphicControl` обладает способностью выводить на экран представляемые им элементы управления, а следовательно, обладает свойством `Canvas` типа

TCanvas. Он также содержит метод Paint(), который производные классы элементов обязаны переопределять.

Класс TCustomControl

Как можно было заметить, имена некоторых классов, производных от TWinControl, начинаются с сочетания "TCustom", например: TCustomComboBox, TCustomControl, TCustomEdit и TCustomListBox.

Нестандартные, или *пользовательские* (custom), элементы управления выполняют почти такие же функции, как и потомки класса TWinControl, за исключением того, что благодаря некоторым специальным визуальным особенностям и характеристикам поведения разработчик получает заготовку, на основе которой можно создавать свои собственные пользовательские компоненты. Они особенно полезны разработчикам компонентов, которым приходится создавать собственные визуальные элементы управления.

Другие классы

Несколько классов не принадлежат к разряду компонентов, но служат для программной поддержки существующих компонентов. Как правило, эти классы являются свойствами других компонентов и происходят непосредственно от класса TPersistent. Некоторые из них имеют типы TString, TCanvas и TCollection.

Классы TStrings и TStringsLists

Абстрактный класс TStrings позволяет манипулировать списками строк, принадлежащих компонентам, подобно тому, как это делается с помощью элемента управления типа TListBox. Однако на самом деле класс TStrings не управляет памятью для хранения и обработки таких строк (этим занимается тот элемент управления, который является владельцем класса TStrings). Класс TStrings лишь определяет методы и свойства, необходимые для доступа к строкам и их обработки, не используя ни функции API, ни системные сообщения.

Заметьте, что TStrings — это абстрактный класс, т.е. в нем нет реализации кода, необходимого для работы со строками, он лишь определяет требуемые методы. Реализация же методов обработки строк относится к компетенции потомков этого класса.

Чтобы внести ясность, приведем такой пример. Допустим, имеется несколько компонентов со свойствами типа TStrings — TListBox.Items, TMemo.Lines и TComboBox.Items. Каждое из этих свойств имеет тип TStrings. Но как вызвать методы этих свойств, если они еще не воплощены в коде? Хороший вопрос. Ответ таков: хотя все эти свойства определены как TStrings, переменная, к которой относится, например, свойство TListBox.FItems, будет создана как экземпляр производного класса, методы которого и будут вызываться. Чтобы прояснить этот момент, отметим, что переменная FItems является закрытым полем свойства Items класса TListBox:

```
TCustomListBox = class(TWinControl)
  private
    FItems: TStrings;
```

Модуль StdCtrls.pas, являющийся частью библиотеки VCL, определяет класс TListBoxStrings как производный от класса TStrings. Листинг 10.1 демонстрирует его определение.

Листинг 10.1. Объявление класса TListBoxStrings

```
TListBoxStrings = class(TStrings)
  private
    ListBox: TCustomListBox;
  protected
    procedure Put(Index: Integer; const S: string); override;
    function Get(Index: Integer): string; override;
    function GetCount: Integer; override;
    function GetObject(Index: Integer): TObject; override;
    procedure PutObject(Index: Integer;
                        AObject: TObject); override;
    procedure SetUpdateState(Updating: Boolean); override;
  public
    function Add(const S: string): Integer; override;
    procedure Clear; override;
    procedure Delete(Index: Integer); override;
    procedure Exchange(Index1, Index2: Integer); override;
    function IndexOf(const S: string): Integer; override;
    procedure Insert(Index: Integer; const S: string); override;
    procedure Move(CurIndex, NewIndex: Integer); override;
end;
```

Затем StdCtrls.pas определяет реализацию каждого метода этого производного класса. Когда класс TListBox создает экземпляры для своей переменной FItems, на самом деле создается экземпляр его производного класса, к свойству FItems которого и происходит обращение:

```
constructor TCustomListBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
  // Создается экземпляр TListBoxStrings
  FItems := TListBoxStrings.Create;
  ...
end;
```

Следует четко уяснить, что класс TStrings лишь определяет свои методы, но не содержит их реализацию — ее осуществляют производные классы. Всем разработчикам компонентов полезно изучить данную технологию. Когда возникают какие-либо вопросы или сомнения по этому поводу, всегда можно обратиться к исходному коду библиотек VCL или CLX, чтобы увидеть, как именно *Borland* реализовала такие методы.

Если речь не идет о разработке компонента, то при необходимости манипулирования списком строк следует воспользоваться другим потомком класса TStrings — вполне самодостаточным классом TStringList. Он обрабатывает список строк, размещаемых вне компонента. Его главным достоинством является полная совместимость с классом TStrings. Это означает, что экземпляр класса TStringList можно

присвоить непосредственно соответствующему свойству элемента управления класса TStrings. В следующем фрагменте кода показано, как можно создать экземпляр класса TStringList:

```
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;
```

Чтобы добавить строки во вновь созданный экземпляр класса TStringList, поступите следующим образом:

```
MyStringList.Add('Red');
MyStringList.Add('White');
MyStringList.Add('Blue');
```

Если необходимо добавить одинаковые строки одновременно в компоненты TMemo и TListBox, можно воспользоваться совместимостью свойств TStrings, принадлежащих различным компонентам, и выполнить присвоение одной строкой кода:

```
Memo1.Lines.Assign(MyStringList);
ListBox1.Items.Assign(MyStringList);
```

Вместо непосредственного присвоения, подобного Memo1.Lines := MyStringList, можно скопировать экземпляр класса TStrings с помощью метода Assign(). В табл. 10.5 приведены наиболее часто используемые методы классов типа TStrings.

Таблица 10.5. Наиболее популярные методы класса TStrings

Метод	*Описание*
`Add(const S: String): Integer`	Добавляет строку S в список и возвращает ее позицию в нем
`AddObject(const S: string; Aobject: TObject): Integer`	Добавляет строку и объект в строку или в объект списка строк
`AddStrings(Strings: TStrings)`	Копирует строки из указанного объекта TStrings в конец существующего списка строк
`Assign(Source: TPersistent)`	Замещает существующие строки строками, заданными параметром Source

Окончание табл. 10.5.

Метод	*Описание*
`Clear`	Удаляет все строки из списка
`Delete(Index: Integer)`	Удаляет строку, находящуюся в позиции `Index`
`Exchange(Index1, Index2: Integer)`	Меняет местами строки с позициями `Index1` и `Index2`
`IndexOf(const S: String): Integer`	Возвращает позицию строки `S` в списке
`I nsert(Index: Integer; const S: String)`	Вставляет строку `S` в позицию `Index`
`Move(CurIndex, NewIndex: Integer)`	Перемещает строку с позицией `CurIndex` в позицию `NewIndex`
`LoadFromFile(const FileName: String)`	Считывает текстовый файл по имени `FileName` и помещает его строки в список
`SaveToFile(const FileName: string)`	Сохраняет список в текстовом файле `FileName`

Класс TCanvas

Свойство `Canvas` типа `TCanvas` применяется в оконных элементах управления и представляет собой его поверхность, используемую для прорисовки содержимого окна. Класс `TCanvas` инкапсулирует так называемый *контекст устройства* (device context) окна. Этот класс содержит множество функций и объектов, необходимых для прорисовки поверхности окна. Более подробная информация по данной теме приведена в главе 8, "GDI, шрифты и графика", предыдущего издания — *Delphi 5 Руководство разработчика*, находящегося на прилагаемом CD.

Информация о типах времени выполнения (RTTI)

Общая концепция информации о типах времени выполнения (RTTI — Runtime Type Information) рассматривалась в главе 2, "Язык программирования Object Pascal". Теперь наступил момент более углубленно изучить информацию RTTI, использование которой позволяет расширить обычные возможности языка Object Pascal. Рассмотрим, как получать информацию о текущих типах объектов и данных в выполняемых программах, а также то, как это делается в интегрированной среде разработки Delphi.

Итак, каким образом проявляет себя RTTI? Судить об этом можно, по крайней мере, по двум областям применения. Первая — работа с интегрированной средой разработки Delphi. С помощью RTTI интегрированная среда разработки узнает все данные об объекте или компоненте, с которым она работает, и помещает их в окно инспектора объектов. Конечно, не вся эта информация относится к RTTI, но для удобства будем рассматривать только ее. Вторая область — выполняемый код. В главе 2, "Язык программирования Object Pascal", уже упоминались операторы `is` и `as`.

Для иллюстрации типичного использования информации RTTI рассмотрим оператор `is`.

Предположим, что все компоненты `TEdit` создаваемой формы необходимо сделать доступными только для чтения. Это достаточно просто: пройдитесь в цикле по всем компонентам, используя оператор `is` для выявления компонентов класса `TEdit`, а затем присвойте их свойствам `ReadOnly` значения `True`. Например:

```
for i := 0 to ComponentCount - 1 do
  if Components[i] is TEdit then
    TEdit(Components[i]).ReadOnly := True;
```

Оператор `as` обычно применяется для выполнения определенных действий над параметром `Sender` обработчика события, используемого в различных компонентах. Если все компоненты произошли от общего предка, к свойству которого необходимо получить доступ, то в обработчике события можно использовать оператор `as` для безопасного приведения типа параметра `Sender` к соответствующему потомку, что позволит получить доступ к нужному свойству. Например:

```
procedure TForm1.ControlOnClickEvent(Sender: TObject);
var
  i: integer;
begin
  (Sender as TControl).Enabled := False;
end;
```

Эти примеры с *безопасным использованием типов* (typesafe programming) иллюстрируют применение расширенных возможностей языка Object Pascal, опосредованным образом использующих RTTI. Теперь рассмотрим задачу, решение которой требует непосредственного использования информации RTTI.

Допустим, что существует форма с различными компонентами: одни из них предназначены для работы с данными, а другие — нет. Но действия необходимо выполнить только над компонентами, работающими с данными. С одной стороны, можно было бы в цикле перебрать весь массив `Components` формы и проверить каждый из компонентов на соответствие требуемому типу. А для этого пришлось бы перебрать все типы компонентов, используемых для работы с данными. С другой стороны, у компонентов, работающих с данными, может не быть общего предка (базового класса), соответствие которому было бы легко проверить. Для такого случая, например, неплохо было бы иметь тип `TDataAwareControl`, но его, к сожалению, не существует.

Гораздо проще определить, интересует ли нас данный компонент, проверив существование свойства `DataSource` (известно, что это свойство присуще всем компонентам, работающим с данными). Для такой проверки потребуется непосредственное использование информации RTTI.

Далее в этой главе более подробно рассматриваются возможности использования информации RTTI для решения подобных проблем.

Модуль TypInfo.pas — определитель RTTI

Информация о типе существует для любого объекта (потомка класса `TObject`). Она содержится в памяти и при необходимости считывается интегрированной средой разработки или динамической библиотекой (runtime library). В модуле `Typ-`

Info.pas определяются структуры, позволяющие запрашивать информацию о типах. Некоторые методы класса TObject, описанные в главе 2, "Язык программирования Object Pascal", перечислены в табл. 10.6.

Таблица 10.6. Методы класса TObject

Функция	*Возвращаемый тип*	*Что возвращается*
ClassName()	String	Имя класса объекта
ClassType()	TClass	Тип объекта
InheritsFrom()	Boolean	Логический индикатор, определяющий, происходит ли класс от другого заданного класса
ClassParent()	TClass	Тип базового класса объекта
InstanceSize()	Word	Размер экземпляра объекта в байтах
ClassInfo()	Pointer	Указатель на находящуюся в памяти информацию (RTTI) об объекте

Теперь пришло время подробно остановиться на функции ClassInfo(), определяемой следующим образом:

```
class function ClassInfo: Pointer;
```

Данная функция возвращает указатель на информацию RTTI для вызывающего класса, а именно — на структуру типа PTypeInfo. Этот тип определен в модуле TypeInfo.pas как указатель на структуру TTypeInfo. Оба определения приведены в следующем фрагменте TypeInfo.pas:

```
PPTypeInfo = ^PTypeInfo;
  PTypeInfo = ^TTypeInfo;
  TTypeInfo = record
    Kind: TTypeKind;
    Name: ShortString;
    {TypeData: TTypeData}
  end;
```

Закомментированное поле TypeData содержит информацию о типе данного класса. Тип этой информации зависит от значения поля Kind. Такое поле может принимать одно из значений перечисления TTypeKind:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
             tkString, tkSet, tkClass, tkMethod, tkWChar,
             tkLString, tkWString, tkVariant, tkArray, tkRecord,
             tkInterface);
```

Вновь обратимся к коду модуля TypeInfo.pas и ознакомимся с подтипами некоторых перечисленных выше значений. Например, tkFloat может содержать такие подтипы:

```
TFloatType = (ftSingle, ftDouble, ftExtended, ftComp, ftCurr);
```

Таким образом, по значению поля Kind можно определить, к какому типу относится переменная TypeData. Определение структура TTypeData в модуле TypeInfo.pas приведено в листинге 10.2.

Листинг 10.2. Структура TTypeData

```
PTypeData = ^TTypeData;
TTypeData = packed record
  case TTypeKind of
    tkUnknown, tkLString, tkWString, tkVariant: ();
    tkInteger, tkChar, tkEnumeration, tkSet, tkWChar: (
      OrdType: TOrdType;
      case TTypeKind of
        tkInteger, tkChar, tkEnumeration, tkWChar: (
          MinValue: Longint;
          MaxValue: Longint;
          case TTypeKind of
            tkInteger, tkChar, tkWChar: ();
            tkEnumeration: (
              BaseType: PPTypeInfo;
              NameList: ShortStringBase));
        tkSet: (
          CompType: PPTypeInfo));
    tkFloat: (FloatType: TFloatType);
    tkString: (MaxLength: Byte);
    tkClass: (
      ClassType: TClass;
      ParentInfo: PPTypeInfo;
      PropCount: SmallInt;
      UnitName: ShortStringBase;
     {PropData: TPropData});
    tkMethod: (
      MethodKind: TMethodKind;
      ParamCount: Byte;
      ParamList: array[0..1023] of Char
     {ParamList: array[1..ParamCount] of
        record
          Flags: TParamFlags;
          ParamName: ShortString;
          TypeName: ShortString;
        end;
        ResultType: ShortString});
    tkInterface: (
      IntfParent : PPTypeInfo; { ancestor }
      IntfFlags : TIntfFlagsBase;
      Guid : TGUID;
      IntfUnit : ShortStringBase;
     {PropData: TPropData});
    tkInt64: (
      MinInt64Value, MaxInt64Value: Int64);
  end;
```

Как видите, структура TTypeData представляет собой большую запись с вариантами. Для тех, кто знаком с подобными записями и указателями, нетрудно разобраться и с принципами RTTI. Они кажутся сложными только из-за того, что технология RTTI не документирована.

НА ЗАМЕТКУ

Иногда *Borland* не документирует свои технологии, так как они подлежат изменению в следующей версии продукта. При использовании таких возможностей (например недокументированной технологии RTTI) может оказаться, что созданный код будет не вполне совместим с будущими версиями Delphi.

Теперь продемонстрируем использование структур RTTI для получения информации о типах.

Получение информации о типах

Чтобы продемонстрировать технологию получения информации о типе объекта в процессе выполнения приложения, был создан проект, код главной формы которого представлен в листинге 10.3.

Листинг 10.3. Главная форма проекта ClassInfo.dpr

```
unit MainFrm;

interface
uses
  Windows, Messages, SysUtils, Classes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls, QExtCtrls;

type

  TMainForm = class(TForm)
    lbSampClasses: TListBox;
    lbPropList: TListBox;
    lbBaseClassInfo: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure lbSampClassesClick(Sender: TObject);
    procedure FormDblClick(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo;
```

```
{$R *.XFM}

function CreateAClass(const AClassName: string): TObject;
{ Этот метод показывает, как создать класс по его имени. Не
забудьте зарегистрировать данный класс с помощью метода Register-
Classe(), как это сделано в методе инициализации данного модуля. }
var
  C : TFormClass;
  SomeObject: TObject;
begin
  C := TFormClass(FindClass(AClassName));
  SomeObject := C.Create(nil);
  Result := SomeObject;
end;


procedure GetBaseClassInfo(AClass: TObject; AStrings: TStrings);
{ Этот метод позволяет получить некоторые основные данные RTTI
объекта и возвращает эту информацию параметру AStrings. }
var
  ClassTypeInfo: PTypeInfo;
  ClassTypeData: PTypeData;
  EnumName: String;
begin
  ClassTypeInfo := AClass.ClassInfo;
  ClassTypeData := GetTypeData(ClassTypeInfo);
  with AStrings do begin
    Add(Format('Class Name:     %s', [ClassTypeInfo.Name]));
    EnumName := GetEnumName(TypeInfo(TTypeKind),
                            Integer(ClassTypeInfo.Kind));
    Add(Format('Kind:           %s', [EnumName]));
    Add(Format('Size:           %d', [AClass.InstanceSize]));
    Add(Format('Defined in:     %s.pas',
               [ClassTypeData.UnitName]));
    Add(Format('Num Properties: %d', [ClassTypeData.PropCount]));
  end;
end;

procedure GetClassAncestry(AClass: TObject; AStrings: TStrings);
{ Этот метод определяет предков данного объекта и возвращает имена
классов в параметр AStrings. }
var
  AncestorClass: TClass;
begin
  AncestorClass := AClass.ClassParent;
  { Перебирает все родительские классы, начиная с Sender и до
    конца иерархии наследования. }
  AStrings.Add('Class Ancestry');
  while AncestorClass <> nil do begin
    AStrings.Add(Format('    %s',[AncestorClass.ClassName]));
    AncestorClass := AncestorClass.ClassParent;
  end;
end;
```

```
procedure GetClassProperties(AClass: TObject; AStrings: TStrings);
{ Этот метод позволяет получить имена свойств и методов данного
объекта и вернуть эту информацию параметру AStrings. }
var
  PropList: PPropList;
  ClassTypeInfo: PTypeInfo;
  ClassTypeData: PTypeData;
  i: integer;
  NumProps: Integer;
begin
  ClassTypeInfo := AClass.ClassInfo;
  ClassTypeData := GetTypeData(ClassTypeInfo);
  if ClassTypeData.PropCount <> 0 then begin
    // Резервирует память, необходимую для хранения указателей на
    // структуры TPropInfo, исходя из количества свойств.
    GetMem(PropList, SizeOf(PPropInfo) * ClassTypeData.PropCount);
    try
      // Заполняет список PropList указателями
      // на структуры TPropInfo.
      GetPropInfos(AClass.ClassInfo, PropList);
      for i := 0 to ClassTypeData.PropCount - 1 do
        // Отфильтровываются свойства, являющиеся событиями
        // (свойства, хранящие указатели на методы).
        if not (PropList[i]^.PropType^.Kind = tkMethod) then
          AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
                       PropList[i]^.PropType^.Name]));
      // Получить свойства, являющиеся событиями
      // (свойства, хранящие указатели на методы).
      NumProps := GetPropList(AClass.ClassInfo, [tkMethod],
                              PropList);
      if NumProps <> 0 then begin
        AStrings.Add('');
        AStrings.Add('   EVENTS   ================ ');
        AStrings.Add('');
      end;
      // Параметр AStrings заполняется именами событий.
      for i := 0 to NumProps - 1 do
        AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
                     PropList[i]^.PropType^.Name]));
    finally
      FreeMem(PropList,
              SizeOf(PPropInfo) * ClassTypeData.PropCount);
    end;
  end;

end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Добавить в список несколько классов.
  lbSampClasses.Items.Add('TButton');
  lbSampClasses.Items.Add('TForm');
```

```
  lbSampClasses.Items.Add('TListBox');
  lbSampClasses.Items.Add('TPaintBox');
  lbSampClasses.Items.Add('TFindDialog');
  lbSampClasses.Items.Add('TOpenDialog');
  lbSampClasses.Items.Add('TTimer');
  lbSampClasses.Items.Add('TComponent');
  lbSampClasses.Items.Add('TGraphicControl');
end;

procedure TMainForm.lbSampClassesClick(Sender: TObject);
var
  SomeComp: TObject;
begin
  lbBaseClassInfo.Items.Clear;
  lbPropList.Items.Clear;
  // Создать экземпляр выбранного класса.
  SomeComp :=
      CreateAClass(lbSampClasses.Items[lbSampClasses.ItemIndex]);
  try
    GetBaseClassInfo(SomeComp, lbBaseClassInfo.Items);
    GetClassAncestry(SomeComp, lbBaseClassInfo.Items);
    GetClassProperties(SomeComp, lbPropList.Items);
  finally
    SomeComp.Free;
  end;
end;

procedure TMainForm.FormDblClick(Sender: TObject);
begin
  ShowMessage('hello');
end;

initialization
begin
  RegisterClasses([TButton, TForm, TListBox, TPaintBox,
                   TFindDialog, TOpenDialog, TTimer,
                   TComponent, TGraphicControl]);
end;

end.
```

НА ЗАМЕТКУ

Пример, демонстрирующий применение RTTI для CLX, находится на прилагаемом CD в каталоге `CLX` для настоящей главы.

В этой главной форме содержится три списка. Список `lbSampClasses` содержит имена классов нескольких объектов, информацию о типах которых необходимо получить. При выборе объекта из списка `lbSampClasses` в список `lbBaseClassInfo` заносится основная информация о размере и происхождении этого объекта, а в список `lbPropList` — сведения о свойствах выбранного объекта.

Для получения информации о классе используются три вспомогательные процедуры.

- `GetBaseClassInfo()` — заполняет список основной информацией об объекте — типе, размере, модуле, в котором он определен, и количестве свойств.
- `GetClassAncestry()` — заполняет список именами объектов-предков.
- `GetClassProperties()` — заполняет список свойствами данного класса и их типами.

Каждая процедура использует в качестве параметров экземпляр объекта и список строк.

При выборе пользователем одного из классов в списке `lbSampClasses` его процедура обработки события `OnClick` (`lbSampClassesClick()`) вызывает вспомогательную функцию `CreateAClass()`, которая создает экземпляр класса, заданного именем типа класса. Затем этот экземпляр объекта передается по назначению, чтобы в нужный список было занесено значение соответствующего свойства `TListBox.Items`.

СОВЕТ

Функция `CreateAClass()` способна создать класс по его имени. Но, как уже было сказано, передаваемый этой функции в качестве параметра класс необходимо зарегистрировать с помощью процедуры `RegisterClasses()`.

Получение информации о типах объектов

Процедура `GetBaseClassInfo()` передает значение, возвращаемое функцией `TObject.ClassInfo()`, функции `GetTypeData()`, определенной в модуле `TypeInfo.pas`. Эта процедура возвращает указатель на структуру `TTypeData`, построенную для класса, структура `PTypeInfo` которого была передана данной процедуре (см. листинг 10.2). Процедура `GetBaseClassInfo()` просто указывает на различные поля структур `TTypeInfo` и `TTypeData` для расширения списка `AStrings`. Обратите внимание на использование функции `GetEnumName()` для получения строки по имени перечислимого типа. Это также функция RTTI, определенная в файле `TypInfo.pas`. О RTTI перечислимых типов рассказывается в следующем разделе.

СОВЕТ

Используйте функцию `GetTypeData()`, определенную в файле `TypInfo.pas`, чтобы получить указатель на структуру `TTypeInfo` данного класса. Результат вызова функции `TObject.ClassInfo()` следует передать функции `GetTypeData()`.

СОВЕТ

Чтобы получить имя перечислимого типа, представленного в виде строки, можно воспользоваться функцией `GetEnumName()`. Функция `GetEnumValue()` возвращает значение перечислимого типа по заданному имени.

Получение информации о происхождении объекта

Процедура GetClassAncestry() заполняет список строк именами базовых классов данного объекта. Это довольно простая операция, использующая процедуру ClassParent() данного объекта и возвращающая указатель на тип TClass базового класса или nil, если достигнута вершина иерархии наследования (класс-прародитель). Процедура GetClassAncestry() "проходит" по всей иерархии наследования и добавляет имена базовых классов (предков) в соответствующий список строк.

Получение информации о типах свойств объекта

Если объект обладает свойствами, то значение TTypeData.PropCount соответствует их общему количеству. Существует несколько подходов для получения информации о свойствах данного класса, но здесь приводится лишь два из них.

Процедура GetClassProperties() начинается так же, как и предыдущих два метода, — с передачи результата функции ClassInfo() в функцию GetTypeData() для получения указателя на структуру TTypeData класса. Затем, исходя из значения ClassTypeData.PropCount, выделяется память для переменной PropList, определенной как указатель на массив PPropList, который, в свою очередь, определен в модуле TypeInfo.pas:

```
type
    PPropList = ^TPropList;
    TPropList = array[0..16379] of PPropInfo;
```

Массив TPropList хранит указатели на данные TPropInfo каждого свойства. Тип TPropInfo определен в модуле TypeInfo.pas следующим образом:

```
PPropInfo = ^TPropInfo;
  TPropInfo = packed record
    PropType: PPTypeInfo;
    GetProc: Pointer;
    SetProc: Pointer;
    StoredProc: Pointer;
    Index: Integer;
    Default: Longint;
    NameIndex: SmallInt;
    Name: ShortString;
  end;
```

Поле TPropInfo и является информацией о типе свойства.

Процедура GetClassProperties() использует функцию GetPropInfos() для заполнения массива указателями на информацию RTTI обо всех свойствах данного объекта. Затем, перебирая в цикле все элементы массива, считывает имена и типы свойств из соответствующих данных RTTI. Обратите внимание на следующую строку:

```
if not (PropList[i]^.PropType^.Kind = tkMethod) then
```

Таким образом отфильтровываются свойства, являющиеся событиями (указатели на методы). Эти свойства будут добавлены в соответствующий список позднее, попутно продемонстрировав альтернативный метод получения информации RTTI о свойстве. Чтобы получить массив TPropList для свойств конкретного типа, в заключительной части метода GetClassProperties() используется функция GetProp-

`List()`. В данном случае представляют интерес лишь свойства типа `tkMethod`. Функция `GetPropList()` также определена в модуле `TypeInfo.pas` (обратите внимание на комментарии в исходном коде).

СОВЕТ

Используйте функцию `GetPropInfos()`, если хотите получить указатель на информацию RTTI обо *всех* свойствах данного объекта. Для получения информации о свойствах определенного типа используйте функцию `GetPropList()`.

На рис. 10.3 изображена главная форма с информацией о типах времени выполнения.

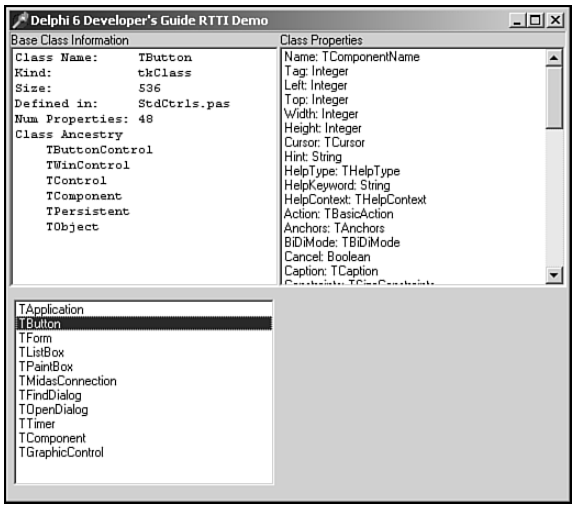

Рис. 10.3. *Главная форма с информацией о типах*

Проверка наличия у объекта определенного свойства

Ранее была поставлена задача проверки существования определенного свойства у данного объекта. Тогда речь шла о свойстве `DataSource`. Используя функции, определенные в модуле `TypInfo.pas`, можно написать функцию проверки того, предназначен ли элемент управления для работы с данными:

```
function IsDataAware(AComponent: TComponent): Boolean;
var
  PropInfo: PPropInfo;
begin
  // Искать свойство по имени DataSource
  PropInfo := GetPropInfo(AComponent.ClassInfo, 'DataSource');
  Result := PropInfo <> nil;

  // Двойная проверка, чтобы удостовериться в
  // происхождении от класса TDataSource.
```

```
  if Result then
    if not ((PropInfo^.Proptype^.Kind = tkClass) and (GetTypeData(
      PropInfo^.PropType^).ClassType.InheritsFrom(TDataSource)))
      then
      Result := False;
end;
```

Здесь используется функция GetPropInfo(), возвращающая указатель TPropInfo на данное свойство. Эта функция возвращает значение nil, если свойство не существует. Для дополнительной проверки необходимо удостовериться, что свойство DataSource действительно происходит от класса TDataSource.

Эту функцию можно усовершенствовать так, чтобы она проверяла наличие свойства с любым заданным именем:

```
function HasProperty(AComponent: TComponent;
                     APropertyName: String): Boolean;
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComponent.ClassInfo, APropertyName);
  Result := PropInfo <> nil;
end;
```

Обратите внимание, что такой подход применим только к тем свойствам, которые объявлены как published. Для непубликуемых свойств данные RTTI отсутствуют.

Получение информации о типах указателей на методы

Существует возможность получить информацию RTTI и для указателей на методы. Например, можно вернуть тип метода (процедуры, функции и т.п.) и его параметры. В листинге 10.4 показано, как получить информацию RTTI для выбранной группы методов.

Листинг 10.4. Получение информации RTTI для методов

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, DBClient, MidasCon, MConnect;

type

  TMainForm = class(TForm)
    lbSampMethods: TListBox;
    lbMethodInfo: TMemo;
    lblBasicMethodInfo: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure lbSampMethodsClick(Sender: TObject);
```

```
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo, DBTables, Provider;

{$R *.DFM}

type
  // Необходимо переопределить эту запись, так как в typinfo.pas
  // она закомментирована.

  PParamRecord = ^TParamRecord;
  TParamRecord = record
    Flags:     TParamFlags;
    ParamName: ShortString;
    TypeName:  ShortString;
  end;

procedure GetBaseMethodInfo(ATypeInfo: PTypeInfo;
                            AStrings: TStrings);
{ Этот метод получает часть основных данных RTTI из
  структуры TTypeInfo и добавляет их в параметр AStrings. }
var
  MethodTypeData: PTypeData;
  EnumName: String;
begin
  MethodTypeData := GetTypeData(ATypeInfo);
  with AStrings do begin
    Add(Format('Class Name:     %s', [ATypeInfo^.Name]));
    EnumName := GetEnumName(TypeInfo(TTypeKind),
                            Integer(ATypeInfo^.Kind));
    Add(Format('Kind:           %s', [EnumName]));
    Add(Format('Num Parameters: %d',[MethodTypeData.ParamCount]));
  end;
end;

procedure GetMethodDefinition(ATypeInfo: PTypeInfo;
                              AStrings: TStrings);
{ Этот метод позволяет получить информацию об указателе метода.
  Используем эту информацию для реконструкции определения метода.}
var
  MethodTypeData: PTypeData;
  MethodDefine:   String;
  ParamRecord:    PParamRecord;
  TypeStr:        ^ShortString;
  ReturnStr:      ^ShortString;
```

```
  i: integer;
begin
  MethodTypeData := GetTypeData(ATypeInfo);

  // Определение типа метода
  case MethodTypeData.MethodKind of
    mkProcedure:      MethodDefine := 'procedure ';
    mkFunction:       MethodDefine := 'function ';
    mkConstructor:    MethodDefine := 'constructor ';
    mkDestructor:     MethodDefine := 'destructor ';
    mkClassProcedure: MethodDefine := 'class procedure ';
    mkClassFunction:  MethodDefine := 'class function ';
  end;

  // Указатель на первый параметр
  ParamRecord    := @MethodTypeData.ParamList;
  i := 1; // первый параметр

  // Перебрать параметры метода и, если они определены правильно,
  // добавить их в список строк.
  while i <= MethodTypeData.ParamCount do begin
    if i = 1 then
      MethodDefine := MethodDefine+'(';
    if pfVar in ParamRecord.Flags then
      MethodDefine := MethodDefine+('var ');
    if pfconst in ParamRecord.Flags then
      MethodDefine := MethodDefine+('const ');
    if pfArray in ParamRecord.Flags then
      MethodDefine := MethodDefine+('array of ');
// Не будем изменять pfAddress, а лишь убедимся в том, что
// параметр Self передается с установленным флагом.
{
    if pfAddress in ParamRecord.Flags then
      MethodDefine := MethodDefine+('*address* ');
}
    if pfout in ParamRecord.Flags then
      MethodDefine := MethodDefine+('out ');

    // Использовать арифметику указателей для получения строки
    // типа параметра.
    TypeStr := Pointer(Integer(@ParamRecord^.ParamName) +
               Length(ParamRecord^.ParamName)+1);

    MethodDefine := Format('%s%s: %s', [MethodDefine,
                           ParamRecord^.ParamName, TypeStr^]);

    inc(i); // Увеличить счетчик.

    // Переход к следующему параметру. Обратите внимание на
    // использование арифметических операций над указателями для
    // расчета положения следующего параметра.
    ParamRecord := PParamRecord(Integer(ParamRecord) +
                   SizeOf(TParamFlags) +
```

```
                    (Length(ParamRecord^.ParamName) + 1) +
                    (Length(TypeStr^)+1));

    // Если параметры все еще есть, то продолжать
    if i <= MethodTypeData.ParamCount then begin
      MethodDefine := MethodDefine + '; ';
    end else
      MethodDefine := MethodDefine + ')';
  end;

  // Если метод является функцией, то он должен возвращать
  // значение, которое также помещается в строку определения
  // метода. Возвращаемое значение следует за последним
  // параметром.
  if MethodTypeData.MethodKind = mkFunction then begin
    ReturnStr := Pointer(ParamRecord);
    MethodDefine := Format('%s: %s;', [MethodDefine, ReturnStr^])
  end else
    MethodDefine := MethodDefine+';';

  // И, наконец, добавить строку в список.
  with AStrings do begin
    Add(MethodDefine)
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Добавить ряд типов методов в список, а также сохранить
    указатель на данные RTTI в массиве списка Objects }
  with lbSampMethods.Items do begin
    AddObject('TNotifyEvent', TypeInfo(TNotifyEvent));
    AddObject('TMouseEvent', TypeInfo(TMouseEvent));
    AddObject('TBDECallBackEvent', TypeInfo(TBDECallBackEvent));
    AddObject('TDataRequestEvent', TypeInfo(TDataRequestEvent));
    AddObject('TGetModuleProc', TypeInfo(TGetModuleProc));
    AddObject('TReaderError', TypeInfo(TReaderError));
  end;
end;

procedure TMainForm.lbSampMethodsClick(Sender: TObject);
begin
  lbMethodInfo.Lines.Clear;
  with lbSampMethods do begin
    GetBaseMethodInfo(PTypeInfo(Items.Objects[ItemIndex]),
                      lbMethodInfo.Lines);
    GetMethodDefinition(PTypeInfo(Items.Objects[ItemIndex]),
                        lbMethodInfo.Lines);
  end;
end;

end.
```

Как видно из листинга 10.4, в список lbSampMethods помещены имена методов использованного объекта. Указатели на RTTI этих методов находятся в массиве списка Objects. Они получены с помощью функции TypeInfo(), предназначенной для поиска указателя на информацию о типах времени выполнения для заданного идентификатора типа. Если пользователь выбирает один из этих методов, данные RTTI из массива Objects используются для реконструкции определения метода. Более подробная информация по этой теме приведена в комментариях листинга.

СОВЕТ

Для получения указателя на генерируемую компилятором информацию RTTI для определенного идентификатора типа применяйте функцию TypeInfo(). Например, следующая строка позволяет получить указатель на информацию RTTI для типа TButton:

```
TypeInfoPointer := TypeInfo(TButton);
```

Получение информации о перечислимых типах

Итак, наиболее сложная часть RTTI уже позади. Для более простых типов также можно получить информацию о типах времени выполнения. В следующих разделах показано, как получить данные RTTI для целого и перечислимого типов, а также для множества.

Информация RTTI о целочисленных типах

Получить информацию RTTI о целочисленных типах очень просто. Листинг 10.5 иллюстрирует этот процесс.

Листинг 10.5. Получение информации RTTI о целочисленных типах

```
procedure TMainForm.lbSampsClick(Sender: TObject);
var
  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;
  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
begin
  memInfo.Lines.Clear;
  with lbSamps do begin

    // Получить указатель на структуру TTypeInfo
    OrdTypeInfo := PTypeInfo(Items.Objects[ItemIndex]);
    // Получить указатель на структуру TTypeData
    OrdTypeData := GetTypeData(OrdTypeInfo);

    // Получить строку по имени типа
    TypeNameStr := OrdTypeInfo.Name;
```

```
    // Получить строку с разновидностью типа
    TypeKindStr := GetEnumName(TypeInfo(TTypeKind),
                               Integer(OrdTypeInfo^.Kind));

    // Получить минимальные и максимальные значения для типа
    MinVal := OrdTypeData^.MinValue;
    MaxVal := OrdTypeData^.MaxValue;

    // Добавить информацию в поле memo
    with memInfo.Lines do begin
      Add('Type Name: '+TypeNameStr);
      Add('Type Kind: '+TypeKindStr);
      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));
    end;
  end;
end;
```

Здесь функция `TypeInfo()` используется для получения указателя на структуру `TTypeInfo` для типа данных `Integer`. Затем этот указатель передается функции `GetTypeData()` для получения указателя на структуру `TTypeData`. Обе эти структуры используются для помещения в список информации RTTI о целочисленном типе. Более подробный демонстрационный вариант `IntegerRTTI.dpr` можно найти на прилагаемом CD, в каталоге, относящемся к данной главе.

Информация RTTI о перечислимых типах

Получить информацию RTTI о перечислимых типах также несложно. Как видите, листинг 10.6 практически идентичен листингу 10.5, за исключением дополнительного цикла `for`, отображающего значения перечислимого типа.

Листинг 10.6. Получение информации RTTI о перечислимом типе

```
procedure TMainForm.lbSampsClick(Sender: TObject);
var
  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;
  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
  i: integer;
begin
  memInfo.Lines.Clear;
  with lbSamps do begin

    // Получить указатель на структуру TTypeInfo
    OrdTypeInfo := PTypeInfo(Items.Objects[ItemIndex]);
    // Получить указатель на структуру TTypeData
    OrdTypeData := GetTypeData(OrdTypeInfo);

    // Получить строку по имени типа
```

```
    TypeNameStr := OrdTypeInfo.Name;
    // Получить строку с разновидностью типа
    TypeKindStr := GetEnumName(TypeInfo(TTypeKind),
                               Integer(OrdTypeInfo^.Kind));

    // Получить минимальные и максимальные значения для типа
    MinVal := OrdTypeData^.MinValue;
    MaxVal := OrdTypeData^.MaxValue;

    // Добавить информацию в поле memo
    with memInfo.Lines do begin
      Add('Type Name: '+TypeNameStr);
      Add('Type Kind: '+TypeKindStr);
      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));
      // Отобразить значения и имена перечислимых типов
      if OrdTypeInfo^.Kind = tkEnumeration then
      for i := MinVal to MaxVal do
        Add(Format('  Value: %d   Name: %s',
            [i, GetEnumName(OrdTypeInfo, i)]));
    end;
  end;
end;
```

Более подробный демонстрационный вариант `EnumRTTI.dpr` можно найти на прилагаемом CD, в каталоге, относящемся к данной главе.

Информация RTTI о множествах

Получение информации RTTI о множестве ненамного сложнее предыдущих технологий. В листинге 10.7 представлен программный код, обеспечивающий функционирование главной формы проекта `SetRTTI.dpr`, который можно найти на прилагаемом CD, в каталоге, относящемся к данной главе.

Листинг 10.7. Получение информации RTTI о типах множеств

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Grids;

type
  TMainForm = class(TForm)
    lbSamps: TListBox;
    memInfo: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure lbSampsClick(Sender: TObject);
  private
```

```
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo, Buttons;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Добавить несколько примеров перечислимых типов
  with lbSamps.Items do begin
    AddObject('TBorderIcons', TypeInfo(TBorderIcons));
    AddObject('TGridOptions', TypeInfo(TGridOptions));
  end;
end;

procedure GetTypeInfoForOrdinal(AOrdTypeInfo: PTypeInfo;
                                AStrings: TStrings);
var
  // OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;
  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
  i: integer;
begin

  // Получить указатель на структуру TTypeData
  OrdTypeData := GetTypeData(AOrdTypeInfo);

  // Получить строку по имени типа
  TypeNameStr := AOrdTypeInfo.Name;
  // Получить строку с разновидностью типа
  TypeKindStr := GetEnumName(TypeInfo(TTypeKind),
                 Integer(AOrdTypeInfo^.Kind));

  // Получить минимальные и максимальные значения для типа
  MinVal := OrdTypeData^.MinValue;
  MaxVal := OrdTypeData^.MaxValue;

  // Добавить информацию в поле memo
  with AStrings do begin
    Add('Type Name: '+TypeNameStr);
    Add('Type Kind: '+TypeKindStr);

    // Рекурсивный вызов функции для отображения перечислимых
    // значений множества.
```

```
    if AOrdTypeInfo^.Kind = tkSet then begin
      Add('==========');
      Add('');
      GetTypeInfoForOrdinal(OrdTypeData^.CompType^, AStrings);
    end;

    // Отображение значения и имени перечислимых типов,
    // принадлежащих множеству.
    if AOrdTypeInfo^.Kind = tkEnumeration then begin
      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));
      for i := MinVal to MaxVal do
        Add(Format('  Value: %d   Name: %s',
            [i, GetEnumName(AOrdTypeInfo, i)]));
    end;
  end;
end;

procedure TMainForm.lbSampsClick(Sender: TObject);
begin
  memInfo.Lines.Clear;
  with lbSamps do
    GetTypeInfoForOrdinal(PTypeInfo(Items.Objects[ItemIndex]),
                          memInfo.Lines);
end;

end.
```

В список этой демонстрационной программы было помещено два типа множеств. Указатель на структуры `TTypeInfo` для этих двух типов был помещен в массив `Objects` с помощью функции `TypeInfo()`. Когда пользователь выбирает из списка один из этих элементов, вызывается процедура `GetTypeInfoForOrdinal()`, которой передаются в качестве параметров как указатель типа `PTypeInfo`, так и свойство `memInfo.Lines` для заполнения данными RTTI.

Процедура `GetTypeInfoForOrdinal()` выполняет стандартные действия для получения указателя на структуру `TTypeData`, принадлежащую заданному типу. Начальная информация о типе хранится в параметре типа `TStrings`, а затем происходит рекурсивный вызов той же процедуры `GetTypeInfoForOrdinal()` с передачей в качестве первого параметра значения `OrdTypeData^.CompType^` — указателя на перечислимый тип множества. Получаемые при этом данные RTTI аналогично добавляются в то же самое свойство типа `TStrings`.

Присвоение значений свойствам с помощью RTTI

Ознакомившись со способами поиска и определения свойств компонентов, будет весьма полезно узнать, как с помощью информации RTTI присвоить этим свойствам значения. Такая задача довольно проста. В модуле `TypInfo.pas` содержится много вспомогательных процедур, позволяющих опрашивать и устанавливать публикуемые (`public`) свойства компонентов. Речь идет о тех же самых вспомогательных процедурах, которые используются интегрированной средой разработки Delphi (IDE) (в окне

Object Inspector). Было бы очень полезно открыть модуль TypInfo.pas и ознакомиться с этими процедурами. Некоторые из них приведены ниже.

Предположим, что необходимо присвоить целочисленное значение некоторому свойству данного компонента. При этом допустим, что неизвестно, существует ли такое свойство в рассматриваемом компоненте. Вот как выглядит процедура присваивания целочисленного значения свойству компонента, при условии, что это свойство существует:

```
procedure SetIntegerPropertyIfExists(AComp: TComponent;
                                     APropName: String;
                                     AValue: Integer);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComp.ClassInfo, APropName);
  if PropInfo <> nil then begin
    if PropInfo^.PropType^.Kind = tkInteger then
      SetOrdProp(AComp, PropInfo, Integer(AValue));
  end;
end;
```

Данная процедура принимает три параметра. Первый параметр, AComp, представляет собой компонент, свойство которого следует модифицировать. Второй параметр, APropName, — это имя свойства, которому необходимо присвоить значение третьего параметра, AValue. В данной процедуре используется функция GetPropInfo(), предназначенная для считывания указателя TPropInfo на заданное свойство. Если этого свойства не существует, то функция GetPropInfo() вернет значение nil. Если же свойство существует, то с помощью второго оператора if определяется корректность типа заданного свойства. Тип свойства tkInteger определен в модуле TypInfo.pas вместе с другими возможными типами свойств, как показано ниже:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
             tkString, tkSet, tkClass, tkMethod, tkWChar,
             tkLString, tkWString, tkVariant, tkArray, tkRecord,
             tkInterface, tkInt64, tkDynArray);
```

Наконец, присвоение значения свойству выполняется с использованием еще одной процедуры — SetOrdProp() — из модуля TypInfo.pas, которая как раз и предназначена для установки значений свойств перечислимого типа. Обращение к этой процедуре может выглядеть примерно следующим образом:

```
SetIntegerPropertyIfExists(Button2, 'Width', 50);
```

Метод SetOrdProp() называют *методом установки* (setter method), поскольку он устанавливает заданное свойство равным заданному значению. Существует также и метод *чтения значения* свойства (getter method). В модуле TypInfo.pas есть несколько таких вспомогательных процедур с соответствующими именами (Set*XXX*Prop()), предусмотренных для всех возможных типов свойств (табл. 10.7).

Таблица 10.7. Методы чтения и установки свойств

Тип свойства	*Метод установки*	*Метод чтения*
Порядковый	`SetOrdProp()`	`GetOrdProp()`
Перечислимый	`SetEnumProp()`	`GetEnumProp()`
Объект	`SetObjectProp()`	`GetObjectProp()`
Строка	`SetStrProp()`	`GetStrProp()`
С плавающей запятой	`SetFloatProp()`	`GetFloatProp()`
Вариант	`SetVariantProp()`	`GetVariantProp()`
Метод (Событие)	`SetMethodProp()`	`GetMethodProp()`
`Int64`	`SetInt64Prop()`	`GetInt64Prop()`

И опять обращаю внимание на то, что в модуле `TypInfo.pas` содержится много других вспомогательных методов, которые могут оказаться очень полезными.

В следующем фрагменте показано, как присваивается значение свойству объекта:

```
procedure SetObjectPropertyIfExists(AComponent: TComponent;
                                    APropName: String;
                                    AValue: TObject);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComponent.ClassInfo, APropName);
  if PropInfo <> nil then begin
    if PropInfo^.PropType^.Kind = tkClass then
      SetObjectProp(AComponent, PropInfo, AValue);
  end;
end;
```

Этот метод можно вызвать следующим образом:

```
var
  F: TFont;
begin
  F := TFont.Create;
  F.Name   := 'Arial';
  F.Size   := 24;
  F.Color  := clRed;
  SetObjectPropertyIfExists(Panel1, 'Font', F);
end;
```

А вот как можно присвоить значение свойству метода:

```
procedure SetMethodPropertyIfExists(AComp: TComponent;
                                    APropName: String;
                                    AMethod: TMethod);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComp.ClassInfo, APropName);
```

```
  if PropInfo <> nil then begin
    if PropInfo^.PropType^.Kind = tkMethod then
      SetMethodProp(AComp, PropInfo, AMethod);
  end;
end;
```

Данный метод требует использования типа `TMethod`, который определен в модуле `SysUtils.pas`. Чтобы вызвать метод, назначающий обработчик события одного компонента другому, можно использовать метод `GetMethodProp()`, считывающий значение `TMethod` из исходного компонента:

```
SetMethodPropertyIfExists(Button5, 'OnClick',
                          GetMethodProp(Panel1, 'OnClick'));
```

На прилагаемом CD находится проект `SetProperties.dpr`, демонстрирующий использование этих методов.

Резюме

В настоящей главе обсуждались библиотека визуальных компонентов (VCL) и библиотека межплатформенных компонентов (CLX). Кроме того, была рассмотрена иерархия классов, а также характеристики компонентов различных уровней иерархии. К тому же подробно обсуждалась информация о типах времени выполнения (RTTI).

ГЛАВА

11

Разработка компонентов VCL

В ЭТОЙ ГЛАВЕ...

Способность Delphi 6 легко создавать специальные компоненты является одним из главных преимуществ этой среды разработки. Пользователи большинства других систем программирования вынуждены довольствоваться стандартными элементами управления Windows или же приобретать совершенно иной набор сложных элементов управления, разработанных сторонними производителями. Возможность включать в приложения Delphi свои собственные специальные компоненты означает полный контроль над пользовательским интерфейсом приложения. Применение пользовательских элементов управления оставляет последнее слово в вопросах внешнего вида приложения и производимого ею впечатления за разработчиком.

Во всех версиях Delphi, начиная с первой, существует возможность создавать компоненты для библиотеки VCL. Delphi 6 позволяет создавать компоненты и для библиотеки CLX, но более подробная информация по этой теме приведена в главе 13, "Разработка компонентов CLX".

Разработчики компонентов, безусловно, по достоинству оценят содержимое настоящей главы. Здесь подробно рассматриваются все аспекты разработки компонента — начиная от самой концепции и до установки его в интегрированную среду разработки Delphi. Описаны также проблемы, подстерегающие разработчика в процессе создания компонента, а также полезные советы, которые помогут их избежать.

Даже для тех, кто разрабатывает, в основном, не компоненты, а приложения, данная глава тоже содержит много полезной информации. Использовав в программе несколько собственных компонентов, можно придать приложению некую изюминку и повысить его продуктивность. Рано или поздно придется встретиться с ситуацией, когда ни один из стандартных компонентов не будет удовлетворять поставленным задачам. Тогда-то и пригодится возможность создавать собственные компоненты. Вполне возможно разработать компонент, который идеально соответствует поставленной задаче, а затем многократно использовать его в других программах.

Концепция разработки компонентов

Настоящий раздел содержит основные сведения, необходимые для разработки собственного компонента. Практическое применение этих сведений будет продемонстрировано на примерах создания некоторых полезных компонентов.

Решение о необходимости создания компонента

Зачем мучиться, создавая новый собственный компонент, когда можно воспользоваться существующими или слепить на скорую руку что-нибудь попроще из подсобных средств? Для разработки нового компонента существует несколько важных причин.

- Если необходимо разработать новый элемент пользовательского интерфейса и в дальнейшем применять его в разных приложениях.
- Если нужно сделать приложение устойчивым к ошибкам, разделив его логически на модули, построенные на основании объектно-ориентированных классов.
- Если среди существующих компонентов Delphi и элементов ActiveX нет такого, который полностью удовлетворял бы всем требованиям.

- Если существуют потенциальные пользователи создаваемого компонента и его можно распространить среди других программистов либо за деньги, либо ради собственного удовольствия.
- Если хочется глубже разобраться в Delphi, библиотеке VCL и функциях интерфейса API Win32.

Лучше всего учиться создавать пользовательские компоненты у тех, кто их изобрел. Исходный код библиотеки VCL — настоящая сокровищница знаний, и мы настоятельно рекомендуем изучить его, особенно тем, кто решил серьезно заняться разработкой компонентов. Исходный код библиотеки VCL присутствует в двух версиях Delphi: в Professional и Enterprise.

Многих пугает сама мысль о создании собственных компонентов, и, поверьте, совершенно напрасно. Легко это или тяжело — зависит только от вас. Конечно, существуют очень сложные компоненты, но, в принципе, создать полезный компонент довольно просто.

Этапы разработки компонента

Если проблема уже определена и намечен способ ее решения, основанный на использовании собственного компонента, то создание такого компонента — от концепции до ее воплощения — будет происходить в несколько этапов.

- Прежде всего нужна идея полезного компонента, отличного от уже существующих.
- Затем следует построить алгоритм, по которому компонент будет работать.
- Начните с подготовительных мероприятий, не бросайтесь сразу с головой в проектирование компонента. Спросите себя: “Что нужно сделать, чтобы заставить данный компонент функционировать?”.
- Мысленно разложите конструкцию компонента на независимые логические части. Это не только упростит и упорядочит создание компонента, но и позволит написать понятный программный код со стройной организацией. При разработке компонента имейте в виду, что, возможно, кто-то захочет создать на базе его класса производный компонент.
- Сначала проверьте компонент на пробном проекте. Если добавить его в палитру компонентов сразу, то вы, вероятно, пожалеете об этом.
- Наконец, добавьте компонент и его пиктограмму в палитру компонентов. После небольшой настройки он будет готов для переноса в приложения Delphi.

Вот шесть основных этапов создания компонента Delphi.

1. Выбор базового класса.
2. Создание модуля компонента.
3. Добавление в новый компонент свойств, методов и событий.
4. Проверка.
5. Регистрация компонента в среде Delphi.
6. Создание файла справки для компонента.

В настоящей главе рассматриваются лишь первые пять этапов, а создание файлов справки выходит за ее пределы. Но это не означает, что шестой этап менее важен, чем предыдущие. Рекомендуем изучить наиболее популярные средства создания файлов справки программ. Компания *Borland* также поместила необходимую информацию по данному вопросу в собственную интерактивную справочную систему. Обратитесь к разделу "Providing Help for Your Component" ("Создание справочной информации для пользовательского компонента") этой системы.

Выбор базового класса

В главе 10, "Архитектура компонентов: VCL и CLX", обсуждалась иерархия компонентов библиотеки VCL и назначение различных классов, расположенных на разных уровнях этой иерархии. Были описаны четыре базовых класса, от которых может происходить класс компонента: стандартные элементы управления, пользовательские элементы управления, визуальные элементы управления и невизуальные компоненты. Например, если достаточно просто расширить поведение существующего элемента Win32, скажем `TMemo`, то можно создать компонент, расширив возможности стандартного класса. Если необходимо создать абсолютно новый класс компонента, то придется иметь дело с пользовательским элементом управления. Компоненты на основе визуальных элементов применяются для создания визуальных эффектов, реализуемых без использования ресурсов Win32. И, наконец, если необходимо создать компонент, не имеющий визуального представления в форме готового приложения, но, тем не менее, присутствующий в окне инспектора объектов, то разработать придется невизуальный компонент. Различные классы библиотеки VCL представляют разные типы компонентов. Возможно, придется вернуться к главе 10, "Архитектура компонентов: VCL и CLX", если концепции понятны не до конца. В табл. 11.1 содержится лишь краткий перечень.

Таблица 11.1. Базовые классы компонентов библиотеки VCL

Класс VCL	*Тип пользовательских элементов управления*
`TObject`	Хотя классы, непосредственно происходящие от класса `TObject`, не являются компонентами, тем не менее они заслуживают внимания. Класс `TObject` используется как базовый для создания многих объектов, с которыми едва ли придется встречаться в процессе разработки. Например класс `TIniFile`
`TComponent`	Это исходная точка для многих невизуальных компонентов. Основное их достоинство — это возможность самостоятельно загружать и сохранять потоки данных в интегрированной среде Delphi во время разработки
`TGraphicControl`	Используйте данный класс для создания пользовательского компонента, не имеющего дескриптора окна. Потомки класса `TGraphicControl` отображаются в клиентской области своих родительских элементов, а поэтому не требуют использования системных ресурсов

Окончание табл. 11.1.

Класс VCL	*Тип пользовательских элементов управления*
`TWinControl`	Это базовый класс для создания всех компонентов, обладающих дескриптором окна. Данный класс содержит общие свойства и события оконных элементов управления
`TCustomControl`	Этот класс происходит от класса `TWinControl`. Он обладает свойством `Canvas` и содержит метод `Paint()`, предоставляя расширенный контроль над внешним видом компонента. Данный класс используется, в основном, для создания пользовательских оконных элементов
`TCustomClassName`	В библиотеке VCL содержится несколько классов, часть свойств которых не являются публикуемыми. Это позволяет разработчикам создать несколько пользовательских компонентов на базе одного и того же класса и в каждом из них публиковать только заранее определенные свойства, необходимые для данного конкретного класса
`TComponentName`	Некоторый существующий класс, подобный `TEdit`, `TPanel` или `TScrollBox`. Если необходимо расширить поведение какого-нибудь элемента управления, то в качестве базового класса для своего компонента используйте уже существующий, например `TEdit` или иной подходящий пользовательский класс, а не создавайте новый элемент "с нуля". Большинство создаваемых компонентов попадет именно в эту категорию

Очень важно понимать особенности различных классов и возможности существующих компонентов. В большинстве случаев оказывается, что существующие классы способны обеспечить бо́льшую часть функций, необходимых новому компоненту. Только досконально изучив возможности существующих компонентов, можно приступать к выбору базового класса для нового компонента. Эти сведения относятся к разряду знаний, приобретаемых лишь на практике, поэтому простого чтения литературы будет недостаточно. Придется приложить значительные усилия для исследования каждого компонента и класса библиотеки VCL Delphi. Единственный способ для этого — использовать компоненты библиотеки VCL в своих приложениях, пусть даже в чисто экспериментальных целях.[1]

Создание модуля компонента

Определившись с выбором базового класса создаваемого компонента, можно приступать к созданию модуля этого компонента. В ряде следующих разделов последовательно рассматриваются все этапы разработки нового компонента. Причем основное внимание будет уделено сущности этих этапов, а не конкретным функциям, поэтому

[1] *Наилучший способ научиться программированию – это начать программировать.* – Прим. ред.

возможности создаваемого для примера компонента будут минимальны, но вполне достаточны для иллюстрации этапов разработки.

Данный компонент под названием `TddgWorthless` будет потомком класса `TCustomControl`, а значит, будет обладать дескриптором окна и способностью к отображению. От базового класса `TCustomControl` этот компонент унаследовал несколько свойств, методов и событий.

Самый простой способ начать создание модуля нового компонента заключается в использовании эксперта компонентов (Component Expert), показанного на рис. 11.1.

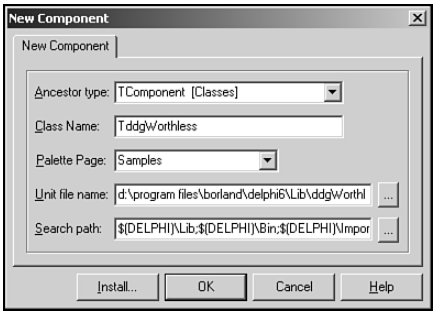

Рис. 11.1. *Эксперт компонентов*

Окно эксперта компонентов можно вывести на экран, выбрав в меню Component пункт New Component. Укажите в нем имя базового класса, имя класса компонента, вкладку палитры, на которой должен отображаться компонент, а также имя модуля компонента. Затем щелкните на кнопке OK — и Delphi автоматически создаст модуль компонента с готовым объявлением его типа и процедурой регистрации. В листинге 11.1 приведен модуль, созданный системными средствами Delphi.

Листинг 11.1. Модуль `Worthless.pas` — пример компонента Delphi

```
unit Worthless;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs;
type
  TddgWorthless = class(TCustomControl)
  private
    { Закрытые объявления }
  protected
    { Защищенные объявления }
  public
    { Открытые объявления }
  published
    { Публикуемые объявления }
  end;
procedure Register;
implementation
procedure Register;
```

```
begin
  RegisterComponents('DDG', [TddgWorthless]);
end;
end.
```

Как видно из примера, пока класс TddgWorthless — просто заготовка компонента. В следующих разделах в класс TddgWorthless будут добавлены свойства, методы и события.

Создание свойств

Применение свойств компонентов описано в главе 10, "Архитектура компонентов: VCL и CLX", а в настоящем разделе рассмотрим, как добавлять различные типы свойств к создаваемым компонентам.

Типы свойств

Большинство типов свойств описано в главе 10. Добавим в компонент TddgWorthless свойства каждого типа, чтобы проиллюстрировать различия между ними. Каждый тип в окне инспектора объектов редактируется по-разному. Более подробная информация об этих типах и способах их редактирования приведена в настоящей главе далее.

Добавление в компонент простых свойств

Под простыми свойствами следует понимать числа, строки и символы. Они могут непосредственно редактироваться пользователем в окне инспектора объектов и не требуют специальных методов доступа. В листинге 11.2 представлен вариант компонента TddgWorthless с тремя простыми свойствами.

Листинг 11.2. Простые свойства

```
TddgWorthless = class(TCustomControl)
  private
    // Внутренние данные
    FIntegerProp: Integer;
    FStringProp: String;
    FCharProp: Char;
  published
    // Простые типы свойств
    property IntegerProp: Integer read FIntegerProp
                                write FIntegerProp;
    property StringProp: String read FStringProp
                               write FStringProp;
    property CharProp: Char read FCharProp write FCharProp;
end;
```

Используемый здесь синтаксис должен уже быть знаком, поскольку он подробно описан в главе 10, "Архитектура компонентов: VCL и CLX". Здесь представлены внутренние данные компонента, объявленные в разделе private. Свойства, относящиеся к

этим полям, объявлены в разделе published; это означает, что при установке компонента в Delphi, такие свойства можно будет редактировать в окне инспектора объектов.

НА ЗАМЕТКУ

При создании компонентов используется соглашение об именовании, в соответствии с которым имена полей начинаются с буквы F, а имена компонентов и типов — с буквы T. Если соблюдать эти несложные соглашения, то разобраться в коде будет значительно проще.

Добавление в компонент перечислимых свойств

Определенные пользователем перечислимые и булевы свойства также можно редактировать в окне инспектора объектов. Для этого следует дважды щелкнуть на поле Value этого окна и выбрать подходящее значение свойства в раскрывающемся списке. В качестве примера можно привести свойство Align, присущее большинству визуальных компонентов. Для создания свойства перечислимого типа сначала нужно определить сам перечислимый тип, например следующим образом:

```
TEnumProp = (epZero, epOne, epTwo, epThree);
```

Затем следует определить внутреннее поле для хранения значения, задаваемого пользователем. В листинге 11.3 демонстрируются два перечислимых свойства компонента TddgWorthless.

Листинг 11.3. Свойства перечислимого типа

```
TddgWorthless = class(TCustomControl)
  private
    // Перечислимые типы данных
    FEnumProp: TEnumProp;
    FBooleanProp: Boolean;
  published
    property EnumProp: TEnumProp read FEnumProp write FEnumProp;
    property BooleanProp: Boolean read FBooleanProp
                                write FBooleanProp;
end;
```

Для простоты остальные свойства исключены из компонента. Если бы этот компонент был установлен, то его перечислимые свойства отобразились бы в окне инспектора объектов так, как показано на рис. 11.2.

Добавление в компонент свойства типа множества

Свойство типа множества при редактировании в окне инспектора объектов выглядит как множество, определенное в синтаксисе языка Pascal. Простейший способ его редактирования – развернуть свойство в окне инспектора объектов, в результате чего каждый его элемент станет отдельным логическим свойством. Для создания свойства типа множества необходимо в первую очередь определить его тип:

```
TSetPropOption = (poOne, poTwo, poThree, poFour, poFive);
TSetPropOptions = set of TSetPropOption;
```

Сначала определяется набор элементов множества соответствующего перечислимого типа `TSetPropOption`, а уже затем — само множество `TSetPropOptions`.

Теперь можно добавить свойство `TSetPropOptions` в компонент `TddgWorthless`:

```
TddgWorthless = class(TCustomControl)
  private
    FOptions: TSetPropOptions;
  published
    property Options: TSetPropOptions read Foptions
                                      write FOptions;
end;
```

На рис. 11.3 показано, как выглядит это свойство в развернутом виде в окне инспектора объектов.

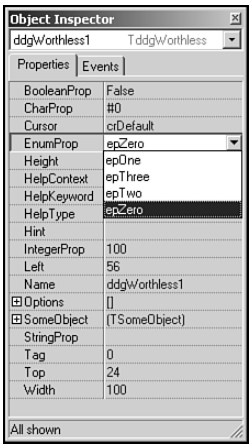

Рис. 11.2. *Отображение перечислимых свойств компонента `TddgWorthless` в окне инспектора объектов6*

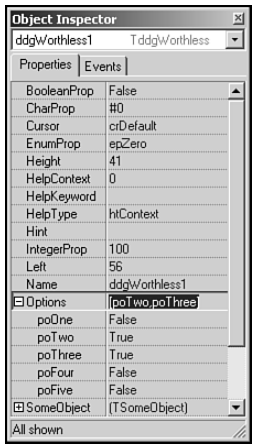

Рис. 11.3. *Представление свойства типа множества в окне инспектора объектов*

Добавление в компонент свойства-объекта

Свойствами могут являться объекты или другие компоненты. Например, у компонента `TShape` есть свойства-объекты `TBrush` и `TPen`. Когда свойство является объектом, оно может быть развернуто в окне инспектора объектов таким образом, чтобы его собственные свойства также могли быть модифицированы. Объектные свойства должны быть потомками класса `TPersistent`, для того чтобы их публикуемые свойства (т.е. свойства, объявленные в разделе `published`) могли быть записаны в поток данных и отображены в окне инспектора объектов.

Для определения объектного свойства компонента `TddgWorthless` необходимо сначала определить класс, который будет использован в качестве типа свойства. Объявление этого класса приведено в листинге 11.4.

Листинг 11.4. Определение `TSomeObject`

```
TSomeObject = class(TPersistent)
  private
    FProp1: Integer;
    FProp2: String;
  public
    procedure Assign(Source: TPersistent);
  published
    property Prop1: Integer read FProp1 write FProp1;
    property Prop2: String read FProp2 write FProp2;
end;
```

Класс `TSomeObject` является прямым потомком класса `TPersistent`, но в общем случае это необязательно. Если объект, от которого происходит новый класс, является потомком класса `TPersistent`, то этот объект также может быть использован в качестве свойства другого объекта.

В приведенном выше примере класс `TSomeObject`, используемый для создания объектного свойства, имеет два собственных простых свойства: `Prop1` и `Prop2`. В его состав также входит процедура `Assign()`, назначение которой поясняется ниже.

Теперь можно добавить в компонент `TddgWorthless` внутреннее поле типа `TSomeObject`. Так как это свойство представляет собой объект, его нужно создать. В противном случае, когда пользователь поместит в форму компонент `TddgWorthless` того экземпляра класса `TSomeObject`, который он мог бы редактировать, просто не будет существовать. В листинге 11.5 показано объявление компонента `TddgWorthless` с его новым объектным свойством.

Листинг 11.5. Добавление свойств-объектов

```
TddgWorthless = class(TCustomControl)
private
  FSomeObject: TSomeObject;
  procedure SetSomeObject(Value: TSomeObject);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
published
  property SomeObject: TSomeObject read FSomeObject
                                   write SetSomeObject;
end;
```

Обратите внимание на то, что в код включены переопределенные конструктор `Create()` и деструктор `Destroy()`. Кроме того, объявлен метод доступа `SetSomeObject()`, предназначенный для записи свойства `SomeObject`. Метод доступа "для записи" часто называют просто *методом записи* (writer method) (или *методом установки* (setter)), а метод доступа "для чтения" — *методом чтения* (reader) (или *методом выборки* (getter method)). Как указывалось в главе 10, "Архитектура компонентов: VCL и CLX",

методы записи должны иметь один параметр такого же типа, что и свойство, которому они принадлежат. Существует соглашение начинать имена методов записи со слова `Set`.

Конструктор `TddgWorthless.Create()` определяем следующим образом:

```
constructor TddgWorthless.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSomeObject := TSomeObject.Create;
end;
```

Здесь вначале был вызван унаследованный конструктор `Create()`, а затем создан экземпляр класса `TSomeObject`. Поскольку конструктор `Create()` вызывается как во время разработки, когда пользователь помещает компонент в форму, так и при выполнении приложения, то можно быть уверенным, что объект `FSomeObject` всегда действителен.

Кроме того, необходимо переопределить деструктор `Destroy()`, который перед освобождением компонента `TddgWorthless` должен предварительно освободить объект `TSomeObject`. Вот код переопределенного деструктора:

```
destructor TddgWorthless.Destroy;
begin
  FSomeObject.Free;
  inherited Destroy;
end;
```

Теперь, изучив принципы создания экземпляра объекта `TSomeObject`, рассмотрим, что произойдет при выполнении следующего программного кода:

```
var
  MySomeObject: TSomeObject;
begin
  MySomeObject := TSomeObject.Create;
  ddgWorthless.SomeObjectj := MySomeObject;
end;
```

Если свойство `TddgWorthless.SomeObject` было бы определено без метода записи, подобного приведенному ниже, то при попытке пользователя присвоить полю `SomeObject` его собственный объект, прежний экземпляр объекта, на который ссылался `FSomeObject`, был бы потерян:

```
property SomeObject: TSomeObject read FSomeObject
                                 write FSomeObject;
```

Напомним, что экземпляры объектов — это на самом деле указатели, ссылающиеся на реальный объект (см. главу 2, "Язык программирования Object Pascal"). Выполняя присвоение, как в предыдущем примере, будет получен указатель на другой экземпляр объекта, в то время как предыдущий экземпляр никуда из памяти не исчезает[2]. При разработке компонентов обычно пытаются исключить необходимость строгого соблюдения пользователями формальных правил при доступе к свойствам. Чтобы избежать этого подводного камня и защитить новый компонент от неумелого использования, для объектных свойств создают методы доступа. Указанное гарантирует, что

[2] *Речь идет о явлении утечки памяти.* – Прим. ред.

при присвоении таким свойствам новых значений системные ресурсы не будут потеряны. Метод доступа к объекту SomeObject предназначен как раз для этого:

```
procedure TddgWorthLess.SetSomeObject(Value: TSomeObject);
begin
  if Assigned(Value) then
    FSomeObject.Assign(Value);
end;
```

Метод SetSomeObject() вызывает метод FSomeObject.Assign(), передавая ему указатель на новый объект TSomeObject. Вот как выглядит реализация метода TSomeObject.Assign():

```
procedure TSomeObject.Assign(Source: TPersistent);
begin
  if Source is TSomeObject then begin
    FProp1 := TSomeObject(Source).Prop1;
    FProp2 := TSomeObject(Source).Prop2;
    inherited Assign(Source);
  end;
end;
```

В методе TSomeObject.Assign() вначале выполняется проверка того, действительно ли пользователь в качестве параметра передал экземпляр объекта TSomeObject. Если это так, то из параметра Source копируются соответствующие значения свойств. Это еще один способ присвоения значений объектов другим объектам, используемым в библиотеке VCL. Реализуя метод Assign() для нового компонента, имеет смысл просмотреть реализацию данного метода в различных классах исходного кода библиотеки VCL — это может подсказать подходящую идею.

СОВЕТ

Никогда не присваивайте свойству значение внутри его собственного метода записи. Рассмотрим, например, следующее объявление свойства:

```
property SomeProp: integer read FSomeProp write SetSomeProp;
 ....
procedure SetSomeProp(Value:integer);
begin
  SomeProp := Value;  // Это приведет к бесконечной рекурсии
end;
```

Поскольку доступ осуществляется к самому свойству, а не ко внутреннему полю, происходит повторный вызов метода SetSomeProp(), приводящий к бесконечному повторению рекурсивных вызовов. В конце концов программа аварийно завершается из-за переполнения стека. Поэтому в методах записи свойств всегда работайте только с внутренними полями!

Добавление в компонент свойства-массива

Доступ к некоторым свойствам можно осуществлять так, как если бы они представляли собой массивы. То есть они содержат список элементов, к каждому из которых можно обращаться по значению его индекса. Такими элементами могут быть любые типы объектов. Примерами являются, в частности, свойства TScreen.Fonts,

TMemo.Lines и TDBGrid.Columns. Эти свойства требуют наличия собственных редакторов свойств. Более подробная информация о создании редакторов свойств приведена в главе 12, "Создание расширенного компонента VCL". Сейчас же рассмотрим, как определить свойство, которое может быть индексировано как массив, но, тем не менее, совсем не содержит списков. Оставим пока в стороне компонент TddgWorthless и обратимся к новому компоненту TddgPlanets. Он содержит два свойства: PlanetName и PlanetPosition. Свойство PlanetName — это массив, возвращающий название планеты по заданному значению целочисленного индекса (номеру). Массив PlanetPosition использует не целочисленный, а строковый индекс. Если строка представляет собой название существующей планеты, то массив PlanetPosition возвратит положение планеты в Солнечной системе.

Например, выполнение следующего оператора с использованием свойства TddgPlanets.PlanetName приведет к отображению строки "Neptune":

```
ShowMessage(ddgPlanets.PlanetName[8]);
```

Сравните это выражение с обычным оператором, выводящим на экран сообщение From the sun, Neptune is planet number: 8 ("Нептун — 8-я планета Солнечной системы"):

```
ShowMessage('From the sun, Neptune is planet number: ' +
            IntToStr(ddgPlanets.PlanetPosition['Neptune']));
```

Прежде чем рассмотреть исходный код данного компонента, приведем основные характеристики свойств-массивов, отличающие их от других, уже рассмотренных свойств.

- Свойства-массивы объявляются с использованием одного или нескольких индексных параметров. Тип индексов должен быть простым (это может быть целое число или строка, но не запись и не класс).
- Подпрограммы доступа к свойству (read и write) должны быть методами. Они не могут быть внутренними полями компонента.
- Если в определении свойства-массива используется несколько индексов (т.е. свойство представлено многомерным массивом), методы доступа должны содержать параметры для каждого индекса в том же порядке, что и в свойстве.

Программный код класса TddgPlanets приведен в листинге 11.6.

Листинг 11.6. Класс TddgPlanets, иллюстрирующий создание свойств-массивов

```
unit planets;

interface

uses
  Classes, SysUtils;

type
  TddgPlanets = class(TComponent)
  private
    // Методы доступа к свойству-массиву
    function GetPlanetName(const AIndex: Integer): String;
```

```
    function GetPlanetPosition(const APlanetName:
                               String): Integer;
  public
    { Массив индексирован целым значением. Этот вариант
      используется для индексации массива по умолчанию. }
    property PlanetName[const AIndex:
                        Integer]: String read GetPlanetName; default;
    // Массив индексирован строковым значением
    property PlanetPosition[const APlantetName:
                         String]: Integer read GetPlanetPosition;
  end;

implementation

const
  // Объявление массива-константы с названиями планет
  PlanetNames: array[1..9] of String[7] = ('Mercury', 'Venus',
                          'Earth', 'Mars', 'Jupiter', 'Saturn',
                          'Uranus', 'Neptune', 'Pluto');

function TddgPlanets.GetPlanetName(const AIndex: Integer): String;
{ Возвращает название планеты, указанной индексом. Если индекс
выходит за пределы допустимого диапазона, передается исключение. }
begin
  if (AIndex < 0) or (AIndex > 9) then
    raise Exception.Create('Wrong number, enter a number 1-9')
  else
    Result := PlanetNames[AIndex];
end;

function TddgPlanets.GetPlanetPosition(const APlanetName:
                                       String): Integer;
var
  i: integer;
begin
  Result := 0;
  i := 0;
  { Сравнивает PName с названием каждой планеты и возвращает
    индекс подходящего значения массива-константы. При
    несовпадении возвращает нуль. }
  repeat
    inc(i);
  until (i = 10) or (CompareStr(UpperCase(APlanetName),
                      UpperCase(PlanetNames[i])) = 0);

  if i <> 10 then // Имя планеты найдено
    Result := i;
end;

end.
```

Этот компонент может подсказать идею создания свойств-массивов с целой или строковой переменной, используемой в качестве индекса. При обращении к свойству чтения значение возвращает функция, в отличие от обращения к другим свойствам, возвращающим значение внутреннего поля непосредственно. Комментарии в коде помогут лучше понять схему построения компонента.

Значения по умолчанию

Свойству можно назначить *значение по умолчанию*, или *стандартное значение* (default value), выполнив соответствующее присваивание в конструкторе компонента. Таким образом, если добавить следующие операторы в конструктор компонента `TddgWorthless`, то значение его свойства `FIntegerProp` при помещении компонента в форму всегда будет равно `100`:

```
FIntegerProp := 100;
```

Теперь настало время рассмотреть директивы `Default` и `NoDefault`, присутствующие в объявлениях свойств. Если обратиться к исходному коду библиотеки VCL, то можно заметить, что некоторые объявления свойств содержат директиву `Default`, как и в случае свойства `TComponent.Ftag`:

```
property Tag: Longint read FTag write FTag default 0;
```

Не путайте действие этого оператора с присвоением значения по умолчанию в конструкторе. Например, измените объявление свойства `IntegerProp` компонента `TddgWorthless` следующим образом:

```
property IntegerProp: Integer read FIntegerProp
                              write FIntegerProp default 100;
```

Этот оператор не присваивает свойству значение `100`. Здесь просто определяется, будет ли сохранено значение свойства при сохранении формы, содержащей компонент `TddgWorthless`. Если значение свойства `IntegerProp` не равно `100`, то оно будет сохранено в файле `.DFM`. В противном случае оно не будет сохранено, так как `100` — это значение, которое присваивается данному свойству заново создаваемого объекта перед считыванием его свойств из потока данных. Для ускорения загрузки форм рекомендуется везде, где только это возможно, использовать директиву `Default`. Важно понять, что директива `Default` не присваивает значение свойству, — это необходимо сделать в конструкторе компонента, как было показано выше.

Директива `NoDefault` используется для переобъявления свойства (со значением по умолчанию) таким образом, чтобы оно, независимо от исходного значения, всегда записывалось в поток данных. Например, можно переобъявить компонент так, чтобы для свойства `Tag` вообще не задавалось значения по умолчанию:

```
TSample = class(TComponent) published property Tag NoDefault;
```

Не следует без особой необходимости применять директиву `NoDefault`. Например, свойство `TForm.PixelsPerInch` всегда должно сохраняться для правильного отображения формы при выполнении программы. Необходимо также отметить, что для свойств строкового, вещественного (с плавающей точкой) и типа `int64` значения по умолчанию не могут быть объявлены.

Для изменения значения свойства по умолчанию достаточно переопределить свойство новым значением (но без методов чтения и записи).

Свойство-массив по умолчанию

Вполне возможно объявить свойство-массив таким образом, что оно будет стандартным свойством того компонента, которому оно принадлежит. Это позволит обращаться с экземпляром объекта так, как если бы тот был переменной типа массива. Например, в компоненте `TddgPlanets` объявлено свойство `TddgPlanets.PlanetName` с ключевым словом `Default`. Благодаря этому пользователь компонента может не применять имя свойства `PlanetName` для получения соответствующего значения. Достаточно просто поместить индекс возле идентификатора объекта. Две следующие строки кода приводят к одному и тому же результату:

```
ShowMessage(ddgPlanets.PlanetName[8]);
ShowMessage(ddgPlanets[8]);
```

У объекта может быть только одно стандартное свойство-массив, и потомки данного объекта не могут его переопределять.

Создание событий

В главе 10, "Архитектура компонентов: VCL и CLX", события были описаны как специальные свойства, связанные с кодом, выполняющимся при совершении определенных действий. В данном разделе события рассматриваются более детально. Здесь описано, каким образом возникают события и как можно определить собственные свойства-события в разрабатываемом компоненте.

Происхождение событий

В самом общем виде *событие* (event) можно определить как любое происшествие, вызванное вмешательством пользователя, операционной системы или логикой программы. Событие может быть связано с некоторым программным кодом, отвечающим на это происшествие. Совокупность события и кода, выполняющегося в ответ на это событие, называется *свойством-событием* (event property) и реализуется в виде указателя на некоторый метод. Метод, на который указывает это свойство-событие, называется *обработчиком события* (event handler).

Например, когда пользователь щелкает кнопкой мыши, в систему Win32 посылается сообщение `WM_MOUSEDOWN`. Система Win32 передает это сообщение элементу управления, для которого оно предназначено и на которое он должен тем или иным образом ответить. Элемент управления может ответить на такое событие, проверив сначала наличие кода, предусмотренного для выполнения. Для этого он проверяет, ссылается ли свойство-событие на какой-либо код. Если да, то элемент выполняет данный код, называемый обработчиком события.

Событие `OnClick` — лишь одно из стандартных свойств-событий, определенных в Delphi. Оно, как и другие свойства-события, имеет соответствующий *метод диспетчеризации событий* (event-dispatching method). Обычно эти методы объявляются как защищенные методы того компонента, которому они принадлежат (в разделе `protected`). Они выполняют операции по определению того, ссылается ли данное свойство-событие на какой-либо код, предоставленный пользователем компонента. Для свойства `OnClick` таким методом является метод `Click()`. Свойство `OnClick` и метод `Click()` определены в классе `TControl` следующим образом:

```
TControl = class(TComponent)
private
  FOnClick: TNotifyEvent;
protected
  procedure Click; dynamic;
  property OnClick: TNotifyEvent read FOnClick write FOnClick;
end;
```

А вот как выглядит реализация метода `TControl.Click()`:

```
procedure TControl.Click;
begin
  if Assigned(FOnClick) then FOnClick(Self);
end;
```

Основное, что следует четко понять — это то, что свойства-события являются не более чем указателями на методы. Заметьте, что свойство `FOnClick` имеет тип `TNotifyEvent`, который определен следующим образом:

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Как видите, тип `TNotifyEvent` представляет собой процедуру с одним параметром `Sender` типа `TObject`, а директива `of object` делает эту процедуру методом. Указанное означает, что в процедуру передается дополнительный *неявный* параметр, который не указывается в списке параметров, передаваемых данной процедуре. Это параметр `Self`, ссылающийся на объект, которому принадлежит данный метод. Когда вызывается метод `Click()` компонента, он проверяет, действительно ли переменная `FOnClick` ссылается на метод, и, если да, то вызывает последний.

Разработчик компонента пишет весь программный код, определяющий событие, свойство-событие и метод диспетчеризации. Пользователь компонента создает лишь обработчик события. В функции метода диспетчеризации события входит проверка того, назначил ли пользователь событию какой-либо код, и выполнение этого кода, если таковой существует.

В главе 10, "Архитектура компонентов: VCL и CLX", было описано, как обработчики событий присваиваются свойствам-событиям во время разработки и выполнения программы. В следующем разделе рассматривается, как определять собственные события, свойства-события и методы их диспетчеризации.

Определение свойств–событий

Прежде чем определять свойство-событие, нужно решить, необходим ли новый тип события. Чтобы принять такое решение, полезно ознакомиться с типичными свойствами-событиями компонентов библиотеки VCL Delphi. В большинстве случаев можно определить компонент как производный от уже существующего компонента и просто использовать его свойства-события или сделать доступным одно из его защищенных свойств-событий. Но если обнаружится, что ни одно из существующих событий не подходит, то лишь тогда имеет смысл определять собственное.

В качестве примера рассмотрим следующий сценарий. Предположим, требуется, чтобы в новом компоненте имелось событие, происходящее каждые полминуты по системным часам. Конечно, есть возможность воспользоваться компонентом `TTimer` для проверки системного времени и выполнения необходимых действий в установленные моменты времени. Но разрабатываемый код можно встроить непосредственно в

пользовательский компонент, а затем сделать его доступным другим пользователям, которым осталось бы только подготовить собственный код обработки события OnHalf-Minute.

Код компонента TddgHalfMinute приведен в листинге 11.7 и призван проиллюстрировать способы определения собственных событий.

Листинг 11.7. Создание события TddgHalfMiute

```
unit halfmin;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls;

type
  { Определить процедуру для обработчика события. Свойство-событие
    будет иметь тот процедурный тип, который предусматривает
    передачу двух параметров - генерирующего событие объекта и
    значения TDataTime, фиксирующего время генерации события.
    В данном случае это событие будет происходить каждые
    полминуты. }
  TTimeEvent = procedure(Sender: TObject;
                         TheTime: TDateTime) of object;

  TddgHalfMinute = class(TComponent)
  private
    FTimer: TTimer;
    { Определить поле, которое будет хранить ссылку на обработчик
      события пользователя. Предоставленный пользователем
      обработчик события должен иметь процедурный тип TTimeEvent.}
    FOnHalfMinute: TTimeEvent;
    FOldSecond, FSecond: Word; // Переменные, используемые в коде
    { Определяем процедуру FTimerTimer, которая будет связана со
      свойством-событием FTimer.OnClick. Эта процедура должна
      иметь тип TNotifyEvent, как и само
      свойство-событие TTimer.OnClick. }
    procedure FTimerTimer(Sender: TObject);
  protected
    { Определение метода диспетчеризации события OnHalfMinute. }
    procedure DoHalfMinute(TheTime: TDateTime); dynamic;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Определить реальное свойство, отображаемое в окне инспектора
      объектов. }
    property OnHalfMinute: TTimeEvent read FOnHalfMinute
                                      write FOnHalfMinute;
  end;

implementation
```

```
constructor TddgHalfMinute.Create(AOwner: TComponent);
{ Конструктор Create создает экземпляр компонента TTimer по имени
  FTimer, а затем задает различные свойства экземпляра FTimer,
  в том числе его обработчик события OnTimer, являющийся методом
  FTimerTimer() компонента TddgHalfMinute. Заметьте, что свойство
  FTimer.Enabled устанавливается равным значению true при
  выполнении программы и значению false, если компонент находится
  в режиме разработки. }
begin
  inherited Create(AOwner);
  { Если компонент находится в режиме разработки, FTimer не
    активизируется. }
  if not (csDesigning in ComponentState) then begin
    FTimer := TTimer.Create(self);
    FTimer.Enabled := True;
    { Установить остальные свойства, включая обработчик
      события FTimer.OnTimer.}
    FTimer.Interval := 500;
    FTimer.OnTimer := FTimerTimer;
   end;
end;

destructor TddgHalfMinute.Destroy;
begin
  FTimer.Free;
  inherited Destroy;
end;

procedure TddgHalfMinute.FTimerTimer(Sender: TObject);
{ Этот метод служит обработчиком события FTimer.OnTimer. Он
  назначается свойству FTimer.OnTimer динамически в конструкторе
  TddgHalfMinute. Данный метод считывает системное время и
  проверяет, не кратно ли оно 30 секундам. Если это условие
  выполняется, то он вызывает метод диспетчеризации DoHalfMinute
  события OnHalfMinute. }
var
  DT: TDateTime;
  Temp: Word;
begin
  DT := Now; // Получить системное время
  FOldSecond := FSecond; // Сохранить старое значение секунд
  // Получить значение секундной составляющей системного времени.
  DecodeTime(DT, Temp, Temp, FSecond, Temp);

  { Если текущее время не совпадает со временем последнего
  вызова метода и оно кратно 30 секундам, вызывается
  метод DoOnHalfMinute. }
  if FSecond <> FOldSecond then
    if ((FSecond = 30) or (FSecond = 0)) then
      DoHalfMinute(DT)
end;
```

```
procedure TddgHalfMinute.DoHalfMinute(TheTime: TDateTime);
begin
  if Assigned(FOnHalfMinute) then
    FOnHalfMinute(Self, TheTime);
end;

end.
```

При создании собственного события необходимо решить, какую именно информацию следует передавать пользователям данного компонента в качестве параметра обработчика события. Например, при создании обработчика события `TEdit.OnKeyPress` его код может выглядеть следующим образом:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
end;
```

В этом случае передается не только ссылка на объект, ответственный за возникновение события, но и параметр `Char`, определяющий нажатую клавишу. В недрах подпрограмм библиотеки VCL данное событие возникает в результате поступления сообщения Win32 `WM_CHAR`, содержащего дополнительную информацию о нажатой клавише. Delphi извлекает из этого сообщения необходимые данные и делает их доступными пользователям компонента в качестве параметров обработчика события. Такая схема взаимодействия позволяет разработчику компонента преобразовывать слишком сложную информацию и делать ее доступной пользователям компонента в простом и понятном виде.

Обратите внимание на передаваемый по ссылке параметр в методе `Edit1KeyPress()`. Может вызвать интерес тот факт, что этот метод был объявлен процедурой, а не функцией, возвращающей значение `Char`. Хотя методы могут быть функциями, не следует объявлять как функции события, поскольку это внесет дополнительную путаницу: обращаясь к указателю метода-функции, нельзя быть уверенным, ссылается ли он на результат функции или является указателем функции. Между прочим, одно событие, представляющее собой функцию, все же уцелело со времен первых версий Delphi — это событие `TApplication.OnHelp`.

Просмотрев листинг 11.7, можно заметить, что процедурный тип `TOnHalfMinute` определен следующим образом:

```
TTimeEvent = procedure(Sender: TObject;
                       TheTime: TDateTime) of object;
```

Этот процедурный тип определяет тип обработчика события `OnHalfMinute`. Здесь пользователь передает ссылку на объект, вызывающий событие, и значение времени `TDataTime`, когда это событие произошло.

Поле хранения `FOnHalfMinute` ссылается на пользовательский обработчик события и отображается в окне инспектора объектов на этапе проектирования как свойство `OnHalfMinute`.

Основным назначением компонента, использующего объект `TTimer`, является определение секундной составляющей времени через каждые полсекунды. Если это значение равно `0` или `30`, вызывается метод `DoHalfMinute()`, отвечающий за проверку

существования обработчика события и последующий его вызов. Прочтите комментарии, приведенные в коде программы, — они содержат много полезной информации.

Установив компонент в палитру компонентов Delphi, можно будет помещать его в форму и добавлять следующий обработчик события `OnHalfMinute`:

```
procedure TForm1.ddgHalfMinuteHalfMinute(Sender: TObject;
                                         TheTime: TDateTime);
begin
  ShowMessage('The Time is '+TimeToStr(TheTime));
end;
```

Этот код демонстрирует, как новый тип события получает свой обработчик.

Создание методов

Добавление в компонент методов не отличается от их добавления в любой другой объект. Тем не менее существует несколько моментов, которые следует учитывать при разработке компонентов.

Никакой взаимозависимости!

Одна из главных целей, преследуемых при создании компонентов, — упростить его использование конечным пользователем. Поэтому нужно максимально исключить взаимозависимость методов. Например, не нужно заставлять пользователя вызывать определенный метод при использовании компонента или вызывать методы в определенном порядке. Кроме того, методы, вызываемые пользователем, не должны переводить компонент в такое состояние, при котором другие методы и события не действуют. Наконец, методам следует давать осмысленные имена, чтобы пользователю не приходилось гадать, что делает тот или иной метод.

Степень доступности метода

При разработке компонента необходимо решить, какие методы следует объявить закрытыми (`private`), открытыми (`public`) или защищенными (`protected`). Нужно принять во внимание потребности не только пользователей компонента, но и тех, кто будет использовать класс этого компонента в качестве базового для следующего пользовательского компонента. Информация в табл. 11.2 поможет принять правильное решение.

Таблица 11.2. `Private`, `Protected`, `Public` или `Published`?

Директива	*Для чего предназначена*
`Private`	Переменные и методы экземпляра, которые нежелательно предоставлять типу потомка для доступа или модификации. Для того, чтобы помочь пользователю избежать неприятных ситуаций, доступ к таким переменным обычно предоставляется с помощью свойств с директивами `read` и `write`, определяющими методы доступа. Естественно, не следует предоставлять пользователям доступ ко всем методам реализации свойств

Окончание табл. 11.2.

Директива	*Для чего предназначена*
`Protected`	Переменные, методы и свойства экземпляров, к которым производные классы (но не пользователи данного класса) смогут получать доступ и изменять их. Обычно свойства помещают в раздел `protected` в базовом классе для того, чтобы при необходимости сделать их публикуемыми в производном классе
`Public`	Методы и свойства, доступные любому пользователю объекта данного класса. Если доступ к некоторым свойствам необходимо иметь во время выполнения программы, а не во время разработки, то поместите их в раздел `Public`
`Published`	Свойства, к которым необходим доступ в окне инспектора объектов во время разработки. Для всех свойств, объявленных в этом разделе, генерируется информация о типах времени выполнения (RTTI)

Конструкторы и деструкторы

Создавая новый класс компонента, можно переопределить конструктор его базового класса и установить собственный. При этом необходимо соблюдать некоторые меры предосторожности.

Переопределение конструкторов

Объявляя собственный конструктор для потомка класса `TComponent`, всегда используйте директиву `override`, как показано ниже.

```
TSomeComopnent = class(TComponent)
private
  { Закрытые объявления }
protected
  { Защищенные объявления }
public
  constructor Create(AOwner: TComponent); override;
published
  { Публикуемые объявления }
end;
```

НА ЗАМЕТКУ

На уровне класса `TComponent` конструктор `Create()` является виртуальным. Некомпонентные классы имеют статические конструкторы, которые вызываются конструкторами классов `TComponent`. Следовательно, при создании некомпонентного производного класса, конструктор не может быть переопределен, поскольку он не является виртуальным:

```
TMyObject = class(TPersistant)
```

В этом случае достаточно просто переобъявить конструктор.

Несмотря на то что с точки зрения синтаксиса директива override необязательна, ее отсутствие может вызвать проблемы при использовании компонента. Дело в том, что в процессе использования компонента (как во время разработки, так и во время выполнения) невиртуальный конструктор не будет вызван кодом, создающим этот компонент по ссылке на класс (например в потоковой системе).

Не забудьте также удостовериться, что в коде конструктора вызывается унаследованный конструктор:

```
constructor TSomeComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  // Поместите ваш код здесь
end;
```

Поведение компонента во время разработки

Помните, что при создании компонента всегда вызывается его конструктор. Это относится и к созданию компонента во время разработки (при помещении его в форму). Кстати, можно потребовать отмены выполнения некоторых действий, когда компонент находится в состоянии разработки. Например, в конструкторе компонента TddgHalfMinute создается компонент TTimer. В общем-то ничего страшного в этом нет, но существует возможность добиться того, чтобы TTimer вызывался только во время выполнения.

Для определения текущего состояния компонента следует проверить его свойство ComponentState. В табл. 11.3 приведены данные о возможных состояниях компонента, взятые из интерактивной справочной системы Delphi 6.

Таблица 11.3. Значения состояния компонента

Флаг	*Состояние компонента*
csAncestor	Устанавливается, если компонент используется в форме-предке. Это значение устанавливается только при установленном флаге csDesigning
csDesigning	Режим разработки, т.е. компонент находится в форме в окне конструктора форм
csDestroying	Объект в процессе удаления
csFixups	Установлен, если данный компонент связан с компонентом другой формы, которая еще не загружена. Флаг сбрасывается, когда все подобные связи установлены
csLoading	Загрузка из объекта файловой системы
csReading	Считывание значений свойств из потока
csUpdating	Компонент обновляется, чтобы отобразить изменения, внесенные в форму-предок. Устанавливается только при установленном флаге csAncestor
csWriting	Запись свойств компонента в поток

Состояние csDesigning чаще всего используется для проверки того, находится ли компонент в режиме разработки. Это можно сделать с помощью следующих операторов:

```
inherited Create(AOwner);
if csDesigning in ComponentState then
  { Выполнение необходимых действий }
```

Следует заметить, что состояние csDesigning не определяется до тех пор, пока не будет вызван унаследованный конструктор и данный компонент не будет создан его владельцем. Эти условия почти всегда выполняются при работе с компонентом в конструкторе форм.

Переопределение деструкторов

Переопределяя деструктор, очень важно освободить все ресурсы, распределенные данным компонентом *до того*, как будет вызван унаследованный деструктор:

```
destructor TMyComponent.Destroy;
begin
  FTimer.Free;
  MyStrings.Free;
  inherited Destroy;
end;
```

СОВЕТ

Вот простое, но удобное правило: в переопределенном конструкторе первым вызывается унаследованный конструктор, а в переопределенном деструкторе, наоборот, унаследованный деструктор вызывается последним. Соблюдение этого правила гарантирует, что класс будет корректно настроен перед его модификацией и все связанные с ним ресурсы будут освобождены перед завершением работы с этим классом.
Из данного правила есть исключения, но с ними вряд ли придется встретиться на практике.

Регистрация компонента

В процессе регистрации компонент помещается в палитру компонентов Delphi. Если для разработки компонента использовалось окно New Component, то для регистрации ничего предпринимать не нужно — Delphi сама создаст необходимый код. Если же компонент написан вручную, то в модуль созданного компонента необходимо включить процедуру Register().

В этом случае достаточно добавить процедуру Register() в раздел interface модуля создаваемого компонента.

Процедура Register() просто вызывает процедуру RegisterComponents() для каждого регистрируемого компонента. Процедуре RegisterComponents() передается два параметра: имя вкладки, в которой будет размещаться компонент, и массив типов компонентов. В листинге 11.8 показано, как использовать эту процедуру.

Листинг 11.8. Регистрация компонентов

```
Unit MyComp;
interface
type
  TMyComp = class(TComponent)
    ...
  end;
  TOtherComp = class(TComponent)
    ...
  end;
procedure Register;
implementation
{ Методы класса TMyComp }
{ Методы класса TOtherComp }
procedure Register;
begin
  RegisterComponents('DDG', [TMyComp, TOtherComp]);
end;
end.
```

При выполнении этого кода компоненты `TMyComp` и `TOtherComp` регистрируются и помещаются во вкладку DDG палитры компонентов Delphi.

Палитра компонентов

В Delphi версий 1 и 2 вся библиотека с компонентами, пиктограммами и редакторами, использовавшимися во время разработки, находилась в одном файле. Впрочем, работать лишь с одним файлом было очень удобно, тем не менее по мере того, как в библиотеку добавлялись все новые и новые компоненты, управлять ею становилось все тяжелее и все больше времени уходило на ее перекомпоновку при каждом добавлении.

С появлением пакетов в Delphi 3 стало возможным разбить библиотеку компонентов на несколько отдельных частей. Хотя иметь дело со множеством файлов сложнее, чем с одним, это решение существенно расширяет возможности управления конфигурацией и значительно сокращает время, необходимое для перекомпоновки библиотеки при добавлении в нее нового компонента.

По умолчанию все новые компоненты добавляются в пакет `DclUser6`, но можно создавать и устанавливать новые пакеты времени разработки, выбрав в меню File пункты New и Package. На прилагаемом компакт-диске находится пакет разработки `DdgDT6.dpk`, включающий в себя компоненты, описываемые в этой книге. Пакет времени выполнения называется `DdgRT6.dpk`.

Если поддержка режима разработки подразумевает нечто большее, чем просто вызов `RegisterComponents()` (например необходимы редакторы свойств, или редакторы компонентов, или регистрация экспертов), то следует перенести процедуру `Register()` и регистрируемые ею элементы в модуль, отдельный от самих компонентов. Если этого не сделать, то такой “всеобщий” модуль, скомпилированный в пакет времени выполнения и содержащий процедуру `Register`, которая ссылается на классы или процедуры, существующие лишь во время разработки, окажется непригодным к использованию. Именно поэтому программная поддержка времени разработки должна быть отделена от соответствующей поддержки времени выполнения.

Проверка компонента

Несмотря на весь оптимизм и радость, вызванные завершением создания компонента, не спешите добавлять вновь созданный компонент в палитру компонентов, пока он не будет тщательно отлажен. Следует обязательно провести предварительную проверку, создав проект, в котором используется динамический экземпляр нового компонента. Дело в том, что во время разработки компонент находится в среде IDE. Если в нем содержится ошибка, приводящая к сбою в памяти, то это может, ко всему прочему, привести к зависанию самой среды разработки Delphi. В листинге 11.9 содержится модуль, предназначенный для проверки компонента `TddgExtendedMemo`, создание которого описано в настоящей главе далее. Этот проект находится на прилагаемом CD под именем `TestEMem.dpr`.

Листинг 11.9. Проверка компонента `TddgExtendedMemo`

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, exmemo, ExtCtrls;

type

  TMainForm = class(TForm)
    btnCreateMemo: TButton;
    btnGetRowCol: TButton;
    btnSetRowCol: TButton;
    edtColumn: TEdit;
    edtRow: TEdit;
    Panel1: TPanel;
    procedure btnCreateMemoClick(Sender: TObject);
    procedure btnGetRowColClick(Sender: TObject);
    procedure btnSetRowColClick(Sender: TObject);
  public
    EMemo: TddgExtendedMemo;  // Объявление компонента.
    procedure OnScroll(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
procedure TMainForm.btnCreateMemoClick(Sender: TObject);
{ Динамическое создание компонента. Удостоверьтесь, что всем
свойствам присвоены надлежащие значения, необходимые для правильной
работы компонента. Эти значения зависят от проверяемого компонента.
}
```

```
begin
  if not Assigned(EMemo) then begin
    EMemo := TddgExtendedMemo.Create(self);
    EMemo.Parent := Panel1;
    EMemo.ScrollBars := ssBoth;
    EMemo.WordWrap := True;
    EMemo.Align := alClient;
    // Присвоить обработчики проверяемым событиям.
    EMemo.OnVScroll := OnScroll;
    EMemo.OnHScroll := OnScroll;
  end;
end;

{ Создайте все методы, необходимые для проверки поведения
компонента во время выполнения, в том числе методы доступа ко всем
новым свойствам и методам компонента.

Создайте также обработчики пользовательских событий, чтобы их можно
было проверить. Поскольку компонент создается во время выполнения,
обработчики событиям придется присвоить вручную, как это сделано в
конструкторе Create(). }

procedure TMainForm.btnGetRowColClick(Sender: TObject);
begin
  if Assigned(EMemo) then
    ShowMessage(Format('Row: %d  Column: %d',
                [EMemo.Row, EMemo.Column]));
  EMemo.SetFocus;
end;

procedure TMainForm.btnSetRowColClick(Sender: TObject);
begin
  if Assigned(EMemo) then begin
    EMemo.Row := StrToInt(edtRow.Text);
    EMemo.Column := StrToInt(edtColumn.Text);
    EMemo.SetFocus;
  end;
end;

procedure TMainForm.OnScroll(Sender: TObject);
begin
  MessageBeep(0);
end;

end.
```

Имейте в виду, что даже проверка компонента в режиме разработки не позволяет избежать неприятных инцидентов. Некоторые действия программы могут привести к зависанию самой среды разработки Delphi — если забыть, например, вызвать унаследованный конструктор `Create()`.

НА ЗАМЕТКУ

Не думайте, что во время разработки компонент создается и устанавливается самой средой разработки Delphi — он становится "пригодным к употреблению" только после выполнения конструктора `Create()`. Следовательно, не нужно рассматривать метод `Loaded()` как часть процесса создания компонента. Метод `Loaded()` вызывается только при загрузке компонента из потока (например, когда его помещают в форму во время разработки). Метод `Loaded()` отмечает конец процесса загрузки из потока. Если компонент был создан, а не загружен из потока, то метод `Loaded()` не вызывается.

Создание пиктограммы компонента

Ни один пользовательский компонент не обходится без собственной *пиктограммы* (icon) в палитре компонентов. Для создания пиктограммы используется встроенный *графический редактор* Delphi (Image Editor) или любой другой редактор растровых изображений. Создайте рисунок размером 24x24 пикселя и нарисуйте подходящее изображение, а затем сохраните его в формате файла DCR. Файл с расширением `.dcr` — не что иное, как переименованный файл `.RES`. Следовательно, если сохранить пиктограмму в файле `.RES`, то достаточно будет просто изменить его расширение на `.dcr`.

СОВЕТ

Даже если драйвер используемой видеоплаты поддерживает режим 256 цветов и более, сохраните пиктограмму как растровое изображение с 16 цветами, если компонент предполагается передавать другим пользователям. На мониторах с 16 цветами 256-цветные изображения выглядят, как правило, просто ужасно.

Теперь, поместив пиктограмму в файл DCR, присвойте ей имя класса данного компонента, но используйте при этом прописные буквы. Сохраните файл ресурсов под тем же именем, что и у модуля компонента, но с расширением `.dcr`. Таким образом, если новый компонент называется `TXYZComponent`, то имя пиктограммы — `TXYZCOMPONENT`. Если имя модуля компонента `XYZCOMP.PAS`, то имя файла ресурсов — `XYZCOMP.DCR`. Поместите этот файл в папку, в которой находится модуль компонента. При перекомпиляции модуля пиктограмма будет автоматически связана с библиотекой компонентов.

Примеры разработки компонентов

В оставшихся разделах настоящей главы речь пойдет о создании нескольких реальных компонентов. Созданные компоненты будут использованы для двух основных задач. Во-первых, они станут иллюстрацией использования всех технологий, описанных в предыдущей части главы. А во-вторых, приведенные здесь компоненты можно смело использовать в своих приложениях. При этом ничто не мешает расширить их функциональные возможности, для того чтобы они полностью удовлетворяли возникающие потребности.

Расширение возможностей компонентов-оболочек для классов Win32

В некоторых случаях может потребоваться расширение функциональных возможностей существующих компонентов, особенно компонентов, инкапсулирующих классы элементов управления Win32. Рассмотрим, как этого можно добиться на примере расширения функций элементов управления `TMemo` и `TListBox`.

Компонент TddgExtendedMemo — расширение компонента TMemo

Несмотря на известную гибкость, компонент `TMemo` не дает возможности реализовать некоторые полезные функции. В частности, он не позволяет управлять положением курсора вставки с точки зрения строк и столбцов. Поэтому расширим компонент `TMemo` дополнительными открытыми свойствами.

Кроме того, иногда может понадобиться выполнять определенные действия всякий раз, когда пользователь взаимодействует с полосами прокрутки компонента. Для этого будут созданы события, которым можно назначить код, выполняющийся при возникновении событий прокрутки.

Исходный код компонента `TddgExtendedMemo` приведен в листинге 11.10.

Листинг 11.10. `ExtMemo.pas` — исходный код компонента `TddgExtendedMemo`

```
unit ExtMemo;

interface

uses
  Windows, Messages, Classes, StdCtrls;

type

  TddgExtendedMemo = class(TMemo)
  private
    FRow: Longint;
    FColumn: Longint;
    FOnHScroll: TNotifyEvent;
    FOnVScroll: TNotifyEvent;
    procedure WMHScroll(var Msg: TWMHScroll); message WM_HSCROLL;
    procedure WMVScroll(var Msg: TWMVScroll); message WM_VSCROLL;
    procedure SetRow(Value: Longint);
    procedure SetColumn(Value: Longint);
    function GetRow: Longint;
    function GetColumn: Longint;
  protected
    // Методы диспетчеризации событий
    procedure HScroll; dynamic;
    procedure VScroll; dynamic;
  public
```

```
    property Row: Longint read GetRow write SetRow;
    property Column: Longint read GetColumn write SetColumn;
  published
    property OnHScroll: TNotifyEvent read FOnHScroll
                                     write FOnHScroll;
    property OnVScroll: TNotifyEvent read FOnVScroll
                                     write FOnVScroll;
  end;

implementation

procedure TddgExtendedMemo.WMHScroll(var Msg: TWMHScroll);
begin
  inherited;
  HScroll;
end;

procedure TddgExtendedMemo.WMVScroll(var Msg: TWMVScroll);
begin
  inherited;
  VScroll;
end;

procedure TddgExtendedMemo.HScroll;
{ Это метод диспетчеризации события OnHScroll. Он проверяет, связан
ли обработчик с событием OnHScroll, и если да, вызывает его. }
begin
  if Assigned(FOnHScroll) then
    FOnHScroll(self);
end;

procedure TddgExtendedMemo.VScroll;
{ Это метод диспетчеризации события OnVScroll. Он проверяет, связан
ли обработчик с событием OnVScroll, и если да, вызывает его. }
begin
  if Assigned(FOnVScroll) then
    FOnVScroll(self);
end;

procedure TddgExtendedMemo.SetRow(Value: Longint);
{ Сообщение EM_LINEINDEX возвращает позицию первого символа строки,
заданной параметром wParam. В этом экземпляре в качестве wParam
используется параметр Value. Установка возвращаемого значения
SelStart устанавливает позицию курсора вставки в строке, заданной
параметром Value. }
begin
  SelStart := Perform(EM_LINEINDEX, Value, 0);
  FRow := SelStart;
end;

function TddgExtendedMemo.GetRow: Longint;
{ Сообщение EM_LINEFROMCHAR возвращает строку, в которой находится
символ, определяемый значением wParam. Если в качестве wParam
```

```
передается -1, то возвращается номер строки, в которой находится
курсор вставки. }
begin
  Result := Perform(EM_LINEFROMCHAR, -1, 0);
end;

procedure TddgExtendedMemo.SetColumn(Value: Longint);
{ Получает длину текущей строки с помощью сообщения EM_LINELENGTH.
Это сообщение принимает позицию символа как параметр WParam.
Возвращается длина строки, в которой находится этот символ. }
begin
  FColumn := Perform(EM_LINELENGTH,
                     Perform(EM_LINEINDEX, GetRow, 0), 0);
  { Если значение FColumn больше значения, переданного в
    процедуру, то это значение присваивается переменной FColumn. }
  if FColumn > Value then FColumn := Value;
  // SelStart получает значение найденной позиции
  SelStart := Perform(EM_LINEINDEX, GetRow, 0) + FColumn;
end;

function TddgExtendedMemo.GetColumn: Longint;
{ Сообщение EM_LINEINDEX возвращает индекс строки символа,
переданного в качестве параметра wParam. Если wParam равен -1, то
возвращается индекс текущей строки. Отнимая это значение от
SelStart, получаем положение столбца. }
begin
  Result := SelStart - Perform(EM_LINEINDEX, -1, 0);
end;

end.
```

Прежде всего рассмотрим добавленную в компонент `TddgExtendedMemo` информацию о строках и столбцах. Обратите внимание, что в этот компонент было добавлено два закрытых поля: `FRow` и `FColumn`. Данные поля предназначены для хранения значений строки и столбца курсора вставки. Заметьте также, что определены открытые свойства `Row` и `Column`. Эти свойства объявлены в разделе `public`, так как во время разработки они не нужны. Оба они обладают методами доступа — как для записи, так и для чтения. В качестве методов доступа к свойству `Row` используются методы `GetRow()` и `SetRow()`, а к свойству `Column` — методы `GetColumn()` и `SetColumn()`. На практике можно было бы обойтись без полей `FRow` и `FColumn`, так как доступ к свойствам `Row` и `Column` осуществляется с помощью соответствующих методов доступа; но мы оставили их, чтобы иметь возможность для дальнейшего расширения компонента.

Четыре упомянутых выше метода доступа используют различные сообщения *EM_XXXX*. Комментарии в коде поясняют, что происходит в каждом методе и как эти сообщения используются для обеспечения компонента информацией о значениях `Row` и `Column`. Компонент `TddgExtendedMemo` вводит также два события: `OnHScroll` и `OnVScroll`. Событие `OnHScroll` происходит, когда пользователь щелкает на горизонтальной полосе прокрутки, а событие `OnVScroll` — когда пользователь щелкает на вертикальной полосе прокрутки. Для обнаружения этих событий необходимо перехватить сообщения Win32 `WM_HSCROLL` и `WM_VSCROLL`, передаваемые в элемент

управления, когда пользователь щелкает на одной из его полос прокрутки. Для этого созданы два обработчика сообщений — WMHScroll() и WMVScroll(), — которые вызывают методы диспетчеризации событий HScroll() и VScroll(), а те, в свою очередь, проверяют, назначил ли пользователь компонента обработчики событиям OnHScroll и OnVScroll, а затем вызывают их. Если вас удивляет, почему эта проверка не выполняется в обработчиках сообщений, то объяснение заключается в обеспечении возможности вызывать обработчик событий в результате выполнения других действий, например изменения пользователем позиции курсора вставки.

Теперь компонент TddgExtendedMemo можно установить и использовать в любом приложении. Можно также подумать о расширении этого компонента (к примеру, когда пользователь изменяет позицию курсора вставки, владельцу элемента посылается сообщение WM_COMMAND). Функция HiWord(wParam) содержит уведомляющий код, подтверждающий выполнение определенного действия. Данный код мог бы иметь значение EN_CHANGE, наличие которого говорило бы об изменении сообщения, связанного с редактируемым элементом. Затем можно сделать подкласс данного компонента родительским и перехватывать это сообщение в родительской процедуре окна, которая автоматически обновит поля FRow и FColumn. Подклассы — еще одна сложная тема, обсуждение которой еще предстоит.

TddgTabbedListBox — расширение компонента TListBox

Компонент TListBox библиотеки VCL в Object Pascal является оболочкой стандартного элемента управления Win32 LISTBOX. Совершенству нет предела, поэтому, хотя компонент TListBox инкапсулирует бо́льшую часть функций элемента Win32, все же попытаемся его расширить. В настоящем разделе проанализируем шаг за шагом процесс создания пользовательского компонента на базе стандартного компонента TListBox.

Идея

Идея создания нового компонента, как часто бывает, подсказана практикой. Однажды потребовался список с использованием позиций табуляции и горизонтальной полосы прокрутки для чтения строк, оказавшихся длиннее ширины списка. Обе эти возможности поддерживаются интерфейсом API Win32, но не реализованы в компоненте TListBox. Назовем новый компонент TddgTabListbox.

План создания такого компонента прост: создается потомок класса TListBox с корректными свойствами полей, переопределенными методами и новыми методами, предназначенными для получения желаемого поведения компонента.

Код

Первым шагом в создании прокручиваемого списка с позициями табуляции является включение соответствующих оконных стилей в стиль компонента TddgTabListbox при создании окна списка. К необходимым оконным стилям относятся стиль lbs_UseTabStops для табуляции и стиль ws_HScroll для горизонтальной полосы прокрутки. Для добавления оконных стилей к потомку класса TWinControl необходимо переопределить метод CreateParams(), как показано в следующем коде:

```
procedure TddgTabListbox.CreateParams(var Params: TCreateParams);
begin
```

```
  inherited CreateParams(Params);
  Params.Style := Params.Style or lbs_UseTabStops or ws_HScroll;
end;
```

Для установки позиций табуляции компонент `TddgTabListbox` создает сообщение `lb_SetTabStops`, передавая ему количество позиций табуляции и указатель на их массив в параметрах `wParam` и `lParam` соответственно. Эти две переменные будут храниться в классе — в полях `FNumTabStops` и `FTabStops`. Единственная сложность состоит в том, что позиции табуляции в окне списка обрабатываются с использованием *единиц измерения диалоговых окон* (dialog box units). Delphi не поддерживает никаких других единиц измерения, кроме пикселей, поэтому придется приводить позиции табуляции к пикселям. С помощью модуля `PixDlg.pas`, код которого приведен в листинге 11.11, можно преобразовывать единицы измерения диалоговых окон для осей X и Y в экранные пиксели и обратно.

Метод `CreateParams()`

Если необходимо изменить один из параметров (таких как стиль или класс окна), передаваемых функции API `CreateWindowEx()`, то следует воспользоваться методом `CreateParams()`. Функция `CreateWindowEx()` используется для создания дескриптора окна, связанного с потомком класса `TWinControl`. Переопределение метода `CreateParams()` позволит управлять созданием окна на уровне интерфейса API.
Методу `CreateParams()` передается один параметр типа `TCreateParams`:

```
TCreateParams = record
  Caption: PChar;
  Style: Longint;
  ExStyle: Longint;
  X, Y: Integer;
  Width, Height: Integer;
  WndParent: HWnd;
  Param: Pointer;
  WindowClass: TWndClass;
  WinClassName: array[0..63] of Char;
end;
```

Разработчики компонентов осуществляют переопределение метода `CreateParams()` всякий раз, когда возникает необходимость управлять созданием компонента на уровне API. При этом нельзя забывать, что вначале обязательно нужно вызвать унаследованный метод `CreateParams()` для заполнения записи `Params`.

Листинг 11.11. Исходный код модуля `PixDlg.pas`

```
unit Pixdlg;

interface

function DialogUnitsToPixelsX(DlgUnits: word): word;
function DialogUnitsToPixelsY(DlgUnits: word): word;
function PixelsToDialogUnitsX(PixUnits: word): word;
function PixelsToDialogUnitsY(PixUnits: word): word;
```

```
implementation

uses WinProcs;

function DialogUnitsToPixelsX(DlgUnits: word): word;
begin
  Result := (DlgUnits * LoWord(GetDialogBaseUnits)) div 4;
end;

function DialogUnitsToPixelsY(DlgUnits: word): word;
begin
  Result := (DlgUnits * HiWord(GetDialogBaseUnits)) div 8;
end;

function PixelsToDialogUnitsX(PixUnits: word): word;
begin
  Result := PixUnits * 4 div LoWord(GetDialogBaseUnits);
end;

function PixelsToDialogUnitsY(PixUnits: word): word;
begin
  Result := PixUnits * 8 div HiWord(GetDialogBaseUnits);
end;

end.
```

Зная позиции табуляции, можно вычислить протяженность горизонтальной полосы прокрутки. Она должна быть больше самой длинной строки списка. К счастью, в интерфейсе API Win32 есть функция `GetTabbedTextExtent()`, возвращающая именно эту информацию. Если известен размер самой длинной строки, то можно установить диапазон прокрутки, создав сообщение `lb_SetHorizontalExtent`, передающее требуемый размер в качестве параметра `wParam`.

Кроме того, следует написать обработчики для некоторых специальных сообщений Win32. В частности, необходимо обеспечить обработку сообщений, управляющих вставкой и удалением, для того чтобы иметь возможность измерить длину любой новой строки или определить, не удалена ли самая длинная строка. В этом смысле особый интерес будут представлять такие сообщения, как `lb_AddString`, `lb_InsertString` и `lb_DeleteString`.

В листинге 11.12 содержится исходный код модуля `LbTab.pas` компонента `TddgTabListbox`.

Листинг 11.12. `LbTab.pas` — исходный код компонента `TddgTabListBox`

```
unit Lbtab;

interface

uses
  SysUtils, Windows, Messages, Classes, Controls, StdCtrls;
```

```
type

  EddgTabListboxError = class(Exception);

  TddgTabListBox = class(TListBox)
  private
    FLongestString: Word;
    FNumTabStops: Word;
    FTabStops: PWord;
    FSizeAfterDel: Boolean;
    function GetLBStringLength(S: String): word;
    procedure FindLongestString;
    procedure SetScrollLength(S: String);
    procedure LBAddString(var Msg: TMessage);
                                    message lb_AddString;
    procedure LBInsertString(var Msg: TMessage);
                                    message lb_InsertString;
    procedure LBDeleteString(var Msg: TMessage);
                                    message lb_DeleteString;
  protected
    procedure CreateParams(var Params: TCreateParams); override;
  public
    constructor Create(AOwner: TComponent); override;
    procedure SetTabStops(A: array of word);
  published
    property SizeAfterDel: Boolean read FSizeAfterDel
                                write FSizeAfterDel default True;
  end;

implementation

uses PixDlg;

constructor TddgTabListBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSizeAfterDel := True;
  { Установить позиции табуляции по умолчанию. }
  FNumTabStops := 1;
  GetMem(FTabStops, SizeOf(Word) * FNumTabStops);
  FTabStops^ := DialogUnitsToPixelsX(32);
end;

procedure TddgTabListBox.SetTabStops(A: array of word);
{ Эта процедура устанавливает позиции табуляции списка равными
значениям, заданным в открытом массиве слов A. Новые позиции
табуляции указаны в пикселях и отсортированы в порядке возрастания.
При невозможности установить новую позицию табуляции передается
исключение. }
var
  i: word;
  TempTab: word;
  TempBuf: PWord;
```

```
begin
  { Сохранить новые значения во временных переменных на случай
  возникновения исключения при установке позиций табуляции. }
  TempTab := High(A) + 1;       // Количество позиций табуляции
  GetMem(TempBuf, SizeOf(A));  // Выделение памяти
  Move(A, TempBuf^, SizeOf(A));// Копирование новых позиций таб.
  { Перевод пикселей в единицы диалогового окна и... }
  for i := 0 to TempTab - 1 do
    A[i] := PixelsToDialogUnitsX(A[i]);
  { передача новых позиций табуляции в список. Обратите внимание:
    использовать следует только единицы диалогового окна. }
  if Perform(lb_SetTabStops, TempTab, Longint(@A)) = 0 then begin
    { Если значение выражения равно нулю, то новые позиции
      табуляции установить невозможно, временный буфер табуляции
      очищается и передается исключение. }
    FreeMem(TempBuf, SizeOf(Word) * TempTab);
    raise EddgTabListboxError.Create('Failed to set tabs.')
  end else begin
    { Если выражение не равно нулю, то это означает, что
      установлены новые позиции табуляции и можно освободить
      предыдущие. }
    FreeMem(FTabStops, SizeOf(Word) * FNumTabStops);
    { Копирование значений из временных переменных ... }
    FNumTabStops := TempTab;  // Установка количества позиций таб.
    FTabStops := TempBuf;     // Установка табуляции из буфера
    FindLongestString;        // Переустановка полосы прокрутки
    Invalidate;               // Перерисовка
  end;
end;

procedure TddgTabListBox.CreateParams(var Params: TCreateParams);
{ Необходимо добавить стили для табуляции и горизонтальной
прокрутки. Эти стили будут использованы
                                    функцией API CreateWindowEx(). }
begin
  inherited CreateParams(Params);
  { Стиль lbs_UseTabStops позволяет использовать в списке
    табуляцию; стиль ws_HScroll позволяет выполнять горизонтальную
    прокрутку списка. }
  Params.Style := Params.Style or lbs_UseTabStops or ws_HScroll;
end;

function TddgTabListBox.GetLBStringLength(S: String): word;
{ Эта функция возвращает длину строки S в пикселях. }
var
  Size: Integer;
begin
  // Получить длину строки текста
  Canvas.Font := Font;
  Result := LoWord(GetTabbedTextExtent(Canvas.Handle, PChar(S),
                   StrLen(PChar(S)), FNumTabStops, FTabStops^));
  { Добавляет немного пространства в конец полосы прокрутки для
    улучшения ее внешнего вида. }
```

```
  Size := Canvas.TextWidth('X');
  Inc(Result, Size);
end;

procedure TddgTabListBox.SetScrollLength(S: String);
{ Эта процедура изменяет длину полосы прокрутки, если строка S
длиннее самой длинной предыдущей строки. }
var
  Extent: Word;
begin
  Extent := GetLBStringLength(S);
  // Если эта строка оказалась самой длинной...
  if Extent > FLongestString then begin
    // установить самую длинную строку
    FLongestString := Extent;
    // установить размер полосы прокрутки
    Perform(lb_SetHorizontalExtent, Extent, 0);
  end;
end;

procedure TddgTabListBox.LBInsertString(var Msg: TMessage);
{ Эта процедура вызывается в ответ на сообщение lb_InsertString,
которое посылается списку каждый раз, когда вставляется новая
строка. Поле Msg.lParam содержит указатель на вставляемую строку.
Если новая строка длиннее любой имеющейся, настраивается размер
полосы прокрутки. }
begin
  inherited;
  SetScrollLength(PChar(Msg.lParam));
end;

procedure TddgTabListBox.LBAddString(var Msg: TMessage);
{ Эта процедура вызывается в ответ на сообщение lb_AddString,
которое передается списку каждый раз, когда добавляется новая
строка. Поле Msg.lParam содержит указатель на добавляемую строку с
завершающим нулевым символом. Если новая строка длиннее любой
строки списка, то настраивается размер полосы прокрутки. }
begin
  inherited;
  SetScrollLength(PChar(Msg.lParam));
end;

procedure TddgTabListBox.FindLongestString;
var
  i: word;
  Strg: String;
begin
  FLongestString := 0;
  { Цикл поиска самой длинной строки }
  for i := 0 to Items.Count - 1 do begin
    Strg := Items[i];
    SetScrollLength(Strg);
  end;
```

```
end;

procedure TddgTabListBox.LBDeleteString(var Msg: TMessage);
{ Эта процедура вызывается в ответ на сообщение lb_DeleteString,
которое передается в список при каждом удалении строки. Msg.wParam
содержит индекс удаляемой строки. Присвоение свойству SizeAfterDel
значения False запрещает обновление полосы прокрутки. Это
увеличивает производительность при частом удалении строк. }
var
  Str: String;
begin
  if FSizeAfterDel then begin
    Str := Items[Msg.wParam]; // Получить удаляемую строку
    inherited;                // Удалить строку
    { Является ли удаленная строка самой длинной? }
    if GetLBStringLength(Str) = FLongestString then
      FindLongestString;
  end else
    inherited;
end;

end.
```

В этом компоненте особый интерес представляет метод `SetTabStops()`, использующий в качестве параметра открытый массив типа `word`. Это разрешает пользователям устанавливать такое количество позиций табуляции, какое им потребуется, например:

```
ddgTabListboxInstance.SetTabStops([50, 75, 150, 300]);
```

Если текст в списке выходит за видимую часть окна, автоматически появляется горизонтальная полоса прокрутки.

Компонент TddgRunButton — создание свойств

Если в среде 16-разрядной Windows, помимо текущей, требовалось запустить еще одну программу, то для этого можно было использовать функцию API `WinExec()`. Хотя эта функция в среде Win32 все еще поддерживается, но использовать ее не рекомендуется. Для запуска очередного приложения теперь предлагается использовать функции API `CreateProcess()` и `ShellExecute()`. Каждый раз применять для этого функцию API `CreateProcess()` довольно обременительно, поэтому создадим для ее замены собственный метод `ProcessExecute()`, о чем и пойдет речь ниже.

Для иллюстрации использования этого метода создадим компонент `TddgRunButton`. Его назначение заключается в запуске приложения по щелчку пользователя на предоставленной ему кнопке.

Компонент `TddgRunButton` — идеальный пример создания свойств, проверки их значений и инкапсуляции сложных операций. Вдобавок здесь показано, как извлечь пиктограмму приложения из его исполняемого файла и отобразить ее в компоненте `TddgRunButton` во время разработки. Класс компонента `TddgRunButton` является производным от класса компонента `TSpeedButton`. Поскольку этот компонент со-

держит некоторые свойства, которые нежелательно делать доступными в окне инспектора объектов во время разработки, покажем, как можно скрыть уже существующие свойства от пользователя компонента. Конечно, эту методику нельзя считать идеальным подходом, а в теоретическом плане правильнее было бы создать свой собственный новый компонент, и авторы вполне с этим согласны. Однако это один из тех случаев, когда компания *Borland*, при всей своей безграничной мудрости, все же допустила некоторое упущение, не предоставив промежуточного класса между компонентами `TSpeedButton` и `TCustomControl` (последний является предком компонента `TSpeedButton`), — хотя у всех остальных подобных компонентов они есть. Таким образом, пришлось выбирать: или возиться с собственным компонентом, который в значительной мере дублирует возможности `TSpeedButton`, или одолжить их у `TSpeedButton`, но скрыть при этом несколько ненужных свойств. Было выбрано последнее, но только по необходимости. Между тем, это хороший пример того, насколько тщательно следует продумывать возможности и методы расширения компонента, созданного другими разработчиками.

Код компонента `TddgButton` приведен в листинге 11.13.

Листинг 11.13. `RunBtn.pas` — исходный код компонента `TddgRunButton`

```
unit RunBtn;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;

type
  TCommandLine = type string;
  TddgRunButton = class(TSpeedButton)
  private
    FCommandLine: TCommandLine;
    // Свойства, скрытые от инспектора объектов
    FCaption: TCaption;
    FAllowAllUp: Boolean;
    FFont: TFont;
    FGroupIndex: Integer;
    FLayOut: TButtonLayout;
    procedure SetCommandLine(Value: TCommandLine);
  public
    constructor Create(AOwner: TComponent); override;
    procedure Click; override;
  published
    property CommandLine: TCommandLine read FCommandLine
                                       write SetCommandLine;
    // Свойства "только для чтения" скрыты
    property Caption: TCaption read FCaption;
    property AllowAllUp: Boolean read FAllowAllUp;
    property Font: TFont read FFont;
    property GroupIndex: Integer read FGroupIndex;
```

```
    property LayOut: TButtonLayOut read FLayOut;
  end;

implementation

uses ShellAPI;

const
  EXEExtension = '.EXE';

function ProcessExecute(CommandLine: TCommandLine;
                        cShow: Word): Integer;
{ Этот метод инкапсулирует вызов функции CreateProcess(), создающей
новый процесс и его первичный поток. Данный метод используется в
Win32 для запуска других приложений и требует применения структур
TStartInfo и TProcessInformation. Такие структуры не
документированы в интерактивной справочной системе Delphi, но
описаны в файле справки Win32 как STARTUPINFO и
PROCESS_INFORMATION.
Параметр CommandLine указывает путь к файлу, предназначенному для
выполнения.
Параметр cShow задает константу SW_XXXX, определяющую способ
отображения окна. Это значение присваивается полю sShowWindow
структуры TStartupInfo.}
var
  Rslt: LongBool;
  StartUpInfo: TStartUpInfo;  // Задокументирован как STARTUPINFO
  // Задокументирован как PROCESS_INFORMATION
  ProcessInfo: TProcessInformation;
begin
  { Очистить содержимое структуры StartupInfo }
  FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
  { Инициализация структуры StartupInfo необходимыми данными.
    Здесь значение константы SW_XXXX присваивается полю
    wShowWindow структуры StartupInfo. При задании значения этого
    поля в поле dwFlags должен быть установлен флаг
    STARTF_USESSHOWWINDOW. Дополнительная информация о структуре
    TStartupInfo находится в справочной документации
    по системе Win32, в разделе STARTUPINFO. }
  with StartupInfo do begin
    cb := SizeOf(TStartupInfo); // Задать размер структуры
    dwFlags := STARTF_USESHOWWINDOW or STARTF_FORCEONFEEDBACK;
    wShowWindow := cShow
  end;

  { Создать процесс, вызвав функцию CreateProcess(). Эта функция
    заполняет структуру ProcessInfo информацией о новом процессе и
    его первичном потоке. Подробная информация о структуре
    TProcessInfo содержится в документации по Win32 в
    разделе PROCESS_INFORMATION. }
  Rslt := CreateProcess(PChar(CommandLine), nil, nil, nil, False,
      NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo);
  { Если Rslt равно true, значит, вызов функции CreateProcess
```

```
    прошел успешно. В противном случае GetLastError вернет код
    ошибки. }
  if Rslt then
    with ProcessInfo do begin
      { Ожидание запуска процесса. }
      WaitForInputIdle(hProcess, INFINITE);
      CloseHandle(hThread); // Освободить дескриптор hThread
      CloseHandle(hProcess);// Освободить дескриптор hProcess
      Result := 0;          // Result равен 0, запуск успешный
    end
  else Result := GetLastError; // Присвоить Result код ошибки.
end;

function IsExecutableFile(Value: TCommandLine): Boolean;
{ Данный метод позволяет проверить, представляет ли Value
исполняемый файл. Для этого расширение файла проверяется на
соответствие строке 'EXE' }
var
  Ext: String[4];
begin
  Ext := ExtractFileExt(Value);
  Result := (UpperCase(Ext) = EXEExtension);
end;

constructor TddgRunButton.Create(AOwner: TComponent);
{ Конструктор устанавливает стандартные значения свойств размеров в
соответствии 45x45 }
begin
  inherited Create(AOwner);
  Height := 45;
  Width  := 45;
end;

procedure TddgRunButton.SetCommandLine(Value: TCommandLine);
{ Данный метод записи присваивает полю FCommandLine значение Value,
но только если это значение является корректным именем исполняемого
файла. Он также устанавливает для компонента TddgRunButton
пиктограмму файла, заданного параметром Value. }
var
  Icon: TIcon;
begin
  { Вначале проверяется, находится ли Value в заданном месте и
    действительно ли он представляет собой исполняемый файл. }
  if not IsExecutableFile(Value) then
    Raise Exception.Create(Value+' is not an executable file.');
  if not FileExists(Value) then
    Raise Exception.Create('The file: '+Value+' cannot be found.');

  FCommandLine := Value;  // Сохранение Value в FCommandLine

  { Отображение пиктограммы файла, заданного параметром Value, в
    пиктограмме компонента TddgRunButton. Это требует создания
    экземпляра класса TIcon для помещения пиктограммы. Затем она
```

```
    копируется оттуда в свойство Canvas компонента TddgRunButton.
    Для получения пиктограммы приложения следует вызвать
    функцию API Win32 ExtractIcon(). }
  Icon := TIcon.Create; // Создать экземпляра класса TIcon
  try
    { Выделение пиктограммы из файла приложения. }
    Icon.Handle := ExtractIcon(hInstance, PChar(FCommandLine), 0);
    with Glyph do begin
      { Установка свойств TddgRunButton таким образом, чтобы
        пиктограмму из Icon можно было скопировать в этот
        компонент.
        Сначала очистить содержимое Canvas на случай, если прежде
        там находилась другая пиктограмма. }
      Canvas.Brush.Style := bsSolid;
      Canvas.FillRect(Canvas.ClipRect);
      { Установить ширину и высоту пиктограммы }
      Width := Icon.Width;
      Height := Icon.Height;
      // Отобразить пиктограмму на поверхности TddgRunButton.
      Canvas.Draw(0, 0, Icon);
    end;
  finally
    Icon.Free; // Удалить экземпляр TIcon
  end;
end;

procedure TddgRunButton.Click;
var
  WERetVal: Word;
begin
  inherited Click; // Вызов унаследованного метода Click
  { Выполнение метода ProcessExecute и проверка возвращаемого им
    значения. Если возвращаемое значение не равно 0, то передается
    исключение, поскольку произошла ошибка. Исключение выводит на
    экран сообщение о коде ошибки. }
  WERetVal := ProcessExecute(FCommandLine, sw_ShowNormal);
  if WERetVal <> 0 then begin
    raise Exception.Create('Error executing program. Error Code:;'
                           + IntToStr(WERetVal));
  end;
end;

end.
```

Компонент TddgRunButton обладает свойством CommandLine типа String. Значение этого свойства хранится в закрытом поле FCommandLine.

СОВЕТ

Определение типа `TCommandLine` заслуживает отдельного обсуждения. В данном случае используется следующий синтаксис:

```
TCommandLine = type string;
```

Такое определение предлагает компилятору рассматривать `TCommandLine` как уникальный тип и вместе с тем совместимый с остальными строковыми типами. Новый тип может получить собственную информацию о типах времени выполнения и собственный редактор свойства. Эта же технология может быть использована и для других типов, например:

```
TMySpecialInt = type Integer;
```

Создание редактора для свойства `CommandLine` описано в следующей главе. Пока не будем отвлекаться на это, так как данная тема сложная и ее придется обсудить особо.

Методом записи для свойства `CommandLine` является метод `SetCommandLine()`. Были представлены также две вспомогательные функции: `IsExecutableFile()` и `ProcessExecute()`.

Функция `IsExecutableFile()` по расширению переданного ей файла проверяет, является ли он исполняемым.

Создание и выполнение процесса

Функция `ProcessExecute()` инкапсулирует функцию API Win32 `CreateProcess()`, запускающую другое приложение. Запускаемое приложение определяется параметром `CommandLine`, содержащим путь к файлу. Второй параметр содержит одну из констант `SW_XXXX`, которая задает способ отображения окна. В табл. 11.4 приведены различные константы `SW_XXXX` с объяснением их значений, взятым из интерактивной справочной системы.

Таблица 11.4. Константы `SW_XXXX`

Константа `SW_XXXX`	*Значение*
`SW_HIDE`	Скрывает текущее окно. Активным становится другое окно
`SW_MAXIMIZE`	Отображает окно развернутым до максимума
`SW_MINIMIZE`	Сворачивает окно
`SW_RESTORE`	Отображает окно с тем размером, который оно имело перед последним разворачиванием или сворачиванием
`SW_SHOW`	Отображает окно в его текущем размере и позиции
`SW_SHOWDEFAULT`	Отображает окно в виде, который определен структурой `TStartupInfo`, передаваемой функции `CreateProcess()`
`SW_SHOWMAXIMIZED`	Активизирует или отображает развернутое окно
`SW_SHOWMINIMIZED`	Активизирует или отображает свернутое окно
`SW_SHOWMINNOACTIVE`	Отображает окно свернутым, но активное в данный момент окно остается таковым

Окончание табл. 11.4.

Константа SW_XXXX	*Значение*
SW_SHOWNA	Показывает окно в его текущем состоянии. Активное окно остается таковым
SW_SHOWNOACTIVATE	Отображает окно в таком виде, в каком оно отображалось последний раз. Текущее активное окно остается активным
SW_SHOWNORMAL	Активизирует или отображает окно в таком виде, в каком оно отображалось в последний раз. Позиция окна сохраняется, если оно было предварительно развернуто или свернуто

Функция ProcessExecute() является достаточно удобной утилитой, которую следует выделить в отдельный модуль для использования другими приложениями.

Методы класса TddgRunButton

Конструктор TddgRunButton.Create() после вызова унаследованного конструктора просто устанавливает размеры объекта, принимаемые по умолчанию.

Метод SetCommandLine(), представляющий собой метод записи параметра CommandLine, выполняет несколько задач. Во-первых, он проверяет, является ли значение, присваиваемое параметру CommandLine, именем исполняемого файла и, если нет, передает исключение.

Проверенное значение присваивается полю FCommandLine. Затем функция SetCommandLine() извлекает пиктограмму из файла приложения и рисует ее на поверхности компонента TddgRunButton. Для этого используется функция API Win32 ExtractIcon(). Объяснения данной технологии содержится в комментариях кода.

Метод TddgRunButton.Click() — это метод диспетчеризации события TSpeedButton.OnClick. Здесь необходимо вызвать сначала унаследованный метод Click(), вызывающий обработчик события OnClick (если он назначен). После вызова метода Click() вызывается функция ProcessExecute(), после чего проверяется возвращаемое ею значение, чтобы выяснить, был ли этот вызов успешным. Если нет — передается исключение.

Компонент-контейнер TddgButtonEdit

Иногда требуется создать компонент, состоящий из одного или нескольких других компонентов. Компонент Delphi TDBNavigator — отличный пример таких компонентов, поскольку он состоит из компонента TPanel и нескольких компонентов TSpeedButton. Данный раздел посвящен созданию компонента, являющегося комбинацией компонентов TEdit и TSpeedButton. Назовем этот компонент TddgButtonEdit.

Проектные решения

Ввиду того, что язык Object Pascal обладает объектной моделью с наследованием лишь от одного предка, компонент TddgButtonEdit должен быть самостоятельным компонентом, включающим компоненты TEdit и TSpeedButton. Более того, поскольку этот компонент должен содержать оконные элементы управления, он и сам должен

быть оконным элементом управления. Поэтому TddgButtonEdit необходимо сделать потомком класса TWinControl. В этом случае экземпляры компонентов TEdit и TSpeedButton следует создавать в конструкторе компонента TddgButtonEdit, используя следующий код:

```
constructor TddgButtonEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEdit        := TEdit.Create(Self);
  FEdit.Parent := self;
  FEdit.Height := 21;

  FSpeedButton := TSpeedButton.Create(Self);
  FSpeedButton.Left := FEdit.Width;
  // На два пикселя меньше вертикального размера TEdit
  FSpeedButton.Height := 19;
  FSpeedButton.Width  := 19;
  FSpeedButton.Caption := '...';
  FSpeedButton.Parent := Self;

  Width  := FEdit.Width+FSpeedButton.Width;
  Height := FEdit.Height;
end;
```

При создании компонента, содержащего другие компоненты, определенную трудность представляет выделение свойств внутренних компонентов из свойств компонента-контейнера. Например, компоненту TddgButtonEdit потребуется свойство Text. Для изменения шрифта этого текста понадобится свойство Font. Наконец, кнопке элемента необходимо событие OnClick. Естественно, не стоит пытаться самостоятельно реализовать эти свойства в компоненте-контейнере, если они уже существуют во внутренних компонентах. Наша задача — вывести необходимые свойства внутренних элементов управления на поверхность контейнера, не переписывая их интерфейса.

Предоставление свойств вложенных объектов

Эта задача сводится к простой, но трудоемкой процедуре создания методов записи и чтения для каждого из свойств внутренних компонентов, предоставляемых компоненту-контейнеру. Например, компонент TddgButtonEdit можно снабдить свойством Text с соответствующими методами доступа следующим образом:

```
TddgButtonEdit = class(TWinControl)
private
  FEdit: TEdit;
protected
  procedure SetText(Value: String);
  function  GetText: String;
published
  property Text: String read GetText write SetText;
end;
```

Методы SetText() и GetText() получают прямой доступ к свойству Text вложенного элемента TEdit:

```
function TddgButtonEdit.GetText: String;
begin
  Result := FEdit.Text;
end;

procedure TddgButtonEdit.SetText(Value: String);
begin
  FEdit.Text := Value;
end;
```

Предоставление событий

Помимо свойств, контейнеру необходимо предоставить события внутренних компонентов. Например, если пользователь щелкает мышью на элементе TSpeedButton, то придется обработать его событие OnClick. Предоставление событий ничем не отличается от предоставления свойств; ведь, в конце концов, события — это тоже свойства.

Для начала необходимо снабдить компонент TddgButtonEdit собственным событием OnClick. Для удобства назовем его OnButtonClick. И тогда методы записи и чтения этого события просто перенаправят назначенный обработчик событию OnClick внутреннего компонента TSpeedButton.

В листинге 11.14 приведен исходный код компонента-контейнера TddgButtonEdit.

Листинг 11.14. Компонент-контейнер TddgButtonEdit

```
unit ButtonEdit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons;

type
  TddgButtonEdit = class(TWinControl)
  private
    FSpeedButton: TSpeedButton;
    FEdit: TEdit;
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
    procedure SetText(Value: String);
    function  GetText: String;
    function GetFont: TFont;
    procedure SetFont(Value: TFont);
    function GetOnButtonClick: TNotifyEvent;
    procedure SetOnButtonClick(Value: TNotifyEvent);
  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;
  published
    property Text: String read GetText write SetText;
```

```
    property Font: TFont read GetFont write SetFont;
    property OnButtonClick: TNotifyEvent read GetOnButtonClick
                                          write SetOnButtonClick;
  end;

implementation

procedure TddgButtonEdit.WMSize(var Message: TWMSize);
begin
  inherited;
  FEdit.Width := Message.Width-FSpeedButton.Width;
  FSpeedButton.Left := FEdit.Width;
end;

constructor TddgButtonEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEdit        := TEdit.Create(Self);
  FEdit.Parent := self;
  FEdit.Height := 21;

  FSpeedButton := TSpeedButton.Create(Self);
  FSpeedButton.Left := FEdit.Width;
  // На два пикселя меньше высоты TEdit
  FSpeedButton.Height := 19;
  FSpeedButton.Width  := 19;
  FSpeedButton.Caption := '...';
  FSpeedButton.Parent := Self;

  Width  := FEdit.Width+FSpeedButton.Width;
  Height := FEdit.Height;
end;

destructor  TddgButtonEdit.Destroy;
begin
  FSpeedButton.Free;
  FEdit.Free;
  inherited Destroy;
end;

function TddgButtonEdit.GetText: String;
begin
  Result := FEdit.Text;
end;

procedure TddgButtonEdit.SetText(Value: String);
begin
  FEdit.Text := Value;
end;

function TddgButtonEdit.GetFont: TFont;
begin
  Result := FEdit.Font;
```

```
end;

procedure TddgButtonEdit.SetFont(Value: TFont);
begin
  if Assigned(FEdit.Font) then
    FEdit.Font.Assign(Value);
end;

function TddgButtonEdit.GetOnButtonClick: TNotifyEvent;
begin
  Result := FSpeedButton.OnClick;
end;

procedure TddgButtonEdit.SetOnButtonClick(Value: TNotifyEvent);
begin
  FSpeedButton.OnClick := Value;
end;

end.
```

Компонент TddgDigitalClock — создание событий компонента

Компонент `TddgDigitalClock` иллюстрирует процесс создания событий, определенных пользователем, и предоставления доступа к ним. Здесь используется та же технология, что и при создании событий описанного выше компонента `TddgHalfMinute`.

Компонент `TddgDigitalClock` является производным от компонента `TPanel`. Мы решили, что класс `TPanel` идеально подходит для предка создаваемого компонента, так как содержит свойства `Bevel`*XXXX*, благодаря которым компоненту `TddgDigitalClock` можно придать привлекательный внешний вид. К тому же для отображения системного времени можно использовать свойство `TPanel.Caption`.

Компонент `TddgDigitalClock` содержит следующие события, для обработки которых пользователь может назначить необходимый код:

`OnHour` — происходит каждый час;

`OnHalfPast` — происходит каждые полчаса;

`OnMinute` — происходит каждую минуту;

`OnHalfMinute` — происходит каждые полминуты;

`OnSecond` — происходит каждую секунду.

Компонент `TddgDigitalClock` использует встроенный в него компонент `TTimer`. Его обработчик события `OnTimer` реализует логику отображения информации о времени и вызывает методы диспетчеризации перечисленных выше событий. В листинге 11.15 приведен исходный код модуля `DdgClock.pas`.

Листинг 11.15. `DdgClock.pas` — исходный код компонента `TddgDigitalClock`

```
{$IFDEF VER110}
{$OBJEXPORTALL ON}
{$ENDIF}

unit DDGClock;

interface

uses
  Windows, Messages, Controls, Forms, SysUtils, Classes, ExtCtrls;

type
  { Объявление типа события, которому в качестве параметров будут
    передаваться источник события и переменная TDateTime. }
  TTimeEvent = procedure(Sender: TObject;
                         DDGTime: TDateTime) of object;

  TddgDigitalClock = class(TPanel)
  private
    { Поля данных }
    FHour,
    FMinute,
    FSecond: Word;
    FDateTime: TDateTime;
    FOldMinute,
    FOldSecond: Word;
    FTimer: TTimer;
    { Обработчики событий }
    FOnHour: TTimeEvent;       // Происходит каждый час
    FOnHalfPast: TTimeEvent;   // Происходит каждые полчаса
    FOnMinute: TTimeEvent;     // Происходит каждую минуту
    FOnSecond: TTimeEvent;     // Происходит каждую секунду
    FOnHalfMinute: TTimeEvent; // Происходит каждые 30 секунд
    { Определение обработчика события OnTimer для внутренней
      переменной FTimer типа TTimer. }
    procedure TimerProc(Sender: TObject);
  protected
    { Переопределение метода Paint }
    procedure Paint; override;

    { Определение методов диспетчеризации различных событий. }
    procedure DoHour(Tm: TDateTime); dynamic;
    procedure DoHalfPast(Tm: TDateTime); dynamic;
    procedure DoMinute(Tm: TDateTime); dynamic;
    procedure DoHalfMinute(Tm: TDateTime); dynamic;
    procedure DoSecond(Tm: TDateTime); dynamic;

  public
    { Переопределение конструктора Create и деструктора Destroy. }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Определение свойств-событий. }
```

```
    property OnHour: TTimeEvent read FOnHour write FOnHour;
    property OnHalfPast: TTimeEvent read FOnHalfPast
                                  write FOnHalfPast;
    property OnMinute: TTimeEvent read FOnMinute write FOnMinute;
    property OnHalfMinute: TTimeEvent read FOnHalfMinute
                                    write FOnHalfMinute;
    property OnSecond: TTimeEvent read FOnSecond write FOnSecond;
  end;

implementation

constructor TddgDigitalClock.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); // Вызвать унаследованный конструктор
  Height := 25; // Установить размеры по умолчанию
  Width := 120;
  BevelInner := bvLowered; // Установить стандартные свойства
  BevelOuter := bvLowered;
  { Назначить унаследованному свойству Caption пустую строку. }
  inherited Caption := '';
  { Создать экземпляр компонента TTimer и установить его
    свойство Interval и обработчик события OnTime. }
  FTimer:= TTimer.Create(self);
  FTimer.interval:= 200;
  FTimer.OnTimer:= TimerProc;
end;

destructor TddgDigitalClock.Destroy;
begin
  FTimer.Free;       // Освободить экземпляр компонента TTimer
  inherited Destroy; // Вызывать унаследованный метод Destroy
end;

procedure TddgDigitalClock.Paint;
begin
  inherited Paint; // Вызывать унаследованный метод Paint
  { Теперь унаследованному свойству Caption назначить
    текущее время. }
  inherited Caption := TimeToStr(FDateTime);
end;

procedure TddgDigitalClock.TimerProc(Sender: TObject);
var
  HSec: Word;
begin
  { Сохранить старые значения минут и секунд для дальнейшего
    использования. }
  FOldMinute := FMinute;
  FOldSecond := FSecond;
  FDateTime := Now; // Получить текущее время.
  { Извлечение отдельных составляющих времени }
  DecodeTime(FDateTime, FHour, FMinute, FSecond, Hsec);

  refresh; // Перерисовать компонент для отображения нового
```

```
          // времени.
  { Вызов обработчиков событий времени }
  if FMinute = 0 then
    DoHour(FDateTime);
  if FMinute = 30 then
    DoHalfPast(FDateTime);
  if (FMinute <> FOldMinute) then
    DoMinute(FDateTime);
  if FSecond <> FOldSecond then
    if ((FSecond = 30) or (FSecond = 0)) then
      DoHalfMinute(FDateTime)
    else
      DoSecond(FDateTime);
end;

{ Приведенные ниже методы диспетчеризации событий проверяют наличие
соответствующих обработчиков событий отсчета времени и вызывают их.
}
procedure TddgDigitalClock.DoHour(Tm: TDateTime);
begin
  if Assigned(FOnHour) then
    TTimeEvent(FOnHour)(Self, Tm);
end;

procedure TddgDigitalClock.DoHalfPast(Tm: TDateTime);
begin
  if Assigned(FOnHalfPast) then
    TTimeEvent(FOnHalfPast)(Self, Tm);
end;

procedure TddgDigitalClock.DoMinute(Tm: TDateTime);
begin
  if Assigned(FOnMinute) then
    TTimeEvent(FOnMinute)(Self, Tm);
end;

procedure TddgDigitalClock.DoHalfMinute(Tm: TDateTime);
begin
  if Assigned(FOnHalfMinute) then
    TTimeEvent(FOnHalfMinute)(Self, Tm);
end;

procedure TddgDigitalClock.DoSecond(Tm: TDateTime);
begin
  if Assigned(FOnSecond) then
    TTimeEvent(FOnSecond)(Self, Tm);
end;

end.
```

Логическая структура этого компонента разъясняется в комментариях к исходному коду. Применяемые здесь методы не отличаются от рассмотренных выше при созда-

нии событий. В компонент TddgDigitalClock лишь добавлено бо́льшее количество событий и включена логика для определения того, какое из событий произошло.

Добавление форм в палитру компонентов

Добавление форм в *хранилище объектов* (Object Repository) — удобный способ последующей работы с ними. Но иногда разрабатывают форму, которую приходится часто использовать и которая не предполагает наследования и внесения дополнительных функций. В этом случае Delphi 6 позволяет использовать такие формы, как компоненты палитры компонентов. В частности, компоненты TFontDialog и TOpenDialog — примеры форм, доступных в палитре компонентов. В действительности эти диалоговые окна не являются формами Delphi — они содержатся в библиотеке CommDlg.dll, но, тем не менее, идея остается той же.

Чтобы добавить форму в палитру компонентов, необходимо создать для нее компонент-оболочку, превращающий ее в независимый устанавливаемый компонент. В качестве примера рассмотрим диалоговое окно, автоматически проверяющее пароль пользователя. Это очень простой пример, но нашей целью является демонстрация не установки в качестве компонентов сложных диалоговых окон, а общей идеи добавления диалоговых окон в палитру компонентов. Предлагаемую методику можно применять для добавления в палитру компонентов диалоговых окон любой сложности.

Сначала нужно создать форму, которая впоследствии будет помещена в компонент-оболочку. Используемая здесь форма определена в файле PwDlg.pas. В нем же содержится и код компонента-оболочки.

В листинге 11.16 приведен код модуля PwDlg.pas, содержащего определение формы TPasswordDlg и ее компонента-оболочки TddgPasswordDialog.

Листинг 11.16. PwDlg.pas — форма TPasswordDlg и ее оболочка TddgPasswordDialog

```
unit PwDlg;

interface

uses
  Windows, SysUtils, Classes, Graphics, Forms, Controls, StdCtrls,
  Buttons;

type
  TPasswordDlg = class(TForm)
    Label1: TLabel;
    Password: TEdit;
    OKBtn: TButton;
    CancelBtn: TButton;
  end;

  { Объявление компонента-оболочки. }
  TddgPasswordDialog = class(TComponent)
  private
    PassWordDlg: TPasswordDlg; // Экземпляр TPassWordDlg
```

```
    FPassWord: String;          // Поле для хранения пароля
  public
    function Execute: Boolean; // Функция запуска диалогового окна
  published
    property PassWord: String read FPassword write FPassword;
  end;

implementation
{$R *.DFM}

function TddgPasswordDialog.Execute: Boolean;
begin
  { Создать экземпляр компонента TPasswordDlg }
  PasswordDlg := TPasswordDlg.Create(Application);
  try
    Result := False;  // Инициализация Result значением false
    { Отображение диалогового окна. Если пароль правильный,
      возвращается значение true. }
    if PasswordDlg.ShowModal = mrOk then
      Result := PasswordDlg.Password.Text = FPassword;
  finally
    PasswordDlg.Free;  // Удаление экземпляра PasswordDlg
  end;
end;

end.
```

Компонент `TddgPasswordDialog` называется *компонентом-оболочкой* или просто оболочкой (wrapper), потому что он превращает форму в компонент, который можно установить в палитру компонентов Delphi 6.

Класс компонента `TddgPasswordDialog` является производным непосредственно от класса `TComponent`. Как упоминалось в предыдущей главе, класс `TComponent` — это класс самого низкого уровня, с которым можно работать в конструкторе форм интегрированной среды Delphi. В классе `TddgPasswordDialog` есть две закрытые переменные: `PasswordDlg` типа `TPasswordDlg` и `FPassWord` типа `String`. Переменная `PasswordDlg` — это экземпляр класса `TPasswordDlg`, отображаемый компонентом-оболочкой. Переменная `FPassWord` является *внутренним полем хранения* (internal storage field) строки пароля.

Переменная `FPassWord` получает данные через свойство `PassWord`. Таким образом, в действительности свойство `PassWord` не хранит данных, а лишь служит интерфейсом к переменной `FPassWord`.

Метод `TddgPasswordDialog.Execute()` создает экземпляр объекта `TPasswordDlg` и отображает его в виде модального диалогового окна. Когда диалоговое окно закрывается, строка, введенная в элемент `TEdit`, сравнивается со строкой, хранящейся в переменной `FPassword`.

Приведенный ниже код помещается в конструкцию `try..finally`. Часть кода, находящаяся в блоке `finally`, гарантирует, что память, выделенная для размещения экземпляра компонента `TPasswordDlg`, будет освобождена в любом случае, невзирая на возможные ошибки.

Поместив компонент TddgPasswordDialog в палитру компонентов, можно создать проект, использующий этот компонент. Компонент TddgPasswordDialog можно будет выбирать и помещать в форму так же, как и любой другой компонент. Проект, создание которого описано в предыдущем разделе, содержит компонент TddgPasswordDialog и кнопку, код обработчика события OnClick которого приведен ниже:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ddgPasswordDialog.Execute then      // Запуск PasswordDialog
    ShowMessage('You got it!')           // Правильный пароль
  else
    ShowMessage('Sorry, wrong answer!'); // Неправильный пароль
end;
```

В окне инспектора объектов отображаются три свойства компонента TddgPasswordDialog: Name, Password и Tag. Для использования этого компонента нужно присвоить свойству Password определенное значение. Когда проект будет запущен на выполнение, TddgPasswordDialog получит от пользователя пароль и сравнивает его со значением свойства Password.

Резюме

Знание принципов работы компонентов — основа понимания Delphi. В этой книге еще предстоят встречи со многими пользовательскими компонентами. Надеемся, что теперь компоненты не кажутся “черными ящиками”. В следующей главе рассматриваются еще более сложные способы создания компонентов.

ГЛАВА

12

Создание расширенного компонента VCL

В ЭТОЙ ГЛАВЕ...

В предыдущей главе рассматривались вопросы разработки пользовательских компонентов Delphi. В данном разделе вниманию читателя предлагаются некоторые более сложные методики разработки компонентов, освоив которые можно поднять свое мастерство в этой области на более высокий уровень. Использование новейших технологий демонстрируется на примерах построения псевдовизуальных компонентов, детализированных редакторов свойств, редакторов компонентов и коллекций.

Псевдовизуальные компоненты

Читатель уже знаком с визуальными компонентами, наподобие `TButton` и `TEdit`, и с невизуальными — такими как `TTable` и `TTimer`. В настоящем разделе рассматриваются компоненты, занимающие промежуточное положение между визуальными и невизуальными, — назовем их *псевдовизуальными компонентами* (pseudo-visual components).

Расширенные подсказки

Примером псевдовизуального компонента, рассматриваемого в этом разделе, является расширение *всплывающего окна подсказки* (hint) Delphi. Этот компонент назван псевдовизуальным, поскольку палитра компонентов во время разработки не воспринимает его как визуальный объект, но во время выполнения данный компонент отображается как всплывающее окно подсказки.

Замена обычного стиля окна подсказки новым требует выполнения следующих действий.

1. Создать потомок класса `THintWindow`.
2. Удалить старый класс окна подсказки.
3. Назначить новый класс окна подсказки.
4. Создать новый класс окна подсказки.

Создание потомка класса THintWindow

Прежде чем приступить к написанию кода потомка класса `THintWindow`, следует решить, чем поведение нового класса окна подсказки будет отличаться от поведения обычного окна подсказки. В данном случае вместо обычного прямоугольного окна подсказки будет создано эллиптическое. Попутно можно ознакомиться с интересной методикой — созданием непрямоугольных окон. В листинге 12.1 приводится код модуля `RndHint.pas`, содержащего компонент `TDDGHintWindow`, который является потомком класса `THintWindow`.

Листинг 12.1. `RndHint.pas` — пример эллиптического окна подсказки

```
unit RndHint;

interface

uses Windows, Classes, Controls, Forms, Messages, Graphics;
```

```
type
  TddgHintWindow = class(THintWindow)
  private
    FRegion: THandle;
    procedure FreeCurrentRegion;
  public
    destructor Destroy; override;
    procedure ActivateHint(Rect: TRect;
                           const AHint: string); override;
    procedure Paint; override;
    procedure CreateParams(var Params: TCreateParams); override;
  end;

implementation

destructor TddgHintWindow.Destroy;
begin
  FreeCurrentRegion;
  inherited Destroy;
end;

procedure TddgHintWindow.CreateParams(var Params: TCreateParams);
{ Необходимо удалить стандартную границу (border), созданную на
уровне API Windows при создании окна. }
begin
  inherited CreateParams(Params);
  // Удалить границу
  Params.Style := Params.Style and not ws_Border;
end;

procedure TddgHintWindow.FreeCurrentRegion;
{ Области окна, подобно другим объектам API, должны быть
освобождены по завершении их использования. Запомните, что нельзя
удалить в окне область, являющуюся текущей, поэтому в данном методе
перед удалением объекта области текущей областью назначают
нулевую.}
begin
  if FRegion <> 0 then begin       // Если область существует, то
    SetWindowRgn(Handle, 0, True); // сделать текущей нулевую,
    DeleteObject(FRegion);         // удалить заданную область
    FRegion := 0;                  // и обнулить поле Fregion.
  end;
end;

procedure TddgHintWindow.ActivateHint(Rect: TRect;
                                      const AHint: string);
{ Вызывается при активизации подсказки (при помещении курсора мыши
поверх соответствующего элемента управления). }
begin
  with Rect do                    // добавить немного пространства
    Right := Right + Canvas.TextWidth('WWWW');
  BoundsRect := Rect;
  FreeCurrentRegion;
```

```
  { Создание прямоугольной области со скругленными углами
    для отображения окна подсказки. }
  FRegion := CreateRoundRectRgn(0, 0, Width, Height,
                                      Width, Height);
  if FRegion <> 0 then
    SetWindowRgn(Handle, FRegion, True); // Установка области окна
  inherited ActivateHint(Rect, AHint);   // Вызов унаслед. метода
end;

procedure TddgHintWindow.Paint;
{ Этот метод вызывается обработчиком события WM_PAINT. Он отвечает
за прорисовку окна подсказки. }
var
  R: TRect;
begin
  R := ClientRect;           // Получить размер прямоугольника,
  Inc(R.Left, 1);            // немного сдвинуть левую сторону
  Canvas.Brush.Color := clInfoBk;  // установить цвет фона области
  FillRgn(Canvas.Handle, FRegion, Canvas.Brush.Handle);
  Canvas.Font.Color := clInfoText; // установить цвет текста
  { Вывести строку в центре скругленного прямоугольника }
  DrawText(Canvas.Handle, PChar(Caption), Length(Caption), R,
        DT_NOPREFIX or DT_WORDBREAK or DT_CENTER or DT_VCENTER);
end;

var
  OldHintClass: THintWindowClass;

function SetNewHintClass(AClass:
                         THintWindowClass): THintWindowClass;
var
  DoShowHint: Boolean;
begin
  // Возвращаемое значение - старое окно подсказки
  Result := HintWindowClass;
  DoShowHint := Application.ShowHint;
  if DoShowHint then
    Application.ShowHint := False;// удалить старое окно подсказки
  HintWindowClass := AClass;     // назначить новое окно подсказки
  if DoShowHint then
    Application.ShowHint := True; // создать новое окно подсказки
end;

initialization
  OldHintClass := SetNewHintClass(TddgHintWindow);

finalization
  SetNewHintClass(OldHintClass);

end.
```

Организация переопределенных методов `CreateParams()` и `Paint()` очевидна. Метод `CreateParams()` позволяет настроить структуру стилей окна перед тем, как оно будет создано на уровне API. В этом методе с помощью операции `not` удаляется по маске стиль `WS_BORDER` для того, чтобы вокруг окна не прорисовывалась прямоугольная граница. Метод `Paint()` отвечает за прорисовку окна. В данном случае метод должен поместить свойство подсказки `Caption` в центр заголовка окна. Для текста установлен цвет `clInfoText`, который является определенным системой цветом подсказки.

Эллиптическое окно

Вся магия создания непрямоугольных окон заключена в методе `ActivateHint()`. В действительности же это вовсе не магия, а просто два вызова функций API Win32: `CreateRoundRectRgn()` и `SetWindowRgn()`.

Функция `CreateRoundRectRgn()` определяет прямоугольную область со скругленными углами внутри заданного окна. *Область* (region) — это специальный объект API, позволяющий выполнить прорисовку, закрашивание, обрезание и анализ необходимости вывода подсказки в некоторой области. Помимо функции `CreateRoundRectRgn()`, существуют и другие функции API Win32, предназначенные для создания областей различных типов:

- `CreateEllipticRgn()`
- `CreateEllipticRgnIndirect()`
- `CreatePolygonRgn()`
- `CreatePolyPolygonRgn()`
- `CreateRectRgn()`
- `CreateRectRgnIndirect()`
- `CreateRoundRectRgn()`
- `ExtCreateRegion()`

Кроме того, можно использовать функцию `CombineRgn()` для объединения нескольких различных областей в один сложный участок. Все эти функции подробно описаны в интерактивной справке по интерфейсу API Win32.

Затем вызывается функция `SetWindowRgn()`, которой в качестве параметра передается дескриптор вновь созданной области. Эта функция заставляет операционную систему стать владельцем области, и все последующие перерисовки для данного окна будут происходить только внутри указанной области. Таким образом, если область определена как скругленный прямоугольник, перерисовка будет выполняться только в нем.

СОВЕТ

Стоит упомянуть о двух побочных эффектах применения функции `SetWindowRgn()`. Во-первых, поскольку перерисовывается лишь часть окна, то у него, вероятно, не будет рамки и заголовка. Следует быть готовым к тому, что придется обеспечить пользователя альтернативным способом перемещения, изменения размера и закрытия окна — без помощи рамки и заголовка. Во-вторых, поскольку областью, определенной в функции `SetWindowRgn()`, владеет операционная система, обращаться с ней необходимо осторожно, т.е. не изменять и не удалять ее во время использования. В компоненте `TDDGHintWindow` сначала вызывается метод `FreeCurrentRegion()` и лишь затем уничтожается старое или создается новое окно.

Активизация потомка класса THintWindow

Код инициализации модуля RndHint делает компонент TDDGHintWindow активным окном подсказки для всего приложения. Установка свойства Application.ShowHint в состояние False приводит к отмене старого окна подсказки. Далее следует назначить класс, являющийся потомком класса THintWindow, глобальной переменной HintWindowClass. Теперь можно установить свойство Application.ShowHint в состояние True. В результате этого будет создано новое окно, представляющее собой экземпляр пользовательского класса.

Применение TddgHintWindow

Применение псевдовизуального компонента отличается от методов использования обычных визуальных или невизуальных компонентов. Поскольку вся работа по созданию экземпляра компонента выполняется в разделе инициализации (initialization) его модуля, компонент не может быть добавлен в пакет разработки и установлен в палитру компонентов. Поэтому необходимо просто добавлять имя его модуля в раздел uses исходных файлов проекта.

Анимационные компоненты

Завершая разработку очередного приложения в Delphi, мы были вполне удовлетворены полученными результатами, вот только окно About казалось нам несколько "скучным". Необходимо было что-то придумать. Внезапно кому-то из нас пришла в голову мысль создать новый компонент, который позволил бы включать в диалоговое окно About бегущую полосу с титрами.

Компонент строки титров

Давайте уделим некоторое время анализу того, как работает компонент титров. Он может прокручивать пакет строк в окне компонента как титры в конце фильма. В качестве базового класса для компонента TddgMarquee следует использовать класс TCustomPanel, так как он обладает необходимыми основными возможностями и красивой трехмерной рамкой.

Компонент TddgMarquee переводит текстовые строки в растровое изображение, находящееся в памяти, и затем копирует части этого изображения на свою поверхность для имитации эффекта прокрутки. Для этого используется функция API BitBlt(), копирующая часть изображения из памяти на поверхность элемента управления (в соответствии с его размером), начиная с верхней части. Затем данная область перемещается на несколько пикселей вниз и снова копируется из памяти. Копирование и смещение повторяется до тех пор, пока не будет достигнут конец изображения в памяти.

Теперь необходимо решить, какие дополнительные классы следует интегрировать в компонент TddgMarquee, чтобы он стал анимационным. Во-первых, для хранения прокручиваемых строк потребуется объект TStringList. Во-вторых, в памяти должно содержаться растровое изображение прокручиваемых строк. Для этого вполне подойдет компонент VCL TBitmap.

Создание кода компонента

Как и для обычных компонентов, сначала необходимо составить план класса TddgMarquee. Разобьем задачу на несколько подзадач. Логически компонент TddgMarquee может быть разделен на следующие пять частей.

- Механизм переноса текста в растровое изображение, хранящееся в памяти.
- Механизм копирования изображения текста из памяти на поверхность компонента.
- Таймер, управляющий процессом прокрутки.
- Конструктор, деструктор класса и связанные с ними методы.
- Завершающие штрихи в виде вспомогательных свойств и методов.

Создание изображения в памяти

Создавая экземпляр класса TBitmap, следует заранее узнать его размер, чтобы он мог разместить в памяти весь список строк. Этот размер можно определить, умножив высоту строки на их общее количество. Функция API GetTextMetrics(), которой в качестве параметра передается дескриптор объекта Canvas (поверхности), предназначена для определения размеров строки текста, отображаемого конкретным шрифтом. В процессе выполнения эта функция заполняет данными запись TTextMetric:

```
var
  Metrics: TTextMetric;
begin
  GetTextMetrics(Canvas.Handle, Metrics);
```

НА ЗАМЕТКУ

Функция API GetTextMetrics() соответствующим образом модифицирует запись типа TTextMetric, содержащую много полезной информации о выбранном в данный момент шрифте контекста устройства. Эта функция предоставляет информацию не только о высоте и ширине шрифта, но и о его начертании: полужирном, курсиве, перечеркнутом, о названии шрифта.
Метод TextHeight() класса TCanvas здесь не работает. Этот метод определяет только размеры конкретной строки текста, а не шрифта в целом.

Высота ячейки символа в текущем шрифте хранится в поле tmHeight записи Metric. Если сложить это значение со значением поля tmInternalLeading, хранящим межстрочное расстояние выбранного шрифта, то получается высота строки текста *растрового изображения в памяти* (memory canvas):

```
LineHi := Metrics.tmHeight + Metrics.tmInternalLeading;
```

Общая высота изображения в памяти определяется умножением значения LineHi на количество строк и добавлением удвоенной высоты элемента управления TddgMarquee для создания пустого изображения в начале и конце титров. Предположим, что элемент типа TStringList, в котором находятся все строки, носит имя FItems. А теперь поместим все размеры изображения в структуру TRect:

```
var
  VRect: TRect;
begin
  { Прямоугольник Vrect - это растровое изображение в памяти }
  VRect := Rect(0, 0, Width, LineHi * FItems.Count + Height * 2);
end;
```

После того как в памяти будет создан экземпляр растрового изображения необходимого размера, выполняется его дальнейшая инициализация, заключающаяся в установке шрифта (значение свойства `TddgMarquee.Font`), цвета фона (`TddgMarquee.Color`) и стиля (`Brush.Style`) равным значению `bsClear`.

СОВЕТ

При размещении текста в объекте класса `TCanvas` цвет фона определяется значением свойства `TCanvas.Brush`. Для того чтобы фон был невидимым, нужно установить свойство `TCanvas.Brush.Style` равным значению `bsClear`.

Итак, основная работа завершена, и теперь можно поместить текст в растровое изображение в памяти. Проще всего сделать это с помощью метода `TextOut()` класса `TCanvas`, но применение функции API `DrawText()` предоставляет существенно бо*л*ьшие возможности по управлению форматированием текста. Поскольку в компоненте `TddgMarquee` необходимо управлять выравниванием, будет использована именно эта функция. Для представления стилей выравнивания текста идеально подходит переменная перечислимого типа:

```
type

  TJustification = (tjCenter, tjLeft, tjRight);
```

В следующем фрагменте кода показан метод `PaintLine()` класса `TddgMarquee`, позволяющий поместить текст в изображение в памяти. В данном методе используется переменная `FJust` типа `TJustification`. Вот этот код:

```
procedure TddgMarquee.PaintLine(R: TRect; LineNum: Integer);
{ Этот метод помещает строки текста в область памяти MemBitmap. }
const
  Flags: array[TJustification] of DWORD = (DT_CENTER,
                                           DT_LEFT, DT_RIGHT);
var
  S: string;
begin
  { Упростим код, скопировав очередную строку в локальную
    переменную }
  S := FItems.Strings[LineNum];
  { Прорисовать строку текста в изображение, размещенное в памяти}
  DrawText(MemBitmap.Canvas.Handle, PChar(S), Length(S), R,
           Flags[FJust] or DT_SINGLELINE or DT_TOP);
end;
```

Прорисовка компонента

Теперь, когда уже известно, как создать растровое изображение в памяти и помещать в него текст, можно приступить к копированию этого изображения на поверхность компонента TddgMarquee.

Метод Paint() компонента вызывается в ответ на сообщение Windows WM_PAINT. Именно метод Paint() создает эффект мультипликации. Он позволяет рисовать на поверхности компонента, формируя его визуальное представление.

Задача метода TddgMarquee.Paint() заключается в копировании строк изображения из памяти на поверхность компонента TddgMarquee. Для этого используется функция API BitBlt(), копирующая содержимое контекста одного устройства в контекст другого.

Для определения текущего состояния компонента TddgMarquee используется логическая переменная FActive, указывающая, активизирован ли в данный момент процесс прокрутки. Очевидно, что метод Paint() должен работать по-разному, в зависимости от того, активен данный компонент или нет:

```
procedure TddgMarquee.Paint;
{ Этот виртуальный метод вызывается в ответ на сообщение Windows о
необходимости перерисовки. }
begin
  if FActive then
    { Копировать из памяти на экран. }
    BitBlt(Canvas.Handle, 0, 0, InsideRect.Right,
           InsideRect.Bottom, MemBitmap.Canvas.Handle,
           0, CurrLine, srcCopy)
  else
    inherited Paint;
end;
```

Если компонент активен, то с помощью функции BitBlt() часть хранимого в памяти изображения отображается на экране в свойстве Canvas объекта TddgMarquee. Обратите внимание на переменную CurrLine, передаваемую функции BitBlt() как предпоследний параметр. Значение этого параметра определяет, какую часть изображения в памяти нужно выводить на экран. Постоянно увеличивая или уменьшая значение CurrLine, можно добиться эффекта прокрутки текста в компоненте TddgMarquee вниз или вверх.

Анимация титров

Итак, визуальные аспекты работы компонента TddgMarquee реализованы. Для того чтобы этот компонент заработал, осталось сделать совсем немного. Следует разработать механизм, изменяющий значение CurrLine во времени и запускающий в нужный момент перерисовку компонента. Этого легко добиться, используя компонент Delphi TTimer.

Естественно, перед тем как использовать компонент TTimer, необходимо создать и инициализировать экземпляр этого класса. Компонент TddgMarquee будет обладать собственным экземпляром класса TTimer по имени FTimer, который инициализируется в процедуре DoTimer:

```
procedure DoTimer;
{ Процедура установки таймера компонента TddgMarquee }
begin
  FTimer := TTimer.Create(Self);
  with FTimer do begin
    Enabled := False;
    Interval := TimerInterval;
    OnTimer := DoTimerOnTimer;
  end;
end;
```

В этой процедуре сначала создается неактивный объект `FTimer`. Его свойству `Interval` присваивается значение константы `TimerInterval`. Затем событию `OnTimer` объекта `FTimer` присваивается метод `DoTimerOnTimer` класса `TddgMarquee`. Этот метод будет вызываться при наступлении каждого события `OnTimer`.

НА ЗАМЕТКУ

Присваивая в коде значения событию, необходимо следовать двум правилам.

- Процедура, которая присваивается событию, должна быть методом некоторого объекта (экземпляра). Это не может быть самостоятельная процедура или функция.
- Метод, который присваивают событию, должен иметь тот же список параметров, что и тип события. Например, событие `OnTimer` компонента `TTimer` имеет тип `TNotifyEvent`. Поскольку у `TNotifyEvent` — всего один параметр (`Sender`) типа `TObject`, то любой метод, присваиваемый событию `OnTimer`, должен иметь один параметр типа `TObject`.

Метод `DoTimerOnTimer()` определяется следующим образом:

```
procedure TddgMarquee.DoTimerOnTimer(Sender: TObject);
{ Этот метод выполняется в ответ на событие таймера. }
begin
  IncLine;
  { Перерисовка только внутри границ }
  InvalidateRect(Handle, @InsideRect, False);
end;
```

В этом методе вызывается процедура `IncLine()`, увеличивающая или уменьшающая значение `CurrLine`. Затем функция API `InvalidateRect()` вызывается для *перерисовки* (invalidate или repaint) внутренней части компонента. Авторы предпочли использовать функцию `InvalidateRect()` вместо метода `Invalidate()` класса `TCanvas`, поскольку последний приводит к перерисовке всего компонента, а эта функция — только его части. Это позволяет избавиться от неприятного мерцания (flicker), связанного с перерисовкой всего окна компонента. Запомните: мерцания следует избегать!

Исходный код метода `IncLine()`, обновляющего значение `CurrLine` и определяющего конец прокрутки, приводится ниже.

```
procedure TddgMarquee.IncLine;
{ Этот метод вызывается для приращения индекса текущей строки. }
begin
  if not FScrollDown then begin // если титры прокручиваются вверх
    { Проверка конца прокрутки. }
```

```
    if FItems.Count * LineHi + ClientRect.Bottom -
                               ScrollPixels >= CurrLine then
      { Если еще не конец, то приращение индекса текущей строки. }
      Inc(CurrLine, ScrollPixels)
    else SetActive(False);
  end
  else begin                   // если титры прокручиваются вниз
    { Проверка конца прокрутки. }
    if CurrLine >= ScrollPixels then
      { Если еще не конец, то уменьшение индекса текущей строки. }
      Dec(CurrLine, ScrollPixels)
    else SetActive(False);
  end;
end;
```

Конструктор класса TddgMarquee довольно прост. Он вызывает унаследованный метод Create(), создает экземпляр класса TStringList, устанавливает таймер FTimer и значения по умолчанию для переменных экземпляра. Еще раз напоминаем о необходимости использовать в компонентах унаследованный конструктор Create(). Без него компоненты будут лишены таких немаловажных элементов, как дескриптор и свойство Canvas, возможности работы с потоками данных и взаимодействия с сообщениями Windows. Вот исходный код конструктора Create() класса TddMarquee:

```
constructor TddgMarquee.Create(AOwner: TComponent);
{ Конструктор класса TddgMarquee }

  procedure DoTimer;
  { Процедура установки таймера для класса TddgMarquee }
  begin
    FTimer := TTimer.Create(Self);
    with FTimer do begin
      Enabled := False;
      Interval := TimerInterval;
      OnTimer := DoTimerOnTimer;
    end;
  end;

begin
  inherited Create(AOwner);
  { Создание экземпляра списка строк }
  FItems := TStringList.Create;
  DoTimer;                      // Установка таймера
  { Установка значений по умолчанию для экземпляра }
  Width := 100;
  Height := 75;
  FActive := False;
  FScrollDown := False;
  FJust := tjCenter;
  BevelWidth := 3;
end;
```

Деструктор TddgMarquee еще проще: он дезактивирует компонент, передавая в метод SetActive() параметр False, освобождает память занимаемую таймером и списком строк, а затем вызывает унаследованный метод Destroy():

```
destructor TddgMarquee.Destroy;
{ Деструктор класса TddgMarquee }
begin
  SetActive(False);
  FTimer.Free;     // Освобождение памяти,
  FItems.Free;     // выделенной для объектов.
  inherited Destroy;
end;
```

СОВЕТ

Советуем придерживаться правила, согласно которому при переопределении конструкторов унаследованный конструктор вызывается в начале, а при переопределении деструкторов унаследованный деструктор вызывается в самом конце. Это гарантирует, что класс будет создан до его модификации и все зависимые ресурсы будут освобождены до освобождения класса.
Естественно, из данного правила есть исключения, но для этого нужны достаточно веские причины.

Метод SetActive(), служащий методом записи свойства Active и вызываемый методом IncLine() и деструктором, является механизмом, обеспечивающим начало и останов прокрутки титров:

```
procedure TddgMarquee.SetActive(Value: Boolean);
{ Вызывается для активации/дезактивации прокрутки титров }
begin
  if Value and (not FActive) and (FItems.Count > 0) then begin
    FActive := True;          // Установка флага активности
    MemBitmap := TBitmap.Create;
    FillBitmap;               // Прорисовка растрового изображения
    FTimer.Enabled := True;   // Запуск таймера
  end
  else if (not Value) and FActive then begin
    FTimer.Enabled := False; // Отключить таймер,
    if Assigned(FOnDone) then
      FOnDone(Self);         // передать событие OnDone,
    FActive := False;        // установить FActive в False,
    MemBitmap.Free;          // освободить память изображения,
    Invalidate;              // очистить окно элемента управления.
  end;
end;
```

У компонента TddgMarquee пока отсутствует важное событие, которое сообщало бы пользователю о конце прокрутки. Ничего страшного, это событие FOnDone легко добавить в наш компонент. Вначале будет объявлена переменная экземпляра события некоторого типа в разделе private определения класса. Событие FOnDone будет иметь тип TNotifyEvent:

```
FOnDone: TNotifyEvent;
```

Затем это событие следует объявить в качестве свойства в разделе published определения класса:

```
property OnDone: TNotifyEvent read FOnDone write FOnDone;
```

Вспомним, что директивы read и write здесь определяют функции или переменные для установки и получения значения свойства.

Этих строк кода достаточно, чтобы свойство OnDone появилось во вкладке Events окна инспектора объектов во время разработки. Осталось лишь вызвать пользовательский обработчик события OnDone (как метод, назначенный событию OnDone), что и делается в коде метода Deactivate() компонента TddgMarquee:

```
if Assigned(FOnDone) then FOnDone(Self); //Передача события OnDone
```

Данную строку нужно понимать следующим образом: "Если пользователь компонента назначил событию OnDone какой-то метод, то именно этот метод и нужно вызвать, передав ему текущий экземпляр класса TddgMarquee (т.е. Self) в качестве параметра".

В листинге 12.2 приведен полный исходный код модуля Marquee. Обратите внимание, поскольку рассматриваемый компонент является потомком класса TCustom*XXX*, большинство свойств компонента TCustomPanel необходимо сделать публикуемыми.

Листинг 12.2. Marquee.pas — компонент TddgMarquee

```
unit Marquee;

interface

uses
  SysUtils, Windows, Classes, Forms, Controls, Graphics,
  Messages, ExtCtrls, Dialogs;

const
  ScrollPixels = 3;   // Количество пикселей для каждой прокрутки
  TimerInterval = 50; // Время между прокрутками в м/с

type
  TJustification = (tjCenter, tjLeft, tjRight);

  EMarqueeError = class(Exception);

  TddgMarquee = class(TCustomPanel)
  private
    MemBitmap: TBitmap;
    InsideRect: TRect;
    FItems: TStringList;
    FJust: TJustification;
    FScrollDown: Boolean;
    LineHi : Integer;
    CurrLine : Integer;
    VRect: TRect;
```

```
    FTimer: TTimer;
    FActive: Boolean;
    FOnDone: TNotifyEvent;
    procedure SetItems(Value: TStringList);
    procedure DoTimerOnTimer(Sender: TObject);
    procedure PaintLine(R: TRect; LineNum: Integer);
    procedure SetLineHeight;
    procedure SetStartLine;
    procedure IncLine;
    procedure SetActive(Value: Boolean);
  protected
    procedure Paint; override;
    procedure FillBitmap; virtual;
  public
    property Active: Boolean read FActive write SetActive;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property ScrollDown: Boolean read FScrollDown
                               write FScrollDown;
    property Justify: TJustification read Fjust
                                  write FJust default tjCenter;
    property Items: TStringList read FItems write SetItems;
    property OnDone: TNotifyEvent read FOnDone write FOnDone;
    { Публикация унаследованных свойства: }
    property Align;
    property Alignment;
    property BevelInner;
    property BevelOuter;
    property BevelWidth;
    property BorderWidth;
    property BorderStyle;
    property Color;
    property Ctl3D;
    property Font;
    property Locked;
    property ParentColor;
    property ParentCtl3D;
    property ParentFont;
    property Visible;
    property OnClick;
    property OnDblClick;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnResize;
  end;

implementation

constructor TddgMarquee.Create(AOwner: TComponent);
{ Конструктор класса TddgMarquee }
```

```
  procedure DoTimer;
  { Процедура установки таймера для класса TddgMarquee }
  begin
    FTimer := TTimer.Create(Self);
    with FTimer do begin
      Enabled := False;
      Interval := TimerInterval;
      OnTimer := DoTimerOnTimer;
    end;
  end;

begin
  inherited Create(AOwner);
  { Создание экземпляра списка строк }
  FItems := TStringList.Create;
  DoTimer;                        // Установка таймера
  { Установка значений по умолчанию для экземпляра }
  Width := 100;
  Height := 75;
  FActive := False;
  FScrollDown := False;
  FJust := tjCenter;
  BevelWidth := 3;
end;

destructor TddgMarquee.Destroy;
{ Деструктор класса TddgMarquee }
begin
  SetActive(False);
  FTimer.Free;     // Освобождение памяти,
  FItems.Free;     // выделенной для объектов.
  inherited Destroy;
end;

procedure TddgMarquee.DoTimerOnTimer(Sender: TObject);
{ Этот метод выполняется в ответ на событие таймера. }
begin
  IncLine;
  { Перерисовка только внутри границ. }
  InvalidateRect(Handle, @InsideRect, False);
end;

procedure TddgMarquee.IncLine;
{ Этот метод вызывается для приращения индекса текущей строки. }
begin
  if not FScrollDown then begin // если титры прокручиваются вверх
    { Проверка конца прокрутки. }
    if FItems.Count * LineHi + ClientRect.Bottom -
                               ScrollPixels  >= CurrLine then
      { Если еще не конец, то приращение индекса текущей строки. }
      Inc(CurrLine, ScrollPixels)
    else SetActive(False);
  end
```

```
  else begin                    // если титры прокручиваются вниз
    { Проверка конца прокрутки. }
    if CurrLine >= ScrollPixels then
      { Если еще не конец, то уменьшение индекса текущей строки. }
      Dec(CurrLine, ScrollPixels)
    else SetActive(False);
  end;
end;

procedure TddgMarquee.SetItems(Value: TStringList);
begin
  if FItems <> Value then
    FItems.Assign(Value);
end;

procedure TddgMarquee.SetLineHeight;
{ Этот виртуальный метод устанавливает переменную экземпляра
LineHi. }
var
  Metrics: TTextMetric;
begin
  { Получить размеры шрифта }
  GetTextMetrics(Canvas.Handle, Metrics);
  { Вычисление высоты строки }
  LineHi := Metrics.tmHeight + Metrics.tmInternalLeading;
end;

procedure TddgMarquee.SetStartLine;
{ Этот виртуальный метод инициализирует переменную CurrLine
экземпляра. }
begin
  // Инициализация текущей строки в начало списка, если прокрутка
  // вверх...
  if not FScrollDown then CurrLine := 0
  // ...или в конец списка, если прокрутка вниз.
  else CurrLine := VRect.Bottom - Height;
end;

procedure TddgMarquee.PaintLine(R: TRect; LineNum: Integer);
{ Этот метод помещает строки текста в область памяти MemBitmap. }
const
  Flags: array[TJustification] of DWORD = (DT_CENTER,
                                           DT_LEFT, DT_RIGHT);
var
  S: string;
begin
  { Упростим код, скопировав очередную строку в локальную
    переменную. }
  S := FItems.Strings[LineNum];
  { Прорисовать строку текста в изображение, размещенное в памяти}
  DrawText(MemBitmap.Canvas.Handle, PChar(S), Length(S), R,
           Flags[FJust] or DT_SINGLELINE or DT_TOP);
end;
```

```
procedure TddgMarquee.FillBitmap;
var
  y, i : Integer;
  R: TRect;
begin
  SetLineHeight;                  // Установка высоты каждой строки
  { Прямоугольник Vrect - это растровое изображение в памяти }
  VRect := Rect(0, 0, Width, LineHi * FItems.Count + Height * 2);
  { Прямоугольник InsideRect представляет собой внутреннюю часть
    прямоугольной области. }
  InsideRect := Rect(BevelWidth, BevelWidth,
                     Width - (2 * BevelWidth),
                     Height - (2 * BevelWidth));
  R := Rect(InsideRect.Left, 0, InsideRect.Right, VRect.Bottom);
  SetStartLine;
  MemBitmap.Width := Width;  // Инициализация изображения в памяти
  with MemBitmap do begin
    Height := VRect.Bottom;
    with Canvas do begin
      Font := Self.Font;
      Brush.Color := Color;
      FillRect(VRect);
      Brush.Style := bsClear;
    end;
  end;
  y := Height;
  i := 0;
  repeat
    R.Top := y;
    PaintLine(R, i);
    { Увеличить y на высоту строки (в пикселях). }
    inc(y, LineHi);
    inc(i);
  until i >= FItems.Count;     // повторить для всех строк
end;

procedure TddgMarquee.Paint;
{ Этот виртуальный метод вызывается в ответ на сообщение Windows о
необходимости перерисовки. }
begin
  if FActive then              // Копировать из памяти на экран
    BitBlt(Canvas.Handle, 0, 0, InsideRect.Right,
           InsideRect.Bottom, MemBitmap.Canvas.Handle,
           0, CurrLine, srcCopy)
  else
    inherited Paint;
end;

procedure TddgMarquee.SetActive(Value: Boolean);
{ Вызывается для активации/дезактивации прокрутки титров. }
begin
  if Value and (not FActive) and (FItems.Count > 0) then begin
```

```
    FActive := True;          // Установка флага активности
    MemBitmap := TBitmap.Create;
    FillBitmap;               // Прорисовка растрового изображения
    FTimer.Enabled := True;   // Запуск таймера
  end
  else if (not Value) and FActive then begin
    FTimer.Enabled := False; // Отключить таймер,
    if Assigned(FOnDone) then
      FOnDone(Self);         // передать событие OnDone,
    FActive := False;        // установить FActive в False,
    MemBitmap.Free;          // освободить память изображения,
    Invalidate;              // очистить окно элемента управления.
  end;
end;

end.
```

СОВЕТ

Обратите внимание на использование директивы `default` со свойством `Justify` компонента `TddgMarquee`. Использование директивы `default` оптимизирует работу компонента с потоками данных, что повышает производительность во время разработки. Значения по умолчанию можно создавать для свойств любого упорядоченного типа (`Integer`, `Word`, `Longint`, а также, например, для перечислимых типов), но для свойств неупорядоченных типов, таких как строки, числа с плавающей точкой, массивы, записи и классы, значения по умолчанию не могут быть созданы.
Кроме того, инициализировать значения по умолчанию для свойств необходимо в конструкторе. В противном случае возможны проблемы при работе с потоками данных.

Проверка компонента TddgMarquee

Хотя компонент уже практически написан и находится на стадии тестирования, охладите свой пыл и пока не устанавливайте его в палитру компонентов. Сначала его следует отладить. Необходимо выполнить полную предварительную проверку компонента, разработав специальный проект, создающий и использующий динамический экземпляр нового компонента. В листинге 12.3 содержится код главного модуля проекта `TestMarq`, предназначенного для проверки компонента `TddgMarquee`. Этот простой проект состоит из формы с двумя кнопками.

Листинг 12.3. `TestU.pas` — проверка компонента `TddgMarquee`

```
unit Testu;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, Marquee, StdCtrls, ExtCtrls;

type
```

```
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    Marquee1: TddgMarquee;
    procedure MDone(Sender: TObject);
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.MDone(Sender: TObject);
begin
  Beep;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Marquee1 := TddgMarquee.Create(Self);
  with Marquee1 do begin
    Parent := Self;
    Top := 10;
    Left := 10;
    Height := 200;
    Width := 150;
    OnDone := MDone;
    Show;
    with Items do begin
      Add('Greg');
      Add('Peter');
      Add('Bobby');
      Add('Marsha');
      Add('Jan');
      Add('Cindy');
    end;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Marquee1.Active := True;
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
  Marquee1.Active := False;
end;

end.
```

СОВЕТ

Всегда создавайте пробные проекты для всех новых компонентов. *Никогда* не пытайтесь выполнить входной контроль компонента, добавив его в палитру компонентов. Отлаживая компонент, находящийся в палитре, можно не только потерять много времени на бесполезную перекомпоновку пакета, но и, возможно, столкнуться с зависанием интегрированной среды разработки Delphi из-за ошибки в новом компоненте.

Только после устранения всех ошибок в новом компоненте его можно добавить в палитру компонентов. Как уже было сказано, это довольно просто: достаточно выбрать в меню Component пункт Install Component, а затем ввести имена модуля и пакета в раскрывшемся диалоговом окне Install Component. Щелкните на кнопке OK — и Delphi перекомпонует тот пакет, в который помещается новый компонент, и обновит вид палитры компонентов. Конечно, для того чтобы новой компонент мог оказаться в палитре компонентов, ему потребуется процедура `Register()`. Компонент `TddgMarquee` регистрируется в модуле `DDGReg.pas` пакета `DDGDsgn`, который находится на прилагаемом компакт-диске.

Создание редакторов свойств

В главе 11, "Разработка компонентов VCL", описано, как в окне инспектора объектов выполняется редактирование свойств самых распространенных типов. Средство, с помощью которого выполняется редактирование свойства, называется *редактором свойств* (property editor). Для уже существующих свойств предусмотрено несколько заранее созданных редакторов. Но возможна ситуация, когда ни один из уже существующих редакторов применить не удастся — например, при создании пользовательского свойства. В этом случае потребуется разработать собственный редактор свойств.

Редактировать свойства в окне инспектора объектов можно двумя способами. Один заключается в предоставлении пользователю возможности редактирования свойства как строки текста. Другой требует создания специального диалогового окна, в котором и выполняется редактирование свойства. В некоторых случаях потребуется использовать оба способа для редактирования одного свойства.

Вот основные этапы создания редактора свойств.

1. Создание объекта класса, производного от класса редактора свойств.
2. Редактирование свойства как текста.
3. Редактирование свойства в диалоговом окне (необязательный этап).
4. Определение атрибутов редактора свойств.
5. Регистрация редактора свойств.

Все эти этапы подробно рассматриваются в следующих разделах.

Создание потомка редактора свойств

Delphi определяет несколько редакторов свойств в модуле `DesignEditors.pas`, причем все они происходят от базового класса `TPropertyEditor`. Создаваемый редактор свойства также должен происходить от него или его потомков. В табл. 12.1 перечислены классы, производные от `TPropertyEditor`, используемые для редактирования свойств уже существующих типов.

Таблица 12.1. Редакторы свойств, определенные в модуле `DesignEditors.pas`

Редактор свойств	*Описание*
`TOrdinalProperty`	Базовый класс для всех редакторов свойств обычных типов, таких как `TIntegerProperty`, `TEnumProperty`, `TCharProperty` и т.п.
`TIntegerProperty`	Редактор целочисленных свойств любых размеров
`TCharProperty`	Редактор символьных свойств типа `char` и символьных наборов (например 'A'..'Z')
`TEnumProperty`	Редактор свойств для всех пользовательских перечислимых типов
`TFloatProperty`	Редактор свойств вещественного типа (чисел с плавающей точкой)
`TStringProperty`	Стандартный редактор свойств строковых типов
`TSetElementProperty`	Стандартный редактор свойств для отдельных элементов множеств. Каждый элемент рассматривается как независимый логический тип
`TSetProperty`	Редактор свойств, являющихся множеством. Множество редактируется по отдельным элементам
`TClassProperty`	Редактор свойств, представляющих собой объекты
`TMethodProperty`	Редактор свойств, являющихся указателями на методы, т.е. событиями
`TComponentProperty`	Редактор свойств, ссылающихся на компоненты. Это не то же самое, что и редактор `TClassProperty`. Данный редактор позволяет пользователю задать компонент, на который ссылается свойство, т.е. `ActiveControl`
`TColorProperty`	Редактор свойств типа `TColor` (цвет)
`TFontNameProperty`	Редактор свойств, содержащих имя шрифта. Этот редактор отображает раскрывающийся список названий шрифтов, доступных в данной системе
`TFontProperty`	Редактор свойств типа `TFont`, позволяющий редактировать подчиненные свойства (subproperties), поскольку он является производным от класса `TClassProperty`

Окончание табл. 12.1.

Редактор свойств	*Описание*
`TInt64Property`	Стандартный редактор свойства для всех типов `Int64` и его производных
`TNestedProperty`	Этот редактор свойств способен использовать родительский редактор свойств
`TClassProperty`	Стандартный редактор свойства для объектов
`TMethodProperty`	Стандартный редактор свойства для методов
`TInterfaceProperty`	Стандартный редактор свойства для обращений к интерфейсу
`TComponentNameProperty`	Редактор имен свойств. Этот редактор отключается, если выбрано более одного компонента
`TDateProperty`	Стандартный редактор свойств для раздела даты в свойствах типа `TDateTime`
`TTimePropery`	Редактор свойств для раздела времени свойств типа `TDateTime`
`TDateTimeProperty`	Редактор свойств типа `TDateTime`
`TVariantProperty`	Редактор свойств для типов `variant`

Выбор базового класса для создаваемого редактора свойств основан на желаемом поведении свойства при его редактировании. Например, свойство может обладать теми же функциональными возможностями, которые присущи компоненту `TIntegerProperty`, но при этом необходима дополнительная логика редактирования. Следовательно, в качестве базового класса создаваемого редактора свойств целесообразно использовать класс `TIntegerProperty`.

СОВЕТ

Иногда в создании редактора свойства для специфического типа данных нет никакой необходимости. Например, типу диапазонов автоматически назначается необходимый редактор (скажем, набор значений `1..10` будет передан компоненту `TIntegerProperty`), для перечислимых типов автоматически используются раскрывающиеся списки и т.д. Предпочтительней применять определения типов, а не пользовательские редакторы свойств, так как они отлично поддерживаются и языком программирования во время компиляции, и используемыми по умолчанию стандартными редакторами свойств.

Редактирование свойства как текста

Редактор свойства используется для двух основных целей. Первая достаточно ясна и состоит в том, чтобы позволить пользователю отредактировать значение свойства. Другая, не столь очевидная, — обеспечить *строковое представление* значения свойства в окне инспектора объектов.

Создавая производный класс редактора свойств, необходимо переопределить методы `GetValue()` и `SetValue()`. Метод `GetValue()` возвращает строковое представление значения свойства для отображения в окне инспектора объектов. Метод `SetValue()` осуществляет присвоение свойству значения на основании его строкового представления в окне инспектора объектов.

В качестве примера рассмотрим определение класса `TIntegerProperty` в модуле `DESIGNEDITORS.PAS`:

```
TIntegerProperty = class(TOrdinalProperty)
public
  function GetValue: string; override;
  procedure SetValue(const Value: string); override;
end;
```

Как видите, методы `GetValue()` и `SetValue()` переопределены. Ниже показана реализация метода `GetValue()`:

```
function TIntegerProperty.GetValue: string;
begin
  Result := IntToStr(GetOrdValue);
end;
```

А вот реализация метода `SetValue()`:

```
procedure TIntegerProperty.SetValue(const Value: String);
var
  L: Longint;
begin
  L := StrToInt(Value);
  with GetTypeData(GetPropType)^ do
    if (L < MinValue) or (L > MaxValue) then
      raise EPropertyError.CreateResFmt(SOutOfRange,
                                        [MinValue, MaxValue]);
  SetOrdValue(L);
end;
```

Метод `GetValue()` возвращает строковое представление целочисленного свойства, а инспектор объектов использует эту строку для отображения значения свойства. Метод `GetOrdValue()`, определенный в классе `TPropertyEditor`, служит для возвращения значения свойства, с которым работает редактор.

Метод `SetValue()` принимает строковое значение, введенное пользователем, и присваивает его свойству в нужном формате. Метод `SetValue()` предотвращает некоторые ошибки, осуществляя контроль вводимых пользователем значений. Метод `SetOrdValue()` собственно присваивает свойству, с которым работает редактор, введенное пользователем значение.

В классе `TPropertyEditor` определено несколько методов, подобных методу `GetOrdValue()`, предназначенных для получения строкового представления различных типов. К тому же компонент `TPropertyEditor` содержит методы, аналогичные методу `Set`, предназначенные для установки значений в соответствующих форматах. Потомки класса `TPropertyEditor` наследуют все эти методы, используемые для получения и установки значений свойств, с которыми работают редакторы. Такие методы приведены в табл. 12.2.

Таблица 12.2. Методы чтения и записи свойств класса `TPropertyEditor`

Тип свойства	*Метод "`Get`"*	*Метод "`Set`"*
Число с плавающей точкой	`GetFloatValue()`	`SetFloatValue()`
Событие	`GetMethodValue()`	`SetMethodValue()`
Перечисление	`GetOrdValue()`	`SetOrdValue()`
Строка	`GetStrValue()`	`SetStrValue()`
Вариант	`GetVarValue()`	`SetVarValue()`, `SetVarValueAt()`

Для иллюстрации создания нового редактора свойства вернемся к примеру с планетами Солнечной системы, приведенному в одной из предыдущих глав. Прошлый раз был создан простой компонент `TPlanet`, представляющий одну планету. Компонент `TPlanet` содержит свойство `PlanetName`. Его внутреннее поле типа `Integer` хранит позицию планеты в Солнечной системе, но в окне инспектора объектов оно должно будет отображаться как название планеты.

Это слишком просто, поэтому усложним задачу. Допустим, необходимо предоставить пользователю возможность выбрать один из двух способов задания планеты. Он может набрать строку названия планеты, например **`Venus`** или **`VENUS`**, или **`VeNuS`**, либо ввести позицию планеты в Солнечной системе. Так, для Венеры эта позиция будет равной значению 2.

Ниже приводится исходный код компонента `TPlanet`:

```
type
  TPlanetName = type Integer;

  TPlanet = class(TComponent)
  private
    FPlanetName: TPlanetName;
  published
    property PlanetName: TPlanetName read FPlanetName
                                     write FPlanetName;
  end;
```

Как видите, этот компонент совсем небольшой. У него есть только одно свойство `PlanetName` типа `TPlanetName`. Специальное определение типа `TPlanetName` позволяет ему иметь собственную информацию о типах времени выполнения и оставаться при этом целочисленным типом.

Основные функциональные возможности находятся не в компоненте `TPlanet`, а в редакторе свойств для типа `TPlanetName`. Текст этого редактора приведен в листинге 12.4.

Листинг 12.4. `PlanetPE.PAS` — исходный код редактора `TPlanetNameProperty`

```
unit PlanetPE;

interface

uses
```

```
  Windows, SysUtils, DesignEditors;

type
  TPlanetNameProperty = class(TIntegerProperty)
  public
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;

implementation

const
  { Объявление массива названий планет }
  PlanetNames: array[1..9] of String[7] = ('Mercury', 'Venus',
                           'Earth', 'Mars', 'Jupiter', 'Saturn',
                           'Uranus', 'Neptune', 'Pluto');

function TPlanetNameProperty.GetValue: string;
begin
  Result := PlanetNames[GetOrdValue];
end;

procedure TPlanetNameProperty.SetValue(const Value: String);
var
  PName: string[7];
  i, ValErr: Integer;
begin
  PName := UpperCase(Value);
  i := 1;
  { Сравнить Value с названием каждой планеты в массиве
    PlanetNames. Если найдено соответствие, то переменная i
    принимает значение меньше 10. }
  while (PName <> UpperCase(PlanetNames[i])) and (i < 10) do
    inc(i);
  { Если i меньше 10, то введено правильное название планеты.
    Установка значения и выход из процедуры. }
  if i < 10 then begin // Название планеты введено правильно.
    SetOrdValue(i);
    Exit;
  end
  { Если i больше 10, то пользователь ввел недопустимый номер или
    несуществующее название планеты. Использовать функцию Val для
    проверки введенного значения на число. Если ValErr не равен
    нулю, то неправильно введено название планеты. В противном
    случае проверить введенный номер на принадлежность
    диапазону (0 < i < 10). }
  else begin
    Val(Value, i, ValErr);
    if ValErr <> 0 then
      raise Exception.Create(
         Format('Sorry, Never heard of the planet %s.', [Value]));
    if (i <= 0) or (i >= 10) then
       raise Exception.Create(
```

```
                'Sorry, that planet is not in OUR solar system.');
    SetOrdValue(i);
  end;
end;

end.
```

Вначале создается редактор свойства TPlanetNameProperty, являющийся потомком TIntegerProperty. Кстати, в раздел uses этого модуля обязательно нужно включить модули DesignEditors и DesignIntf.

Определим массив строковых констант для представления планет Солнечной системы, в зависимости от их положения относительно Солнца. Эти строки будут отвечать за строковое представление планет в окне инспектора объектов.

Как уже отмечалось, необходимо переопределить методы GetValue() и SetValue(). Метод GetValue() возвращает строку из массива PlanetNames. Данный массив индексирован по значениям свойств. Конечно, значение свойства должно находиться в пределах диапазона 1–9. Поэтому пользователю запрещено вводить в метод SetValue() номер, выходящий за пределы этого диапазона.

Метод SetValue() получает строку, введенную в окне инспектора объектов. Эта строка может быть как названием планеты, так и номером, определяющим позицию планеты. Логика кода определяет, правильно ли введено название планеты или ее номер, и если да, то соответствующее значение присваивается свойству методом SetOrdValue(). Если пользователь ввел неправильное название планеты или недопустимый номер, то передается соответствующее исключение.

Вот и все определение редактора свойств. Впрочем нет, еще не все — для того чтобы свойство "узнало" о своем новом редакторе, его необходимо зарегистрировать.

Регистрация редактора свойств

Для регистрации редактора свойства следует воспользоваться процедурой RegisterPropertyEditor(). Данный метод объявляется следующим образом:

```
procedure RegisterPropertyEditor(PropertyType: PTypeInfo;
                                 ComponentClass: TClass;
                                 const PropertyName: string;
                                 EditorClass: TPropertyEditorClass);
```

Первый параметр PropertyType — это указатель на информацию о типах времени выполнения редактируемого свойства. Такую информацию можно получить с помощью функции TypeInfo(). Параметр ComponentClass используется для обозначения класса, к которому применяется редактор свойства. PropertyName определяет имя свойства компонента, а параметр EditorClass — тип используемого редактора свойства. Для свойства TPlanet.PlanetName эта функция выглядит следующим образом:

```
RegisterPropertyEditor(TypeInfo(TPlanetName),
                       TPlanet, 'PlanetName',
                       TPlanetNameProperty);
```

СОВЕТ

Пока (в иллюстративных целях) этот конкретный редактор свойств регистрируется только для использования со свойством `PlanetName` компонента `TPlanet`. Но данный редактор можно зарегистрировать таким образом, чтобы он был пригоден для работы с любым свойством типа `TPlanetName`. Для этого следует в качестве параметра `ComponentClass` ввести значение `nil`, а в качестве параметра `PropertyName` — `''`.

Регистрацию редактора свойств можно оформить вместе с регистрацией компонента в модуле компонента, как показано в листинге 12.5.

Листинг 12.5. `Planet.pas` — компонент `TPlanet`

```
unit Planet;

interface

uses
  Classes, SysUtils;

type
  TPlanetName = type Integer;

  TddgPlanet = class(TComponent)
  private
    FPlanetName: TPlanetName;
  published
    property PlanetName: TPlanetName read FPlanetName
                                     write FPlanetName;
  end;

implementation

end.
```

СОВЕТ

Регистрация редактора свойств в процедуре `Register()` модуля компонента связывает код редактора с компонентом при помещении последнего в пакет. Для сложных компонентов средства времени разработки зачастую занимают больше места, чем код самих компонентов. Несмотря на то, что размер кода не играет роли для таких маленьких компонентов как этот, имейте в виду, что *все* перечисленное в разделе `interface` (интерфейс) модуля компонента (в том числе и процедура `Register()` вместе со всеми классами, регистрируемыми ею, такими как тип класса редактора свойств) будет скомпилировано в пакет вместе с компонентом. Поэтому регистрацию редактора свойств нужно осуществлять в отдельном регистрационном модуле. Некоторые разработчики создают для своих компонентов как пакеты времени разработки, так и пакеты времени выполнения, причем редакторы свойств и другие средства времени разработки присутствуют только в пакете разработки. Программный код приме-

ров, обсуждаемых в этой книге, содержится в пакете времени выполнения DdgRT6 и пакете разработки DdgDT6.

Редактирование свойства в диалоговом окне

Иногда необходимы более широкие возможности редактирования, чем редактирование на месте, в окне инспектора объектов. В этом случае в качестве редактора свойств можно использовать диалоговое окно. Например, большинство компонентов Delphi имеют свойство Font. Конечно, создатели Delphi могли вынудить пользователей вручную набирать имя шрифта и сопутствующую информацию. Но было бы неразумно ожидать, что пользователь знает, какую именно информацию ему следует вводить. Гораздо проще обеспечить пользователя диалоговым окном, в котором он сможет выбрать необходимые атрибуты, относящиеся к шрифту, и увидеть результат их применения до утверждения окончательного варианта.

Чтобы проиллюстрировать использование диалогового окна для редактирования свойства, расширим функциональные возможности компонента TddgRunButton, создание которого описано в главе 11, "Разработка компонентов VCL". При установке значения свойства CommandLine в окне инспектора объектов пользователю будет достаточно щелкнуть на кнопке с многоточием, чтобы открыть диалоговое окно Open File и выбрать в нем файл для работы с компонентом TddgRunButton.

Пример диалогового редактора свойств: расширение возможностей компонента TddgRunButton

Текст компонента TddgRunButton приведен в листинге 11.13 главы 11, "Разработка компонентов VCL". Не будем полностью повторять его здесь, а лишь обратим внимание на несколько моментов. Свойство TddgRunButton.CommandLine определено как имеющее тип TCommandLine. Этот тип определен следующим образом:

```
TCommandLine = type string;
```

Здесь специальное объявление типа используется для того, чтобы получить уникальную информацию о типах времени выполнения. Это позволяет определить редактор свойств специально для типа TCommandLine. Кроме того, поскольку тип TCommandLine аналогичен строковому, то для него подойдет редактор свойств, предназначенный для строковых типов.

Не забывайте, что проверка наличия ошибок присвоения уже включена в методы доступа редактора свойств TCommandLineProperty для типа TCommandLine. Следовательно, нет необходимости повторять ее в логике редактора свойств.

В листинге 12.6 приведен код редактора свойств TCommandLineProperty.

Листинг 12.6. RunBtnPE.pas — модуль TCommandLineProperty

```
unit runbtnpe;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
```

```
  Forms, Dialogs, StdCtrls, Buttons, DesignEditors, DesignIntf,
  TypInfo;

type
  { Редактор происходит от класса TStringProperty и унаследовал
    его возможности редактирования строк. }
  TCommandLineProperty = class(TStringProperty)
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;

implementation

function TCommandLineProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]; // Отображение диалогового окна
end;                    // в методе Edit.

procedure TCommandLineProperty.Edit;
{ Метод Edit отображает окно TOpenDialog, из которого пользователь
получает имя исполняемого файла, присваиваемое свойству. }
var
  OpenDialog: TOpenDialog;
begin
  { Создать окно TOpenDialog }
  OpenDialog := TOpenDialog.Create(Application);
  try
    { Отображать только исполняемые файлы }
    OpenDialog.Filter := 'Executeable Files|*.EXE';
    { Если пользователь выбрал файл, присвоить его свойству. }
    if OpenDialog.Execute then
      SetStrValue(OpenDialog.FileName);
  finally
    OpenDialog.Free // Освободить экземпляр класса TOpenDialog.
  end;
end;

end.
```

Ознакомившись с компонентом `TCommandLineProperty`, можно понять, что этот редактор свойств очень прост. Он происходит от класса `TStringProperty` и, следовательно, поддерживает средства редактирования строк. Поэтому необязательно вызывать диалоговое окно в инспекторе объектов. Пользователь может вручную набрать командную строку. К тому же не понадобится переопределять методы `SetValue()` и `GetValue()`, так как в компоненте `TStringProperty` это уже сделано надлежащим образом. Тем не менее необходимо переопределить метод `GetAttributes()` для того, чтобы инспектор свойств "знал", что свойство можно редактировать с помощью диалогового окна. Метод `GetAttributes()` заслуживает более подробного обсуждения.

Установка атрибутов редактора свойств

Каждый редактор свойств должен сообщить инспектору объектов о способе редактирования свойства и об использовании специальных атрибутов, если таковые имеются. В большинстве случаев достаточно атрибутов, унаследованных от базового класса редактора. Но иногда придется переопределить метод `GetAttributes()`, возвращающий набор флагов атрибутов свойства (флаги `TPropertyAttributes`). Этот набор указывает, для каких атрибутов используются специальные редакторы свойств. Флаги типа `TPropertyAttribute` приведены в табл. 12.3.

Таблица 12.3. Флаги `TPropertyAttribute`

Атрибут	*Как редактор свойств работает с инспектором объектов*
`paValueList`	Возвращает перечислимый список значений свойства. Метод `GetValue()` заполняет список. Справа от значения свойства находится кнопка с изображением стрелки, направленной вниз. Применяется для таких перечислимых свойств, как `TForm.BorderStyle`, и таких групп целочисленных констант, как `TColor` и `TCharSet`
`paSubProperties`	Подчиненные свойства отображаются с отступом ниже текущего свойства в формате иерархической структуры. Атрибут `paValueList` также должен быть установлен. Устанавливается для свойств типа "множество" или типа "класс", таких как `TOpenDialog.Options` и `TForm.Font`
`paDialog`	Появляется кнопка с многоточием, по щелчку на которой метод `Edit()` вызывает диалоговое окно. Применяется для свойств наподобие `TForm.Font`
`paMultiSelect`	Свойства отображаются, даже если в конструкторе форм выбрано больше одного компонента; это позволяет пользователю изменять значения свойств сразу нескольких компонентов. Некоторые свойства, например `Name`, не поддерживают этой возможности
`paAutoUpdate`	Метод `SetValue()` вызывается при каждом изменении свойства. Если этот флаг не установлен, `SetValue()` вызывается при нажатии клавиши <Enter> или при выходе из свойства в окне инспектора объектов. Применяется для таких свойств, как `TForm.Caption`
`paFullWidthName`	Сообщает инспектору объектов о том, что это значение не нуждается в дополнительном представлении, а значит, его имя следует отображать, используя всю ширину окна инспектора объектов
`paSortList`	Инспектор объектов сортирует список, возвращаемый функцией `GetValues()`
`paReadOnly`	Значение свойства не может быть изменено
`paRevertable`	Свойству можно вернуть его прежнее значение. Но некото-

рые свойства, например вложенные (такие как TFont), восстановлению не подлежат

НА ЗАМЕТКУ

Обратите внимание на установленные флаги TPropertyAttribute для различных редакторов свойства в файле DesignEditors.pas.

Установка атрибута paDialog для свойства типа TCommandLineProperty

Поскольку компонент TCommandLineProperty используется для отображения диалогового окна, необходимо сообщить инспектору объектов об этой возможности, установив атрибут paDialog в методе TCommandLineProperty.GetAttributes(). В таком случае справа от значения свойства CommandLine в окне инспектора объектов появится кнопка с многоточием. Если пользователь щелкнет на этой кнопке, то будет вызван метод TCommandLineProperty.Edit().

Регистрация редактора свойств TCommandLineProperty

Завершающим этапом реализации редактора свойств TCommandLineProperty является его регистрация с помощью описанной ранее в настоящей главе процедуры RegisterPropertyEditor(). Эта процедура была добавлена в процедуру Register() в модуле DDGReg.pas, входящем в состав пакета DdgDsgn:

```
RegisterComponents('DDG', [TddgRunButton]);
  RegisterPropertyEditor(TypeInfo(TCommandLine), TddgRunButton,
                         '', TCommandLineProperty);
```

Помните о том, что в раздел uses необходимо добавить модули DsgnIntf и RunBtnPE.

Редакторы компонентов

Редакторы компонентов модифицируют поведение компонентов во время разработки, позволяя добавлять элементы в контекстное меню, связанное с конкретным компонентом, а также изменять стандартные действия, выполняемые по двойному щелчку мышью на компоненте в конструкторе форм. Вполне возможно, что вы уже использовали редакторы полей компонентов TTable, TQuery и TStoredProc и, следовательно, сами того не ведая, работали с редакторами компонентов.

Класс TComponentEditor

Хоть это и малоизвестно, но для каждого компонента, выбранного в конструкторе форм, создается свой собственный редактор. Тип созданного редактора компонентов зависит от типа компонента, но все редакторы компонентов происходят от класса TComponentEditor. Этот класс определен в модуле DsgnIntf следующим образом:

```
TComponentEditor= class(TBaseComponentEditor, IComponentEditor)
  private
    FComponent: TComponent;
    FDesigner: IDesigner;
  public
    constructor Create(AComponent: TComponent;
                       ADesigner: IDesigner); override;
    procedure Edit; virtual;
    procedure ExecuteVerb(Index: Integer); virtual;
    function GetComponent: TComponent;
    function GetDesigner: IDesigner;
    function GetVerb(Index: Integer): string; virtual;
    function GetVerbCount: Integer; virtual;
    function IsInInlined: Boolean;
    procedure Copy; virtual;
    procedure PrepareItem(Index: Integer;
                          const AItem: IMenuItem); virtual;
    property Component: TComponent read FComponent;
    property Designer: IDesigner read GetDesigner;
end;
```

Свойства

Свойство `TComponentEditor.Component` — это экземпляр редактируемого компонента. Так как все свойства происходят от базового типа `TComponent`, то для доступа к полям, введенным в производных классах, необходимо выполнить приведение типа.

Свойство `Designer` — это экземпляр компонента `TFormDesigner`, управляющего приложением во время разработки. Полное определение данного класса можно найти в модуле `DesignEditors.pas`.

Методы

Метод `Edit()` вызывается при двойном щелчке на компоненте во время разработки. Чаще всего этот метод вызывает тот или иной тип диалогового окна при разработке. Стандартное поведение такого метода предполагает вызов функции `ExecuteVerb(0)`, если функция `GetVerbCount()` возвращает значение, большее или равное `1`. Если в методе `Edit()` или в любом другом методе компонент подвергается изменениям, необходимо вызвать метод `Designer.Modified()`.

Здесь в именах процедур и функций используется термин *verb* (глагол, действие), поскольку речь идет о методах объектов, то есть о тех действиях, которые объект может предпринимать. Сама интегрированная среда разработки Delphi ничего не знает о новых объектах или компонентах, поэтому по мере их добавления ей необходимо "сообщить" об их появлении. Для этого было разработано несколько методов, которые применяются для идентификации действий объекта. Чтобы сообщить Delphi о новом компоненте, обычно используются такие методы, как `GetVerbCount`, `GetVerb` и `ExecuteVerb`. Они подходят для большинства компонентов.

Метод `GetVerbCount()` вызывается для получения количества элементов, добавляемых в контекстное меню.

Метод `GetVerb()` принимает целочисленный параметр `Index` и возвращает строку, содержащую текст, который появляется в соответствующей позиции контекстного меню.

После выбора элемента в контекстном меню вызывается метод ExecuteVerb(). Этому методу в параметре Index передается номер выбранного пункта контекстного меню (считая от нуля). В ответ должно быть выполнено действие, определяемое выбранным пунктом меню.

Метод Paste() вызывается при помещении компонента в буфер обмена. Delphi помещает в буфер обмена ресурсный (записанный в поток) образ компонента. Этот метод можно также использовать для помещения в буфер обмена данных различного формата.

Класс TDefaultEditor

Если для нового компонента собственный редактор компонентов не зарегистрирован, то для его редактирования будет использован стандартный редактор, экземпляр класса TDefaultEditor. Этот редактор переопределяет поведение метода Edit() так, что он ищет свойства компонента и генерирует (или находит) событие OnCreate, OnChanged или OnClick (в зависимости от того, какое из них будет найдено первым).

Пример простого компонента

Рассмотрим простой пользовательский компонент:

```
type
  TComponentEditorSample = class(TComponent)
  protected
    procedure SayHello; virtual;
    procedure SayGoodbye; virtual;
  end;

procedure TComponentEditorSample.SayHello;
begin
  MessageDlg('Hello, there!', mtInformation, [mbOk], 0);
end;

procedure TComponentEditorSample.SayGoodbye;
begin
  MessageDlg('See ya!', mtInformation, [mbOk], 0);
end;
```

Как видите, такой небольшой компонент может делать немногое. Это невизуальный компонент, происходящий непосредственно от класса TComponent и содержащий два метода — SayHello() и SayGoodbye(), — просто отображающих окна сообщений.

Пример редактора для простого компонента

Для того чтобы компонент смотрелся солиднее, его можно снабдить собственным редактором. Такой редактор будет вызываться во время разработки для выполнения методов компонента. Для этого в нем придется переопределить по крайней мере три метода класса TComponentEditor: ExecuteVerb(), GetVerb() и GetVerbCount(). Приведем исходный код этого компонента:

```
type
  TSampleEditor = class(TComponentEditor)
  private
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

procedure TSampleEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TComponentEditorSample(Component).SayHello;
                                                    // вызов функции
    1: TComponentEditorSample(Component).SayGoodbye;
                                                    // вызов функции
  end;
end;

function TSampleEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0: Result := 'Hello';       // возвращает строку hello
    1: Result := 'Goodbye';     // возвращает строку goodbye
  end;
end;

function TSampleEditor.GetVerbCount: Integer;
begin
  Result := 2;      // возможно два действия
end;
```

Метод `GetVerbCount()` возвращает число 2, а это означает, что редактор компонентов готов выполнить два действия. Метод `GetVerb()` возвращает строку каждого из соответствующих пунктов в контекстном меню. Метод `ExecuteVerb()` вызывает соответствующий метод компонента, основываясь на значении индекса, передаваемого ему в качестве параметра.

Регистрация редактора компонентов

Подобно компонентам и редакторам свойств, редакторы компонентов также должны быть зарегистрированы в интегрированной среде разработки Delphi — в методе `Register()` модуля. Для регистрации редактора компонентов вызывается процедура с соответствующим именем — `RegisterComponentEditor()`, — объявленная следующим образом:

```
procedure RegisterComponentEditor(ComponentClass: TComponentClass;
                          ComponentEditor: TComponentEditorClass);
```

Первый параметр данной процедуры определяет тип компонента, для которого необходимо зарегистрировать редактор; второй параметр задает имя этого редактора.

В листинге 12.7 приведен код модуля `CompEdit.pas`, который содержит компонент, его редактор и вызов регистрации.

Листинг 12.7. `CompEdit.pas` — пример редактора компонента

```
unit CompEdit;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
  Dialogs, DesignEditors;

type
  TComponentEditorSample = class(TComponent)
  protected
    procedure SayHello; virtual;
    procedure SayGoodbye; virtual;
  end;

  TSampleEditor = class(TComponentEditor)
  public
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

implementation

{ TComponentEditorSample }
procedure TComponentEditorSample.SayHello;
begin
  MessageDlg('Hello, there!', mtInformation, [mbOk], 0);
end;

procedure TComponentEditorSample.SayGoodbye;
begin
  MessageDlg('See ya!', mtInformation, [mbOk], 0);
end;

{ TSampleEditor }
const
  vHello = 'Hello';
  vGoodbye = 'Goodbye';

procedure TSampleEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TComponentEditorSample(Component).SayHello;
       // вызов функции
    1: TComponentEditorSample(Component).SayGoodbye;
       // вызов функции
  end;
end;
```

```
function TSampleEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0: Result := vHello;       // возвращает строку hello
    1: Result := vGoodbye;     // возвращает строку goodbye
  end;
end;

function TSampleEditor.GetVerbCount: Integer;
begin
  Result := 2;      // возможно два действия
end;

end.
```

Работа с потоками данных непубликуемых компонентов

В главе 11, "Разработка компонентов VCL", отмечалось, что интегрированная среда разработки Delphi автоматически записывает и считывает публикуемые (published) свойства компонента из файла DFM. Что же делать, если в файле DFM необходимо сохранять и непубликуемые данные? К счастью, компоненты Delphi содержат механизм, позволяющий записывать и считывать определенные программистом данные из файла DFM.

Определение свойств

Первым шагом определения сохраняемых непубликуемых свойств является переопределение метода DefineProperties() компонента. Этот метод каждый компонент наследует от класса TPersistent, в котором он определяется следующим образом:

```
procedure DefineProperties(Filer: TFiler); virtual;
```

По умолчанию данный метод управляет чтением и записью публикуемых свойств в файл DFM. Этот метод можно переопределить и, после вызова унаследованного (inherited) метода, вызвать определенные в компоненте TFiler методы DefineProperty() или DefineBinaryProperty() — один раз для каждой порции данных, которые нужно поместить в файл DFM. Определение таких методов приводится ниже.

```
procedure DefineProperty(const Name: string;
                         ReadData: TReaderProc;
                         WriteData: TWriterProc;
                         HasData: Boolean); virtual;
procedure DefineBinaryProperty(const Name: string;
                               ReadData, WriteData: TStreamProc;
                               HasData: Boolean); virtual;
```

Функция DefineProperty() используется для того, чтобы сделать сохраняемыми такие стандартные типы данных, как строковые, целые, логические, символьные типы, числа с плавающей точкой и перечислимые типы. Метод DefineBinaryProperty() используется для обеспечения доступа к необработанным бинарным данным (графическим или звуковым), записанным в файл DFM.

В обеих этих функциях параметр Name идентифицирует имя свойства, которое должно быть записано в файл DFM. Оно может и не совпадать с внутренним именем поля, содержащего эти данные. Параметры ReadData и WriteData функции DefineProperty() отличаются от соответствующих параметров функции DefineBinaryProperty() по типу, но предназначены для одного и того же: они вызываются для записи или считывания данных из файла DFM. (Подробнее рассмотрим их несколько позже.) Параметр HasData определяет, имеет ли свойство данные, которые необходимо сохранить.

Параметры ReadData и WriteData функции DefineProperty() имеют типы TReaderProc и TWriterProc соответственно. Эти типы определены следующим образом:

```
type
  TReaderProc = procedure(Reader: TReader) of object;
  TWriterProc = procedure(Writer: TWriter) of object;
```

Классы TReader и TWriter (специализированные потомки класса TFiler) имеют дополнительные методы чтения и записи своих типов. Методы этих типов "наводят мосты" между публикуемыми данными компонента и файлом DFM.

Параметры ReadData и WriteData метода DefineBinaryProperty() имеют тип TStreamProc, определенный так:

```
type
  TStreamProc = procedure(Stream: TStream) of object;
```

Поскольку методам типа TStreamProc передается только один параметр типа TStream, то двоичные данные легко можно считывать и записывать в поток. Подобно типам других методов, методы этого типа связывают с файлом DFM нестандартные данные.

Пример использования функции DefineProperty()

Для того чтобы свести воедино всю изложенную информацию по данному вопросу, в листинге 12.8 приведен код модуля DefProp.pas. Этот модуль иллюстрирует использование функции DefineProperty() для сохранения содержимого двух полей данных, объявленных в разделе private: строкового и целочисленного.

Листинг 12.8. DefProp.pas — пример использования функции DefineProperty()

```
unit DefProp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
```

```
  Dialogs;

type
  TDefinePropTest = class(TComponent)
  private
    FString: String;
    FInteger: Integer;
    procedure ReadStrData(Reader: TReader);
    procedure WriteStrData(Writer: TWriter);
    procedure ReadIntData(Reader: TReader);
    procedure WriteIntData(Writer: TWriter);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TDefinePropTest.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Поместить данные в закрытые поля. }
  FString := 'The following number is the answer...';
  FInteger := 42;
end;

procedure TDefinePropTest.DefineProperties(Filer: TFiler);
begin
  inherited DefineProperties(Filer);
  { Определение новых свойств и их методов чтения/записи. }
  Filer.DefineProperty('StringProp', ReadStrData,
                       WriteStrData, FString <> '');
  Filer.DefineProperty('IntProp', ReadIntData,
                       WriteIntData, True);
end;

procedure TDefinePropTest.ReadStrData(Reader: TReader);
begin
  FString := Reader.ReadString;
end;

procedure TDefinePropTest.WriteStrData(Writer: TWriter);
begin
  Writer.WriteString(FString);
end;

procedure TDefinePropTest.ReadIntData(Reader: TReader);
begin
  FInteger := Reader.ReadInteger;
end;

procedure TDefinePropTest.WriteIntData(Writer: TWriter);
```

```
begin
  Writer.WriteInteger(FInteger);
end;

end.
```

СОВЕТ

Всегда для чтения и записи строковых данных используйте методы `ReadString()` и `WriteString()` классов `TReader` и `TWriter`. Никогда не используйте похожие на них методы `ReadStr()` и `WriteStr()`, потому что они могут повредить файл `DFM`.

Компонент TddgWaveFile: пример использования функции DefineBinaryProperty()

Как уже говорилось, функцию `DefineBinaryProperty()` лучше всего использовать для сохранения вместе с компонентом графической или звуковой информации. Фактически в библиотеке VCL эта методика используется для сохранения изображений, связанных с компонентом, например значка (`Glyph`) компонента `TBitBtn` или пиктограммы (`Icon`) компонента `TForm`. В этом разделе изложено, как использовать подобную методику для сохранения звуковых данных, связанных с компонентом `TddgWaveFile`.

НА ЗАМЕТКУ

`ddgWaveFile` — это полнофункциональный компонент, содержащий пользовательское свойство, редактор данного свойства и редактор компонента, позволяющий воспроизводить звуки во время разработки. Об этом речь пойдет несколько позже, а пока рассмотрим механизм сохранения бинарных данных свойства.

Ниже приведен исходный код метода `DefineProperties()` компонента `TddgWaveFile`.

```
procedure TddgWaveFile.DefineProperties(Filer: TFiler);

{ Определяет двоичное свойство "Data" для поля FData. Благодаря
этому значение поля FData может быть считано и записано в файл DFM.
}

  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then
      Result := not (Filer.Ancestor is TddgWaveFile) or
                not Equal(TddgWaveFile(Filer.Ancestor))
    else
      Result := not Empty;
  end;
```

```
begin
  inherited DefineProperties(Filer);
  Filer.DefineBinaryProperty('Data', ReadData,
                             WriteData, DoWrite);
end;
```

Данный метод определяет бинарное свойство Data, которое считывается и записывается с помощью методов ReadData() и WriteData() компонента. Причем данные записываются только в том случае, когда функция DoWrite() возвращает значение True. Более подробная информация об этой функции приведена в настоящей главе далее.

Методы ReadData() и WriteData() определены следующим образом:

```
procedure TddgWaveFile.ReadData(Stream: TStream);
{ Читает данные WAV из потока DFM. }
begin
  LoadFromStream(Stream);
end;

procedure TddgWaveFile.WriteData(Stream: TStream);
{ Записывает данные WAV из потока DFM. }
begin
  SaveToStream(Stream);
end;
```

Как видите, оба этих небольших метода просто вызывают методы LoadFromStream() и SaveToStream(), которые также определены в классе компонента TddgWaveFile. Вот исходный код метода LoadFromStream():

```
procedure TddgWaveFile.LoadFromStream(S: TStream);
{ Загружает данные WAV из потока S. Эта процедура освобождает
память, предварительно выделенную для данных FData. }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  FDataSize := 0;
  FData := AllocMem(S.Size);
  FDataSize := S.Size;
  S.Read(FData^, FDataSize);
end;
```

Вначале данный метод определяет, была ли ранее выделена память (проверяя значение поля FDataSize). Если это так (т.е. это значение больше нуля), то память, на которую указывает значение поля FData, освобождается. Затем для поля FData выделяется новый блок памяти и полю FDataSize присваивается размер потока поступающих данных. После чего содержимое потока считывается по указателю поля FData.

Метод SaveToStream() значительно проще вышеописанного. Вот его определение.

```
procedure TddgWaveFile.SaveToStream(S: TStream);
{ Сохраняет данные WAV в поток S. }
begin
  if FDataSize > 0 then S.Write(FData^, FDataSize);
end;
```

Этот метод записывает данные, на которые ссылается указатель FData, в поток S типа TStream.

Локальная функция DoWrite(), вызываемая внутри метода DefineProperties(), определяет необходимость записи в поток значения свойства Data. Конечно, если поле FData "пустует", то ничего в поток записывать не нужно. Кроме того, следует принять дополнительные меры для того, чтобы компонент корректно работал при наследовании формы: необходимо убедиться, что свойство Ancestor объекта TFiler не равно nil. Если это так и свойство указывает на некоторую версию предка текущего компонента, то следует также удостовериться, что данные, приготовленные для записи, отличаются от данных предка. Если не выполнить такую дополнительную проверку, то копия данных (в данном случае файла wav) будет записана в каждую форму потомка и изменение файла wav предка не приведет к изменению форм потомков.

В листинге 12.9 представлен модуль Wavez.pas, содержащий полный исходный код этого компонента.

Листинг 12.9. Wavez.pas — пример компонента, инкапсулирующего файл WAV

```
unit Wavez;

interface

uses
  SysUtils, Classes;

type
  { Для создания редактора используется специализированный
                                                    тип string. }
  TWaveFileString = type string;

  EWaveError = class(Exception);

  TWavePause = (wpAsync, wpsSync);
  TWaveLoop = (wlNoLoop, wlLoop);

  TddgWaveFile = class(TComponent)
  private
    FData: Pointer;
    FDataSize: Integer;
    FWaveName: TWaveFileString;
    FWavePause: TWavePause;
    FWaveLoop: TWaveLoop;
    FOnPlay: TNotifyEvent;
    FOnStop: TNotifyEvent;
    procedure SetWaveName(const Value: TWaveFileString);
    procedure WriteData(Stream: TStream);
    procedure ReadData(Stream: TStream);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  public
    destructor Destroy; override;
    function Empty: Boolean;
```

```
    function Equal(Wav: TddgWaveFile): Boolean;
    procedure LoadFromFile(const FileName: String);
    procedure LoadFromStream(S: TStream);
    procedure Play;
    procedure SaveToFile(const FileName: String);
    procedure SaveToStream(S: TStream);
    procedure Stop;
  published
    property WaveLoop: TWaveLoop read FWaveLoop write FWaveLoop;
    property WaveName: TWaveFileString read FWaveName
                                       write SetWaveName;
    property WavePause: TWavePause read FWavePause
                                   write FWavePause;
    property OnPlay: TNotifyEvent read FOnPlay write FOnPlay;
    property OnStop: TNotifyEvent read FOnStop write FOnStop;
  end;

implementation

uses MMSystem, Windows;

{ TddgWaveFile }

destructor TddgWaveFile.Destroy;
{ Гарантирует освобождение всей выделенной ранее памяти. }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  inherited Destroy;
end;

function StreamsEqual(S1, S2: TMemoryStream): Boolean;
begin
  Result := (S1.Size = S2.Size) and
            CompareMem(S1.Memory, S2.Memory, S1.Size);
end;

procedure TddgWaveFile.DefineProperties(Filer: TFiler);
{ Определяет бинарное свойство "Data" для поля FData. Благодаря
этому данные, на которые указывает поле Fdata, могут быть считаны и
записаны в файл DFM. }

  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then
      Result := not (Filer.Ancestor is TddgWaveFile) or
                not Equal(TddgWaveFile(Filer.Ancestor))
    else
      Result := not Empty;
  end;

begin
  inherited DefineProperties(Filer);
```

```
  Filer.DefineBinaryProperty('Data', ReadData,
                             WriteData, DoWrite);
end;

function TddgWaveFile.Empty: Boolean;
begin
  Result := FDataSize = 0;
end;

function TddgWaveFile.Equal(Wav: TddgWaveFile): Boolean;
var
  MyImage, WavImage: TMemoryStream;
begin
  Result := (Wav <> nil) and (ClassType = Wav.ClassType);
  if Empty or Wav.Empty then begin
    Result := Empty and Wav.Empty;
    Exit;
  end;
  if Result then begin
    MyImage := TMemoryStream.Create;
    try
      SaveToStream(MyImage);
      WavImage := TMemoryStream.Create;
      try
        Wav.SaveToStream(WavImage);
        Result := StreamsEqual(MyImage, WavImage);
      finally
        WavImage.Free;
      end;
    finally
      MyImage.Free;
    end;
  end;
end;

procedure TddgWaveFile.LoadFromFile(const FileName: String);
{ Загружает данные WAV из файла FileName. Обратите внимание: эта
процедура не присваивает значения свойству WaveName. }
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmOpenRead);
  try
    LoadFromStream(F);
  finally
    F.Free;
  end;
end;

procedure TddgWaveFile.LoadFromStream(S: TStream);
{ Загружает данные WAV из потока S. Эта процедура освобождает
память, выделенную перед этим для FData. }
begin
```

```
  if not Empty then FreeMem(FData, FDataSize);
  FDataSize := 0;
  FData := AllocMem(S.Size);
  FDataSize := S.Size;
  S.Read(FData^, FDataSize);
end;

procedure TddgWaveFile.Play;
{ Воспроизводит звук WAV из FData с учетом параметров, содержащихся
в FWaveLoop и FWavePause. }
const
  LoopArray: array[TWaveLoop] of DWORD = (0, SND_LOOP);
  PauseArray: array[TWavePause] of DWORD = (SND_ASYNC, SND_SYNC);
begin
  { Убедимся, что компонент содержит данные. }
  if Empty then
    raise EWaveError.Create('No wave data');
  if Assigned(FOnPlay) then FOnPlay(Self);    // произошло событие
  { Попытка воспроизведения звука wave. }
  if not PlaySound(FData, 0, SND_MEMORY or
                   PauseArray[FWavePause] or
                   LoopArray[FWaveLoop]) then
    raise EWaveError.Create('Error playing sound');
end;

procedure TddgWaveFile.ReadData(Stream: TStream);
{ Считывает данные WAV из потока DFM. }
begin
  LoadFromStream(Stream);
end;

procedure TddgWaveFile.SaveToFile(const FileName: String);
{ Сохраняет данные WAV в файл FileName. }
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmCreate);
  try
    SaveToStream(F);
  finally
    F.Free;
  end;
end;

procedure TddgWaveFile.SaveToStream(S: TStream);
{ Сохраняет данные WAV в поток S. }
begin
  if not Empty then
    S.Write(FData^, FDataSize);
end;

procedure TddgWaveFile.SetWaveName(const Value: TWaveFileString);
{ Метод записи свойства WaveName. Этот метод отвечает за установку
```

```
свойства WaveName и загрузку данных WAV из файла Value. }
begin
  if Value <> '' then begin
    FWaveName := ExtractFileName(Value);
    { Не загружать из файла при загрузке из потока DFM, так
      как поток DFM уже содержит данные. }
    if (not (csLoading in ComponentState)) and
        FileExists(Value) then
      LoadFromFile(Value);
  end
  else begin
    { Если строка Value пуста, то необходимо освободить память,
      выделенную для данных WAV. }
    FWaveName := '';
    if not Empty then
      FreeMem(FData, FDataSize);
    FDataSize := 0;
  end;
end;

procedure TddgWaveFile.Stop;
{ Останавливает воспроизведение текущего звука WAV. }
begin
  if Assigned(FOnStop) then FOnStop(Self);  // произошло событие
  PlaySound(Nil, 0, SND_PURGE);
end;

procedure TddgWaveFile.WriteData(Stream: TStream);
{ Записывает данные WAV в поток DFM. }
begin
  SaveToStream(Stream);
end;

end.
```

Категории свойств

Как уже отмечалось в главе 1, "Программирование в Delphi", новинкой Delphi 5 являлись *категории свойств* (property categories). Теперь свойства компонентов библиотеки VCL можно отнести к той или иной категории, а инспектор объектов получает возможность рассортировать свойства по категориям. Принадлежность свойства к определенной категории можно зарегистрировать с помощью функций `RegisterPropertyInCategory()` и `RegisterPropertiesInCategory()`, объявленных в модуле `DesignIntf`. Первая из названных функций позволяет отнести к заданной категории одно свойство, а вторая — сразу несколько.

Функция `RegisterPropertyInCategory()` является перегружаемой, в результате чего можно использовать четыре ее версии, предназначенные для различных потребностей. Всем версиям этой функции передается в качестве первого параметра класс `TPropertyCategoryClass`, описывающий необходимую категорию. Что касается остальной части списка передаваемых параметров, то каждой из версии переда-

ется различная комбинация (состоящая из имени свойства, типа свойства и класса компонента). Это позволяет программисту выбрать наилучший метод регистрации созданных им свойств. Приводим объявления различных версий функции RegisterPropertyInCategory():

```
function RegisterPropertyInCategory(
          ACategoryClass: TPropertyCategoryClass;
          const APropertyName: string): TPropertyFilter; overload;
function RegisterPropertyInCategory(
          ACategoryClass: TPropertyCategoryClass;
          AComponentClass: TClass;
          const APropertyName: string): TPropertyFilter overload;
function RegisterPropertyInCategory(
          ACategoryClass: TPropertyCategoryClass;
          APropertyType: PTypeInfo;
          const APropertyName: string): TPropertyFilter; overload;
function RegisterPropertyInCategory(
          ACategoryClass: TPropertyCategoryClass;
          APropertyType: PTypeInfo): TPropertyFilter; overload;
```

Эти функции достаточно интеллектуальны, чтобы надлежащим образом воспринимать символы шаблона, благодаря чему можно отнести к определенной категории все свойства, имена которых соответствуют, к примеру, маске 'Data*'. Для получения полного списка поддерживаемых Delphi символов шаблона и информации об их поведении обратитесь к интерактивной справочной системе Delphi (раздел, описывающий класс TMask).

Предусмотрено три варианта перегружаемой функции RegisterPropertiesInCategory():

```
function RegisterPropertiesInCategory(
    ACategoryClass: TPropertyCategoryClass;
    const AFilters: array of const): TPropertyCategory; overload;
function RegisterPropertiesInCategory(
    ACategoryClass: TPropertyCategoryClass;
    AComponentClass: TClass;
    const AFilters: array of string): TPropertyCategory; overload;
function RegisterPropertiesInCategory(
    ACategoryClass: TPropertyCategoryClass;
    APropertyType: PTypeInfo;
    const AFilters: array of string):TPropertyCategory; overload;
```

Классы категорий

Тип TPropertyCategoryClass представляет собой ссылку на класс категории свойства TPropertyCategory. Для всех стандартных категорий свойств в библиотеке VCL базовым классом является класс TPropertyCategory. Существует 12 стандартных категорий свойств, и все они описаны в табл. 12.4.

Таблица 12.4. Классы стандартных категорий свойств

Имя класса	*Описание*
TActionCategory	Свойства, связанные с действиями времени выполнения. К этой категории относятся свойства Enabled и Hint компонента TControl
TDatabaseCategory	Свойства, относящиеся к операциям с базами данных. К этой категории относятся свойства DatabaseName и SQL компонента TQuery
TDragNDropCategory	Свойства, связанные с операциями перетаскивания и закрепления (стыковки окон). К данной категории относятся свойства DragCursor и DragKind компонента TControl
THelpCategory	Свойства, связанные с использованием интерактивной справки и подсказок. К этой категории относятся свойства HelpContext и Hint компонента TWinControl
TLayoutCategory	Свойства, связанные с визуальным отображением элемента управления во время разработки. К данной категории относятся свойства Top и Left компонента TControl
TLegacyCategory	Свойства, связанные с уже устаревшими операциями. К этой категории относятся свойства Ctl3D и ParentCtl3D компонента TWinControl
TLinkageCategory	Свойства, имеющие отношение к присоединению или связыванию одного компонента с другим. К этой категории относится свойство DataSet компонента TDataSource
TLocaleCategory	Свойства, связанные с локализацией. К этой категории относятся свойства BiDiMode и ParentBiDiMode компонента TControl
TLocalizableCategory	Свойства, относящиеся к операциям с базами данных. К этой категории относятся свойства DatabaseName и SQL компонента TQuery
TMiscellaneousCategory	Свойства, которые либо не подходят ни под одну из существующих категорий, либо не нуждаются в причислении к какой-либо категории, либо не зарегистрированы в явном виде в определенной категории. К этой категории принадлежат свойства AllowAllUp и Name компонента TSpeedButton
TVisualCategory	Свойства, связанные с визуальным отображением элемента управления во время выполнения. К данной категории относятся свойства Align и Visible компонента TControl

Окончание табл. 12.4.

Имя класса	*Описание*
TInputCategory	Свойства, связанные с вводом данных (они необязательно имеют отношение к операциям с базами данных). К этой категории относятся свойства Enabled и ReadOnly компонента TEdit

Допустим, что создан компонент по имени TNeato, у которого есть свойство Keen, и требуется зарегистрировать это свойство в качестве члена категории Action, представленной классом TActionCategory. Это можно сделать, добавив в процедуру Register() обращение к функции RegisterPropertyInCategory(), как показано ниже.

```
RegisterPropertyInCategory(TActionCategory, TNeato, 'Keen');
```

Пользовательские категории

Как уже было сказано, категория свойства представляется в программном коде как класс, который является потомком класса TPropertyCategory. Возникает вопрос: трудно ли создать свои собственные категории свойств? Оказывается, это не так уж и сложно. В большинстве случаев для этого достаточно переопределить виртуальные функции Name() и Description() класса TPropertyCategory и тогда можно получать информацию, относящуюся к данной категории.

В качестве примера создадим новую категорию Sound, к которой можно причислить некоторые свойства компонента TddgWaveFile (он уже упоминался в настоящей главе ранее). Объявление класса новой категории, TSoundCategory, представлено в листинге 12.10. Этот листинг содержит текст файла WavezEd.pas, включающего в себя определение категории компонента, редактор его свойств и редактор самого компонента.

Листинг 12.10. WavezEd.pas — редактор свойств для компонента TddgWaveFile

```
unit WavezEd;

interface

uses PropertyCategories, DesignEditors, DesignIntf;

type
  { Редактор для свойства WaveName компонента TddgWaveFile }
  TWaveFileStringProperty = class(TStringProperty)
  public
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
  end;

  { Редактор для компонента TddgWaveFile. Позволяет воспроизводить
    и останавливать звучание данных WAV с помощью локального меню
```

```
    в среде IDE. }
  TWaveEditor = class(TComponentEditor)
  private
    procedure EditProp(const Prop: IProperty);
  public
    procedure Edit; override;
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

implementation

uses TypInfo, Wavez, Classes, Controls, Dialogs;

{ TWaveFileStringProperty }

procedure TWaveFileStringProperty.Edit;
{ Выполняется по щелчку на кнопке с многоточием рядом со строкой
свойства WavName в окне инспектора объектов. Этот метод позволяет
пользователю выбрать файл в диалоговом окне OpenDialog и задать его
в качестве значения свойства. }
begin
  with TOpenDialog.Create(nil) do
    try
      { Установить свойства диалогового окна }
      Filter := 'Wav files|*.wav|All files|*.*';
      DefaultExt := '*.wav';
      { Поместить текущее значение в свойство FileName диалогового
        окна. }
      FileName := GetStrValue;
      { Выполнить диалог и установить значение свойства, если это
        диалоговое окно закрывается по щелчку на кнопке OK. }
      if Execute then
        SetStrValue(FileName);
    finally
      Free;
    end;
end;

function TWaveFileStringProperty.GetAttributes:↳
                                        TPropertyAttributes;
{ Указывает на то, что редактор свойств будет вызывать диалоговое
окно. }
begin
  Result := [paDialog];
end;

{ TWaveEditor }

const
  VerbCount = 2;
  VerbArray: array[0..VerbCount - 1] of string[7] = ('Play',
```

```
                                                        'Stop');

procedure TWaveEditor.Edit;
{ Вызывается, когда пользователь дважды щелкает на компоненте во
время разработки.
Этот метод вызывает метод GetComponentProperties, чтобы обратиться
к методу Edit редактора свойства WaveName. }
var
  Components: IDesignerSelections;
begin
  Components := TDesignerSelections.Create;
  Components.Add(Component);
  GetComponentProperties(Components, tkAny, Designer,
                         EditProp, nil);
end;

procedure TWaveEditor.EditProp(const Prop: IProperty);
{ Вызывается один раз для каждого свойства в ответ на вызов функции
GetComponentProperties. Этот метод ищет редактор свойства WaveName
и вызывает свой метод Edit. }
begin
  Prop.Edit;
  Designer.Modified;    // Сообщает конструктору о модификации
end;

procedure TWaveEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TddgWaveFile(Component).Play;
    1: TddgWaveFile(Component).Stop;
  end;
end;

function TWaveEditor.GetVerb(Index: Integer): string;
begin
  Result := VerbArray[Index];
end;

function TWaveEditor.GetVerbCount: Integer;
begin
  Result := VerbCount;
end;

end.
```

После определения класса категории остается лишь зарегистрировать свойства для данной категории, используя одну из функций регистрации. Для компонента `TddgWaveFile` это делается в процедуре `Register()` с помощью следующего фрагмента кода:

```
RegisterPropertiesInCategory(TSoundCategory, TddgWaveFile,
                             ['WaveLoop', 'WaveName', 'WavePause']);
```

Списки компонентов: классы TCollection и TCollectionItem

Довольно часто компоненты содержат или владеют списками нескольких элементов —данных, записей, объектов и даже других компонентов. В некоторых случаях удобно инкапсулировать такой список в специальный объект и сделать этот объект свойством владельца компонента. Примером подобного подхода служит свойство Lines компонента TMemo. Данное свойство имеет тип TStrings, инкапсулирующий список строк. При этом объект TStrings отвечает за механизм работы с потоками, используемый для записи строк в файл формы при сохранении этой формы.

Как следует поступить, если требуется сохранить список элементов компонентного или объектного типов, которые еще не инкапсулированы существующим классом, таким как TStrings? В этом случае можно было бы создать класс, обеспечивающий взаимодействие элементов списка с потоками данных и сделать его свойством компонента-владельца. В качестве альтернативного варианта можно переопределить стандартный механизм взаимодействия с потоками данных компонента-владельца так, чтобы он знал, как работать с данным списком элементов. Но самое лучшее решение — воспользоваться преимуществами, предоставляемыми классами TCollection и TCollectionItem.

Класс TCollection — это объект, используемый для хранения списка объектов типа TCollectionItem. Сам класс TCollection — это не компонент, а просто потомок класса TPersistent. Обычно класс TCollection связан с существующим компонентом.

Чтобы воспользоваться возможностями класса TCollection для хранения списка элементов, необходимо создать экземпляр класса, производный от TCollection, назвав его, например, TNewCollection. Этот класс будет служить типом свойства компонента. Затем следует создать новый класс из класса TCollectionItem, который можно назвать TNewCollectionItem. Класс TNewCollection будет содержать список объектов типа TNewCollectionItem. Вся прелесть такого подхода состоит в том, что для того, чтобы принадлежащие классу TNewCollectionItem данные были обработаны обычными средствами взаимодействия с потоками данных, достаточно их опубликовать (объявить в разделе published компонента TNewCollectionItem). Delphi знает, как записывать опубликованные свойства в поток и считывать их из него.

Примером использования класса TCollection может служить компонент TStatusBar, являющийся потомком класса TWinControl. Одно из его свойств, Panels, имеет тип TStatusPanels. Класс TStatusPanels является потомком класса TCollection и определяется следующим образом:

```
type
  TStatusPanels = class(TCollection)
  private
    FStatusBar: TStatusBar;
    function GetItem(Index: Integer): TStatusPanel;
    procedure SetItem(Index: Integer; Value: TStatusPanel);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(StatusBar: TStatusBar);
```

```
    function Add: TStatusPanel;
    property Items[Index: Integer]: TStatusPanel read GetItem
                                        write SetItem; default;
  end;
```

Класс TStatusPanels хранит список потомков класса TCollectionItem — объектов класса TStatusPanel. Ниже приведен код определения этого класса.

```
type
  TStatusPanel = class(TCollectionItem)
  private
    FText: string;
    FWidth: Integer;
    FAlignment: TAlignment;
    FBevel: TStatusPanelBevel;
    FStyle: TStatusPanelStyle;
    procedure SetAlignment(Value: TAlignment);
    procedure SetBevel(Value: TStatusPanelBevel);
    procedure SetStyle(Value: TStatusPanelStyle);
    procedure SetText(const Value: string);
    procedure SetWidth(Value: Integer);
  public
    constructor Create(Collection: TCollection); override;
    procedure Assign(Source: TPersistent); override;
  published
    property Alignment: TAlignment read FAlignment
                        write SetAlignment default taLeftJustify;
    property Bevel: TStatusPanelBevel read FBevel
                        write SetBevel default pbLowered;
    property Style: TStatusPanelStyle read FStyle
                        write SetStyle default psText;
    property Text: string read FText write SetText;
    property Width: Integer read FWidth write SetWidth;
  end;
```

Свойства типа TStatusPanel, объявленные в разделе published класса, автоматически получают возможность взаимодействовать с потоками в Delphi. В классе TStatusPanel предусмотрена передача параметра типа TCollection конструктору Create(), благодаря чему обеспечивается связь с классом TCollection. В данном случае конструктору класса TStatusPanel передается компонент TStatusBar, с которым объект TStatusPanel должен быть связан. Класс TCollection обладает механизмом взаимодействия с потоками данных, являющихся компонентами класса TCollectionItem, а также определяет некоторые методы и свойства для манипулирования элементами, поддерживаемыми в классе TCollection. Их описание можно найти в интерактивной справке.

Для иллюстрации использования этих двух новых классов был создан компонент TddgLaunchPad, позволяющий пользователю хранить список компонентов TddgRunButton (компонент TddgRunButton описан в главе 11, "Разработка компонентов VCL").

Компонент TddgLaunchPad — потомок компонента TScrollBox. Одно из свойств нового компонента — RunButtons — является потомком класса TCollection и поддерживает список компонентов TRunBtnItem. Компонент TRunBtnItem — это потомок класса TCollectionItem, свойства которого используются для создания компонента TddgRun-

Button, помещенного в компонент TddgLaunchPad. В следующих разделах рассматриваются важные моменты, связанные с его созданием.

Определение класса TCollectionItem: компонент TRunBtnItem

Вначале необходимо определить элементы, которые будут содержаться в списке. Для TddgLaunchPad это — компоненты типа TddgRunButton. Таким образом, каждый экземпляр компонента TRunBtnItem должен быть связан с компонентом TddgRunButton. Ниже приводится фрагмент определения класса TRunBtnItem:

```
type
  TRunBtnItem = class(TCollectionItem)
  private
    FCommandLine: String;   // Хранит командную строку
    FLeft: Integer;         // Хранит позиционные свойства
    FTop: Integer;          // для компонента TddgRunButton.
    FRunButton: TddgRunButton; // Указатель на TddgRunButton
    ...
  public
    constructor Create(Collection: TCollection); override;
  published
    { Публикуемые свойства для работы с потоками. }
    property CommandLine: String read FCommandLine
                                write SetCommandLine;
    property Left: Integer read FLeft write SetLeft;
    property Top: Integer read FTop write SetTop;
  end;
```

Обратите внимание, что компонент TRunBtnItem содержит указатель на компонент TddgRunButton и помещает в раздел published только те свойства, которые необходимы для создания компонента TddgRunButton. Можно было бы подумать, что если компонент TRunBtnItem связывается с компонентом TddgRunButton, то достаточно просто опубликовать этот компонент и предоставить остальную работу механизму взаимодействия с потоками. Однако при этом возникают определенные проблемы, связанные с тем, что механизм взаимодействия с потоками по-разному обрабатывает классы TComponent и TPersistent. Основное правило обработки здесь заключается в том, что система взаимодействия с потоками данных отвечает за создание новых экземпляров каждого найденного в потоке класса, произошедшего от класса TComponent; в то же время предполагается, что экземпляры класса TPersistent уже существуют и в создании новых необходимости нет. Следуя такому правилу, записываем в поток информацию, необходимую для компонента TddgRunButton, а затем создаем экземпляр компонента TddgRunButton в конструкторе класса TRunBtnItem.

Определение класса TCollection: компонент TRunButtons

Следующим шагом будет определение объекта, обслуживающего список компонентов TRunBtnItem. Как уже говорилось, этот объект должен быть потомком класса TCollection. Назовем этот класс TRunButtons. Ниже следует его определение.

```
type
  TRunButtons = class(TCollection)
  private
    // Хранит указатель на TddgLaunchPad
    FLaunchPad: TddgLaunchPad;
    function GetItem(Index: Integer): TRunBtnItem;
    procedure SetItem(Index: Integer; Value: TRunBtnItem);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(LaunchPad: TddgLaunchPad);
    function Add: TRunBtnItem;
    procedure UpdateRunButtons;
    property Items[Index: Integer]: TRunBtnItem read GetItem
                                          write SetItem; default;
end;
```

Класс TRunButtons связывает себя с компонентом TddgLaunchPad, который будет рассмотрен чуть ниже. Это происходит в конструкторе Create(), который, как можно заметить, принимает компонент TddgLaunchPad в качестве параметра. Обратите внимание на различные свойства и методы, добавление которых позволило компоненту манипулировать отдельными классами TRunBtnItem, в частности свойством Items, представляющим собой массив для списка компонентов TRunBtnItem.

Использование классов TRunBtnItem и TRunButtons станет более понятным после рассмотрения реализации компонента TddgLaunchPad.

Реализация классов TddgLaunchPad, TRunBtnItem и TRunButtons

Компонент TddgLaunchPad имеет свойство типа TRunButtons. Его реализация, как и реализация компонентов TRunBtnItem и TRunButtons, приведена в листинге 12.11.

Листинг 12.11. LnchPad.pas — иллюстрация реализации компонента TddgLaunchPad

```
unit LnchPad;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, RunBtn, ExtCtrls;
```

```
type

  TddgLaunchPad = class;

  TRunBtnItem = class(TCollectionItem)
  private
    FCommandLine: string;      // Хранит командную строку
    FLeft: Integer;            // Хранит позиционные свойства
    FTop: Integer;             // для компонента TRunButton
    FRunButton: TddgRunButton; // Указатель на TddgRunButton
    FWidth: Integer;           // Хранит размеры
    FHeight: Integer;
    procedure SetCommandLine(const Value: string);
    procedure SetLeft(Value: Integer);
    procedure SetTop(Value: Integer);
  public
    constructor Create(Collection: TCollection); override;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    property Width: Integer read FWidth;
    property Height: Integer read FHeight;
  published
    { Публикуемые свойства для работы с потоками. }
    property CommandLine: String read FCommandLine
                               write SetCommandLine;
    property Left: Integer read FLeft write SetLeft;
    property Top: Integer read FTop write SetTop;
  end;

  TRunButtons = class(TCollection)
  private
    // Хранит указатель на TddgLaunchPad
    FLaunchPad: TddgLaunchPad;
    function GetItem(Index: Integer): TRunBtnItem;
    procedure SetItem(Index: Integer; Value: TRunBtnItem);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(LaunchPad: TddgLaunchPad);
    function Add: TRunBtnItem;
    procedure UpdateRunButtons;
    property Items[Index: Integer]: TRunBtnItem read GetItem
                                        write SetItem; default;
  end;

  TddgLaunchPad = class(TScrollBox)
  private
    FRunButtons: TRunButtons;
    TopAlign: Integer;
    LeftAlign: Integer;
    procedure SetRunButtons(Value: TRunButtons);
    procedure UpdateRunButton(Index: Integer);
  public
```

```
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure GetChildren(Proc: TGetChildProc;
                          Root: TComponent); override;
  published
    property RunButtons: TRunButtons read FRunButtons
                                     write SetRunButtons;
  end;

implementation

{ TRunBtnItem }

constructor TRunBtnItem.Create(Collection: TCollection);
{ Конструктор принимает в качестве параметра коллекцию
     типа TCollection, которая владеет этим элементом TRunBtnItem. }
begin
  inherited Create(Collection);
  { Создает экземпляр компонента FRunButton. Делает объект панели
    загрузки владельцем и родителем. Затем инициализирует его
    различные свойства. }
  FRunButton := TddgRunButton.Create(
                              TRunButtons(Collection).FLaunchPad);
  FRunButton.Parent := TRunButtons(Collection).FLaunchPad;
  FWidth := FRunButton.Width;   // Хранит размер по горизонтали
  FHeight := FRunButton.Height; // и вертикали.
end;

destructor TRunBtnItem.Destroy;
begin
  FRunButton.Free;   // Удалить экземпляр TddgRunButton.
  inherited Destroy; // Вызов унаследованного деструктора Destroy.
end;

procedure TRunBtnItem.Assign(Source: TPersistent);
{ Необходимо переопределить метод TCollectionItem.Assign таким
образом, чтобы он знал, как копировать данные из одного экземпляра
TRunBtnItem в другой. Если это так, то вызывать унаследованный
метод Assign() не нужно. }
begin
  if Source is TRunBtnItem then begin
    { Вместо присвоения командной строки полю FCommandLine
      выполнить присвоение значения соответствующему свойству с
      помощью вызова метода доступа. Метод доступа осуществляет
      все необходимые сопутствующие действия. }
    CommandLine := TRunBtnItem(Source).CommandLine;
    { Копировать значения в остальные поля и выйти из процедуры. }
    FLeft := TRunBtnItem(Source).Left;
    FTop := TRunBtnItem(Source).Top;
    Exit;
  end;
  inherited Assign(Source);
end;
```

```
procedure TRunBtnItem.SetCommandLine(const Value: string);
{ Это метод записи для свойства TRunBtnItem.CommandLine. Он
гарантирует, что экземпляру FRunButton компонента TddgRunButton
будет присвоено заданное значение строки из параметра Value. }
begin
  if FRunButton <> nil then begin
    FCommandLine := Value;
    FRunButton.CommandLine := FCommandLine;
    { Это приводит к вызову метода TRunButtons.Update для
      каждого элемента TRunBtnItem. }
    Changed(False);
  end;
end;

procedure TRunBtnItem.SetLeft(Value: Integer);
{ Метод доступа к свойству TRunBtnItem.Left. }
begin
  if FRunButton <> nil then begin
    FLeft := Value;
    FRunButton.Left := FLeft;
   end;
end;

procedure TRunBtnItem.SetTop(Value: Integer);
{ Метод доступа к свойству TRunBtnItem.Top. }
begin
  if FRunButton <> nil then begin
    FTop := Value;
    FRunButton.Top := FTop;
   end;
end;

{ TRunButtons }

constructor TRunButtons.Create(LaunchPad: TddgLaunchPad);
{ Конструктор заставляет FLaunchPad ссылаться на параметр типа
TddgLaunchPad. LaunchPad является владельцем этой коллекции, и для
доступа к ней необходимо иметь соответствующий указатель. }
begin
  inherited Create(TRunBtnItem);
  FLaunchPad := LaunchPad;
end;

function TRunButtons.GetItem(Index: Integer): TRunBtnItem;
{ Метод доступа для TRunButtons.Items, возвращающий экземпляр
компонента TRunBtnItem. }
begin
  Result := TRunBtnItem(inherited GetItem(Index));
end;

procedure TRunButtons.SetItem(Index: Integer; Value: TRunBtnItem);
{ Метод доступа к свойству TRunButton.Items, позволяющий выполнить
присвоение элементу с заданным индексом. }
begin
```

```
  inherited SetItem(Index, Value)
end;

procedure TRunButtons.Update(Item: TCollectionItem);
{ Метод TCollection.Update вызывается объектами
типа TCollectionItem при изменении одного из элементов коллекции.
Изначально этот метод является абстрактным. Он должен быть
переопределен для включения необходимой логики обработки
изменения элемента TCollectionItem. Используем его для перерисовки
элемента при вызове метода TddgLaunchPad.UpdateRunButton. }
begin
  if Item <> nil then
    FLaunchPad.UpdateRunButton(Item.Index);
end;

procedure TRunButtons.UpdateRunButtons;
{ Открытая процедура UpdateRunButtons предоставляет пользователю
компонентов TRunButtons возможность принудительной перерисовки всех
кнопок. Этот метод вызывает метод TddgLaunchPad.UpdateRunButton для
каждого экземпляра TRunBtnItem. }
var
  i: integer;
begin
  for i := 0 to Count - 1 do
    FLaunchPad.UpdateRunButton(i);
end;

function TRunButtons.Add: TRunBtnItem;
{ Этот метод должен быть переопределен таким образом, чтобы
возвращать экземпляр TRunBtnItem при вызове унаследованного метода
Add. Для этого осуществляется приведение типа результата
унаследованного метода. }
begin
  Result := TRunBtnItem(inherited Add);
end;

{ TddgLaunchPad }

constructor TddgLaunchPad.Create(AOwner: TComponent);
{ Инициализация экземпляра TRunButtons и внутренних переменных,
используемых для позиционирования элемента TRunBtnItem при его
отображении. }
begin
  inherited Create(AOwner);
  FRunButtons := TRunButtons.Create(Self);
  TopAlign := 0;
  LeftAlign := 0;
end;

destructor TddgLaunchPad.Destroy;
begin
  FRunButtons.Free;  // Освобождение экземпляра TRunButtons.
  inherited Destroy; // Вызов унаследованного деструктора.
end;
```

```
procedure TddgLaunchPad.GetChildren(Proc: TGetChildProc;
                                    Root: TComponent);
{ Переопределить метод GetChildren, чтобы компонент TddgLaunchpad
игнорировал любые принадлежащие ему элементы TRunButtons, поскольку
они не нуждаются в потоковой обработке в контексте компонента
TddgLaunchPad. Вся информация, необходимая для создания экземпляров
TRunButton, уже записана в поток в виде публикуемых свойств класса
TRunBtnItem, являющегося потомка класса TCollectionItem. Этот метод
предотвращает двойную запись в поток компонента TRunButton. }
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    { Игнорирует кнопки запуска и поле прокрутки. }
    if not (Controls[i] is TddgRunButton) then
      Proc(TComponent(Controls[I]));
end;

procedure TddgLaunchPad.SetRunButtons(Value: TRunButtons);
{ Метод доступа для свойства RunButtons. }
begin
  FRunButtons.Assign(Value);
end;

procedure TddgLaunchPad.UpdateRunButton(Index: Integer);
{ Этот метод отвечает за отображение экземпляров TRunBtnItem. Он
гарантирует, что элементы TRunBtnItem не будут выходить за пределы
панели TddgLaunchPad, а при необходимости будут созданы новые
строки таких элементов. Это происходит только при
добавлении/удалении элементов TRunBtnItems. Пользователь имеет
возможность так изменить размеры панели TddgLaunchPad, что она
будет меньше ширины строки с элементами TRunBtnItem. }
begin
  { Вначале рисуется первый элемент, обе его позиции обнуляются. }
  if Index = 0 then begin
    TopAlign := 0;
    LeftAlign := 0;
  end;
  { Если ширина текущей строки элементов TRunBtnItem больше
    ширины панели TddgLaunchPad, то создается новая строка
    элементов TRunBtnItem. }
  if (LeftAlign + FRunButtons[Index].Width) > Width then begin
    TopAlign := TopAlign + FRunButtons[Index].Height;
    LeftAlign := 0;
  end;
  FRunButtons[Index].Left := LeftAlign;
  FRunButtons[Index].Top := TopAlign;
  LeftAlign := LeftAlign + FRunButtons[Index].Width;
end;

end.
```

Реализация компонента TRunBtnItem

Конструктор `TRunBtnItem.Create()` создает экземпляр компонента `TddgRunButton`. Каждый элемент `TRunBtnItem` в коллекции обслуживает собственный экземпляр `TddgRunButton`. Следующие две строки конструктора требуют пояснения:

```
FRunButton := TddgRunBut-
ton.Create(TRunButtons(Collection).FLaunchPad);
FRunButton.Parent := TRunButtons(Collection).FLaunchPad;
```

При выполнении первой строки кода создается экземпляр компонента `TddgRunButton` — `FRunButton`. Владельцем экземпляра `FRunButton` является переменная `FLaunchPad`, которая представляет собой экземпляр компонента `TddgLaunchPad` и поле объекта `TCollection`, передаваемого в качестве параметра. Необходимо использовать экземпляр `FLaunchPad` в качестве владельца экземпляра `FRunButton`, потому что ни экземпляр компонента `TRunBtnItem`, ни объект `TRunButton` не могут быть владельцами (так как они являются потомками класса `TPersistent`). Запомните, что владельцем может быть только потомок класса `TComponent`!

Хотелось бы обратить внимание на проблему, возникающую при использовании экземпляра класса `FLaunchPad` в качестве владельца экземпляра `FRunButton`. Экземпляр `FLaunchPad` сделан владельцем `FRunButton` во время разработки. Обычное поведение механизма взаимодействия с потоками заставит Delphi при сохранении формы записать в поток экземпляр класса `FRunButton` как компонент, принадлежащий экземпляру `FLaunchPad`. Но этого как раз и не нужно, поскольку экземпляр `FRunButton` создается в конструкторе компонента `TRunBtnItem` на основе информации, записанной в поток в контексте компонента `TRunBtnItem`. Ниже показано, каким образом предотвратить запись в поток компонентов `TddgRunButton` компонентом `TddgLaunchPad` во избежание подобного нежелательного поведения.

Вторая строка приведенного выше фрагмента кода назначает экземпляр `FLaunchPad` родителем экземпляра `FRunButton`, чтобы тот отвечал за отображение компонента `FRunButton`.

Деструктор `TRunBtnItem.Destroy()` освобождает экземпляр `FRunButton` перед вызовом унаследованного деструктора.

При определенных обстоятельствах необходимо переопределить метод `TrunBtnItem.Assign()`. Примером таких обстоятельств может служить ситуация, когда приложение запускается первый раз и форма считывается из потока. Именно с помощью метода `Assign()` экземпляру компонента `TRunBtnItem` выдается указание присвоить потоковые значения своих свойств свойствам компонента, которого он контролирует (в данном случае это компонент `TddgRunButton`).

Другие методы — это просто методы доступа к различным свойствам компонента `TRunBtnItem` (они разъясняются в комментариях, содержащихся в листинге).

Реализация компонента TRunButtons

Метод `TRunButtons.Create()` присваивает указателю `FLaunchPad` ссылку на параметр типа `TddgLaunchPad` — `LaunchPad`, для того чтобы на него можно было ссылаться в дальнейшем.

Метод `TRunButtons.Update()` вызывается при изменении одного из экземпляров компонента `TRunBtnItem`. Этот метод содержит определенный алгоритм, вы-

полняющийся при такого рода изменении, и используется для вызова метода компонента TddgLaunchPad, который перерисовывает экземпляры TRunBtnItem. Был также добавлен открытый метод UpdateRunButtons(), который принудительно (по желанию пользователя) перерисовывает элементы.

Остальные методы компонента TRunButtons являются методами доступа к свойствам, они описаны в комментариях к листингу 12.11.

Реализация компонента TddgLaunchPad

Конструктор и деструктор компонента TddgLaunchPad не отличаются большой сложностью. Метод TddgLaunchPad.Create() создает экземпляр объекта TRunButtons и передает ему себя в качестве параметра конструктора. Деструктор TddgLaunchPad.Destroy() освобождает экземпляр компонента TRunButtons.

Переопределение метода TddgLaunchPad.GetChildren() — очень важный момент. Благодаря ему предотвращается запись в поток экземпляров компонента TddgRunButton, хранимых в коллекции, как компонентов с владельцем TddgLaunchPad. Запомните: это необходимо, так как они должны создаваться не в контексте объекта TddgLaunchPad, а в контексте экземпляров компонента TRunBtnItem. Поскольку ни один компонент TddgRunButton не передается в процедуру Proc, они не будут записываться или считываться из потока.

Метод TddgLaunchPad.UpdateRunButton() используется для отображения экземпляров TddgRunButton, которые содержатся в коллекции. Логика, реализованная в этом методе, гарантирует, что кнопки никогда не "выйдут" за пределы панели TddgLaunchPad. Поскольку компонент TddgLaunchPad — потомок класса TScrollBox, прокрутка будет осуществляться в вертикальном направлении.

Остальные методы — просто методы доступа к свойствам. Они описаны в комментариях листинга 12.11.

Наконец, регистрируем редактор свойств класса коллекции TRunButtons в процедуре Register() данного модуля. В следующем разделе как раз и рассматривается этот редактор свойств, а также один из способов редактирования списка компонентов в диалоговом окне редактора свойств.

Редактирование списка компонентов TCollectionItem в диалоговом окне редактора свойств

Теперь, когда уже определен компонент TddgLaunchPad, класс коллекции TRunButtons и класс коллекции TRunBtnItem, необходимо обеспечить пользователя способом добавления компонентов TddgRunButton в коллекцию TRunButtons. Лучше всего для этого подходит редактор свойств списка, обслуживаемого коллекцией TRunButtons.

Такое диалоговое окно предназначено для проведения непосредственных манипуляций компонентами TRunBtnItem, содержащимися в коллекции RunButtons типа TddgLaunchPad. В списке PathListBox отображаются различные строки CommandLine для каждого элемента TddgRunButton, "вложенного" в компонент TRunBtnItem. В компоненте TddgRunButton отображается текущий выбранный элемент списка. Диалоговое

окно также содержит кнопки, позволяющие пользователю добавлять и удалять элемент, принимать изменения и отменять операции. По мере того как пользователь вносит изменения в диалоговое окно, они отображаются в компоненте TddgLaunchPad.

СОВЕТ

Для редакторов свойств используется соглашение о включении в них кнопки Apply для подтверждения изменения формы. Здесь не рассматривается добавление кнопки Apply, но читатель может самостоятельно сделать это в качестве упражнения. Чтобы посмотреть, как работает кнопка Apply, обратитесь к исходному коду редактора свойства Panels компонента TStatusBar, расположенного во вкладке Win32 палитры компонентов.

В листинге 12.12 представлен исходный код редактора свойства RunButtons.TddgLaunchPad и его диалогового окна.

Листинг 12.12. LPadPE.pas — редактор свойств TRunButtons

```
unit LPadPE;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Buttons, RunBtn, StdCtrls, LnchPad, DesignIntf,
  DesignEditors, ExtCtrls, TypInfo;

type
  { Вначале объявить диалоговое окно редактора. }
  TLaunchPadEditor = class(TForm)
    PathListBox: TListBox;
    AddBtn: TButton;
    RemoveBtn: TButton;
    CancelBtn: TButton;
    OkBtn: TButton;
    Label1: TLabel;
    pnlRBtn: TPanel;
    procedure PathListBoxClick(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure RemoveBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
  private
    TestRunBtn: TddgRunButton;
    FLaunchPad: TddgLaunchPad;// Используется как резервная копия
    FRunButtons: TRunButtons; // Ссылается на реальный TRunButtons
    Modified: Boolean;
    procedure UpdatePathListBox;
  end;

  { Объявление потомка TPropertyEditor и переопределение
```

```
    необходимых методов. }
  TRunButtonsProperty = class(TPropertyEditor)
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure Edit; override;
  end;

{ Эта функция будет вызываться редактором свойств. }
function EditRunButtons(RunButtons: TRunButtons): Boolean;

implementation

{$R *.DFM}

function EditRunButtons(RunButtons: TRunButtons): Boolean;
{ Создает экземпляр диалогового окна TLaunchPadEditor для
непосредственной модификации коллекции TRunButtons. }
begin
  with TLaunchPadEditor.Create(Application) do
    try
      // Указывает на действительный TRunButtons
      FRunButtons := RunButtons;
      { Копирование компонентов TRunBtnItems в резервный экземпляр
        FLaunchPad, который будет использован, если пользователь
        отменит операцию редактирования. }
      FLaunchPad.RunButtons.Assign(RunButtons);
      { Отображение в окне списка компонентов TRunBtnItem. }
      UpdatePathListBox;
      ShowModal; // Отобразить форму.
      Result := Modified;
    finally
      Free;
    end;
end;

{ TLaunchPadEditor }

procedure TLaunchPadEditor.FormCreate(Sender: TObject);
begin
  { Создает резервные экземпляры компонента TLaunchPad для
    использования в том случае, если пользователь отменит
    редактирование элементов TRunBtnItem. }
  FLaunchPad := TddgLaunchPad.Create(Self);

  // Создает экземпляр компонента TddgRunButton и выравнивает его
  // относительно содержащей его панели.
  TestRunBtn := TddgRunButton.Create(Self);
  TestRunBtn.Parent := pnlRBtn;

  TestRunBtn.Width  := pnlRBtn.Width;
  TestRunBtn.Height := pnlRBtn.Height;
end;
```

```
procedure TLaunchPadEditor.FormDestroy(Sender: TObject);
begin
  TestRunBtn.Free;
  FLaunchPad.Free; // Освобождает экземпляр TLaunchPad.
end;

procedure TLaunchPadEditor.PathListBoxClick(Sender: TObject);
{ Если пользователь щелкает на элементе списка TRunBtnItem,
проверить компонент TRunButton, соответствующий выбранному
элементу. }
begin
  if PathListBox.ItemIndex > -1 then
    TestRunBtn.CommandLine :=
        PathListBox.Items[PathListBox.ItemIndex];
end;

procedure TLaunchPadEditor.UpdatePathListBox;
{ Повторная инициализация компонента PathListBox для обновления
списка элементов TRunBtnItem. }
var
  i: integer;
begin
  PathListBox.Clear; // Вначале очистить окно списка.
  for i := 0 to FRunButtons.Count - 1 do
    PathListBox.Items.Add(FRunButtons[i].CommandLine);
end;

procedure TLaunchPadEditor.AddBtnClick(Sender: TObject);
{ Если был щелчок на кнопке Add, вызывается диалог TOpenDialog для
получения имени и пути выполняемого файла (этот файл добавляется в
PathListBox), а также добавляется новый элемент FRunBtnItem. }
var
  OpenDialog: TOpenDialog;
begin
  OpenDialog := TOpenDialog.Create(Application);
  try
    OpenDialog.Filter := 'Executable Files|*.EXE';
    if OpenDialog.Execute then begin
      { Добавить в PathListBox. }
      PathListBox.Items.Add(OpenDialog.FileName);
      FRunButtons.Add; // Создать новый экземпляра TRunBtnItem.
      { Установить фокус на новый элемент в списке PathListBox }
      PathListBox.ItemIndex := FRunButtons.Count - 1;
      { Установка командной строки для нового элемента TRunBtnItem
        равной имени файла, заданного
        значением PathListBox.ItemIndex. }
      FRunButtons[PathListBox.ItemIndex].CommandLine :=
        PathListBox.Items[PathListBox.ItemIndex];
      { Вызов обработчика события PathListBoxClick для проверки
        компонента TRunButton, соответствующего вновь добавленному
        элементу. }
      PathListBoxClick(nil);
      Modified := True;
```

```
    end;
  finally
    OpenDialog.Free
  end;
end;

procedure TLaunchPadEditor.RemoveBtnClick(Sender: TObject);
{ Удаление выбранного пути/имени файла из PathListBox одновременно
с удалением соответствующего элемента TRunBtnItem из экземпляра
коллекции FRunButtons. }
var
  i: integer;
begin
  i := PathListBox.ItemIndex;
  if i >= 0 then begin
    PathListBox.Items.Delete(i); // Удаление элемента из списка
    FRunButtons[i].Free;         // Удаление элемента из коллекции
    TestRunBtn.CommandLine := '';// Удаление кнопки проверки
    Modified := True;
  end;
end;

procedure TLaunchPadEditor.CancelBtnClick(Sender: TObject);
{ Если пользователь отменяет операцию, элементы TRunBtnItem
резервного экземпляра LaunchPad копируются в исходный экземпляр
TLaunchPad, а затем, после установки для ModalResult значения
mrCancel, форма закрывается. }
begin
  FRunButtons.Assign(FLaunchPad.RunButtons);
  Modified := False;
  ModalResult := mrCancel;
end;

{ TRunButtonsProperty }

function TRunButtonsProperty.GetAttributes: TPropertyAttributes;
{ Сообщает инспектору объектов о том, что данный редактор свойства
будет использовать диалоговое окно. По щелчку на кнопке с
многоточием в окне Object Inspector будет вызываться метод Edit. }
begin
  Result := [paDialog];
end;

procedure TRunButtonsProperty.Edit;
{ Вызов метода EditRunButton() и передача ему указателя на
редактируемый экземпляр компонента TRunButton. Этот указатель
получен с помощью метода GetOrdValue. Затем следует перерисовка
диалогового окна LaunchDialog с использованием вызова
                          метода TRunButtons.UpdateRunButtons. }
begin
  if EditRunButtons(TRunButtons(GetOrdValue)) then
    Modified;
  TRunButtons(GetOrdValue).UpdateRunButtons;
```

```
end;

function TRunButtonsProperty.GetValue: string;
{ Переопределение метода GetValue таким образом, чтобы тип класса
редактируемого свойства отображался в окне инспектора объектов. }
begin
  Result := Format('(%s)', [GetPropType^.Name]);
end;

end.
```

В приведенном модуле сначала определяется диалоговое окно `TLaunchPadEditor`, а затем — редактор свойства `TRunButtonsProperty`. Рассмотрим сначала редактор свойства, так как он и вызывает это диалоговое окно.

Редактор свойства `TRunButtonsProperty` мало отличается от редактора свойства с диалоговым окном, описанного выше. В нем переопределяются методы `GetAttributes()`, `Edit()` и `GetValue()`.

Метод `GetAttributes()` просто возвращает значение типа `TPropertyAttributes`, определяющее, что данный редактор вызывает диалоговое окно, а также помещает кнопку с многоточием в окно инспектора объектов.

Метод `GetValue()` использует функцию `GetPropType()` для получения указателя на информацию о типах времени выполнения редактируемого свойства. Он возвращает поле имени этой информации, представляющее строку типа данного свойства. Данная строка отображается внутри скобок в окне инспектора объектов, как это принято в Delphi.

Наконец, метод `Edit()` вызывает функцию `EditRunButtons()`, определенную в данном модуле. Ей в качестве параметра передается указатель на свойство `TRunButtons` (с помощью функции `GetOrdValue`). При возврате из функции вызывается метод `UpdateRunButtons()` для перерисовки изображения коллекции `RunButtons`, отражающий все произошедшие в ней изменения.

Функция `EditRunButtons()` создает экземпляр компонента `TLaunchPadEditor` и помещает в его поле `FRunButtons` указатель на `TRunButtons`, переданный в нее в качестве параметра. Этот указатель используется для внесения изменений в коллекцию `TRunButtons`. Затем функция копирует коллекцию свойств `TRunButtons` во внутренний экземпляр компонента `TddgLaunchPad` по имени `FLaunchPad`. Функция использует этот экземпляр как резервный на тот случай, если пользователь отменит операцию редактирования.

Выше упоминалась возможность добавления кнопки Apply в диалоговое окно. Для этого вместо непосредственной модификации реальной коллекции можно отредактировать экземпляр коллекции `RunButtons` компонента `FLaunchPad`. Тогда, если пользователь отменит операцию редактирования, ничего не произойдет; если же он нажмет кнопку Apply или OK, то внесенные изменения подтвердятся.

Конструктор формы `Create()` создает внутренний экземпляр компонента `TddgLaunchPad`. Деструктор `Destroy()` освобождает его перед уничтожением формы.

Метод `PathListBoxClick()` представляет собой обработчик события `OnClick` компонента `PathListBox`. Этот метод заставляет компонент `TestRunBtn` (тестовый компонент `TddgRunButton`) отображать элемент, выбранный в данный момент в спи-

ске PathListBox, а также путь к выполняемому файлу. Пользователь может нажать кнопку такого экземпляра TddgRunButton для запуска приложения.

Метод UpdatePathListBox() инициализирует список PathListBox элементами коллекции.

Метод AddButtonClick() представляет собой обработчик события OnClick кнопки Add. Данный обработчик события вызывает диалоговое окно File Open, чтобы принять от пользователя имя выполняемого файла. При этом в список PathListBox добавляется путь к этому файлу. Метод AddButtonClick() также создает экземпляр компонента TRunBtnItem в коллекции и присваивает командную строку свойству TRunBtnItems.CommandLine. Это значение тут же передается в компонент TddgRunButton, заключенный в данный компонент TRunBtnItems.

Метод RemoveBtnClick() представляет собой обработчик события OnClick кнопки Remove. Он удаляет выбранный элемент из списка PathListBox и экземпляр TRunBtnItem из коллекции.

Метод CancelBtnClick() представляет собой обработчик события OnClick кнопки Cancel. Он копирует резервную коллекцию из экземпляра FLaunchPad в действительную коллекцию TRunButtons и закрывает форму.

Объекты TCollection и TCollectionItem могут оказаться весьма полезными во многих отношениях. Внимательно их изучите, а как только потребуется сохранить список компонентов, можно будет применить уже готовое решение.

Резюме

В настоящей главе рассматривались более сложные методики и приемы, используемые при разработке компонентов в Delphi. Здесь были продемонстрированы способы расширения подсказки, анимационные компоненты, редакторы компонентов, редакторы свойств и коллекции компонентов. Вооружившись всей этой информацией, можно разработать любой необходимый компонент.

ГЛАВА

13

Разработка компонентов CLX

В ЭТОЙ ГЛАВЕ...

Предыдущие три главы были посвящены созданию в Delphi пользовательских компонентов. Если говорить точнее, глава 11, "Разработка компонентов VCL", и 12, "Создание расширенного компонента VCL", были посвящены вопросам создания пользовательских компонентов VCL. Однако, как было сказано в главе 10, "Архитектура компонентов: VCL и CLX", в Delphi 6 используется две иерархии классов компонентов: VCL и CLX. Поэтому настоящий раздел полностью посвящен разработке пользовательских компонентов CLX. К счастью, многое из того, что имеет отношение к созданию компонентов VCL, также применимо и к разработке компонентов CLX.

Что такое CLX?

CLX — это аббревиатура термина *Component Library for Cross-Platform* (библиотека межплатформенных компонентов), который был впервые употреблен в новом инструменте ускоренной разработки приложений Kylix для Linux. Но библиотека CLX не является просто аналогом VCL для Linux. Архитектура CLX также используется и в Delphi 6, что является основанием для разработки межплатформенных приложений с использованием Delphi 6 и Kylix.

В Delphi библиотека VCL обычно ассоциируется с компонентами, расположенными в палитре компонентов. Это и не удивительно, так как большинство из компонентов VCL являются визуальными элементами управления. Впрочем, библиотека CLX представляет собой не просто иерархию визуальных компонентов. В частности, она разделена на четыре независимые части: BaseCLX, VisualCLX, DataCLX и NetCLX.

BaseCLX содержит базовые модули и классы, которые используются как в Kylix, так и в Delphi. Например, частью BaseCLX являются модули `System`, `SysUtils` и `Classes`. VisualCLX — это аналог VCL, который основан, однако, не на стандартных библиотеках Windows `User32.dll` и `ComCtl32.dll`, а на библиотеке `Qt`. DataCLX содержит компоненты для доступа к данным, которые ориентированы на новую технологию dbExpress. И, наконец, NetCLX содержит компоненты, реализующие новую межплатформенную технологию WebBroker.

Те, кто знаком с предыдущими версиями Delphi, знают, что модули, заключенные в BaseCLX, использовались начиная с Delphi 1. Таким образом, можно сказать, что они являются также и частью VCL. Фактически корпорация *Borland* приняла во внимание возникшую путаницу в названиях, и в Delphi 6 эти базовые модули были отнесены к библиотеке RTL.

Базовые модули используются как в приложениях VCL, так и в приложениях CLX. В последних они применяются неявно, в составе VisualCLX.

Настоящая глава посвящена VisualCLX. В частности, здесь рассматриваются способы расширения архитектуры VisualCLX за счет создания собственных компонентов CLX. Как уже упоминалось ранее, в основе VisualCLX лежит библиотека Qt, разработанная компанией *Troll Tech*. Эта библиотека содержит классы C++, предназначенные для разработки *пользовательского интерфейса* (UI — User Interface), и не зависит от платформы. Точнее библиотека Qt может использоваться в системах Windows и X Window (Linux). Фактически Qt — это наиболее распространенная библиотека классов, которая используется при разработке графических пользовательских интерфейсов в среде Linux. Например, данная библиотека использовалась при разработке программы KDE Window Manager.

Конечно, существуют и другие межплатформенные библиотеки классов, но компания *Borland* решила в качестве основания для VisualCLX избрать Qt. На это существует несколько причин. Во-первых, классы Qt очень похожи на классы компонентов VCL. Например, свойства определяются двумя методами записи/чтения. Кроме того, в Qt события обрабатываются при помощи так называемого *сигнального механизма* (signal). К тому же графическая модель Qt очень похожа на ту, которая используется в VCL. И, наконец, в Qt определен большой набор стандартных элементов управления, которые в номенклатуре Qt называются *widgets*. Многие из этих элементов управления стали основой для разработки специалистами *Borland* новых компонентов на языке Object Pascal.

Архитектура CLX

Итак, библиотека VisualCLX состоит из классов Object Pascal, по сути являющихся оболочками классов библиотеки Qt. Это очень похоже на то, как классы библиотеки VCL инкапсулируют стандартные интерфейсы API и элементы управления Windows. Одной из главных целей создания CLX было максимальное упрощение переноса существующих приложений VCL в архитектуру CLX. В результате иерархия классов CLX имеет общие черты с иерархией классов VCL (рис. 13.1 и 13.2). Темно-серые прямоугольники на рис. 13.1 соответствуют основным базовым классам VCL.

Тем не менее, иерархии классов в этих двух библиотеках не являются полностью идентичными. В частности, в CLX добавлено несколько новых классов, а некоторые из них были перемещены в другие ветви иерархии. На рис. 13.2 эти отличия отмечены светло-серыми прямоугольниками. Например, класс компонента CLX `TTimer` больше не является прямым потомком класса `TComponent`, как это было в VCL. Теперь он является производным от нового класса `THandleComponent`, который, в свою очередь, является базовым для тех невизуальных компонентов, которые основаны на элементах управления Qt. Также обратите внимание на компонент CLX `TLabel`, который больше не является графическим элементом управления. Теперь его класс стал производным от нового класса `TFrameControl`. Библиотека Qt предоставляет широкие возможности настройки границ элементов управления, реализованных в классе `TFrameControl`.

Таким образом, класс `TWidgetControl` в CLX является эквивалентом класса `TWinControl` в VCL. Изменение названия класса связано с переходом на базу библиотеки Qt, в которой вместо приставки `Win` используется слово `Widget`. Этим была подчеркнута независимость элементов управления VisualCLX от системы Windows.

Следует отметить, что класс `TWinControl` также определен в CLX как псевдоним для класса `TWidgetControl`. При разработке Kylix и CLX одной из самой первых идей была возможность использования одного ресурсного файла для определения как компонентов CLX, так и компонентов VCL. Предполагалось, что во время компиляции приложения будут использованы директивы условной компиляции, которые, определив платформу, внесут в раздел `uses` соответствующие модули (VCL или CLX). Однако такой подход годится только для очень простых компонентов. На практике, при разработке совместно используемых модулей, в исходный код приложений приходится вносить более значительные изменения.

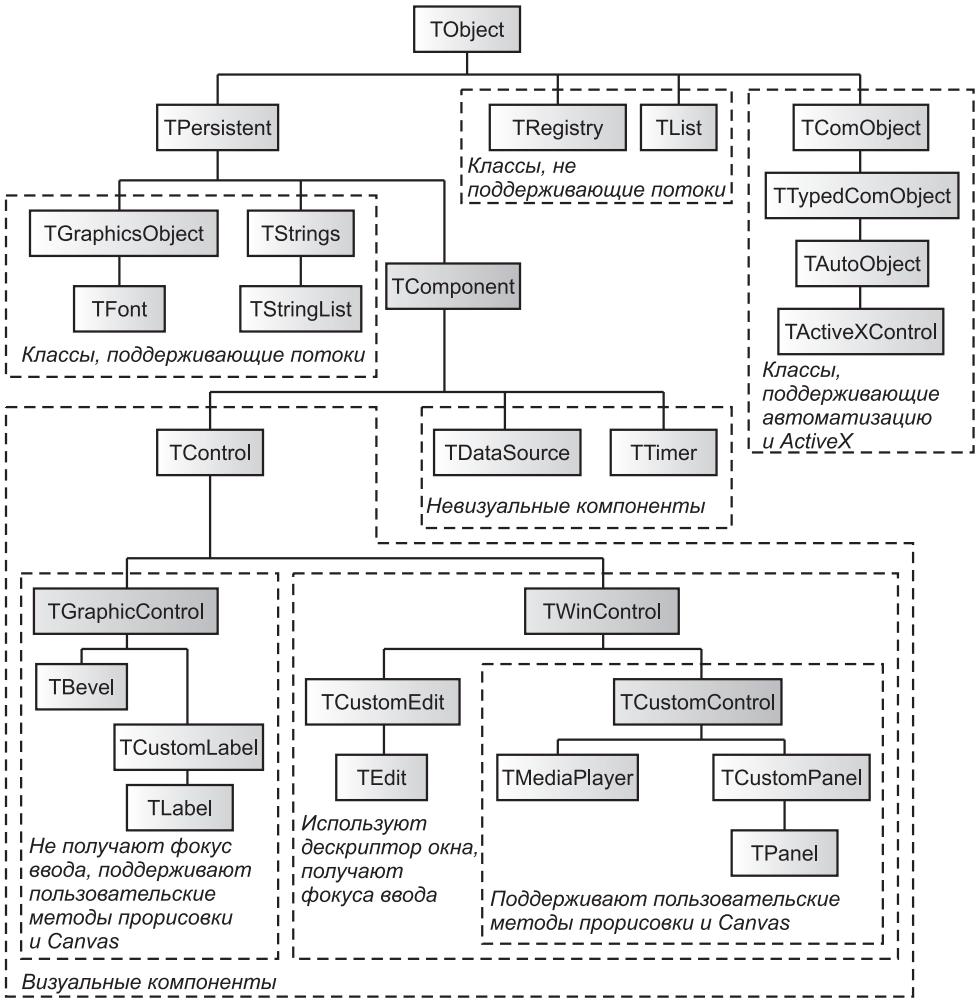

Рис. 13.1. *Иерархия базовых классов VCL*

НА ЗАМЕТКУ

Разработка компонента, который использовал бы один и тот же исходный файл для VCL и CLX, — это не то же самое, что разработка компонента CLX, который может использоваться одновременно и в Delphi 6 и в Kylix. В настоящей главе описывается именно последний процесс.

К счастью, изменения в иерархии классов, представленной на рис. 13.2, мало влияют на сам процесс разработки приложений. Для большинства компонентов VCL, входящих в состав Delphi, существуют эквиваленты в VisualCLX (например `TEdit`, `TListBox`, `TComboBox` и т.д.). Изменения в иерархии классов больше затрагивают разработчиков компонентов.

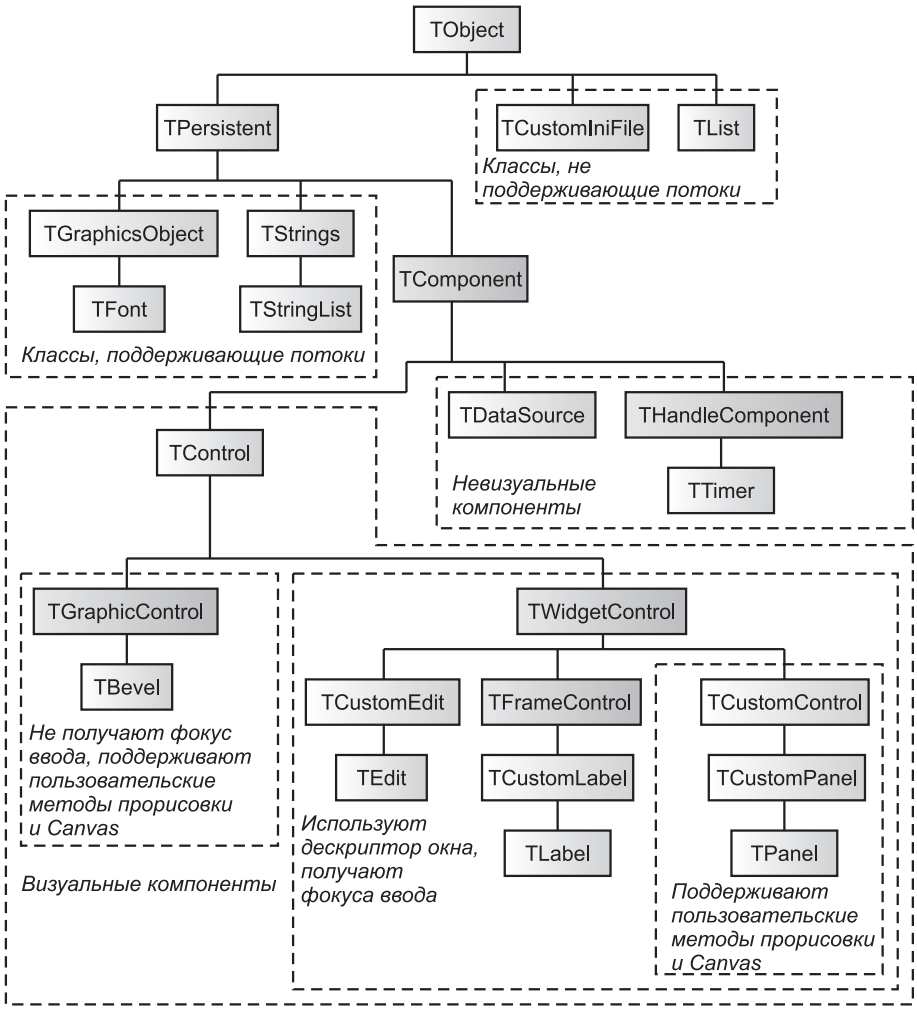

Рис. 13.2. *Иерархия базовых классов CLX*

НА ЗАМЕТКУ

Чрезвычайно удобным средством для изучения новой иерархии классов является броузер объектов (Object Browser), входящий в состав Delphi 6 и Kylix. Если открыть в нем класс `TWinControl`, являющийся псевдонимом класса `TWidgetControl`, то можно увидеть, фактически, две идентичных иерархии классов.

Между структурами VCL и CLX гораздо больше сходства, чем можно увидеть на представленных рисунках. Например, класс `TCanvas` в обеих структурах практически идентичен (хотя его внутренняя реализация для VCL и CLX, конечно же, отличается). Для VCL класс `TCanvas` предоставляет доступ к графическому контексту устройства вывода в системе Windows через свойство `TCanvas.Handle`. Для CLX класс `TCanvas`

через то же свойство `TCanvas.Handle` предоставляет доступ к модулю прорисовки библиотеки Qt. Таким образом свойство `Handle` используется для доступа к низкоуровневым функциям графического интерфейса устройств вывода как в приложениях VCL, так и в приложениях CLX.

Компоненты CLX были разработаны так, чтобы упростить преобразование уже существующих приложений VCL в приложения CLX. В результате интерфейсы категории `public` и `published` во многих компонентах практически идентичны для обоих архитектур. Это означает, что такие события как `OnClick`, `OnChange` и `OnKeyPress` (а также соответствующие им методы обработки `Click()`, `Change()` и `KeyPress()`), реализованы и в VCL, и в CLX.

Преобразование приложений

Библиотека CLX имеет много общего с VCL. Тем не менее, между ними существует и немало отличий, связанных с различием платформ. Этот касается в основном разработчиков компонентов, так как им придется учитывать в программном коде различия операционных систем. Например, все обращения к функциям интерфейса API Win32 (описаны в модуле `Windows.pas`) должны быть заменены, если компонент будет использоваться в Kylix.

Кроме того, при преобразовании приложений следует учитывать несколько вопросов, связанных с динамическими библиотеками RTL (например чувствительность к регистру символов в именах файлов и символы-разделители в путях). Некоторые компоненты VCL перенести в среду Linux просто невозможно. Это относится, в частности, к компоненту VCL для доступа к интерфейсу электронной почты Messaging API (MAPI). MAPI в Linux не существует, вследствие чего для создания подобного компонента необходимо использовать другие механизмы.

При преобразовании приложений для использования в среде Linux встречаются некоторые нюансы, связанные с самой платформой. Например, модель COM непосредственно в Linux не используется, однако поддерживаются некоторые ее интерфейсы. Также не рекомендуется применять в компонентах CLX технологию Owner-Draw, использующуюся во многих элементах управления VCL для среды Windows, потому что она конфликтует со стилями Qt. К другим особенностям VCL, которые не поддерживаются в CLX, относятся: закрепление объектов, поддержка двунаправленных наборов данных и азиатских символов.

К тому же разработчики часто сталкиваются с тем, что для одних и тех же элементов управления в CLX и VCL применяются различные модули. Например, модулю VCL `Controls.pas` в CLX соответствует модуль `QControls.pas`. В данном случае проблема заключается в том, что при разработке компонента или приложения CLX, в Delphi 6 происходит обращение и к модулям VCL. Это часто приводит к случайному смешению вызовов модулей CLX и VCL в модулях компонентов. В некоторых случаях такие компоненты нормально работают в среде Windows, однако при их переносе в Kylix компилятор выдаст сообщение об ошибке, так как модули VCL в Linux не используются.

НА ЗАМЕТКУ

Специалисты *Borland* рекомендуют разрабатывать компоненты CLX при помощи Kylix в среде Linux. Это позволит избежать случайного применения модулей VCL. Но многие разработчики, скорее всего, все равно предпочтут работать в Delphi 6, а затем проверять компоненты в Kylix.

Кроме всего прочего, при разработке компонентов и приложений CLX необходимо учитывать чувствительность системы Linux к регистру символов. В частности, это имеет значение при работе с именами файлов и путями. Эту особенность необходимо учитывать также и в разделе `uses`, при указании имен модулей, поскольку при отличии регистра символов в именах модули не будут найдены компилятором Kylix. Пакет Delphi не чувствителен к регистру символов, однако при работе с ним к разработчикам тоже предъявляются соответствующие требования. Например, регистр символов должен строго соблюдаться в имени процедуры `Register()`, которая используется в модулях, экспортируемых из пакета.

Обойдемся без сообщений

В системе Linux (или, если говорить точнее, XWindows) нет такой системы сообщений, как в Windows (например `wm_LButtonDown`, `wm_SetCursor` или `wm_Char`). Когда компонент CLX используется в среде Linux, системные сообщения обрабатываются соответствующими классами Qt. При этом очень важным моментом является то, что сообщения обрабатываются классами Qt даже в среде Windows.

Учитывая вышеизложенное, методы обработки сообщений, которые используются в компонентах VCL (в частности `CMTextChanged()`), должны быть заменены динамическими методами (например `TextChanged()`). Более подробно данный вопрос будет рассмотрен в следующих разделах. Сейчас же отметим лишь то, что в некоторых случаях сообщения в VCL обрабатываются совсем не так, как в CLX.

Простые компоненты

В настоящем разделе будут подробно рассмотрены некоторые компоненты VCL, преобразованные в компоненты CLX. Первый из них — это компонент `Spinner`, в котором реализованы такие базовые возможности, как прорисовка, обработка нажатий клавиш, перенос фокуса, работа с мышью и обработка событий.

Следующие три компонента являются расширениями базового компонента `Spinner`, причем каждый следующий расширяет предыдущий. В первом компоненте-потомке добавлена обработка событий мыши и отображение пользовательских курсоров; во втором — отображение изображений, извлеченных из компонента `ImageList`; а в третьем — подключение к одному их полей набора данных.

НА ЗАМЕТКУ

Все модули, представленные в настоящей главе, могут использоваться как в Delphi 6, так и в Kylix.

Компонент TddgSpinner

На рис. 13.3 показаны три экземпляра компонента класса TddgSpinner, которые могут использоваться в приложениях CLX. В традиционном компоненте SpinEdit кнопки инкремента/декремента значения располагаются одна над другой. В представленном расширенном компоненте эти кнопки находятся на противоположных сторонах.

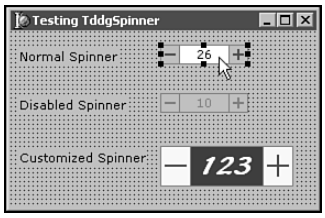

Рис. 13.3. *Компонент CLX TddgSpinner используется для ввода целочисленных значений*

В листинге 13.1 представлен полный исходный код модуля QddgSpin.pas, где реализован компонент TddgSpinner. Класс данного компонента является производным от класса VCL TCustomControl, который реализован также и в CLX. В связи с тем что класс TddgSpinner происходит теперь от класса CLX TCustomControl, его можно применять как в Windows, так и в Linux.

Названия классов редко изменяются при переносе компонентов в CLX, а вот к именам модулей обычно добавляется буква "Q", что указывает на их отношение к библиотеке Qt.

НА ЗАМЕТКУ

Во всех листингах в комментариях указан код VCL.
Комментарии, которые начинаются с VCL->CLX:, указывают на специфические фрагменты программного кода, имеющие отношение к преобразованию из VCL в CLX.

Листинг 13.1. QddgSpin.pas — исходный код компонента TddgSpinner

```
unit QddgSpin;

interface

uses
  SysUtils, Classes, Types, Qt, QControls, QGraphics;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ImgList;
  *)

type
  TddgButtonType = ( btMinus, btPlus );
  TddgSpinnerEvent = procedure (Sender: TObject;
                                NewValue: Integer;
                                var AllowChange: Boolean ) of object;
```

```
TddgSpinner = class( TCustomControl )
private
  // Данные экземляра компонента
  FValue: Integer;
  FIncrement: Integer;
  FButtonColor: TColor;
  FButtonWidth: Integer;
  FMinusBtnDown: Boolean;
  FPlusBtnDown: Boolean;

  // Указатели на методы обработки пользовательских событий
  FOnChange: TNotifyEvent;
  FOnChanging: TddgSpinnerEvent;

  (*
  // VCL->CLX:  Эти обработчики сообщений в CLX не используются

  // Метод обработки сообщений окна
  procedure WMGetDlgCode( var Msg: TWMGetDlgCode );
    message wm_GetDlgCode;

  // Метод обработки сообщений компонента
  procedure CMEnabledChanged( var Msg: TMessage );
    message cm_EnabledChanged;
  *)
protected
  procedure Paint; override;
  procedure DrawButton( Button: TddgButtonType; Down: Boolean;
                        Bounds: TRect ); virtual;

  // Вспомогательные методы
  procedure DecValue( Amount: Integer ); virtual;
  procedure IncValue( Amount: Integer ); virtual;

  function CursorPosition: TPoint;
  function MouseOverButton( Btn: TddgButtonType ): Boolean;

  // VCL->CLX:  EnabledChanged заменяет обработчик сообщения
  //            компонента cm_EnabledChanged
  procedure EnabledChanged; override;

  // Методы обработки новых событий
  procedure Change; dynamic;
  function CanChange( NewValue: Integer ): Boolean; dynamic;

  // Переопределенные методы обработки событий
  procedure DoEnter; override;
  procedure DoExit; override;
  procedure KeyDown(var Key: Word;
                    Shift: TShiftState); override;

  procedure MouseDown( Button: TMouseButton; Shift: TShiftState;
                       X, Y: Integer ); override;
```

```
  procedure MouseUp( Button: TMouseButton; Shift: TShiftState;
                     X, Y: Integer ); override;

  (*
  // VCL->CLX:  Следующие объявления в CLX были изменены

  function DoMouseWheelDown( Shift: TShiftState;
                          MousePos: TPoint ): Boolean; override;

  function DoMouseWheelUp( Shift: TShiftState;
                          MousePos: TPoint ): Boolean; override;
  *)

  function DoMouseWheelDown( Shift: TShiftState;
                    const MousePos: TPoint ): Boolean; override;

  function DoMouseWheelUp( Shift: TShiftState;
                    const MousePos: TPoint ): Boolean; override;

  // Методы доступа к свойствам
  procedure SetButtonColor( Value: TColor ); virtual;
  procedure SetButtonWidth( Value: Integer ); virtual;
  procedure SetValue( Value: Integer ); virtual;
public
  // Не забудьте указать для конструктора директиву override
  constructor Create( AOwner: TComponent ); override;
published
  // Объявления новых свойств
  property ButtonColor: TColor
    read FButtonColor
    write SetButtonColor
    default clBtnFace;

  property ButtonWidth: Integer
    read FButtonWidth
    write SetButtonWidth
    default 18;

  property Increment: Integer
    read FIncrement
    write FIncrement
    default 1;

  property Value: Integer
    read FValue
    write SetValue;

  // Объявления новых событий
  property OnChange: TNotifyEvent
    read FOnChange
    write FOnChange;

  property OnChanging: TddgSpinnerEvent
```

```
      read FOnChanging
      write FOnChanging;

    // Унаследованные свойства и события
    property Color;
    (*
    property DragCursor;     // VCL->CLX: В CLX этого свойства нет
    *)
    property DragMode;
    property Enabled;
    property Font;
    property Height default 18;
    property HelpContext;
    property Hint;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property TabOrder;
    property TabStop default True;
    property Visible;
    property Width default 80;
    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDrag;
  end;

implementation

{=========================}
{== Методы TddgSpinner  ==}
{=========================}

constructor TddgSpinner.Create( AOwner: TComponent );
begin
  inherited Create( AOwner );

  // Инициализация данных экземпляра
  FButtonColor := clBtnFace;
  FButtonWidth := 18;
  FValue := 0;
  FIncrement := 1;
  FMinusBtnDown := False;
  FPlusBtnDown := False;
```

```
  // Инизиализация унаследованных свойств
  Width := 80;
  Height := 18;
  TabStop := True;

  // VCL->CLX:  TWidgetControl устанавливает свойство Color в
  //            состояние clNone
  Color := clWindow;

  // VCL->CLX:  InputKeys используется вместо обработки
  //            сообщения wm_GetDlgCode
  InputKeys := InputKeys + [ ikArrows ];
end;

{== Методы доступа к свойствам ==}

procedure TddgSpinner.SetButtonColor( Value: TColor );
begin
  if FButtonColor <> Value then begin
    FButtonColor := Value;
    Invalidate;
  end;
end;

procedure TddgSpinner.SetButtonWidth( Value: Integer );
begin
  if FButtonWidth <> Value then begin
    FButtonWidth := Value;
    Invalidate;
  end;
end;


procedure TddgSpinner.SetValue( Value: Integer );
begin
  if FValue <> Value then begin
    if CanChange( Value ) then begin
      FValue := Value;
      Invalidate;

      // Передача события Change
      Change;
    end;
  end;
end;

{== Методы прорисовки ==}

procedure TddgSpinner.Paint;
var
  R: TRect;
  YOffset: Integer;
```

```
  S: string;
  XOffset: Integer;             // VCL->CLX: Добавлено для CLX
begin
  inherited Paint;
  with Canvas do begin
    Font := Self.Font;
    Pen.Color := clBtnShadow;
    if Enabled then
      Brush.Color := Self.Color
    else begin
      Brush.Color := clBtnFace;
      Font.Color := clBtnShadow;
    end;

    // Отображение значения
    (*
    // VCL->CLX: SetTextAlign в CLX не используется
    SetTextAlign( Handle, ta_Center or ta_Top );  // функция GDI
    *)

    R := Rect( FButtonWidth - 1, 0,
               Width - FButtonWidth + 1, Height );
    Canvas.Rectangle( R.Left, R.Top, R.Right, R.Bottom );
    InflateRect( R, -1, -1 );

    S := IntToStr( FValue );
    YOffset := R.Top + ( R.Bottom - R.Top -
                         Canvas.TextHeight( S ) ) div 2;

    // VCL->CLX: Вместо функции SetTextAlign используется XOffset
    XOffset := R.Left + ( R.Right - R.Left -
                          Canvas.TextWidth( S ) ) div 2;

    (*
    // VCL->CLX: Процедура TextRect изменена
    TextRect( R, Width div 2, YOffset, S );
    *)
    TextRect( R, XOffset, YOffset, S );

    DrawButton( btMinus, FMinusBtnDown,
                Rect( 0, 0, FButtonWidth, Height ) );
    DrawButton( btPlus, FPlusBtnDown,
                Rect( Width - FButtonWidth, 0, Width, Height ) );

    if Focused then begin
      Brush.Color := Self.Color;
      DrawFocusRect( R );
    end;
  end;
end; {= TddgSpinner.Paint =}

procedure TddgSpinner.DrawButton( Button: TddgButtonType;
                                  Down: Boolean; Bounds: TRect );
```

```
begin
  with Canvas do begin
    if Down then                              // Установка цвета фона
      Brush.Color := clBtnShadow
    else
      Brush.Color := FButtonColor;
    Pen.Color := clBtnShadow;
    Rectangle( Bounds.Left, Bounds.Top,
               Bounds.Right, Bounds.Bottom );

    if Enabled then begin
      (*
      // VCL->CLX: clActiveCaption для CLX заменено на
      //           clActiveHighlightedText
      Pen.Color := clActiveCaption;
      Brush.Color := clActiveCaption;
      *)
      Pen.Color := clActiveBorder;
      Brush.Color := clActiveBorder;
    end
    else begin
      Pen.Color := clBtnShadow;
      Brush.Color := clBtnShadow;
    end;

    if Button = btMinus then begin    // Прорисовка кнопки "Минус"

      Rectangle( 4, Height div 2 - 1,
                 FButtonWidth - 4, Height div 2 + 1 );
    end
    else begin                            // Прорисовка кнопки "Плюс"
      Rectangle( Width - FButtonWidth + 4, Height div 2 - 1,
                 Width - 4, Height div 2 + 1 );
      Rectangle( Width - FButtonWidth div 2 - 1,
                 ( Height div 2 ) - (FButtonWidth div 2 - 4),
                 Width - FButtonWidth div 2 + 1,
                 ( Height div 2 ) + (FButtonWidth div 2 - 4)  );
    end;
    Pen.Color := clWindowText;
    Brush.Color := clWindow;
  end;
end; {= TddgSpinner.DrawButton =}

procedure TddgSpinner.DoEnter;
begin
  inherited DoEnter;
  // Элемент управления принимает фокус --
  // обновление изображения для прорисовки границы фокуса
  Repaint;
end;

procedure TddgSpinner.DoExit;
begin
```

```
  inherited DoExit;
  // Элемент управления теряет фокус --
  // обновление изображения для удаления границы фокуса
  Repaint;
end;

// VCL->CLX:  EnabledChanged заменяет обработчик
//            сообщения cm_EnabledChanged

procedure TddgSpinner.EnabledChanged;
begin
  inherited;
  // Перерисовка компонента для отображения изменения
  // его состояния
  Repaint;
end;


{== Методы обработки событий ==}

{=====================================================================
  TddgSpinner.CanChange

  Это - метод обработки события OnChanging. Обратите внимание, это
  - функция, а не процедура. В функции переменной Result значение
  присваивается до того, как вызывается пользовательский
  обработчик события.
=====================================================================}

function TddgSpinner.CanChange( NewValue: Integer ): Boolean;
var
  AllowChange: Boolean;
begin
  AllowChange := True;
  if Assigned( FOnChanging ) then
    FOnChanging( Self, NewValue, AllowChange );
  Result := AllowChange;
end;

procedure TddgSpinner.Change;
begin
  if Assigned( FOnChange ) then
    FOnChange( Self );
end;

// Обратите внимание, методы DecValue и IncValue присваивают
// новое значение свойству Value (а не FValue), что косвенно
// приводит к вызову метода SetValue

procedure TddgSpinner.DecValue( Amount: Integer );
begin
  Value := Value - Amount;
end;
```

```
procedure TddgSpinner.IncValue( Amount: Integer );
begin
  Value := Value + Amount;
end;


{== Методы обработки событий клавиатуры ==}

(*
// VCL->CLX: Заменен в конструкторе на использование InputKeys

procedure TddgSpinner.WMGetDlgCode( var Msg: TWMGetDlgCode );
begin
  inherited;
  Msg.Result := dlgc_WantArrows; // обрабатывает клавиши курсора
end;
*)


procedure TddgSpinner.KeyDown( var Key: Word; Shift: TShiftState);
begin
  inherited KeyDown( Key, Shift );

  // VCL->CLX:  Константы клавиш в CLX изменены.
  //            Вместо префикса vk_ используется Key_

  case Key of
    Key_Left, Key_Down:
      DecValue( FIncrement );

    Key_Up, Key_Right:
      IncValue( FIncrement );
  end;
end;


{== Методы обработки событий мыши ==}

function TddgSpinner.CursorPosition: TPoint;
begin
  GetCursorPos( Result );
  Result := ScreenToClient( Result );
end;

function TddgSpinner.MouseOverButton(Btn:
                                     TddgButtonType): Boolean;
var
  R: TRect;
begin
  // Получить границы соответствующей кнопки
  if Btn = btMinus then
    R := Rect( 0, 0, FButtonWidth, Height )
```

```
  else
    R := Rect( Width - FButtonWidth, 0, Width, Height );

  // Находится ли курсор над кнопкой?
  Result := PtInRect( R, CursorPosition );
end;

procedure TddgSpinner.MouseDown( Button: TMouseButton;
                                 Shift: TShiftState; X, Y: Integer);
begin
  inherited MouseDown( Button, Shift, X, Y );

  if not ( csDesigning in ComponentState ) then
    SetFocus;       // Фокус на Spinner переносится только во
                    // время выполнения программы

  if ( Button = mbLeft ) and
     ( MouseOverButton(btMinus) or
       MouseOverButton(btPlus) ) then begin
    FMinusBtnDown := MouseOverButton( btMinus );
    FPlusBtnDown := MouseOverButton( btPlus );

    Repaint;
  end;
end;

procedure TddgSpinner.MouseUp( Button: TMouseButton;
                               Shift: TShiftState; X, Y: Integer );
begin
  inherited MouseUp( Button, Shift, X, Y );

  if Button = mbLeft then begin
    if MouseOverButton( btPlus ) then
      IncValue( FIncrement )
    else if MouseOverButton( btMinus ) then
      DecValue( FIncrement );

    FMinusBtnDown := False;
    FPlusBtnDown := False;

    Repaint;
  end;
end;

function TddgSpinner.DoMouseWheelDown( Shift: TShiftState;
                                 const MousePos: TPoint ): Boolean;
begin
  inherited DoMouseWheelDown( Shift, MousePos );
  DecValue( FIncrement );
  Result := True;
end;

function TddgSpinner.DoMouseWheelUp( Shift: TShiftState;
```

```
                                    const MousePos: TPoint ): Boolean;
begin
  inherited DoMouseWheelUp( Shift, MousePos );
  IncValue( FIncrement );
  Result := True;
end;

end.
```

Как видите, исходный код для CLX очень похож на исходный код для VCL, но, тем не менее, в нем есть несколько существенных отличий.

Во-первых, подключаются модули, связанные с Qt: `Qt`, `QControls` и `QGraphics`. Также используется новый модуль `Types`, который применяется и в VCL, и в CLX. К счастью, основная часть объявлений класса CLX `TddgSpinner` идентична объявлениям VCL. В частности, это можно сказать об объявлении полей экземпляра, а также о методах обработки событий.

В первую очередь, при переносе компонента в CLX, необходимо внести изменения в код методов обработки сообщений `CMEnabledChanged()` и `WMGetDlgCode()`. Сообщения `cm_EnabledChanged` и `wm_GetDlgCode` в CLX не используются, а значит, их функции должны быть реализованы по-другому.

Как уже упоминалось, в CLX сообщения компонентов, наподобие `cm_EnabledChanged`, заменены соответствующими динамическими методами. Поэтому класс CLX `TControl` при изменении свойства `Enabled` вместо отправки сообщения `cm_EnabledChanged` просто вызывает метод `EnabledChanged()`. Таким образом программный код из метода `CMEnabledChanged()` просто переносится в переопределенный метод `EnabledChanged()`.

Обычной задачей при разработке компонентов является обработка событий клавиш управления курсором. В компоненте `TddgSpinner` эти клавиши используются для увеличения и уменьшения значения. В компоненте VCL данная функция была реализована через обработку сообщения `wm_GetDlgCode`. Как уже упоминалось, сообщение `wm_GetDlgCode` в компонентах CLX не используется, следовательно, необходим какой-нибудь другой подход. Положительным моментом является то, что в классе `TWidgetControl` определено свойство `InputKeys`, где указывается набор клавиш для обработки в конструкторе компонента.

В конструкторе содержится еще одно изменение, связанное с преобразованием из VCL в CLX: класс `TWidgetControl` инициализирует свойство `Color`, которому в классе `TControl` присваивается значение `clNone`. В VCL класс `TWinControl` просто использует унаследованное значение `clWindow`, поэтому свойству `Color` нужно присвоить в конструкторе значение `clWindow`.

Теперь, после того как был модифицирован конструктор, в исходный код осталось внести совсем немного изменений. Большинство методов обработки событий используются также и в CLX. Учитывая это, переход к CLX намного упрощается, если для компонента вместо обработки сообщений Windows использовать переопределенные методы обработки событий VCL.

В начале настоящей главы было сказано, что все методы построения компонентов VCL можно применять и при разработке компонентов CLX. И действительно, объявления свойств, методы доступа и обработчики событий в VCL и CLX аналогичны.

Больше всего изменений при переносе компонента из VCL в CLX необходимо внести в метод `Paint()`.

При переносе компонентов методы отображения, наподобие `Paint()`, обычно подвергаются самым значительным модификациям, даже несмотря на то, что классы `TCanvas` в обеих архитектурах имеют почти идентичные интерфейсы.

При переносе компонента класса `TddgSpinner` в его исходный код следует внести два изменения, связанных с отображением. Во-первых, в версии VCL используется функция GDI `SetTextAlign`, которая автоматически центрирует текстовое поле компонента. Эта функция API Windows в Linux не используется. Кроме того, она не будет работать даже в среде Windows, так как ориентируется на дескриптор графического контекста устройства вывода. Компоненты CLX не имеют доступа к контексту устройства, потому что для них свойство `Canvas.Handle` указывает на объект Qt `Painter`.

К счастью, большинство методов класса `TCanvas` используются и в Windows, и в Linux, поэтому проблема выравнивания текста может быть разрешена достаточно просто обычным вычислением необходимой позиции.

Вторая проблема, связанная с отображением, заключается в использовании метода `DrawButton()`. В частности, символы "+" и "-" на кнопках в VCL прорисовываются с использованием цвета `clActiveCaption`, а идентификатору `clActiveCaption` в модуле `QGraphics.pas` соответствует значение `clActiveHighlightedText`.

НА ЗАМЕТКУ

Для выполнения прорисовки вне метода `Paint()` компонентов CLX необходимо сначала вызвать метод `Canvas.Start()`. После завершения прорисовки вызывается метод `Canvas.Stop()`.

Не все модификации, связанные с переносом компонента, настолько просты, как кажется на первый взгляд. В частности, вместо определенных в VCL констант кодов виртуальных клавиш, наподобие `vk_Left`, в CLX используется совершенно другой набор констант. Это объясняется тем, что коды виртуальных клавиш являются составной частью интерфейса API Windows и, следовательно, не используются в Linux.

Вот и все! Теперь создан полнофункциональный компонент CLX, который может быть использован как в приложениях, разрабатываемых в Delphi 6 для среды Windows, так и в приложениях, разрабатываемых в Kylix для среды Linux. При этом, без сомнения, очень важно то, что на обеих платформах используется один и тот же исходный код.

Дополнения времени разработки

Все действия по переносу компонента `TddgSpinner` из VCL в CLX были достаточно очевидны и просты. Незначительные сложности возникали лишь при использовании свойства `InputKeys`.

Тем не менее, при добавлении компонентам CLX новых функций, различия между VCL и CLX становятся очевидными.

Рассмотрим исходный код компонента, представленный в листинге 13.2. В этом модуле реализован класс `TddgDesignSpinner`, который является потомком класса `TddgSpinner`. Дополнительная функция заключается в изменении формы курсора мыши, когда он оказывается над одной из кнопок (рис. 13.4). Кроме того, компонент

производного класса будет обладать возможностью изменять числовое значение поля при щелчке на кнопке "Плюс" или "Минус" прямо во время разработки (рис. 13.5).

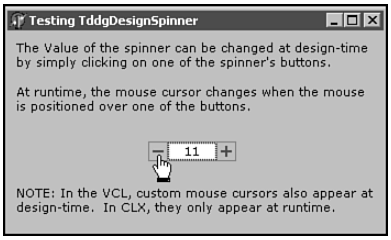

Рис. 13.4. *Компонент `TddgDesignSpinner` изменяет форму курсора мыши, когда он оказывается над одной из кнопок*

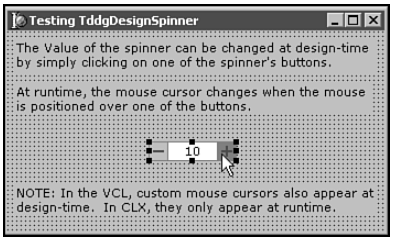

Рис. 13.5. *Компонент `TddgDesignSpinner` позволяет изменять значение свойства `Value` при щелчке на кнопках компонента прямо во время разработки*

Листинг 13.2. `QddgDsnSpin.pas` — исходный код компонента `TddgDesignSpinner`

```
unit QddgDsnSpn;

interface

uses
  SysUtils, Classes, Qt, QddgSpin;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ddgSpin;
  *)

type
  TddgDesignSpinner = class( TddgSpinner )
  private
    // VCL->CLX:  Пользовательский курсор хранится в поле QCursorH
    FThumbCursor: QCursorH;

    (*
    // VCL->CLX: Пользовательские курсоры и события времени
    //           разработки в CLX обрабатываются отдельно.
    //           Следующий фрагмент имеет отношение к VCL.

    FThumbCursor: HCursor;

    // Метод обработки сообщений окна
    procedure WMSetCursor( var Msg : TWMSetCursor );
      message wm_SetCursor;

    // Метод обработки сообщений компонента
    procedure CMDesignHitTest( var Msg: TCMDesignHitTest );
```

```
      message cm_DesignHitTest;
    *)
  protected
    procedure Change; override;

    // VCL->CLX: Следующие два метода переопределены для CLX
    procedure MouseMove( Shift: TShiftState;
                         X, Y: Integer ); override;
    function DesignEventQuery( Sender: QObjectH;
                               Event: QEventH ): Boolean; override;
  public
    constructor Create( AOwner: TComponent ); override;
    destructor Destroy; override;
  end;

implementation

(*
// VCL->CLX:  В CLX ресурсы курсоров не используются
{$R DdgDsnSpn.res}        // Подключение ресурса
                          // пользовательского курсора
*)

uses
  Types, QControls, QForms;  // VCL->CLX: Добавление модулей CLX

// VCL->CLX:  Для представления пользовательских курсоров в CLX
//            используются два массива байтов.
//            Один - для изображения, второй - для маски.

const
  Bits: array[0..32*4-1] of Byte = (
    $00, $30, $00, $00, $00, $48, $00, $00,
    $00, $48, $00, $00, $00, $48, $00, $00,
    $00, $48, $00, $00, $00, $4E, $00, $00,
    $00, $49, $C0, $00, $00, $49, $30, $00,
    $00, $49, $28, $00, $03, $49, $24, $00,
    $04, $C0, $24, $00, $04, $40, $04, $00,
    $02, $40, $04, $00, $02, $00, $04, $00,
    $01, $00, $04, $00, $01, $00, $04, $00,
    $00, $80, $08, $00, $00, $40, $08, $00,
    $00, $40, $08, $00, $00, $20, $10, $00,
    $00, $20, $10, $00, $00, $7F, $F8, $00,
    $00, $7F, $F8, $00, $00, $7F, $E8, $00,
    $00, $7F, $F8, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00 );

  Mask: array[0..32*4-1] of Byte = (
    $00, $30, $00, $00, $00, $78, $00, $00,
    $00, $78, $00, $00, $00, $78, $00, $00,
    $00, $78, $00, $00, $00, $7E, $00, $00,
```

```
    $00, $7F, $C0, $00, $00, $7F, $F0, $00,
    $00, $7F, $F8, $00, $03, $7F, $FC, $00,
    $07, $FF, $FC, $00, $07, $FF, $FC, $00,
    $03, $FF, $FC, $00, $03, $FF, $FC, $00,
    $01, $FF, $FC, $00, $01, $FF, $FC, $00,
    $00, $FF, $F8, $00, $00, $7F, $F8, $00,
    $00, $7F, $F8, $00, $00, $3F, $F0, $00,
    $00, $3F, $F0, $00, $00, $7F, $F8, $00,
    $00, $7F, $F8, $00, $00, $7F, $E8, $00,
    $00, $7F, $F8, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00 );


{====================================}
{== Методы класса TddgDesignSpinner ==}
{====================================}

constructor TddgDesignSpinner.Create( AOwner: TComponent );
var
  BitsBitmap: QBitmapH;
  MaskBitmap: QBitmapH;
begin
  inherited Create( AOwner );

  (*
  // VCL->CLX:  В CLX метод LoadCursor не используется
  FThumbCursor := LoadCursor( HInstance, 'DdgDSNSPN_BTNCURSOR' );
  *)

  // VCL->CLX:  Для создания пользовательского курсора
  //            используется два массива байтов
  BitsBitmap := QBitmap_create( 32, 32, @Bits, False );
  MaskBitmap := QBitmap_create( 32, 32, @Mask, False );
  try
    FThumbCursor := QCursor_create( BitsBitmap,
                                    MaskBitmap, 8, 0 );
  finally
    QBitmap_destroy( BitsBitmap );
    QBitmap_destroy( MaskBitmap );
  end;
end;

destructor TddgDesignSpinner.Destroy;
begin
(*
VCL->CLX: В CLX вместо DestroyCursor используется QCursor_Destroy
DestroyCursor( FThumbCursor );       // Удаление объекта курсора
*)
  QCursor_Destroy( FThumbCursor );
  inherited Destroy;
end;
```

```
// Если мышь расположена над одной из кнопок, то курсор
// принимает форму, заданную в файле ресурса DdgDsnSpn.res

(*
// VCL->CLX:  В CLX сообщение wm_SetCursor не используется
procedure TddgDesignSpinner.WMSetCursor( var Msg: TWMSetCursor );
begin
  if MouseOverButton( btMinus ) or MouseOverButton( btPlus ) then
    SetCursor( FThumbCursor )
  else
    inherited;
end;
*)

// VCL->CLX: Для отображения пользовательского курсора
//           необходимо переопределить метод MouseMove

procedure TddgDesignSpinner.MouseMove( Shift: TShiftState;
                                       X, Y: Integer );
begin
  if MouseOverButton( btMinus ) or MouseOverButton( btPlus ) then
    QWidget_setCursor( Handle, FThumbCursor )
  else
    QWidget_UnsetCursor( Handle );
  inherited;
end;

(*
// VCL->CLX:  Вместо сообщения cm_DesignHitTest в CLX используется
//            переопределенный метод DesignEventQuery.

procedure TddgDesignSpinner.CMDesignHitTest( var Msg:
                                             TCMDesignHitTest );
begin
  // Обработка этого сообщения компонента позволяет изменять
  // значение свойства Value при помощи щелчка на кнопках
  // прямо во время разработки. Если мышь расположена над одной
  // из кнопок, то Msg.Result принимает значение 1. Это указывает
  // Delphi на необходимость "передать" событие мыши компоненту.

  if MouseOverButton( btMinus ) or MouseOverButton( btPlus ) then
    Msg.Result := 1
  else
    Msg.Result := 0;
end;
*)

function TddgDesignSpinner.DesignEventQuery( Sender: QObjectH;
                                             Event: QEventH ): Boolean;
var
  MousePos: TPoint;
begin
```

```
  Result := False;
  if ( Sender = Handle ) and
     ( QEvent_type(Event) in [QEventType_MouseButtonPress,
                              QEventType_MouseButtonRelease,
                              QEventType_MouseButtonDblClick]) then
  begin
    // Примечание: объект MousePos используется для определения
    //             координат курсора мыши (в данном примере
    //             значения не имеет).

    MousePos := Point( QMouseEvent_x( QMouseEventH( Event ) ),
                       QMouseEvent_y( QMouseEventH( Event ) ) );

    if MouseOverButton( btMinus ) or
       MouseOverButton( btPlus ) then
      Result := True
    else
      Result := False;
  end;
end;

procedure TddgDesignSpinner.Change;
var
  Form: TCustomForm;
begin
  inherited Change;

  // Изменения свойства Value, сделанные при помощи мыши,
  // должны отображаться в инспекторе объектов.

  if csDesigning in ComponentState then begin
    Form := GetParentForm( Self );

    (*
    // VCL->CLX: Вместо Form.Designer в CLX
    //           используется DesignerHook
    if ( Form <> nil ) and ( Form.Designer <> nil ) then
      Form.Designer.Modified;
    *)

    if ( Form <> nil ) and ( Form.DesignerHook <> nil ) then
      Form.DesignerHook.Modified;
  end;
end;

end.
```

Судя по комментариям, включенным в исходный код, реализация двух новых возможностей компонента CLX — это не такое уж простое дело, так как в VCL для них использовались сообщения Windows. Как уже упоминалось ранее, Linux не поддерживает сообщения, поэтому в компоненте CLX для реализации новых возможностей должны использоваться другие механизмы.

Прежде всего следует отметить, что в CLX назначить элементу управления курсор намного сложнее, чем в VCL. В среде Windows для этого используется ресурсный файл с курсором, причем для получения дескриптора курсора вызывается функция API `LoadCursor`. Затем этот дескриптор используется при обработке сообщения `wm_SetCursor`, отправляемого системой Windows каждому элементу управления, который нуждается в обновлении указателя мыши.

В CLX используется другой подход. Во-первых, библиотека Qt не поддерживает ресурсы курсоров. В модуле `Qt.pas` определены несколько методов `QCursor_create()`, каждый из которых использует собственный способ создания курсора мыши. Для того, чтобы применить один из стандартных курсоров Qt, необходимо передать соответствующее целочисленное значение в метод `QCursor_create()`. В то же время, для создания пользовательского курсора используются два массива байтов, содержащих битовую схему изображения. В первом массиве хранятся биты черного и белого цветов, а во втором — маска, определяющая прозрачные области в изображении курсора.

Для того чтобы отобразить курсор мыши в соответствующий момент времени, вместо обработки сообщения `wm_SetCursor` используется переопределенный метод обработки события `MouseMove()`. В тот момент, когда курсор мыши находится над одной из кнопок, вызывается функция `QWidget_setCursor()`, изменяющая форму курсора. В противном случае при помощи метода `QWidget_UnsetCursor()` восстанавливается исходная форма курсора.

В VCL обработка сообщений мыши во время разработки реализована через обработку сообщения компонента `cm_DesignHitTest`. К сожалению, это сообщение отсутствует в CLX, поэтому вместо него используется переопределенный метод `DesignEventQuery()`. Этот метод позволяет разработчикам компонентов обращаться к обработчикам событий компонентов Qt во время разработки. Если он возвращает значение `True`, то элемент управления реагирует на событие. В данном примере использовались только входные события мыши, поэтому в первую очередь нужно выяснить, соответствуют ли эти входные события нашим критериям. Если соответствуют, то следует определить, расположен ли курсор мыши над одной из кнопок.

Метод `Change()` в классе `TddgDesignSpinner` должен быть переопределен, потому что изменения свойства `Value` выбранного компонента должны синхронно отображаться в инспекторе объектов. Если метод `Change()` не будет переопределен, то инспектор объектов не сможет отобразить изменения значений, сделанных при помощи щелчка мыши на кнопках непосредственно в конструкторе форм. Единственной коррекцией, внесенной в этот метод, является замена ссылки `Form.Designer` на `Form.DesignerHook`.

Ссылки на компоненты и список ImageList

Следующий компонент, `TddgImgListSpinner`, расширяет функции компонента `TddgDesignSpinner`. В нем реализовано свойство, значением которого является ссылка на компонент `ImageList`, что позволяет использовать вместо символов “+” и “-” на кнопках инкремента и декремента изображения из списка `ImageList` (рис. 13.6).

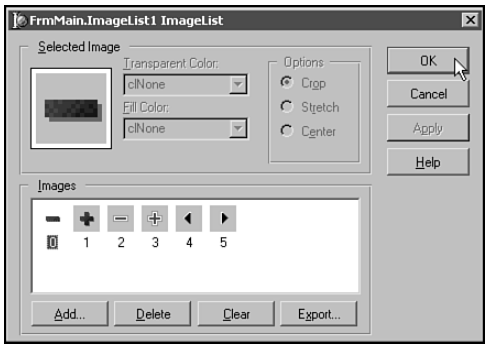

Рис. 13.6. *Компонент `TddgImgListSpinner` может использоваться для кнопок изображения из списка `ImageList`*

В листинге 13.3 представлен исходный код модуля `QddgILSpin.pas`, в котором реализован компонент `TddgImgListSpinner`. В отличие от компонента `TddgDesignSpinner`, при его переносе в CLX, требуется совсем немного изменений.

Листинг 13.3. `QddgILSpin.pas` — исходный код компонента `TddgImgListSpinner`

```
unit QddgILSpin;

interface

uses
  Classes, Types, QddgSpin, QddgDsnSpn, QImgList;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ddgSpin, ddgDsnSpn, ImgList;
  *)

type
  TddgImgListSpinner = class( TddgDesignSpinner )
  private
    FImages: TCustomImageList;
    FImageIndexes: array[ 1..2 ] of Integer;
    FImageChangeLink: TChangeLink;

    // Внутренние обработчики событий
    procedure ImageListChange( Sender: TObject );
  protected
    procedure Notification( AComponent : TComponent;
                            Operation : TOperation ); override;

    procedure DrawButton( Button: TddgButtonType; Down: Boolean;
                          Bounds: TRect ); override;
    procedure CalcCenterOffsets( Bounds: TRect;
                                 var L, T: Integer);

    procedure CheckMinSize;

    // Методы доступа к свойствам
```

```
    procedure SetImages( Value: TCustomImageList ); virtual;
    function GetImageIndex( PropIndex: Integer ): Integer;
                                                   virtual;
    procedure SetImageIndex( PropIndex: Integer;
                             Value: Integer ); virtual;
  public
    constructor Create( AOwner: TComponent ); override;
    destructor Destroy; override;
  published
    property Images: TCustomImageList
      read FImages
      write SetImages;

    property ImageIndexMinus: Integer
      index 1
      read GetImageIndex
      write SetImageIndex;

    property ImageIndexPlus: Integer
      index 2
      read GetImageIndex
      write SetImageIndex;
  end;

implementation

uses
  QGraphics;        // VCL->CLX:  Добавлено для поддержки CLX


{====================================}
{== Методы класса TddgImgListSpinner ==}
{====================================}

constructor TddgImgListSpinner.Create( AOwner: TComponent );
begin
  inherited Create( AOwner );

  FImageChangeLink := TChangeLink.Create;
  FImageChangeLink.OnChange := ImageListChange;
  // Примечание: пользователи компонента не имеют прямого доступа
  // к объекту типа TChangeLink, поэтому они не могут использовать
  // собственные обработчики событий.

  FImageIndexes[ 1 ] := -1;
  FImageIndexes[ 2 ] := -1;
end;

destructor TddgImgListSpinner.Destroy;
begin
  FImageChangeLink.Free;
  inherited Destroy;
end;
```

```
procedure TddgImgListSpinner.Notification( AComponent: TComponent;
                                           Operation: TOperation );
begin
  inherited Notification( AComponent, Operation );
  if ( Operation = opRemove ) and ( AComponent = FImages ) then
    SetImages( nil );           // Вызов метода доступа
end;


function TddgImgListSpinner.GetImageIndex( PropIndex:
                                           Integer ): Integer;
begin
  Result := FImageIndexes[ PropIndex ];
end;

procedure TddgImgListSpinner.SetImageIndex( PropIndex: Integer;
                                            Value: Integer );
begin
  if FImageIndexes[ PropIndex ] <> Value then begin
    FImageIndexes[ PropIndex ] := Value;
    Invalidate;
  end;
end;


procedure TddgImgListSpinner.SetImages( Value: TCustomImageList );
begin
  if FImages <> nil then
    FImages.UnRegisterChanges( FImageChangeLink );

  FImages := Value;

  if FImages <> nil then begin
    FImages.RegisterChanges( FImageChangeLink );
    FImages.FreeNotification( Self );
    CheckMinSize;
  end;
  Invalidate;
end;

procedure TddgImgListSpinner.ImageListChange( Sender: TObject );
begin
  if Sender = Images then begin
    CheckMinSize;
    // Для предотвращения мерцания экрана вместо
    // метода Invalidate вызывается метод Update
    Update;
  end;
end;

procedure TddgImgListSpinner.CheckMinSize;
begin
```

```
  // В области кнопки изображение должно отображаться полностью
  if FImages.Width > ButtonWidth then
    ButtonWidth := FImages.Width;
  if FImages.Height > Height then
    Height := FImages.Height;
end;


procedure TddgImgListSpinner.DrawButton( Button: TddgButtonType;
                                         Down: Boolean;
                                         Bounds: TRect );
var
  L, T: Integer;
begin
  with Canvas do begin
    Brush.Color := ButtonColor;
    Pen.Color := clBtnShadow;
    Rectangle( Bounds.Left, Bounds.Top,
               Bounds.Right, Bounds.Bottom );

    if Button = btMinus then         // Прорисовка кнопки "-"
    begin
      if ( Images <> nil ) and ( ImageIndexMinus <> -1 ) then
      begin
        (*
        // VCL->CLX: Свойство DrawingStyle в классе CLX TImageList
        //           отсутствует, а вместо него используется
        //           свойство BkColor.
        if Down then
          FImages.DrawingStyle := dsSelected
        else
          FImages.DrawingStyle := dsNormal;
        *)
        if Down then
          FImages.BkColor := clBtnShadow
        else
          FImages.BkColor := clBtnFace;

        CalcCenterOffsets( Bounds, L, T );

        (*
        // VCL->CLX:  TImageList.Draw в CLX имеет другой синтаксис
        FImages.Draw( Canvas, L, T, ImageIndexMinus, Enabled );
        *)
        FImages.Draw( Canvas, L, T, ImageIndexMinus, itImage,
                      Enabled );
      end
      else
        inherited DrawButton( Button, Down, Bounds );
    end
    else                             // Прорисовка кнопки "+"
    begin
      if ( Images <> nil ) and ( ImageIndexPlus <> -1 ) then
```

```
      begin
        (*
        // VCL->CLX: Свойство DrawingStyle в классе CLX TImageList
        //           отсутствует. Вместо него используется
        //           свойство BkColor.
        if Down then
          FImages.DrawingStyle := dsSelected
        else
          FImages.DrawingStyle := dsNormal;
        *)
        if Down then
          FImages.BkColor := clBtnShadow
        else
          FImages.BkColor := clBtnFace;

        CalcCenterOffsets( Bounds, L, T );

        (*
        // VCL->CLX:  TImageList.Draw в CLX имеет другой синтаксис
        FImages.Draw( Canvas, L, T, ImageIndexPlus, Enabled );
        *)
        FImages.Draw( Canvas, L, T, ImageIndexPlus, itImage,
                      Enabled );
      end
      else
        inherited DrawButton( Button, Down, Bounds );
    end;
  end;
end; {= TddgImgListSpinner.DrawButton =}


procedure TddgImgListSpinner.CalcCenterOffsets( Bounds: TRect;
                                                var L, T: Integer );
begin
  if FImages <> nil then begin
    L := Bounds.Left + ( Bounds.Right - Bounds.Left ) div 2 -
         ( FImages.Width div 2 );
    T := Bounds.Top + ( Bounds.Bottom - Bounds.Top ) div 2 -
         ( FImages.Height div 2 );
  end;
end;

end.
```

Как обычно, в раздел `uses` нужно включить модули, имеющие отношение к CLX, и удалить модули, относящиеся к VCL. В частности, ссылка на модуль `ImgList` заменяется ссылкой на модуль `QImgList`. Это важно, потому что в VCL класс `TImageList` является производным от класса `TCustomImageList`, реализованного в библиотеке `ComCtl32.dll`. Компания *Borland* создала версию класса `TCustomImageList` для CLX, которая вместо библиотеки `ComCtl32.dll` использует графические примитивы библиотеки Qt.

Преимущества такого подхода очевидны из объявления класса. Объявление класса `TddgImgListSpinner` версии CLX идентично объявлению версии VCL. Более того, идентичными являются реализации всех методов, кроме одного.

Некоторые изменения, конечно же, нужно внести в переопределенный метод прорисовки `DrawButton()`. При этом необходимо решить две задачи. Для решения первой из них необходимо провести сравнительную характеристику классов, существующих как в VCL, так и с CLX. Действительно, одному из классов VCL может соответствовать один из классов CLX, однако это не означает, что в классе CLX будут реализованы все функции класса VCL.

В случае с классом `TCustomImageList` в версии VCL реализовано свойство `DrawingStyle`, которое использовалось в версии VCL класса `TddgImgListSpinner` для прорисовки изображения на кнопке. Но свойство `DrawingStyle` отсутствует в классе `TCustomImageList` версии CLX, потому в реализации компонента должен использоваться другой подход.

Вторая модификация в методе `DrawButton()` связана с тем, что метод `TCustomImageList.Draw()` в архитектуре VCL работает не так, как в архитектуре CLX.

Компоненты CLX для работы с базами данных

В четвертом варианте компонента `Spinner` реализовано взаимодействие с базами данных. При помощи свойств `DataSource` и `DataField` компонент типа `TddgDBSpinner` может подключаться к любому целочисленному полю некоторого набора данных. На рис. 13.7 показан компонент типа `TddgDBSpinner`, подключенный к полю `VenueNo` набора данных `Events`.

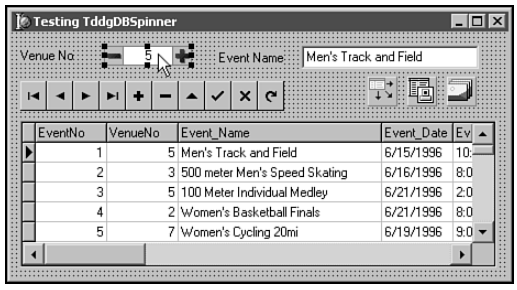

Рис. 13.7. *Компонент `TddgDBSpinner` используется для отображения и редактирования целочисленных полей наборов данных*

Класс `TddgDBSpinner` является производным от класса `TddgImgListSpinner`. Он реализован в модуле `QddgDBSpin.pas`, исходный код которого представлен в листинге 13.4.

Листинг 13.4. `QddgDBSpin.pas` — исходный код компонента `TddgDBSpinner`

```
unit QddgDBSpin;

interface

uses
```

```
  SysUtils, Classes, Qt, QddgILSpin, DB, QDBCtrls;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ddgILSpin, DB, DBCtrls;
  *)

type
  TddgDBSpinner = class( TddgImgListSpinner )
  private
    FDataLink: TFieldDataLink;     // Обеспечивает доступ к данным

    // Внутренние обработчики событий доступа к данным
    procedure DataChange( Sender: TObject );
    procedure UpdateData( Sender: TObject );
    procedure ActiveChange( Sender: TObject );

    (*
    // VCL->CLX:  Методы обработки сообщений компонента,
    // отсутствующие в CLX
    procedure CMExit( var Msg: TCMExit ); message cm_Exit;
    procedure CMDesignHitTest( var Msg: TCMDesignHitTest );
      message cm_DesignHitTest;
    *)
  protected
    procedure Notification( AComponent : TComponent;
                            Operation : TOperation ); override;
    procedure CheckFieldType( const Value: string ); virtual;

    Переопределенные методы обработки событий
    procedure Change; override;
    procedure KeyPress( var Key : Char ); override;

    // VCL->CLX:  Процедура DoExit заменена на CMExit
    procedure DoExit; override;
    // VCL->CLX:  Функция DesignEventQuery
    // заменена на CMDesignHitTest
    function DesignEventQuery( Sender: QObjectH;
                               Event: QEventH ): Boolean; override;

    // Переопределенные вспомогательные методы
    procedure DecValue( Amount: Integer ); override;
    procedure IncValue( Amount: Integer ); override;

    // Методы доступа к свойствам
    function GetField: TField; virtual;
    function GetDataField: string; virtual;
    procedure SetDataField( const Value: string ); virtual;
    function GetDataSource: TDataSource; virtual;
    procedure SetDataSource( Value: TDataSource ); virtual;
    function GetReadOnly: Boolean; virtual;
    procedure SetReadOnly( Value: Boolean ); virtual;

    // Доступ потомков к объектам Field и DataLink
```

```
    property Field: TField
      read GetField;

    property DataLink: TFieldDataLink
      read FDataLink;
  public
    constructor Create( AOwner: TComponent ); override;
    destructor Destroy; override;
  published
    property DataField: string
      read GetDataField
      write SetDataField;

    property DataSource: TDataSource
      read GetDataSource
      write SetDataSource;

    // Это свойство контролирует состояние (ReadOnly)
    // объекта DataLink
    property ReadOnly: Boolean
      read GetReadOnly
      write SetReadOnly
      default False;
  end;

type
  EInvalidFieldType = class( Exception );

resourcestring
  SInvalidFieldType = 'DataField can only be connected to ' +
                      'columns of type Integer, Smallint, Word, ' +
                      'and Float';

implementation

uses
  Types;                        // VCL->CLX:  Добавлено для CLX

{================================}
{== Методы класса TddgDBSpinner ==}
{================================}

constructor TddgDBSpinner.Create( AOwner: TComponent );
begin
  inherited Create( AOwner );

  FDataLink := TFieldDataLink.Create;

  // Для поддержки метода TField.FocusControl, свойство
  // FDataLink.Control должно указывать на Spinner.
  // Свойство Control необходимо для всех компонентов
  // класса TWinControl.
  FDataLink.Control := Self;
```

```
  // Назначение обработчиков событий
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
  FDataLink.OnActiveChange := ActiveChange;

  // Примечание: пользователь компонента не имеет прямого доступа
  // к объекту DataLink, поэтому не может назначать собственные
  // обработчики событий.
end;

destructor TddgDBSpinner.Destroy;
begin
  FDataLink.Free;
  FDataLink := nil;
  inherited Destroy;
end;

procedure TddgDBSpinner.Notification( AComponent: TComponent;
                                      Operation: TOperation );
begin
  inherited Notification( AComponent, Operation );
  if ( Operation = opRemove ) and
     ( FDataLink <> nil ) and
     ( AComponent = FDataLink.DataSource ) then begin
    DataSource := nil;          // Косвенный вызов SetDataSource
  end;
end;

function TddgDBSpinner.GetField: TField;
begin
  Result := FDataLink.Field;
end;

function TddgDBSpinner.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TddgDBSpinner.SetDataField( const Value: string );
begin
  CheckFieldType( Value );
  FDataLink.FieldName := Value;
end;

function TddgDBSpinner.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TddgDBSpinner.SetDataSource( Value: TDataSource );
begin
  if FDatalink.DataSource <> Value then begin
```

```
    FDataLink.DataSource := Value;

    // Необходимо вызвать метод FreeNotification, так как источник
    // данных может находиться в другой форме или в другом модуле.
    if Value <> nil then
      Value.FreeNotification( Self );
  end;
end;

function TddgDBSpinner.GetReadOnly: Boolean;
begin
  Result := FDataLink.ReadOnly;
end;

procedure TddgDBSpinner.SetReadOnly( Value: Boolean );
begin
  FDataLink.ReadOnly := Value;
end;

procedure TddgDBSpinner.CheckFieldType( const Value: string );
var
  FieldType: TFieldType;
begin
  // Поле указанного в Value столбца должно иметь тип
  // ftInteger, ftSmallInt, ftWord или ftFloat.
  // Если это не так, то передается исключение EInvalidFieldType.

  if ( Value <> '' ) and
     ( FDataLink <> nil ) and
     ( FDataLink.Dataset <> nil ) and
     ( FDataLink.Dataset.Active ) then begin
    FieldType := FDataLink.Dataset.FieldByName( Value ).DataType;
    if ( FieldType <> ftInteger ) and
       ( FieldType <> ftSmallInt ) and
       ( FieldType <> ftWord ) and
       ( FieldType <> ftFloat ) then begin
      raise EInvalidFieldType.Create( SInvalidFieldType );
    end;
  end;
end;

procedure TddgDBSpinner.Change;
begin
  // Для FDataLink устанавливается признак модификации
  if FDataLink <> nil then
    FDataLink.Modified;
  inherited Change;        // Передается событие OnChange
end;

procedure TddgDBSpinner.KeyPress( var Key: Char );
begin
  inherited KeyPress( Key );
```

```
  if Key = #27 then begin
    FDataLink.Reset;  // Нажата клавиша Esc
    Key := #0;        // Значение #0 используется для того, чтобы
  end;                // при нажатии Esc окно не закрылось.
end;

procedure TddgDBSpinner.DecValue( Amount: Integer );
begin
  if ReadOnly or not FDataLink.CanModify then begin
    // Запрет изменений, если FDataLink в режиме ReadOnly
    (*
    // VCL->CLX:  MessageBeep - это функция API Windows
    MessageBeep( 0 )
    *)
    Beep;
  end else begin
    // Попытка открыть набор данных для редактирования.
    // Уменьшение значения - только в режиме редактирования
    if FDataLink.Edit then
      inherited DecValue( Amount );
  end;
end;

procedure TddgDBSpinner.IncValue( Amount: Integer );
begin
  if ReadOnly or not FDataLink.CanModify then begin
    // Запрет изменений, если FDataLink в режиме ReadOnly
    (*
    // VCL->CLX:  MessageBeep - это функция API Windows
    MessageBeep( 0 )
    *)
    Beep;
  end else begin
    // Попытка открыть набор данных для редактирования.
    // Увеличение значения - только в режиме редактирования
    if FDataLink.Edit then
      inherited IncValue( Amount );
  end;
end;

{=================================================================
  TddgDBSpinner.DataChange

  Этот метод вызывается вследствие различных событий:

  1. Изменено значения соответствующего поля. Это событие
происходит после изменения значения в столбце, связанном с текущим
элементом управления, и последующего перехода на другой столбец или
запись.
  2. Переход соответствующего набора данных в режим редактирования.
  3. Изменение в наборе данных, указанном в свойстве DataSource.
  4. Переход курсора к новой записи таблицы.
  5. Восстановление записи при вызова метода Cancel.
```

```
  6. Изменено свойство DataField при переходе на другой столбец.
=============================================================}

procedure TddgDBSpinner.DataChange( Sender: TObject );
begin
  if FDataLink.Field <> nil then
    Value := FDataLink.Field.AsInteger;
end;

{=============================================================
  TddgDBSpinner.UpdateData

Этот метод вызывается в том случае, если необходима синхронизация
содержимого поля и значения в компоненте Spinner. Учтите, что
данный метод вызывается только при изменении данных.
=============================================================}

procedure TddgDBSpinner.UpdateData( Sender: TObject );
begin
  FDataLink.Field.AsInteger := Value;
end;

{=============================================================
  TddgDBSpinner.ActiveChange

  Этот метод вызывается при изменении свойства Active подключенного
набора данных.

  Примечание: для "обновления" состояния набора данных
              можно использовать свойство FDataLink.Active.
=============================================================}

procedure TddgDBSpinner.ActiveChange( Sender: TObject );
begin
  // После активизации набора данных в нем проверяется
  // корректность типа поля, указанного в свойстве DataField.

  if ( FDataLink <> nil ) and FDataLink.Active then
    CheckFieldType( DataField );
end;

(*
// VCL->CLX:  Вместо процедуры CMExit используется DoExit

procedure TddgDBSpinner.CMExit( var Msg: TCMExit );
begin
  try  // Попытка обновления записи при потере фокуса
       // компонентом Spinner
    FDataLink.UpdateRecord;
  except
    SetFocus;     // В случае неудачного обновления фокус остается
                  // в элементе управления
    raise;        // Повторная передача исключения
```

```
  end;
  inherited;
end;
*)

procedure TddgDBSpinner.DoExit;
begin
  try  // Попытка обновления записи при потере фокуса
       // компонентом Spinner
    FDataLink.UpdateRecord;
  except
    SetFocus;     // В случае неудачного обновления фокус остается
                  // в элементе управления
    raise;        // Повторная передача исключения
  end;
  inherited;
end;

(*
// VCL->CLX:  Процедура CMDesignHitTest заменена
//            на DesignEventQuery

procedure TddgDBSpinner.CMDesignHitTest(var Msg: TCMDesignHitTest);
begin
  // В компоненте базового класса значение Value можно изменять во
  // время разработки. Для компонентов, взаимодействующих с базами
  // данных, это недопустимо, так как подключенный набор данных
  // переходит в режим редактирования.
  Msg.Result := 0;
end;
*)

function TddgDBSpinner.DesignEventQuery( Sender: QObjectH;
                                         Event: QEventH ): Boolean;
begin
  // В компоненте базового класса значение Value можно изменять во
  // время разработки. Для компонентов, взаимодействующих с базами
  // данных это не допустимо, так как подключенный набор данных
  // переходит в режим редактирования.
  Result := False;
end;

end.
```

К счастью, структура компонентов, взаимодействующих с базами данных, в CLX практически такая же, как и в VCL. Для подключения к данным в компонент CLX просто внедряется объект типа `TFieldDataLink` и реализуется обработка событий `DataChange` и `UpdateData`. Само собой разумеется, что таким же образом реализуются и свойства `DataSource`, `DataField` и `ReadOnly`, но их реализация точно такая же, как в компонентах VCL.

НА ЗАМЕТКУ

Не забудьте указать в разделе `uses` модуль `DBCtrls` вместо модуля `QDBCtrls`. В отличие от модуля `DB`, модуль `DBCtrls` не является общим для VCL и CLX. Класс `TFieldDataLink` определен как в модуле `DBCtrls`, так и в модуле `QDBCtrls`, но в Delphi 6 у версии VCL этого класса не реализована обработка ошибок. Фактически такой компонент может корректно работать даже в среде Windows, однако при его переносе в Kylix компилятор выдаст множество сообщений о синтаксических ошибках.

Существует одна ситуация, о которой следует сказать особо. Многие компоненты VCL, взаимодействующие с базами данных, для того чтобы вызвать метод `UpdateRecord`, обрабатывают сообщение компонента `cm_Exit`.

Однако сообщение `cm_Exit` в CLX не реализовано, потому вместо метода `UpdateRecord` необходимо использовать метод обработки события `DoExit()`.

Класс `TddgDBSpinner` — это прямой наследник класса `TddgImgListSpinner`, который, в свою очередь, является прямым наследником класса `TddgDesignSpinner`. Одна из особенностей класса `TddgDesignSpinner` — это возможность изменения значения компонента `Spinner` при помощи мыши прямо во время разработки. Для компонента, взаимодействующего с базой данных, такая возможность должна быть отключена, так как приводит к переходу подключенного набора данных в режим редактирования. К сожалению, во время разработки приложения вывести набор данных из режима редактирования невозможно, поэтому в классе `TddgDBSpinner` переопределяется метод `DesignEventQuery()`. В результате этот метод просто возвращает значение `False`, чтобы компонент не обрабатывал события мыши во время разработки.

Редакторы компонентов CLX

Структура редакторов компонентов CLX точно такая же, как и у VCL, но кое в чем они отличаются. Прежде всего тем, что модули, в которых реализованы функции времени разработки, разбиты на несколько новых модулей. В частности, модуль `DsgnIntf` был переименован в `DesignIntf`. Кроме того, в большинстве случаев в раздел `uses` нужно добавить новый модуль `DesignEditors`. В модуле `DesignIntf` определены интерфейсы, которые используются конструктором форм и инспектором объектов. В модуле `DesignEditors` реализованы классы редактора основных свойств и редактора компонентов.

К сожалению, не все возможности времени разработки, использовавшиеся в VCL, применяются в CLX. Например, редактор свойства `OwnerDraw` доступен только в VCL. Редакторы, характерные для CLX, реализованы в модуле `CLXEditors`, а редакторы, характерные для VCL, — в модуле `VCLEditors`.

На рис. 13.8 представлен редактор `TddgRadioGroupEditor` компонента CLX `TRadioGroup`, позволяющий пользователю быстро присвоить значение свойству `ItemIndex`. Класс `TddgRadioGroupEditor` определен в модуле `QddgRgpEdt.pas`, исходный код которого представлен в листинге 13.5.

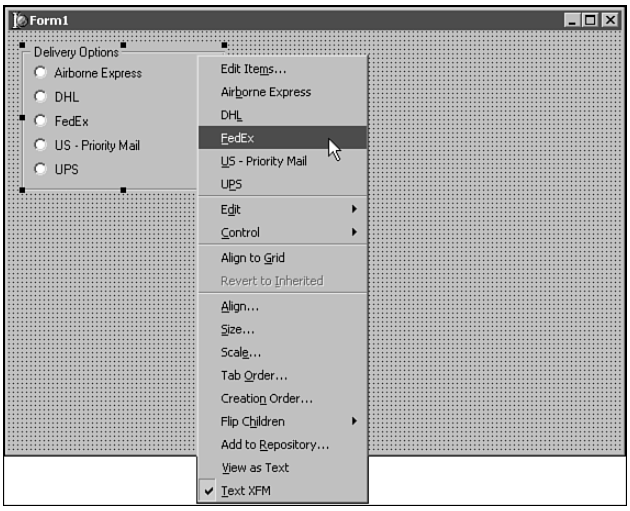

Рис. 13.8. *Выбор элемента для компонента CLX* `RadioGroup` *в специальном редакторе*

Листинг 13.5. `QddgRgpEdt.pas` **— исходный код редактора** `TddgRadioGroupEditor`

```
unit QddgRgpEdt;

interface

uses
  DesignIntf, DesignEditors, QExtCtrls, QDdgDsnEdt;

type
  TddgRadioGroupEditor = class( TddgDefaultEditor )
  protected
    function RadioGroup: TRadioGroup; virtual;
  public
    function GetVerbCount: Integer; override;
    function GetVerb( Index: Integer ) : string; override;
    procedure ExecuteVerb( Index: Integer ); override;
  end;

implementation

uses
  QControls;

{=======================================}
{== Методы класса TddgRadioGroupEditor ==}
{=======================================}

function TddgRadioGroupEditor.RadioGroup: TRadioGroup;
```

```
begin
  // Вспомогательная функция, обеспечивающая быстрый доступ к
  // редактируемому компоненту. Компонент также проверяется на
  // соответствие классу TRadioGroup
  Result := Component as TRadioGroup;
end;

function TddgRadioGroupEditor.GetVerbCount: Integer;
begin
  // Возвращается количество новых элементов меню
  Result := RadioGroup.Items.Count + 1;
end;

function TddgRadioGroupEditor.GetVerb( Index: Integer ): string;
begin
  // Заголовки элементов контекстного меню
  if Index = 0 then
    Result := 'Edit Items...'
  else
    Result := RadioGroup.Items[ Index - 1 ];
end;

procedure TddgRadioGroupEditor.ExecuteVerb( Index: Integer );
begin
  if Index = 0 then
    EditPropertyByName( 'Items' )  // Определено в QDdgDsnEdt.pas
  else begin
    if RadioGroup.ItemIndex <> Index - 1 then
      RadioGroup.ItemIndex := Index - 1
    else
      RadioGroup.ItemIndex := -1;  // Сбросить все элементы
    Designer.Modified;
  end;
end;

end.
```

Методика, используемая в классе `TddgRadioGroupEditor`, применима как для VCL, так и для CLX. В данном примере набор пунктов контекстного меню компонента `TRadioGroup` зависит от элементов выбранных в группе. Выбор пункта меню, соответствующего одному из элементов группы, приводит к изменению значения свойства `ItemIndex`. Если в группе нет ни одного элемента, то меню содержит только пункт Edit Items.

При выборе пользователем пункта меню Edit Items вызывается редактор объекта типа `TStringList`, в котором указывается значение для свойства `Items` компонента `TRadioGroup`. Метод `EditPropertyByName()` не входит в состав CLX или VCL. Он определен в классе `TddgDefaultEditor`. Этот метод используется для вызова редактора любого указанного свойства компонента из редактора данного компонента. Класс `TddgDefaultEditor` реализован в модуле `QddgDsnEdt.pas`, исходный код которого представлен в листинге 13.6.

Листинг 13.6. `QddgDsnEdt.pas` — исходный код редактора `TddgDefaultEditor`

```
unit QddgDsnEdt;

interface

uses
  Classes, DesignIntf, DesignEditors;

type
  TddgDefaultEditor = class( TDefaultEditor )
  private
    FPropName: string;
    FContinue: Boolean;
    FPropEditor: IProperty;
    procedure EnumPropertyEditors(const
                                  PropertyEditor: IProperty);
    procedure TestPropertyEditor( const PropertyEditor: IProperty;
                                  var Continue: Boolean );
  protected
    procedure EditPropertyByName( const APropName: string );
  end;

implementation

uses
  SysUtils, TypInfo;

{====================================}
{== Методы класса TddgDefaultEditor ==}
{====================================}

procedure TddgDefaultEditor.EnumPropertyEditors( const
                                     PropertyEditor: IProperty );
begin
  if FContinue then
    TestPropertyEditor( PropertyEditor, FContinue );
end;

procedure TddgDefaultEditor.TestPropertyEditor( const
                                       PropertyEditor: IProperty;
                                       var Continue: Boolean );
begin
  if not Assigned( FPropEditor ) and
     ( CompareText( PropertyEditor.GetName, FPropName ) = 0 ) then
  begin
    Continue := False;
    FPropEditor := PropertyEditor;
  end;
end;

procedure TddgDefaultEditor.EditPropertyByName( const
```

```
                                                    APropName: string );
var
  Components: IDesignerSelections;
begin
  Components := TDesignerSelections.Create;
  FContinue := True;
  FPropName := APropName;
  Components.Add( Component );
  FPropEditor := nil;
  try
    GetComponentProperties( Components, tkAny, Designer,
                            EnumPropertyEditors );
    if Assigned( FPropEditor ) then
      FPropEditor.Edit;
  finally
    FPropEditor := nil;
  end;
end;

end.
```

Пакеты

Компоненты CLX (так же, как и компоненты VCL) для использования в интегрированной среде Kylix или Delphi должны быть помещены внутрь пакетов. При этом важно отметить, что откомпилированный пакет Delphi 6, содержащий компоненты CLX, не может быть установлен в Kylix. Это связано с тем, что пакеты Windows реализованы в виде специальных библиотек DLL, в то время как пакеты Linux реализованы в виде файлов распределенных объектов с расширением `.so`. К счастью, формат и синтаксис файлов с исходным кодом пакетов идентичны для обеих платформ.

Тем не менее, информация, включаемая в пакеты для среды Windows, отличается от аналогичной информации для среды Linux. Например, в разделе `requires` пакета времени выполнения для Linux обычно указываются пакеты `baseclx` и `visualclx`, а пакет `baseclx` в Delphi 6 не используется. В среде Windows в пакетах времени выполнения, содержащих компоненты CLX, указывается только пакет `visualclx`. Как и в случае с пакетами VCL, при разработке пакетов CLX потребуются пакеты времени выполнения, содержащие новые пользовательские компоненты CLX.

Соглашения об именовании

Представленные в данной главе компоненты CLX содержатся в пакетах, перечисленных в табл. 13.1 и 13.2. В первой из таблиц указаны файлы `.BPL`, созданные в среде Windows, а также пакеты, необходимые при разработке любого пользовательского пакета. В табл. 13.2 перечислены пакеты, необходимые для разработки новых пакетов, и файлы распределенных объектов, созданные в среде Linux. Исходный код пакетов идентичен для обеих таблиц. Обратите внимание на соглашения, которые используются для названий пакетов.

Таблица 13.1. Пакеты CLX для Windows (Delphi 6)

Исходный код пакета	*Откомпилированная версия*	*Раздел* `requires`
`QddgSamples.dpk`	`QddgSamples60.bpl`	`visualclx`

`QddgSamples_Dsgn.dpk`	`QddgSamples_Dsgn60.bpl`	`visualclx`
		`designide`
		`QddgSamples`

`QddgDBSamples.dpk`	`QddgDBSamples60.bpl`	`visualclx`
		`dbrtl`
		`visualdbclx`
		`QddgSamples`

`QddgDBSamples_Dsgn.dpk`	`QddgDBSamples_Dsgn60.bpl`	`visualclx`
		`QddgSamples_Dsgn`
		`QddgDBSamples`

Таблица 13.2. Пакеты CLX для Linux (Kylix)

Исходный код пакета	*Откомпилированная версия*	*Раздел* `requires`
`QddgSamples.dpk`	`bplQddg6Samples.so.6`	`baseclx` `visualclx`

`QddgSamples_Dsgn.dpk`	`bplQddgSamples_Dsgn.so.6`	`baseclx`
		`visualclx`
		`designide`
		`QddgSamples`

`QddgDBSamples.dpk`	`bplQddgDBSamples.so.6`	`baseclx`
		`visualclx`
		`visualdbclx`
		`dataclx`
		`QddgSamples`

`QddgDBSamples_Dsgn.dpk`	`bplQddgDBSamples_Dsgn.so.6`	`baseclx`
		`visualclx`
		`QddgSamples_Dsgn`
		`QddgDBSamples`

В названия пакетов CLX, которые используются в Windows, обычно входит версия продукта. Например, название `QddgSamples60.bpl` означает, что этот файл предназначен для Delphi 6 и, судя по расширению `.bpl`, является библиотекой пакетов *Borland* (Borland Package Library). В Linux была выбрана традиционная для этой операционной системы практика присвоения имен распределенным объектам. Например, вместо расширения, соответствующего типу файла, для идентификации пакета используется префикс `bpl`. *Borland* также иногда в начале названий пакетов времени разработки использует префикс `dcl`. Но авторы не одобряют подобную практику, потому что пакеты времени разработки лучше идентифицировать при помощи суффикса `_Dsgn`. В дополнение следует отметить следующее: для всех пакетов (как времени выполнения, так и времени разработки) в среде Windows используется расширение `.bpl`. В этом случае использование префикса `bpl` для всех пакетов в среде Linux обеспечивает необходимую степень соответствия. Для идентификации распределен-

ных объектов Linux обычно используется суффикс .so, после которого следует номер версии. Выбор префикса и суффикса при генерации откомпилированных пакетов контролируется при помощи соответствующих параметров.

Также обратите внимание, что в названиях исходных файлов отсутствует номер версии. В предыдущих версиях Delphi распространенной практикой было добавление к названию пакета суффикса для определения версии требуемого пакета VCL. Но, начиная с Kylix и Delphi 6, *Borland* добавила несколько новых параметров контроля за названиями компилируемых пакетов. В пакетах Delphi 6, перечисленных в табл. 13.1, для идентификации версии 60 используется новый параметр {$LIBSUFFIX}. При компиляции пакета Delphi автоматически добавляет к названию файла соответствующий суффикс. В пакетах Kylix, перечисленных в табл. 13.2, для добавления префикса bpl используется директива {$SOPREFIX}, а для добавления суффикса 6 — директива {$SOVERSION}.

Пакеты времени выполнения

В листингах 13.7 и 13.8 представлен исходный код двух пакетов времени выполнения. Первый из них содержит компоненты, которые не взаимодействуют, а второй — компоненты, которые взаимодействуют с базами данных. Обратите внимание на использование условных директив со значениями MSWINDOWS и LINUX, а также содержимое раздела requires. Значение MSWINDOWS используется при компиляции в Delphi 6, а LINUX — при компиляции в Kylix.

Листинг 13.7. QddgSamples.dpk — исходный код пакета CLX времени выполнения, содержащего компоненты, которые не взаимодействуют с базами данных

```
package QddgSamples;

{$R *.res}
{$ALIGN 8}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
```

```
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components'}

{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}

{$IFDEF LINUX}
{$SOPREFIX 'bpl'}
{$SOVERSION '6'}
{$ENDIF}

{$RUNONLY}
{$IMPLICITBUILD OFF}

requires
  {$IFDEF LINUX}
  baseclx,
  {$ENDIF}
  visualclx;

contains
  QddgSpin in 'QddgSpin.pas',
  QddgDsnSpn in 'QddgDsnSpn.pas',
  QddgILSpin in 'QddgILSpin.pas';

end.
```

НА ЗАМЕТКУ

Для выделения участков кода, специфических для конкретной платформы, используйте отдельные блоки вида `{$IDFEF}..{$ENDIF}`, как это показано в примере исходного кода пакета. В частности, избегайте использования конструкций, наподобие следующей:

```
{$IFDEF MSWINDOWS}
  // Здесь расположен код для Windows
{$ELSE}
  // Здесь расположен код для Linux
{$ENDIF}
```

Если *Borland* когда-либо решит реализовать поддержку для других платформ, то такая конструкция приведет к использованию кода Linux для любой системы, которая не относится к Windows.

Листинг 13.8. `QddgDBSamples.dpk` — исходный код для пакета CLX времени выполнения, который содержит компоненты, взаимодействующие с базами данных

```
package QddgDBSamples;

{$R *.res}
{$ALIGN 8}
```

```
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components (Data-Aware)'}

{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}

{$IFDEF LINUX}
{$SOPREFIX 'bpl'}
{$SOVERSION '6'}
{$ENDIF}

{$RUNONLY}
{$IMPLICITBUILD OFF}

requires
  {$IFDEF MSWINDOWS}
  dbrtl,
  {$ENDIF}

  {$IFDEF LINUX}
  baseclx,
  dataclx,
  {$ENDIF}

  visualclx,
  visualdbclx,
  QddgSamples;

contains
  QddgDBSpin in 'QddgDBSpin.pas';

end.
```

Пакеты времени разработки

Компоненты можно помещать и в комбинированные пакеты (времени выполнения и разработки), но такой подход использовать не рекомендуется. Фактически пакет времени выполнения и разработки можно использовать только в том случае, если он не содержит редакторов компонентов. А для такого пакета потребуется пакет `designide`, который нельзя включать в состав инсталляционного комплекта.

Поэтому пакеты времени разработки с пакетами времени выполнения лучше не объединять. В листингах 13.9 и 13.10 представлен исходный код двух пакетов времени разработки. Первый из них содержит компоненты не взаимодействующие, а второй — компоненты, взаимодействующие с базами данных. И вновь обратите внимание на директивы условной компиляции со значениями `MSWINDOWS` и `LINUX`, а также на содержимое раздела `requires`.

Листинг 13.9. `QddgSamples_Dsgn.dpk` — исходный код пакета CLX времени разработки, содержащего компоненты, не взаимодействующие с базами данных

```
package QddgSamples_Dsgn;

{$R *.res}
{$R 'QddgSamples_Reg.dcr'}
{$ALIGN 8}
{$ASSERTIONS OFF}
{$BOOLEVAL OFF}
{$DEBUGINFO OFF}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS OFF}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components'}

{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}
{$IFDEF LINUX}
{$SOPREFIX'bpl'}
```

```
{$SOVERSION '6'}
{$ENDIF}

{$DESIGNONLY}
{$IMPLICITBUILD OFF}

requires
    {$IFDEF LINUX}
    baseclx,
    {$ENDIF}
    visualclx,
    designide,
    QddgSamples;

contains
    QddgSamples_Reg in 'QddgSamples_Reg.pas',
    QddgDsnEdt in 'QddgDsnEdt.pas',
    QddgRgpEdt in 'QddgRgpEdt.pas';

end.
```

НА ЗАМЕТКУ

Для того чтобы исходный файл пакета можно было использовать и в Kylix, и в Delphi 6, в названиях пакетов, указанных в разделе `requires`, должен учитываться регистр символов. Например, в VCL часто используется название `DesignIDE`, а в Linux — `designide`. Если в исходном коде пакета используется первый вариант названия, то Kylix не сможет найти файл `designide.dcp`, потому что в среде Linux `DesignIDE.dcp` и `designide.dcp` — это не одно и то же.

Листинг 13.10. `QddgDBSamples_Dsgn.dpk` — исходный код пакета CLX времени разработки, содержащего компоненты, взаимодействующие с базами данных

```
package QddgDBSamples_Dsgn;

{$R *.res}
{$ALIGN 8}
{$ASSERTIONS OFF}
{$BOOLEVAL OFF}
{$DEBUGINFO OFF}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS OFF}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
```

```
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components (Data-Aware)'}

{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}

{$IFDEF LINUX}
{$SOPREFIX 'bpl'}
{$SOVERSION '6'}
{$ENDIF}

{$DESIGNONLY}
{$IMPLICITBUILD OFF}

requires
  {$IFDEF LINUX}
  baseclx,
  {$ENDIF}
  visualclx,
  QddgSamples_Dsgn,
  QddgDBSamples;

contains
  QddgDBSamples_Reg in 'QddgDBSamples_Reg.pas';

end.
```

Модули регистрации

Как можно заметить, в приведенных листингах пакетов времени разработки для регистрации компонентов используются специальные модули регистрации. Обычно при создании компонентов VCL модули регистрации (`QddgSamples_Reg` и `QddgDBSamples_Reg`) включаются только в пакеты времени разработки. В листинге 13.11 представлен исходный код модуля `QddgSamples_Reg.pas`, который отвечает за регистрацию компонентов, не предназначенных для работы с базами данных, а также редакторы компонента `TddgRadioGroupEditor`.

Листинг 13.11. `QddgSamples_Reg.pas` — модуль регистрации для компонентов CLX, не взаимодействующих с базами данных

```
unit QddgSamples_Reg;

interface
```

```
procedure Register;

implementation

uses
  Classes, DesignIntf, DesignEditors, QExtCtrls, QddgSpin,
  QddgDsnSpn, QddgILSpin, QddgRgpEdt;


{==========================}
{== Процедура регистрации ==}
{==========================}

procedure Register;
begin
  {== Регистрация компонентов ==}

  RegisterComponents( 'DDG-CLX',
                      [ TddgSpinner,
                        TddgDesignSpinner,
                        TddgImgListSpinner ] );

  {== Регистрация редакторов компонентов ==}

  RegisterComponentEditor( TRadioGroup, TddgRadioGroupEditor );
end;

end.
```

Пиктограммы компонентов

Для идентификации созданного компонента CLX в палитре компонентов необходимо создать соответствующую пиктограмму. Она представляет собой 16-цветный растровый рисунок размером 24x24 пикселя. В интерактивной справочной системе Kylix и Delphi рекомендуется создавать отдельные файлы ресурсов для каждого модуля компонента.

В любом случае, редактор пакетов ищет соответствующий файл с расширением `.dcr` для каждого указанного в пакете модуля. К сожалению, этот поиск выполняется как для пакетов времени выполнения, так и для пакетов времени разработки. Учитывая это, включать пиктограммы в пакеты времени разработки не имеет смысла, так как они все равно не будут использованы и только займут место.

Следовательно, вместо создания отдельного файла с расширением `.dcr` для модулей каждого компонента лучше просто создать один файл, содержащий пиктограммы всех компонентов. К счастью, файлы ресурсов Kylix такие же, как и в Delphi. Исполняемые файлы, созданные Kylix, могут использоваться только в среде Linux, а вот формат подключаемых ресурсов аналогичен формату Win32. В результате файлу ресурса Windows с расширением `.res` можно просто присвоить расширение `.dcr`. Например, на рис. 13.9 показано содержимое файла `QddgSamples_Reg.dcr` в графическом редакторе Image Editor.

Обратите внимание: имя файла ресурса совпадает с именем модуля регистрации. Благодаря этому при добавлении модуля регистрации в пакет времени разработки добавляется также и файл ресурса. Как уже упоминалось, в пакетах времени выполнения модули регистрации не используются, поэтому в них также не будут включаться и пиктограммы компонентов.

Не надо недооценивать важность хороших пиктограмм компонентов, так как они являются тем аспектом, с которым пользователи сталкиваются в первую очередь. Непрофессиональные пиктограммы наводят на мысль о непрофессиональных компонентах. Если компоненты разрабатываются для коммерческих проектов, то для создания пиктограмм пригласите профессионального дизайнера.

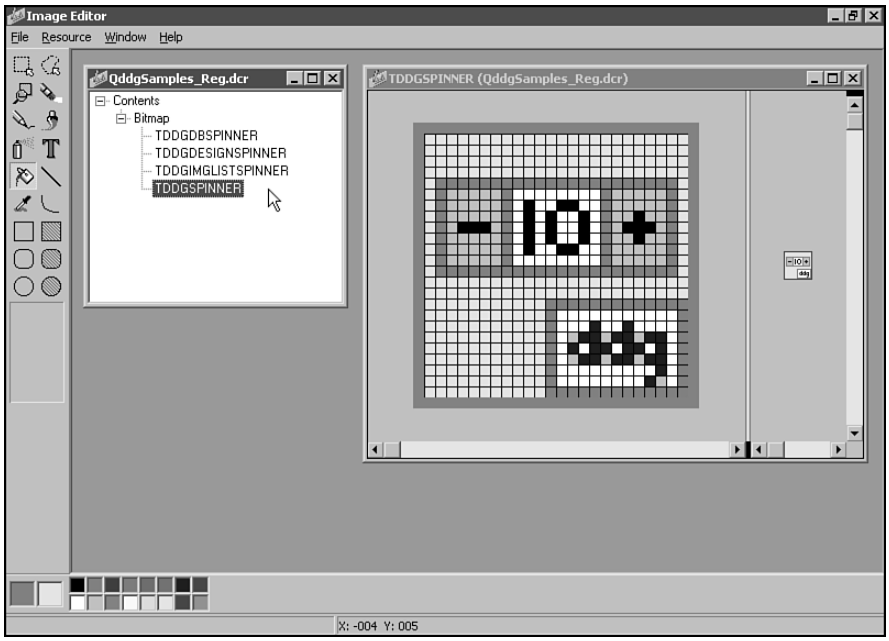

Рис. 13.9. *При создании файлов* `.DCR` *для компонентов CLX можно использовать редактор Image Editor*

Резюме

Преобразовывая компоненты VCL в CLX, в их исходный код следует внести некоторые изменения. Во-первых, везде, где это возможно, используйте уже существующие оболочки VCL. Например, вместо прямых обращений к функциям интерфейса GDI лучше использовать методы класса `TCanvas`. К тому же вместо обработки сообщений окон, наподобие `wm_LButtonDown`, необходимо использовать переопределенные обработчики событий (в данном случае — `MouseDown()`), так как в Linux сообщения, наподобие `wm_LButtonDown`, не используются. Кроме того для изоляции кода, характерного для конкретной платформы, рекомендуется создавать собственные абстрактные классы.

Библиотека CLX была разработана после VCL, однако для переноса существующих компонентов VCL в CLX придется приложить некоторые усилия. Количество вызовов специфических системных функций, наподобие API Win32 или libc, необходимо свести к минимуму или, по крайней мере, они должны быть заключены в соответствующих условных директивах. Несмотря на все вышеизложенное, можно создать компонент CLX на основе одного файла исходного кода, который будет использоваться как в Delphi под Windows, так и в Kylix под Linux.

ГЛАВА

14

Пакеты

В ЭТОЙ ГЛАВЕ...

В Delphi 3 появилась новая возможность — *пакеты* (package), позволяющие располагать элементы приложения в разных модулях, которые могут совместно использоваться многими приложениями. Пакеты подобны *динамически подключаемым библиотекам* (DLL — Dynamic Link Library), но отличаются от них способом применения. Пакеты в основном предназначены для хранения коллекций компонентов в отдельном, совместно используемом модуле *библиотек пакетов Borland* (Borland Package Library — файл `.bpl`). В Delphi пакеты могут использоваться во время выполнения — это позволяет не связывать их код в единое целое с кодом приложения еще на этапе компиляции. Поскольку код этих модулей находится в файлах `.bpl`, а не в файлах `.exe` или `.dll`, то размер последних значительно уменьшается.

Пакеты отличаются от библиотек DLL тем, что они являются частью библиотеки VCL Delphi. Это означает, что приложения, написанные на других языках, не могут использовать данные пакеты (за исключением C++Builder). Одной из причин введения пакетов была необходимость преодолеть ограничения, свойственные Delphi версий 1 и 2. В этих двух версиях Delphi подпрограммы библиотеки VCL увеличивали размер каждого исполняемого файла как минимум на 150–200 Кбайт. Таким образом, если поместить даже незначительную часть приложения в библиотеку DLL, то и файл данной библиотеки, и сам исполняемый файл будут содержать избыточный код. При установке на компьютере большого программного комплекса, состоящего из нескольких приложений, это выливалось в существенную проблему. Пакеты позволяют уменьшить размеры приложений и предоставляют удобный способ для распространения коллекций компонентов.

Для чего предназначены пакеты?

Использование пакетов вызвано рядом причин. О трех наиболее важных из них и пойдет речь в следующих разделах.

Сокращение размера кода

Самой главной причиной использования пакетов является стремление уменьшить размер приложений и библиотек DLL. Delphi поставляется в виде нескольких пакетов, среди которых по логическим признакам распределены компоненты библиотеки VCL. Создаваемые приложения можно компилировать с использованием этих уже существующих пакетов Delphi.

Дробление приложений и уменьшение их размеров

Как известно, многие приложения распространяются по Internet в виде полномасштабных версий, демонстрационных версий или в виде пакетов обновлений к уже существующим приложениям. Представьте себе преимущества такого режима загрузки: пользователь получает из Internet лишь дополнения к приложению, тогда как основная часть этого приложения (установленная ранее) постоянно находится на его компьютере.

Разбив приложение на пакеты, можно предоставить пользователям право получать обновления лишь для тех частей приложения, которые им действительно нужны. Следует, однако, принять во внимание несколько моментов, касающихся взаимоотношений версий пакетов и приложений. Но об этом — чуть позже.

Хранение компонентов

Одной из наиболее распространенных причин использования пакетов является распространение компонентов, созданных сторонними производителями. Те, кто занимается поставкой компонентов, безусловно знают, как создавать пакеты. Кстати, в пакеты можно помещать даже некоторые элементы времени разработки, такие как редакторы свойств и компонентов, мастера и эксперты. Все эти инструменты периода разработки распространяются в виде пакетов.

Пакеты или DLL

Использование DLL для хранения форм администрирования серверов приложений приводит к наличию в DLL его собственной копии модуля `Forms.pas`. Это приводит, в свою очередь, к серьезной ошибке, связанной с обработкой дескрипторов окна Windows, содержащихся внутри DLL. Когда DLL выгружается, ссылка на дескриптор окна все еще поддерживается операционной системой. Следующее сообщение, поступившее из очереди для всех окон верхнего уровня, вызовет такую ошибку в приложении, которая способна привести к аварийному завершению работы операционной системы. Использование пакетов вместо DLL предотвращает возникновение этой проблемы, поскольку пакеты обращаются к копии `Forms.pas` основного приложения, и сообщения из очереди могут передаваться приложению без проблем.

Когда не нужно использовать пакеты?

Не следует использовать пакеты времени выполнения, если нет уверенности в том, что их будут использовать и другие приложения. Иначе эти пакеты займут на диске места больше, чем код, скомпилированный в один исполняемый файл. Почему так получается? При создании приложения среднего размера с использованием пакетов исполняемый файл уменьшается приблизительно с 200 Кбайт до 30, и может показаться, что на диске удалось сохранить 170 Кбайт свободного пространства. Но при распространении этого приложения к нему добавляются используемые им пакеты, в том числе пакет `Vcl60.dcp`, размером почти 2 Мбайта. Естественно, это совсем не та экономия пространства на диске, на которую стоит рассчитывать. По нашему мнению, пакеты следует применять только в том случае, если ими будут пользоваться несколько исполняемых файлов. Помните, что все эти рассуждения относятся лишь к пакетам времени выполнения. Разработчики компонентов, напротив, должны обеспечить своим пользователям пакет разработки, содержащий компонент, который будет доступен в интегрированной среде разработки Delphi.

Типы пакетов

Существует четыре типа пакетов, которые можно создать и использовать.

- **Пакеты времени выполнения** (runtime package). Такие пакеты содержат код, компоненты и другие ресурсы, которые используются приложением во время выполнения. Если приложение создается в расчете на определенный пакет времени выполнения, то в случае отсутствия этого пакета приложение работать не будет.
- **Пакеты разработки** (design package). Эти пакеты содержат компоненты, редакторы свойств или компонентов, эксперты и все прочие элементы, необходимые для разработки приложения в интегрированной среде Delphi. Данный тип пакетов используется только в среде разработки Delphi и никогда не распространяется вместе с создаваемым приложением.
- **Пакеты разработки и времени выполнения**. Эти пакеты, объединяющие в себе возможности пакетов и первого, и второго типа, обычно используются в случае отсутствия специальных элементов времени разработки наподобие редакторов свойств и компонентов или экспертов. Пакеты такого типа можно создавать для упрощения процесса разработки и установки использующих их приложений. Но, если этот пакет все-таки содержит элементы разработки, их использование в приложениях будет связано с дополнительной нагрузкой, вызванной поддержкой режима разработки. Рекомендуем пакеты времени выполнения и пакеты разработки создавать раздельно, чтобы отделить специфические элементы времени разработки, если таковые имеются.
- **Пакеты, не являющиеся ни пакетами времени выполнения, ни пакетами разработки.** Эта редкая разновидность пакетов служит для использования другими пакетами. Они не предназначены ни для непосредственного использования другими приложениями, ни для применения в среде разработки. Таким образом, пакеты могут использовать или включать в себя другие пакеты.

Файлы пакетов

В табл. 14.1 приведено описание типов файлов пакетов в соответствии с их расширением.

Таблица 14.1. Файлы пакетов

Расширение	*Тип файла*	*Описание*
`.dpk`	Исходный файл пакета	Этот файл создается при запуске редактора пакетов. Можно считать его разновидностью файла проекта Delphi `.dpr`
`.dcp`	Символьный файл пакета времени выполнения и разработки	Скомпилированная версия пакета с символьной информацией пакета и его модулей. Кроме того, в нем содержится заголовок с информацией, необходимой среде разработки Delphi

Окончание табл. 14.1.

Расширение	*Тип файла*	*Описание*
`.dcu`	Скомпилированный модуль	Скомпилированная версия модуля, содержащегося в пакете. Для каждого модуля, содержащегося в пакете, создается собственный файл `.dcu`
`.bpl`	Пакет библиотеки времени выполнения и разработки	Это пакет времени выполнения и разработки, эквивалент библиотеки DLL Windows. Если это пакет времени выполнения, то необходимо распространять его вместе с теми приложениями, которые его используют. Если это пакет разработки, то он будет распространяться среди программистов, использующих его для написания программ, вместе со своим "собратом" времени выполнения. Помните: если не предполагается распространять исходный код самого проекта, то можно предоставить соответствующий ему файл `.dcp`

Использование пакетов времени выполнения

Чтобы использовать пакеты времени выполнения в разрабатываемых приложениях Delphi, достаточно установить флажок Build with Runtime Packages (Скомпоновать с пакетами времени выполнения) во вкладке Packages диалогового окна Project Options. При компоновке приложения с этим параметром, вместо статического подключения всех модулей в файл `.exe` или `.dll`, оно получит возможность динамически подключать соответствующие пакеты времени выполнения. В результате получится более стройное приложение. Однако не забывайте при его распространении передавать пользователям и необходимые пакеты.

Установка пакетов в IDE Delphi

Как правило, установку пакетов осуществляют, получив новый набор компонентов от сторонних разработчиков. Прежде всего необходимо поместить файлы пакета в нужное место. Папки, в которых обычно хранятся различные файлы пакетов, приведены в табл. 14.2.

Таблица 14.2. Расположение файлов пакетов

Файл пакета	*Местоположение*
Пакеты времени выполнения (`*.bpl`)	Пакеты времени выполнения должны быть помещены в каталог `\Windows\System` (для Windows 95/98) или `\WinNT\System32` (для Windows NT/2000)
Пакеты разработки (`*.bpl`)	Поскольку возможна установка сразу нескольких пакетов от различных производителей, пакеты разработки должны быть помещены в общую папку — там за ними легче уследить. Например, создайте папку `\Delphi 6\Pkg` и храните свои пакеты разработки там
Символьные файлы пакетов (`*.dcp`)	Их можно поместить туда же, куда и файлы пакетов разработки (`*.bpl`)
Откомпилированные модули (`*.dcu`)	Откомпилированные модули распространяют вместо файлов исходного кода (когда не все секреты можно раскрывать). Рекомендуем хранить файлы `.dcu` сторонних производителей в папке, аналогичной `\Delphi 6\Lib`; например в папке `\Delphi 6\3PrtyLyb`. Эта папка должна быть указана в пути поиска (search path)

Для установки пакета достаточно перейти во вкладку Packages диалогового окна Project Options, которое можно открыть, выбрав в меню Component пункт Install Packages.

Щелкните на кнопке Add и выберите нужный файл с расширением `.bpl`. Его можно найти во вкладке Project. По щелчку на кнопке OK новый пакет будет установлен в среду разработки Delphi. Если в этом пакете содержатся компоненты, то они появятся в новой вкладке палитры компонентов.

Разработка пакетов

Перед созданием нового пакета следует определиться с некоторыми вопросами. Во-первых, необходимо решить, пакет какого типа будет создаваться (времени выполнения, разработки или любого другого). От этого решения зависит выбор одного из представленных ниже сценариев. Во-вторых, нужно решить, как назвать новый пакет и где будут храниться файлы проекта. Не следует размещать готовый пакет в той же папке, в которой он создается. И, наконец, необходимо принять решение, какие модули будет содержать данный пакет и какие пакеты ему будут нужны.

Редактор пакетов

Пакеты обычно создаются с помощью редактора пакетов (Package Editor), который можно запустить, выбрав в диалоговом окне New Items пиктограмму Packages (это окно можно раскрыть, выбрав в меню File пункт New). Окно редактора пакетов содержит две папки — `Contains` и `Requires`.

Папка Contains

В папке `Contains` указывают модули, которые необходимо скомпилировать в новый пакет. Существует несколько правил, которые следует соблюдать при помещении модулей в папку `Contains` пакета.

- Пакет не должен быть указан в разделе `contains` другого пакета или в разделе `uses` модуля другого пакета.
- Модули, прямо или косвенно указанные в разделе `contains` пакета, не могут быть указаны в разделе `Requires` пакета. Эти модули и так будут включены в пакет при его компиляции.
- Нельзя помещать имя модуля в раздел `contains` пакета, если оно уже находится в разделе `contains` другого пакета, используемого этим же приложением.

Папка Requires

В этой папке указывают другие пакеты, необходимые данному пакету. Это подобно использованию раздела `uses` в модулях Delphi. В большинстве случаев создаваемые пакеты будут использовать пакет `VCL60`, содержащий стандартные компоненты библиотеки VCL Delphi. Поэтому следует поместить его в раздел `requires`. Обычно все пользовательские компоненты помещаются в пакет времени выполнения, а затем создается пакет разработки, в раздел `requires` которого и включают данный пакет времени выполнения. Существует несколько правил помещения пакетов в раздел `Requires` другого пакета.

- Избегайте циклических ссылок: `Package1` не может иметь в своем разделе `Requires` пакет `Package1` или другой пакет, содержащий `Package1` в своем разделе `Requires`.
- Цепочка ссылок не должна возвращаться к пакетам, уже указанным в ней.

Редактор пакетов имеет панель инструментов и контекстно-зависимое меню. О назначении каждой из кнопок и команд можно узнать в разделе "*Package Editor*" интерактивной справочной системы Delphi.

Сценарии разработки пакетов

Выше уже отмечалось, что тип разрабатываемого пакета основан на том или ином сценарии его применения. В настоящем разделе предлагается три возможных сценария использования пакетов времени разработки и/или времени выполнения.

Сценарий 1. Пакеты разработки и времени выполнения для компонентов

Сценарий для пакетов разработки и времени выполнения, содержащих компоненты, потребуется разработчикам компонентов в двух случаях.

- Необходимо предоставить программистам Delphi право выбора: компоновать приложение вместе с этими компонентами или распространять их отдельно от него.

- Нежелательно заставлять пользователей компилировать в код приложения элементы времени разработки (редакторы свойств, компонентов и т.п.).

Исходя из этого сценария, следует создать пакет одновременно и разработки, и времени выполнения. На рис. 14.1 изображена схема организации такого пакета. Как видите, пакет разработки `ddgDT60.dpk` включает в себя и элементы времени разработки (редакторы свойств, компонентов и т.п.), и пакет времени выполнения `ddgRT60.dpk`. Этот вложенный пакет (`ddgRT60.dpk`) содержит только выполняемый код новых компонентов. Чтобы достичь подобного устройства пакета, необходимо включить имя пакета времени выполнения в раздел `requires` пакета разработки (рис. 14.1).

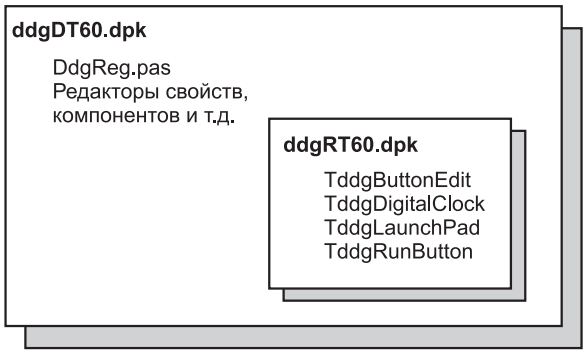

Рис. 14.1. *Пакет разработки содержит элементы разработки и пакеты времени выполнения*

Перед компиляцией каждого пакета необходимо установить соответствующие параметры. Это можно делать в диалоговом окне Package Options, которое можно открыть, выбрав в контекстном меню редактора пакетов пункт Options. Для пакета времени выполнения `DdgRT60.dpk` переключатель Usage Options (Параметры применения) должен быть установлен в положение Runtime Only (Только во время выполнения). Это гарантирует, что компонент времени выполнения не будет установлен в интегрированную среду как пакет разработки (более подробная информация по данной теме приведена далее в настоящей главе). Для пакета разработки `DdgDT60.dpk` этот переключатель должен быть установлен в положение Design Time Only (Только во время разработки). Это обеспечит установку пакета в среде разработки Delphi и предохранит его от использования в качестве пакета времени выполнения.

Вставка пакета времени выполнения в пакет разработки еще не делает содержащиеся в нем компоненты доступными в IDE Delphi. Компоненты нужно зарегистрировать. Как уже отмечалось, при создании компонента Delphi автоматически вносит в его модуль процедуру `Register()`, которая, в свою очередь, вызывает процедуру `RegisterComponents()`. Это и есть процедура, регистрирующая любой компонент в среде IDE Delphi при его установке. При работе с пакетами рекомендуем выделить процедуру `Register()` из компонентов в отдельный модуль регистрации. Данный модуль будет регистрировать все компоненты пакета, вызывая процедуры `RegisterComponents()`. Это не только упрощает управление регистрацией

компонентов, но и позволяет избежать нелегальной установки и использования разработанного пакета, так как незарегистрированные компоненты будут недоступны в IDE.

В частности, все использованные в данной книге компоненты помещены в пакет времени выполнения DdgRT60.dpk. Редакторы свойств, редакторы компонентов и регистрационный модуль (DdgReg.pas) этих компонентов находятся в пакете разработки DdgDT60.dpk. В разделе requires данного пакета указан и пакет DdgRT60.dpk. В листинге 14.1 показано, как может выглядеть регистрационный модуль.

Листинг 14.1. Модуль регистрации компонентов данной книги

```
unit DDGReg;

interface

procedure Register;

implementation

uses
  Classes, ExptIntf, DsgnIntf, TrayIcon, AppBars, ABExpt,
  Worthless, RunBtn, PwDlg, Planets, LbTab, HalfMin, DDGClock,
  ExMemo, MemView, Marquee, PlanetPE, RunBtnPE, CompEdit, DefProp,
  Wavez, WavezEd, LnchPad, LPadPE, Cards, ButtonEdit, Planet,
  DrwPnel;

procedure Register;
begin

  // Регистрация компонентов.
  RegisterComponents('DDG', [ TddgTrayNotifyIcon,
                    TddgDigitalClock, TddgHalfMinute,
                    TddgButtonEdit, TddgExtendedMemo,
                    TddgTabListbox, TddgRunButton, TddgLaunchPad,
                    TddgMemView, TddgMarquee, TddgWaveFile,
                    TddgCard, TddgPasswordDialog, TddgPlanet,
                    TddgPlanets, TddgWorthLess, TddgDrawPanel,
                    TComponentEditorSample, TDefinePropTest]);

  // Регистрация редакторов свойств.
  RegisterPropertyEditor(TypeInfo(TRunButtons), TddgLaunchPad, '',
                         TRunButtonsProperty);
  RegisterPropertyEditor(TypeInfo(TWaveFileString), TddgWaveFile,
                         'WaveName', TWaveFileStringProperty);
  RegisterComponentEditor(TddgWaveFile, TWaveEditor);
  RegisterComponentEditor(TComponentEditorSample, TSampleEditor);
  RegisterPropertyEditor(TypeInfo(TPlanetName),TddgPlanet,
                         'PlanetName', TPlanetNameProperty);
  RegisterPropertyEditor(TypeInfo(TCommandLine), TddgRunButton,
                         '', TCommandLineProperty);

  // Регистрация пользовательских модулей и экспертов библиотек.
```

```
  RegisterCustomModule(TAppBar, TCustomModule);
  RegisterLibraryExpert(TAppBarExpert.Create);

end;

end.
```

Защита компонентов

Для регистрации компонентов достаточно обладать всего лишь пакетом времени выполнения. Для него создается собственный модуль регистрации, в котором регистрируются компоненты данного пакета. Затем этот модуль добавляется в отдельный пакет, содержащий пакет времени выполнения в разделе `requires`. После этого достаточно установить новый пакет в среду Delphi, и все компоненты появятся в палитре компонентов. Тем не менее скомпилировать приложение, использующее такие компоненты, будет невозможно до тех пор, пока не будут установлены соответствующие файлы `.dcu` модулей данных компонентов.

Распространение пакетов

При передаче разработчикам компонентов пакетов без их исходного кода в комплект поставки следует включать файлы обоих откомпилированных пакетов `DdgRT6.bpl` и `DdgDT6.bpl`, оба файла `*.dcp`, а также все скомпилированные модули (`*.dcu`), необходимые для компиляции поставляемых компонентов. Программисты, распространяющие пакеты времени выполнения своих приложений, в которых они используют компоненты из ваших пакетов, также должны будут вместе со своими приложениями распространять пакет `DdgRT6.bpl` и все другие пакеты времени выполнения, используемые этими приложениями.

Сценарий 2. Пакет компонентов только для разработки

Этот сценарий используется в случаях, когда компоненты необходимо распространять в пакетах, не предусматривающих исполнения. В таком случае все компоненты, редакторы компонентов, редакторы свойств и модуль регистрации компонентов помещаются в один файл пакета.

Распространение пакетов

При распространении такого пакета без исходного кода в комплект поставки следует включать скомпилированный файл пакета `DdgDT6.bpl`, файл `DdgDT6.dcp` и все скомпилированные модули (файлы `*.dcu`), необходимые для компиляции распространяемых компонентов. Программисты, использующие эти компоненты, должны будут включить их в свои приложения. Они не смогут распространять ваши компоненты в пакетах времени выполнения.

Сценарий 3. Расширение IDE элементами разработки без компонентов

Данный сценарий применяется, когда среда разработки Delphi дополняется специальными инструментами, такими например, как эксперты. Распространяемые эксперты регистрируются в специальном модуле регистрации. Принцип распространения по такому сценарию очень прост — поставляется только файл `*.bpl`.

Сценарий 4. Дробление приложений

Этот сценарий применяется при дроблении приложения на отдельные логические части, каждая из которых может распространяться отдельно. Существует несколько причин использования такого сценария.

- Этот сценарий легче поддерживать.
- Вначале пользователи могут приобрести приложение, обладающее лишь минимально необходимым набором функций, а впоследствии, когда им потребуется расширить возможности, они докупают пакет, размер которого значительно меньше целого приложения.
- Модернизировать (или исправлять) приложение по частям (patch) значительно легче, а пользователям это позволит избежать повторной установки полной версии приложения.

По такому сценарию распространяются лишь файлы `*.bpl`, необходимые для работы приложения. Такой сценарий подобен предыдущему, но за одним исключением: вместо пакетов для IDE Delphi пользователю предоставляются пакеты собственно приложения. При таком дроблении приложения следует особое внимание уделить вопросам, связанным с версиями пакетов. Об этом пойдет речь в следующем разделе.

Версии пакетов

Версию (versioning) пакета часто понимают неправильно. Версии пакетов похожи на версии модулей. Другими словами, любой пакет приложения должен быть скомпилирован с использованием той же версии Delphi, что и само приложение. Таким образом, пакет, написанный в Delphi 6, нельзя использовать с приложением, созданным в Delphi 5. Разработчики компании *Borland* обычно называют версию пакета *базой кода* (code base). Поэтому пакет, созданный в Delphi 6, можно назвать написанным на базе кода 6.0. Эту идею можно отразить в соглашении об именах, используемых для файлов пакетов.

Директивы компилятора для пакетов

Существует несколько директив компилятора, предназначенных специально для использования в исходном коде пакетов. Одни из них относятся к модулям пакетов, другие — к файлам пакетов. Эти директивы приведены в табл. 14.3 и 14.4.

Таблица 14.3. Директивы компилятора для модулей пакета

Директива	*Назначение*
{$G} или {$IMPORTEDDATA OFF}	Используется для предотвращения помещения модуля в пакет, т.е. в том случае, когда модуль должен быть скомпилирован непосредственно с приложением. Противоположна директиве {$WEAKPACKAGEUNIT}, позволяющей модулю входить в состав пакета, код которого связан с приложением статически
{$DENYPACKAGEUNIT}	То же самое, что и {$G}
{$WEAKPACKAGEUNIT}	Описано в следующем разделе

Таблица 14.4. Директивы компилятора для файлов .dpk пакета

Директива	*Назначение*
{$DESIGNONLY ON}	Компилирует пакет только как пакет разработки
{$RUNONLY ON}	Компилирует пакет только как пакет времени выполнения
{$IMPLICITBUILD OFF}	Предотвращает перекомпоновку пакета в дальнейшем. Используйте эту возможность в пакетах, которые не придется изменять часто

Подробней о директиве {$WEAKPACKAGEUNIT}

Концепция *слабого пакета* (weak package) очень проста. Обычно она используется, если пакет ссылается на библиотеки DLL, которые могут отсутствовать. Например, Vcl60 обращается к API Win32 ядра, входящего в состав ряда операционных систем Windows (присутствует у NT/2000, а у 95/98 отсутствует). Многие из вызываемых ей подпрограмм находятся в библиотеках DLL, присутствующих далеко не на каждом компьютере. Эти вызовы распознаются модулями, содержащими директиву {$WEAKPACKAGEUNIT}. Наличие такой директивы приводит к тому, что исходный код данного модуля будет помещен в пакете в файл DCP, а не в BPL (файл .DCP — аналог файла .DCU, а файл .BPL — аналог файла .DLL). Таким образом, любые ссылки на функции этих "слабо пакетированных" модулей будут статически связаны с приложением, в отличие от динамических ссылок на пакет.

Директива {$WEAKPACKAGEUNIT} употребляется достаточно редко (возможно, с ней и вовсе не придется работать). Она позволяет разработчикам Delphi отрабатывать специфические ситуации. Предположим, существует два компонента (каждый в отдельном пакете), ссылающихся на один и тот же модуль интерфейса библиотеки DLL. Если приложение использует оба компонента, то это приведет к загрузке двух экземпляров библиотеки DLL, что, в свою очередь, вызовет конфликты инициализации и использования глобальных переменных. Решением данной проблемы будет помещение модуля интерфейса в один из стандартных пакетов Delphi (типа Vcl60.bpl). Однако такой подход не позволяет избавиться от других проблем, связанных с отсутствием специализированных библиотек DLL (например PENWIN.DLL). Если библиотека DLL, указанная в модуле интерфейса, отсутствует, то пакет Vcl60.bpl (а значит, и Delphi) окажутся

бесполезными. Разработчики Delphi ликвидируют эту опасность, позволяя `Vcl60.bpl` содержать модуль интерфейса в отдельном пакете, который подключается статически и не загружается динамически при работе с `Vcl60.bpl` в среде Delphi.

Как уже отмечалось, на практике вряд ли придется прибегнуть к этой директиве, если только не предвидится ситуация, с которой довелось столкнуться создателям Delphi, или если не понадобится удостовериться в том, что определенный модуль включен в пакет, но при этом с использующим его приложением статически связан. Это можно применить, например, для оптимизации работы приложения. Обратите внимание: любые слабо пакетированные модули не могут обладать глобальными переменными или кодом в своих разделах инициализации и завершения. Кроме того, для слабо пакетированных модулей вместе с пакетами следует распространять и файлы `*.dcu`.

Соглашения об именах пакетов

Как уже говорилось, версии пакетов должны отражаться в их именах. Настоятельно рекомендуем (хотя вы и не обязаны это делать) использовать в соглашении об именах пакетов базу кода. Например, компоненты этой книги находятся в пакете времени выполнения `DdgRT6.dpk`, имя которого содержит базу кода Delphi 6 — `6`. То же самое можно сказать и о пакете разработки `DdgDT6.dpk`. Предыдущей версией этого пакета была бы `DdgRT5.dpk`[1]. Используя это соглашение, вы избавляете своих пользователей от неприятностей, связанных с согласованием версии пакета и компилятора Delphi. Имя нашего пакета начинается с трехсимвольного идентификатора автора/компании (`Ddg`), за которым следуют символы `RT` (для пакета времени выполнения) или `DT` (для пакета разработки). Вы можете следовать любому соглашению, отвечающему вашим требованиям, но все же включайте в имена пакетов номер версии Delphi.

Расширяемые приложения, использующие пакеты времени выполнения (дополнения)

Пакеты дополнений (add-in packages) позволяют дробить приложения на части (или модули), которые будут распространяться отдельно от основного приложения. Такая схема оказывается особенно привлекательной, поскольку позволяет расширять возможности приложения без внесения изменений в код всего приложения и повторной компиляции. Однако подобный подход требует тщательного архитектурного планирования. Углубление в вопросы такой разработки выходит за рамки этой книги. В данном разделе приведена лишь иллюстрация этого метода.

---

[1] *Авторы абсолютно правы – правила необходимо соблюдать! На самом деле прежней версией пакета разработки был `DDGStd50.dpk`, где "за трехсимвольным идентификатором автора/компании (`DDG`) следуют символы `Std` (для пакета времени выполнения) или `Dsgn` (для пакета разработки)".* – Прим. ред.

Создание форм дополнений

Здесь приложение разбито на три логические части: главное приложение (ChildTest.exe), пакет, содержащий класс TChildForm (AIChildFrm6.bpl), и конкретные классы, производные от класса TChildForm, каждый из которых расположен в своем собственном пакете.

Пакет AIChildFrm6.bpl включает в себя *абстрактный* базовый класс TChildForm. В других пакетах содержатся *конкретные* классы или производные от TChildForm (например TChildForm). Таким образом, авторы называют эти пакеты соответственно *абстрактным пакетом* (base package) и *конкретными пакетами* (concrete packages).

Главное приложение использует абстрактный пакет (AIChildFrm6.bpl). Каждый конкретный пакет также использует абстрактный пакет. Для обеспечения корректной работы главное приложение должно быть скомпилировано с пакетами времени выполнения, включая пакет AIChildFrm6.dcp. Аналогично, каждый конкретный пакет должен использовать пакет AIChildFrm6.dcp. Не будем приводить здесь код исходного класса TChildForm или его конкретных потомков, но отметим, что каждый модуль класса, производного от TChildForm, должен обладать разделами initialization и finalization, которые имеют следующий вид:

```
initialization
  RegisterClass(TCF2Form);
finalization
  UnRegisterClass(TCF2Form);
```

Обращение к процедуре RegisterClass() необходимо для создания класса, производного от TChildForm, доступного системе обработки потоков главного приложения при загрузке им своего пакета. Эта процедура аналогична процедуре RegisterComponents(), которая делает компоненты доступными IDE Delphi, а при выгрузке пакета необходимо обратиться к процедуре UnRegisterClass() для удаления зарегистрированного класса. Заметим, что процедура RegisterClass() делает соответствующий класс доступным только главному приложению. Причем главному приложению все еще остается неизвестным имя класса. Возникает вопрос: как же главное приложение создает экземпляр класса, имени которого оно не знает? Это и является целью данного упражнения — сделать формы доступными главному приложению без жесткого задания имен классов в исходном коде. В листинге 14.2 представлен исходный код главной формы приложения, в котором особого внимания заслуживает реализация форм с помощью пакетов дополнений.

Листинг 14.2. Главная форма приложения, использующего пакеты дополнений

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, ChildFrm, Menus;
```

```
const
  { Дочерняя форма регистрируется в системном реестре Windows. }
  cAddInIniFile  = 'AddIn.ini';
  cCFRegSection  = 'ChildForms';  // Раздел инициализации данных

  FMainCaption   = 'Delphi 6 Developer''s Guide Child Form Demo';

type

  TChildFormClass = class of TChildForm;

  TMainForm = class(TForm)
    pnlMain: TPanel;
    Splitter1: TSplitter;
    pnlParent: TPanel;
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiHelp: TMenuItem;
    mmiForms: TMenuItem;
    procedure mmiExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    // Ссылка на дочернюю форму.
    FChildForm: TChildForm;
    // Список доступных дочерних форм для построения меню.
    FChildFormList: TStringList;
    // Индекс меню Close Form (закрыть форму) по позиции
    // смещения
    FCloseFormIndex: Integer;
    // Дескриптор загруженного в данный момент пакета.
    FCurrentModuleHandle: HModule;
    // Метод создания меню для доступных дочерних форм.
    procedure CreateChildFormMenus;
    // Обработчик загрузки дочерней формы и ее пакета.
    procedure LoadChildFormOnClick(Sender: TObject);
    // Обработчик выгрузки дочерней формы и ее пакета.
    procedure CloseFormOnClick(Sender: TObject);
    // Метод считывания имени потомка класса TChildForm.
    function GetChildFormClassName(const AModuleName:
                                   String): String;
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
uses IniFiles;

{$R *.DFM}
```

```
function RemoveExt(const AFileName: String): String;
{ Вспомогательная функция удаления расширения из имени файла. }
begin
  if Pos('.', AFileName) <> 0 then
    Result := Copy(AFileName, 1, Pos('.', AFileName)-1)
  else
    Result := AFileName;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FChildFormList := TStringList.Create;
  CreateChildFormMenus;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FChildFormList.Free;
  // Выгрузить все загруженные дочерние формы.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);
end;

procedure TMainForm.CreateChildFormMenus;
{ Все доступные дочерние формы регистрируются в системном реестре
Windows. Здесь эта информация используется для создания элементов
меню, обеспечивающих загрузку всех дочерних форм. }
var
  IniFile: TIniFile;
  MenuItem: TMenuItem;
  i: integer;
begin
  inherited;

  { Считать список всех дочерних форм и построить меню на
    основании записей в системном реестре. }
  IniFile := TIniFile.Create(ExtractFilePath(Application.ExeName)
                                                    + cAddInIniFile);
  try
    IniFile.ReadSectionValues(cCFRegSection, FChildFormList);
  finally
    IniFile.Free;
  end;

  { Добавить элементы меню для каждого модуля.
    ОБРАТИТЕ ВНИМАНИЕ: свойство mmMain.AutoHotKeys должно быть
    установлено равным значению maAutomatic. }

  for i := 0 to FChildFormList.Count - 1 do begin
```

```
    MenuItem := TMenuItem.Create(mmMain);
    MenuItem.Caption := FChildFormList.Names[i];
    MenuItem.OnClick := LoadChildFormOnClick;
    mmiForms.Add(MenuItem);
  end;

  // Создать разделитель
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '-';
  mmiForms.Add(MenuItem);

  // Создать элемент меню Close Module (Закрыть модуль)
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '&Close Form';
  MenuItem.OnClick := CloseFormOnClick;
  MenuItem.Enabled := False;
  mmiForms.Add(MenuItem);

  { Сохранить ссылку на индекс элемента меню, необходимый чтобы
    закрыть дочернюю форму. Она будет использована в другом
    методе. }
  FCloseFormIndex := MenuItem.MenuIndex;
end;

procedure TMainForm.LoadChildFormOnClick(Sender: TObject);
var
  ChildFormClassName: String;
  ChildFormClass: TChildFormClass;
  ChildFormName: String;
  ChildFormPackage: String;
begin
  // Заголовок меню представляет имя модуля.
  ChildFormName := (Sender as TMenuItem).Caption;
  // Получить реальное имя файла пакета.
  ChildFormPackage := FChildFormList.Values[ChildFormName];
  // Выгрузить все ранее загруженные пакеты.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);

  try
    // Загрузить заданный пакет
    FCurrentModuleHandle := LoadPackage(ChildFormPackage);
    // Возвратить имя класса, необходимое для создания экземпляра
    ChildFormClassName := GetChildFormClassName(ChildFormPackage);

    { Создать экземпляр класса с помощью процедуры FindClass().
      Заметьте, этот класс должен быть уже зарегистрирован в
      системе обработки потоков с помощью
      процедуры RegisterClass(). Это реализовано в разделе
      инициализации дочерней формы для каждого пакета дочерней
      формы. }
    ChildFormClass := TChildFormClass(
                                  FindClass(ChildFormClassName));
    FChildForm := ChildFormClass.Create(self, pnlParent);
```

```
    Caption := FChildForm.GetCaption;

    { Объединить меню дочерней формы с главным меню }
    if FChildForm.GetMainMenu <> nil then
      mmMain.Merge(FChildForm.GetMainMenu);
    FChildForm.Show;
    mmiForms[FCloseFormIndex].Enabled := True;
  except
    on E: Exception do begin
      CloseFormOnClick(nil);
      raise;
    end;
  end;
end;

function TMainForm.GetChildFormClassName(const AModuleName:
                                          String): String;
{ Имя реального потомка класса TChildForm находится в системном
реестре. Данный метод считывает имя этого класса. }
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create(ExtractFilePath(Application.ExeName)
                                                + cAddInIniFile);
  try
    Result := IniFile.ReadString(RemoveExt(AModuleName),
                                 'ClassName', EmptyStr);
  finally
    IniFile.Free;
  end;
end;

procedure TMainForm.CloseFormOnClick(Sender: TObject);
begin
  if FCurrentModuleHandle <> 0 then begin
    if FChildForm <> nil then begin
      FChildForm.Free;
      FChildForm := nil;
    end;

    // Отмена регистрации всех классов модуля
    UnRegisterModuleClasses(FCurrentModuleHandle);
    // Выгрузка пакета дочерней формы
    UnloadPackage(FCurrentModuleHandle);

    FCurrentModuleHandle := 0;
    mmiForms[FCloseFormIndex].Enabled := False;
    Caption := FMainCaption;
  end;
end;

end.
```

Логика этого приложения довольно проста. В нем для определения доступных пакетов используется системный реестр. Имена найденных пакетов помещаются в строки меню, создаваемого для предоставления возможности загрузки каждого пакета. Кроме того, считывается имя класса формы, содержащейся в каждом пакете.

Основную нагрузку несет на себе обработчик события `LoadChildFormOnClick()`. После определения имени пакета этот метод загружает нужный пакет с помощью функции `LoadPackage()`. (Функция `LoadPackage()` принципиально ничем не отличается от функции `LoadLibrary()`, используемой для загрузки библиотек DLL.) Затем этот метод определяет имя класса для формы, содержащейся в загруженном пакете.

Чтобы создать класс, необходима ссылка на его имя (например `TButton` или `TForm1`). Но в этом главном приложении отсутствует жестко заданное имя класса конкретной дочерней формы типа `TChildForm`. Поэтому имя класса считывается из системного реестра. Главное приложение способно передать имя этого класса функции `FindClass()` и получить от нее ссылку на заданный класс, который уже был зарегистрирован системой обработки потоков. Помните, что регистрация выполняется в разделе инициализации модуля конкретной формы, который вызывается при загрузке соответствующего пакета. Затем, с помощью следующих строк, создается экземпляр класса:

```
ChildFormClass := TChildFormClass(FindClass(ChildFormClassName));
FChildForm := ChildFormClass.Create(self, pnlParent);
```

НА ЗАМЕТКУ

Ссылка на класс (class reference) — это не более чем область в памяти, которая содержит информацию о классе. Это похоже на определение типов для класса. Когда класс регистрируется потоковой системой VCL, вызов функции `RegisterClass()` загружает его в память. Функция `FindClass()` находит в памяти область, занимаемую определением класса по его имени, и возвращает указатель на нее. Это не то же самое, что и экземпляр класса. Экземпляры класса обычно создаются при вызове функции конструктора класса (см. главу 2, “Язык программирования Object Pascal”).

Переменная `ChildFormClass` представляет собой заранее определенную ссылку на класс `TChildForm` и может содержать ссылку на любой потомок класса `TChildForm`.

Обработчик события `CloseFormOnClick()` просто закрывает дочернюю форму и выгружает ее пакет. Остальная часть кода используется в основном для создания меню пакетов и чтения информации из системного реестра.

Углубленное изучение данной технологии позволит создавать очень гибкие структуры приложений.

Экспорт функций из пакетов

Исходя из того, что пакет можно рассматривать как улучшенную разновидность библиотеки DLL, справедливо предположить, что существует возможность экспорта процедур и функций из пакетов, точно так же как и из DLL. Так и есть, эта возможность существует. В настоящем разделе описано, как использовать пакеты подобным образом.

Загрузка формы из функции, расположенной в пакете

Листинг 14.3 представляет собой модуль, содержащийся внутри пакета.

Листинг 14.3. Модуль пакета с двумя экспортируемыми функциями

```
unit FunkFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  TFunkForm = class(TForm)
    Label1: TLabel;
    Button1: TButton;
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

// Объявление функции пакета в соответствии с
// соглашением о вызовах StdCall
procedure FunkForm; stdcall;
function AddEm(Op1, Op2: Integer): Integer; stdcall;

// Экспорт функций.
exports
  FunkForm,
  AddEm;

implementation

{$R *.dfm}

procedure FunkForm;
var
  FunkForm: TFunkForm;
begin
  FunkForm := TFunkForm.Create(Application);
  try
    FunkForm.ShowModal;
  finally
    FunkForm.Free;
  end;
end;
```

```
function AddEm(Op1, Op2: Integer): Integer;
begin
  Result := Op1+Op2;
end;

end.
```

Процедура `FunkForm()` просто отображает объявленную в модуле модальную форму, и ничего более. Функция `AdEm()` получает два операнда и возвращают их сумму. Обратите внимание, что функции объявлены в разделе интерфейса модуля в соответствии с соглашением о вызовах `StdCall`.

Листинг 14.4 содержит исходный код приложения, демонстрирующего вызов функций из пакета.

Листинг 14.4. Пример приложения

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Mask;

const
  cFunkForm = 'FunkForm';
  cAddEm    = 'AddEm';

type
  TForm1 = class(TForm)
    btnPkgForm: TButton;
    meOp1: TMaskEdit;
    meOp2: TMaskEdit;
    btnAdd: TButton;
    lblPlus: TLabel;
    lblEquals: TLabel;
    lblResult: TLabel;
    procedure btnAddClick(Sender: TObject);
    procedure btnPkgFormClick(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

  // Определение сигнатур методов
  TAddEmProc = function(Op1, Op2: Integer): integer; stdcall;
  TFunkFormProc = procedure; stdcall;

var
  Form1: TForm1;
```

```
implementation

{$R *.dfm}

procedure TForm1.btnAddClick(Sender: TObject);
var
  PackageModule: THandle;
  AddEmProc: TAddEmProc;
  Rslt: Integer;
  Op1, Op2: integer;
begin
  PackageModule := LoadPackage('ddgPackFunk.bpl');
  try
    @AddEmProc := GetProcAddress(PackageModule, PChar(cAddEm));
    if not (@AddEmProc = nil) then begin
      Op1 := StrToInt(meOp1.Text);
      Op2 := StrToInt(meOp2.Text);
      Rslt := AddEmProc(Op1, Op2);
      lblResult.Caption := IntToStr(Rslt);
    end;
  finally
    UnloadPackage(PackageModule);
  end;
end;

procedure TForm1.btnPkgFormClick(Sender: TObject);
var
  PackageModule: THandle;
  FunkFormProc: TFunkFormProc;
begin
  PackageModule := LoadPackage('ddgPackFunk.bpl');
  try
    @FunkFormProc := GetProcAddress(PackageModule,
                                    PChar(cFunkForm));
    if not (@FunkFormProc = nil) then
      FunkFormProc;
  finally
    UnloadPackage(PackageModule);
  end;
end;

end.
```

Обратите внимание: вначале необходимо объявить два процедурных типа, `TAddEmProc` и `TFunkFormProc`. Они объявлены точно так же, как и в пакете.

Обсудим сначала обработчик события `btnPkgFormClick()`. Подобный код уже обсуждался в главе 6, “Динамически компонуемые библиотеки”, но здесь вместо обращения к функции `LoadLibrary()` используется `LoadPackage()`. Фактически `LoadPackage()` завершается вызовом `LoadLibrary()`. Затем с помощью функции `GetProcAddress()` получаем ссылку на процедуру. Более подробная информация об

этой функции приведена в главе 6, "Динамически компонуемые библиотеки". Константа `cFunkForm` содержит имя функции в пакете.

Как можно заметить, способ экспорта функций и процедур из пакетов почти точно такой же, как и при экспорте из динамических библиотек.

Как получить информацию о пакете

Вполне возможно запросить у пакета информацию о содержащихся в нем модулях и необходимых ему пакетах. Для этого используются две процедуры: `EnumModules()` и `GetPackageInfo()`. Обоим процедурам необходимы функции обратного вызова (callback function). Листинг 14.5 иллюстрирует применение таких функций. Исходный код примера находится на прилагаемом CD.

Листинг 14.5. Пример получения информации о пакете

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls, DBXpress, DB,
  SqlExpr, DBTables;

type
  TForm1 = class(TForm)
    Button1: TButton;
    TreeView1: TTreeView;
    Table1: TTable;
    SQLConnection1: TSQLConnection;
    procedure Button1Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

type
  TNodeHolder = class
    ContainsNode: TTreeNode;
    RequiresNode: TTreeNode;
  end;

procedure RealizeLength(var S: string);
```

```
begin
  SetLength(S, StrLen(PChar(S)));
end;

procedure PackageInfoProc(const Name: string;
                          NameType: TNameType; Flags: Byte;
                          Param: Pointer);
var
  NodeHolder: TNodeHolder;
  TempStr: String;
begin
  with Form1.TreeView1.Items do begin
    TempStr := EmptyStr;
    if (Flags and ufMainUnit) <> 0 then
      TempStr := 'Main unit'
    else if (Flags and ufPackageUnit) <> 0 then
      TempStr := 'Package unit' else
    if (Flags and ufWeakUnit) <> 0 then
      TempStr := 'Weak unit';
    if TempStr <> EmptyStr then
      TempStr := Format(' (%s)', [TempStr]);
    NodeHolder := TNodeHolder(Param);
    case NameType of
      ntContainsUnit: AddChild(NodeHolder.ContainsNode,
                               Format('%s %s', [Name,TempStr]));
      ntRequiresPackage: AddChild(NodeHolder.RequiresNode, Name);
    end; // case
  end;
end;

function EnumModuleProc(HInstance: integer;
                        Data: Pointer): Boolean;
var
  ModFileName: String;
  ModNode: TTreeNode;
  ContainsNode: TTreeNode;
  RequiresNode: TTreeNode;
  ModDesc: String;
  Flags: Integer;
  NodeHolder: TNodeHolder;

begin
  with Form1.TreeView1 do begin
    SetLength(ModFileName, 255);
    GetModuleFileName(HInstance, PChar(ModFileName), 255);
    RealizeLength(ModFileName);
    ModNode := Items.Add(nil, ModFileName);
    ModDesc := GetPackageDescription(PChar(ModFileName));
    ContainsNode := Items.AddChild(ModNode, 'Contains');
    RequiresNode := Items.Addchild(ModNode, 'Requires');
    if ModDesc <> EmptyStr then begin
      NodeHolder := TNodeHolder.Create;
      try
```

```
        NodeHolder.ContainsNode := ContainsNode;
        NodeHolder.RequiresNode := RequiresNode;
        GetPackageInfo(HInstance, NodeHolder,
                       Flags, PackageInfoProc);
      finally
        NodeHolder.Free;
      end;

      Items.AddChild(ModNode, ModDesc);
      if Flags and pfDesignOnly = pfDesignOnly then
        Items.AddChild(ModNode, 'Design-time package');
      if Flags and pfRunOnly = pfRunOnly then
        Items.AddChild (ModNode, 'Run-time package');
    end;
  end;
  Result := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  EnumModules(EnumModuleProc, nil);
end;

end.
```

Сначала вызывается процедура `EnumModules()`. Она просматривает всю выполняемую программу и пересчитывает все обнаруженные пакеты. Процедуру `EnumModules()` обслуживает функция обратного вызова `EnumModuleProc()`. Эта функция заполняет компонент `TTreeView` информацией о каждом пакете в приложении. Большую часть кода занимает установка параметров компонента `TTreeView`. Функция `GetPackageDescription()` возвращает строку, описывающую ресурсы, содержащиеся в пакетах. Обращение к процедуре `GetPackageInfo()` расположено в функции обратного вызова `PackageInfoProc()`.

Функция `PackageInfoProc()` способна обработать *информационную таблицу* пакета (information table). Она вызывается для каждого модуля в составе пакета и для каждого пакета, обязательного для данного пакета. Здесь компонент `TTreeView` вновь заполняется информацией, полученной в результате исследования значений параметра `Flags` и параметра `NameType`. Более подробная информация по этой теме приведена в интерактивной справочной системе в разделе "*TPackageInfoProc*".

Приведенный код является модификацией листинга из превосходной книги *Марко Канту* (Marco Cantu) *Mastering Delphi 5*, которая должна быть в библиотеке каждого серьезного программиста Delphi.

Резюме

Пакет — это ключевая часть архитектуры Delphi/VCL. Научившись использовать пакеты не только для хранения компонентов, можно создавать весьма элегантные приложения, обладающие гибкой архитектурой.

ГЛАВА

15

Разработка приложений COM

В ЭТОЙ ГЛАВЕ...

Полная поддержка COM-ориентированных технологий — это одно из основных достоинств Delphi. Под термином *COM-ориентированные технологии* (COM-based) подразумевается набор разнообразных технологий, опирающихся на модель *COM* (Component Object Model — модель компонентных объектов) как на некий фундамент. В этот набор входят такие технологии, как серверы и клиенты COM, элементы управления ActiveX, *OLE* (Object Linking and Embedding — связывание и внедрение объектов), *автоматизация* (Automation) и *MTS* (Microsoft Transaction Server — сервер транзакций Microsoft). Но эти новейшие технологии могут несколько озадачить и даже обескуражить разработчика. В настоящей главе речь пойдет о технологиях, основанных на COM, ActiveX и OLE, а также об их использовании при разработке приложений. Еще несколько лет назад рассмотрение подобных тем касалось, в основном, технологии OLE, которая предоставляет метод совместного использования данных различными приложениями, главным образом, внедряя или связывая данные, ассоциированные с одним приложением, с данными, ассоциированными с другим приложением (например внедрение электронной таблицы в документ текстового процессора). Но COM — это нечто гораздо большее, чем OLE-ориентированные трюки с текстовым процессором!

В настоящей главе рассматриваются основы COM-ориентированных технологий, в частности — расширения языка Object Pascal и библиотеки VCL, вызванные необходимостью поддержки технологии COM. Здесь описано применение этих средств для управления серверами автоматизации из приложений Delphi, а также создание собственного сервера автоматизации. Кроме того, дано представление о высокоорганизованных методах автоматизации и MTS. И, наконец, описание класса VCL `TOleContainer`, который инкапсулирует контейнеры ActiveX. Необходимо отметить, что в одной главе невозможно осветить все подробности технологий ActiveX и OLE (это тема для отдельной книги или даже нескольких), тем не менее тут представлены наиболее важные свойства данных технологий и их реализация в Delphi.

Основы COM

Прежде чем приступить к изучению этой темы, читателю необходимо ознакомиться с основными концепциями и терминологией, используемой при описании технологий COM. В настоящем разделе рассматриваются основные понятия и термины, связанные с данными технологиями.

COM: Модель компонентных объектов

Модель компонентных объектов (COM — Component Object Model) представляет собой основу технологий ActiveX и OLE. COM определяет стандарты интерфейсов API и бинарные стандарты для связи объектов, не зависящих от языка программирования или платформы (теоретически). Объекты COM подобны известным уже объектам VCL, за исключением того, что они содержат специализированные методы и свойства, а не поля данных.

Объект COM имеет один или несколько *интерфейсов* (interface), которые, по сути, представляют собой таблицы функций, связанных с этим объектом. Методы интерфейса можно вызывать аналогично методам любого объекта Delphi. Более подробно интерфейсы описаны в этой настоящей далее.

Используемые объекты компонентов могут быть реализованы в любом файле `.exe` или `.dll`. При этом реализация объекта остается для пользователя объекта совершенно прозрачной благодаря предлагаемому COM сервису, называемому *маршалингом* (marshalling). Механизм маршалинга COM берет на себя всю организацию взаимного вызова функций между независимыми процессами и даже различными компьютерами, вследствие чего становится возможным использование 32-разрядных объектов в 16-разрядных приложениях, а также доступ к объекту, расположенному на компьютере А, из приложения, запущенного на компьютере Б. Такое межкомпьютерное взаимодействие называется *распределенной моделью компонентных объектов* (Distributed COM, или DCOM). Более подробное описание механизма этого взаимодействия приведено в настоящей главе далее.

COM, ActiveX или OLE?

"Так в чем же различие между COM, ActiveX и OLE?" — вот один из наиболее частых (и обоснованных) вопросов, которые задают разработчики при знакомстве с этими технологиями. Резонность такого вопроса объясняется еще и тем, что создатель данных технологий, корпорация *Microsoft*, не прилагает особых усилий, чтобы разъяснить их суть. Как уже упоминалось, COM — это стандарты интерфейсов API и бинарные стандарты, которые служат основой для построения всех остальных технологий данного семейства. В 1995 году аббревиатура OLE была общим термином, использовавшимся для описания целого набора технологий, основанных на архитектуре COM. Тогда термин "OLE" применялся только для тех технологий, которые непосредственно имели дело со связыванием и внедрением, а именно: использование контейнеров и серверов, активизация по месту вставки или внедрения, технология "перетащить и опустить" (drag-and-drop), слияние меню. В 1996 году *Microsoft* начинает агрессивную маркетинговую кампанию по внедрению в язык разработчиков термина *ActiveX*. Последний становится всеобъемлющим и используется для описания технологий, отличных от OLE, но основанных на применении COM. Технология ActiveX включает в себя автоматизацию (ранее называвшуюся *OLE-автоматизацией*), элементы управления, документы, контейнеры, сценарии и некоторые технологии Internet. Поскольку началась неразбериха, вызванная стремлением использовать термин "ActiveX" для описания всех "домашних любимцев", корпорация *Microsoft* слегка "дала задний ход" и сейчас иногда называет все технологии, отличные от OLE, но основанные на модели компонентных объектов, просто и незатейливо — *COM-ориентированные технологии* (COM-based technologies).

В компьютерной индустрии критический взгляд на творчество этой корпорации получил свое выражение в следующей фразе: мы говорим "OLE" — подразумеваем "замедление работы и увеличение размера приложений". В результате для маркетинговых решений корпорации *Microsoft* потребовалась новая терминология, предназначенная для новых интерфейсов API, которые были положены в основу будущих операционных систем и технологий Internet. Еще один забавный факт: *Microsoft* просит называть OLE не *Object Linking and Embedding*, а просто *О-Ле*!

Терминология

Технология COM принесла с собой новую терминологию. Поэтому прежде чем погружаться в глубины ActiveX и OLE, следует разобраться со значением некоторых терминов.

Экземпляр объекта COM обычно называют просто *объектом*, а тип, который идентифицирует такой объект, — *компонентным классом* (component class, или coclass). Поэтому для создания экземпляра некоторого *объекта* COM необходим идентификатор *класса* COM (CLSID).

Часть данных, которая совместно используется несколькими приложениями, называется *объектом OLE* (OLE object). Приложения, способные вмещать в себя объекты OLE, называются *контейнерами OLE* (OLE container), а приложения, способные содержать собственные данные в контейнерах OLE, — *OLE-серверами* (OLE server).

Документ, в который входит один или более объектов OLE, обычно называют *составным документом* (compound document). Хотя объекты OLE могут находиться внутри других документов, полноценные приложения, с которыми можно работать в контексте другого документа, называют *документами ActiveX* (ActiveX document).

Как следует из названия технологии, объект OLE может быть *связан* (linked) или *внедрен* (embedded) в составной документ. Связанные объекты сохраняются в отдельном файле на диске. Благодаря средствам связывания объектов несколько контейнеров — или даже приложение-сервер — могут быть связаны с одним и тем же объектом OLE, расположенным на диске. Если одно из приложений модифицирует связанный объект, внесенные изменения распространяется на все приложения, связанные с данным объектом. Внедренные объекты хранятся непосредственно в приложениях, являющихся контейнерами OLE. Только контейнерное приложение будет способно осуществлять редактирование внедренного объекта OLE. Внедрение не позволяет другим приложениям осуществлять доступ к импортированным данным (а следовательно, модифицировать или разрушать их), но это существенно усложняет управление данными, помещенными в контейнер.

Еще один аспект ActiveX, речь о котором пойдет в настоящей главе далее, называется *автоматизацией* (automation). Это средство позволяет приложениям (называемым *контроллерами автоматизации* — automation controller) управлять объектами, ассоциированными с другими приложениями или динамическими библиотеками (именуемыми *сервером автоматизации* — automation server). Автоматизация позволяет управлять объектами в другом приложении и, наоборот, — предоставлять функциональные элементы своего приложения другим приложениям.

Достоинства ActiveX

Самым замечательным свойством технологии ActiveX является простота внедрения средств управления многими типами данных в приложение. Слово “простота” может вызвать улыбку, но это правда! Например, гораздо проще наделить разрабатываемое приложение возможностью использования объектов ActiveX, чем самостоятельно создать средства, которые традиционно применяются при работе с текстовым процессором, электронными таблицами или графическими изображениями.

Технология ActiveX прекрасно вписывается в традицию Delphi многократно использовать программный код, созданный ранее. Не нужно заново писать код для

управления конкретным типом данных, если уже существует работоспособное приложение OLE-сервера. Какой бы сложной ни показалась технология OLE, она все же лучше других альтернативных решений.

Не секрет, что корпорация *Microsoft* инвестировала значительные средства в технологию ActiveX, и теперь разработчики Windows 95/98, Windows NT/2000 и последующих операционных систем данной серии вынуждены ближе познакомиться с технологией ActiveX, чтобы использовать все ее преимущества в своих приложениях. Причем, нравится это кому-то или нет, но модель COM следует воспринимать как объективную реальность и постараться сделать ее удобным помощником в разработке своих приложений.

OLE 1 против OLE 2

Одно из основных различий между объектами OLE, ассоциированными с 16-разрядными серверами OLE 1 и OLE 2, заключается в способе их активизации. Когда активизируется объект, созданный для сервера OLE 1, запускается и получает фокус ввода приложение-сервер, а объект OLE появляется в нем в готовом для редактирования виде. Когда активизируется объект OLE 2, приложение-сервер OLE 2 становится активным неявно, "внутри" приложения-контейнера. Это называется *активизацией по месту вставки* (in-place activation), или *визуальным редактированием* (visual editing).

При активизации объекта OLE 2 меню и панели инструментов приложения-сервера заменяются или сливаются с соответствующими элементами приложения-клиента, а часть окна приложения-клиента, по сути, становится окном приложения-сервера. Этот процесс демонстрируется на конкретном примере, приведенном в настоящей главе далее.

Структурированное хранилище

Стандарт OLE 2 определяет схему хранения информации на диске, известную как *структурированное хранилище* (structured storage). Данная схема предполагает реализацию на файловом уровне тех функций, которые в среде DOS обеспечивались на уровне диска. Структурированное хранилище представляет собой один физический файл на диске, внутреннее строение которого подобно каталогу DOS, и состоит из множества других хранилищ (эквивалентных подкаталогам) и потоков (эквивалентных файлам DOS). Иногда структурированные хранилища называют *составными файлами* (compound files).

Единообразная передача данных

В OLE 2 реализована концепция *объекта данных* (data object), который является базовым объектом, используемым для обмена данными с соблюдением некоторых правил единообразной передачи информации. Принципы *единообразной передачи данных* (UDT — Uniform Data Transfer) определяют передачу данных через *буфер обмена* (clipboard) и реализованы в механизмах "перетащить и опустить", DDE и OLE. Объекты данных предоставляют бо́льшие возможности описания типов содержащихся в них данных, по сравнению с теми, что имелись прежде. На практике недостатки прежних объектов выражались в ограничениях, свойственных существовавшим ранее средст-

вам передачи данных. Фактически технология UDT предназначена для замены технологии DDE. В процессе работы объект данных может учитывать значения собственных свойств, таких как размер, цвет и даже тип устройства, в котором он отображается. Попробуйте-ка реализовать вышесказанное в буфере обмена Windows!

Потоковые модели

Каждый объект COM работает с определенной потоковой моделью, определяющей, каким образом объект может функционировать в многопоточной среде. После регистрации сервера COM в системе каждый из объектов COM, содержащихся на этом сервере, должен зарегистрировать поддерживаемую им потоковую модель. Для объектов COM, созданных в Delphi, потоковая модель выбирается при выполнении последовательности действий с помощью мастеров создания объектов автоматизации, элементов управления ActiveX или объектов COM. Тем самым определяется, как будет зарегистрирован элемент управления. Существуют следующие потоковые модели COM.

- *Однопоточная* (single). Весь сервер COM выполняется в одном потоке.
- *Раздельная* (apartment). Иначе эту модель называют *однопоточно-раздельной* (STA — Single-Threaded Apartment). Каждый объект COM выполняется в контексте собственного потока, а несколько экземпляров объекта COM одинакового типа могут выполняться в отдельных потоках. В результате любые данные, используемые совместно всеми экземплярами объектов (например глобальные переменные), должны быть при необходимости защищены объектами синхронизации потоков.
- *Свободная* (free). Иначе эту модель называют *многопоточно-раздельной* (MTA — Multithreading Apartment). Клиент может вызвать метод объекта в любом потоке и в любое время. В этом случае объект COM должен защищать даже свои собственные данные (а не только глобальные) от одновременного доступа из нескольких потоков.
- *Обе модели*. Поддерживаются обе потоковые модели ("раздельная" и "свободная").

Следует иметь в виду, что выбор желаемой потоковой модели в диалоговом окне мастера не гарантирует, что объект COM будет защищен надлежащим образом в соответствии с выбранной потоковой моделью. Для поддержки определенной потоковой модели необходимо написать код, гарантирующий, что сервер COM будет работать с выбранной моделью правильно. При этом практически всегда нужно использовать объекты синхронизации потоков для защиты доступа к глобальным данным или данным экземпляров в объектах COM. Более подробная информация о синхронизации потоков приведена в главе 5, "Создание многопоточных приложений".

COM+

Корпорация *Microsoft* реализовала свое самое значительное за последнее время обновление технологии COM в виде составной части версии Windows 2000, которую и назвала *COM+*. Цель выпуска стандарта COM+ состоит в упрощении процесса разработки приложений COM на основе интеграции нескольких технологий-сателлитов, из которых самыми значительными являются *сервер транзакций Microsoft* (MTS — Microsoft

Transaction Server) (более подробная информация по этой теме приведена в настоящей главе далее) и *очередь сообщений Microsoft* (MSMQ — Microsoft Message Queue). Интеграция этих технологий со стандартными динамическими средствами COM+ означает, что все разработчики COM+ смогут воспользоваться преимуществами таких средств, как управление транзакциями, поддержка безопасности, удаленное администрирование, обслуживание очереди компонентов, а также сервис, связанный с публикацией и подпиской на события. Поскольку модель COM+ состоит в основном из имеющихся в наличии частей, то это означает полную обратную совместимость, в результате которой все существующие приложения COM и MTS автоматически становятся приложениями COM+. Более подробная информация о технологиях COM+ и MTS приведена в главе 18, "Транзакционные методы разработки с применением COM+ и MTS".

COM и Object Pascal

После краткого обзора основных концепций и терминологии технологий COM, ActiveX и OLE можно переходить к рассмотрению способов реализации этих концепций в Delphi. В настоящем разделе более детально рассматривается как сама технология COM, так и ее согласование с языком Object Pascal и библиотекой VCL.

Интерфейсы

COM определяет стандарты для расположения в памяти функций объектов. Функции располагаются в *виртуальных таблицах* (virtual tables — vtables), т.е. таблицах адресов функций, аналогичных *таблицам виртуальных методов* (VMT — Virtual Method Table) классов в Delphi. Описание каждой виртуальной таблицы на языке программирования называется *интерфейсом* (interface).

Рассмотрим интерфейс с точки зрения отдельного класса. Каждый его элемент можно представить как специфический набор функций или процедур, предназначенных для манипулирования классом. Например, объект COM, представляющий собой растровое изображение, может поддерживать два интерфейса: один, содержащий методы, которые позволяют воспроизводить рисунок на экране монитора и распечатывать его, а другой, управляющий записью и считыванием рисунка в файл или из файла на диске.

По сути, любой интерфейс состоит из двух частей. Первая — это определение интерфейса, которое включает в себя коллекцию, состоящую из одного или нескольких объявлений функций, расположенных в определенном порядке. Определение интерфейса используется как самим объектом, так и пользователем объекта. Вторая часть — реализация интерфейса, представляющая собой практическое воплощение функций, описанных в его определении. Определение интерфейса подобно соглашению, заключенному между объектом COM и использующим его клиентом — оно гарантирует клиенту, что данный объект реализует определенные методы, выстроенные в определенном порядке.

Введенное в Delphi 3 ключевое слово `interface` языка Object Pascal позволяет достаточно просто определять интерфейсы COM. Объявление интерфейсов семантически подобно объявлению классов, за исключением того, что интерфейсы могут содержать только свойства и методы, но не данные. Поскольку интерфейсы не могут содержать данные, их свойства должны записываться и считываться только с помо-

щью методов. Однако самое важное то, что интерфейсы не имеют раздела реализации, поскольку они *лишь определяют* соглашение.

Интерфейс IUnknown

Подобно тому, как все классы Object Pascal в конечном счете являются потомками класса `TObject`, все интерфейсы COM (а следовательно, и все интерфейсы Object Pascal) происходят от интерфейса `IUnknown`. Интерфейс `IUnknown` определен в модуле `System` следующим образом:

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
                            out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

Как видно из приведенного фрагмента кода, помимо ключевого слова `interface`, между объявлениями интерфейса и класса существует еще одно существенное различие, заключающееся в присутствии *глобального уникального идентификатора* (GUID — Globally Unique Identifier), используемого в технологии COM.

СОВЕТ

Чтобы создать в IDE Delphi новый GUID, достаточно нажать в окне редактора кода комбинацию клавиш <Ctrl+Shift+G>.

Глобальный уникальный идентификатор (GUID)

GUID (произносится *гу-ид* (*goo-id*)) представляет собой 128-разрядное целое число, используемое в технологии COM для уникальной идентификации интерфейсов, компонентных классов и других объектов. GUID практически гарантирует настоящую глобальную уникальность благодаря использованию достаточно больших чисел и превосходного алгоритма их генерации. GUID создается с помощью функции API `CoCreateGUID()`, а алгоритм его генерации основан на комбинации следующей информации: текущая дата и время, частота процессора, номер сетевой карты, остаток на банковском счете Билла Гейтса (ну, хорошо, со счетом мы несколько погорячились). Если на компьютере установлена сетевая карта, созданный на этом компьютере GUID будет действительно уникальным, поскольку уникальность каждой сетевой карты гарантируется встроенным в нее глобальным идентификатором (ID). Если же на компьютере нет сетевой карты, ее номер можно заменить другим, синтезировав его с помощью параметров другого установленного в компьютере оборудования.

Поскольку не существует типа данных, позволяющего хранить число из 128 двоичных разрядов, для представления GUID используется запись с типом `TGUID`, которая определена в модуле `System` следующим образом:

```
type
  PGUID = ^TGUID;
  TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
```

```
    D4: array[0..7] of Byte;
  end;
```

Из-за того что присваивать переменным и константам значения GUID в таком формате записи достаточно сложно, Object Pascal позволяет также определить запись `TGUID` как строку следующего формата:

```
'{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}'
```

Благодаря этому следующие объявления эквивалентны:

```
MyGuid: TGUID = (
D1:$12345678;D2:$1234;D3:$1234;D4:($01,$02,$03,$04,$05,$06,$07,$08));

MyGuid: TGUID = '{12345678-1234-1234-12345678}';
```

В COM каждый интерфейс или класс имеет соответствующий GUID, который является уникальным определителем интерфейса. В этом случае два интерфейса или класса, имеющих одинаковые имена и созданных двумя независимыми разработчиками, никогда не будут конфликтовать, потому что соответствующие им GUID всегда будут различны. При определении интерфейса GUID обычно называют *идентификатором интерфейса* (IID — Interface ID), а при определении класса его называют *идентификатором класса* (CLSID — Class ID).

Помимо своего идентификатора IID, в интерфейсе `IUnknown` объявлены три метода: `QueryInterface()`, `_AddRef()` и `_Release()`. В следствии того, что интерфейс `IUnknown` является базовым интерфейсом COM, все остальные интерфейсы неизбежно будут реализовать интерфейс `IUnknown` и его методы. Метод `_AddRef()` следует вызывать в том случае, если клиент получает и желает использовать указатель на данный интерфейс. Вызов этого метода должен сопровождаться вызовом метода `_Release()`, когда клиент заканчивает работу с интерфейсом. В таком случае объект, реализующий данный интерфейс, сможет поддерживать счетчик клиентов, хранящих ссылку на этот объект, или вести *счетчик ссылок* (reference count). Когда количество ссылок станет равным нулю, объект должен будет выгрузить себя самого из памяти. Функция `QueryInterface()` используется для выполнения запроса о том, поддерживает ли данный объект некоторый интерфейс. Если требуемый интерфейс поддерживается, то клиенту возвращается указатель на него. Предположим, что объект `O` поддерживает интерфейсы `I1` и `I2`, и клиент уже имеет указатель на интерфейс `I1` объекта `O`. Для получения от объекта `O` указателя на его интерфейс `I2` необходимо вызвать метод `I1.QueryInterface()`.

НА ЗАМЕТКУ

Опытный разработчик COM может заметить, что символ подчеркивания перед методами `_AddRef()` и `_Release()` не используется в других языках программирования или даже в документации *Microsoft* по COM. Поскольку Object Pascal "знает" интерфейс `IUnknown`, данные методы нельзя вызывать напрямую (об этом чуть позднее), поэтому символ подчеркивания существует главным образом для того, чтобы обратить внимание разработчика и заставить его задуматься, прежде чем выполнять вызов таких методов.

Поскольку каждый интерфейс в Delphi косвенно происходит от интерфейса `IUnknown`, каждый класс Delphi, реализующий интерфейсы, также будет поддерживать

эти три метода интерфейса IUnknown. Такую "грязную работу" можно выполнить вручную или же поручить ее подпрограммам библиотеки VCL, сделав свой класс потомком класса TInterfacedObject, в котором интерфейс IUnknown уже реализован.

Использование интерфейсов

В главе 2, "Язык программирования Object Pascal", и в документации Delphi описывается семантика использования интерфейсов, поэтому приводить ее здесь не имеет смысла. Вместо этого рассмотрим, как интерфейс IUnknown косвенно интегрирован с языком Object Pascal.

При присваивании значения переменной интерфейса компилятор автоматически создает вызов метода интерфейса _AddRef(), чтобы увеличить содержимое счетчика ссылок. Когда переменная интерфейса выходит за пределы области видимости или принимает значение nil, компилятор автоматически создает вызов метода интерфейса _Release(). Рассмотрим такой фрагмент кода:

```
var
  I: ISomeInteface; // Некоторый интерфейс
begin
  // Эта функция возвращает интерфейс
  I := FunctionThatReturnsAnInterface;
  I.SomeMethod;     // Вызов некоторого метода интерфейса
end;
```

А сейчас обратите внимание на следующий код. Здесь код, вводимый разработчиком, выделен полужирным шрифтом, а код, созданный компилятором, обычным.

```
var
    I: ISomeInterface;
begin
  // Интерфейс автоматически инициализируется равным nil
  I := nil;
  try
    // Ваш код должен быть здесь
    I := FunctionThatReturnsAnInterface;
    // Метод _AddRef() вызывается косвенно при
    // присваивании значения переменной I
    I._AddRef;
    I.SomeMethod;
  finally
    // Блок завершения гарантирует, что ссылки
    // на интерфейс будут удалены.
    if I <> nil I._Release;
  end;
end;
```

Компилятор Delphi также достаточно интеллектуален и "знает", когда вызывать методы _AddRef() и _Release(). Это делается при переназначении интерфейсов экземплярам другого интерфейса или при присваивании интерфейсам значения nil. Рассмотрим, к примеру, следующий фрагмент кода:

```
var
  I: ISomeInteface;                    // Некоторый интерфейс
```

```
begin
  // Присвоить I
  I := FunctionThatReturnsAnInterface;  // Получить интерфейс
  I.SomeMethod;                         // Вызов метода интерфейса
  // Переприсвоить I
  I := OtherFunctionThatReturnsAnInterface;  // Другой интерфейс
  I.OtherMethod;               // Вызов метода другого интерфейса
  // Установить I в nil
  I := nil;
end;
```

Обратите внимание на комбинацию, состоящую из кода, написанного разработчиком (полужирный шрифт), и кода, созданного компилятором (обычный шрифт):

```
var
  I: ISomeInterface;
begin
  // Интерфейс автоматически инициализируется равным nil
  I := nil;
  try
    // Ваш код должен быть здесь
    // Присвоить I
    I := FunctionThatReturnsAnInterface;
    // Метод _AddRef() вызывается косвенно при
    // присваивании I значения
    I._AddRef;
    I.SomeMethod;
    // Переприсвоить I
    I._Release;
    I := OtherFunctionThatReturnsAnInterface;
    I._AddRef;
    I.OtherMethod;
    // Установить I в nil
    I._Release;
    I := nil;
  finally
    // Блок завершения гарантирует, что ссылки
    // на интерфейс будут удалены.
    if I <> nil I._Release;
  end;
end;
```

Этот пример помогает понять, из-за чего в Delphi используется символ подчеркивания в методах `_AddRef()` и `_Release()`. Об увеличении или уменьшении количества ссылок интерфейса часто забывали, и это являлось классической ошибкой при программировании с применением технологии COM в те времена, когда не существовало ключевого слова `interface`. Интерфейсы Delphi как раз и призваны помочь разработчикам избавиться от такого рода проблем с помощью косвенного вызова этих методов.

Поскольку компилятор "знает", как и когда генерировать вызовы методов `_AddRef()` и `_Release()`, резонно предположить, что он "знает" и о третьем методе интерфейса `IUnknown`, `QueryInterface()`. Конечно же, это так. Получив указатель

на интерфейс от некоторого объекта, можно использовать оператор as для "приведения" его к типу другого интерфейса, который поддерживается объектом COM. Понятие "приведение типа" наилучшим образом подходит для описания данного процесса, хотя такое применение оператора as является на самом деле не приведением типа в буквальном смысле этого выражения, а внутренним обращением к методу QueryInterface(). Вот как выглядит демонстрация описанного процесса:

```
var
  I1: ISomeInterface;         // Некоторый интерфейс
  I2: ISomeOtherInterface;    // Другой интерфейс
begin
  // Присвоить I1
  I1 := FunctionThatReturnsAnInterface; // Функция, возвращающая
                                        // интерфейс.
  // Метод QueryInterface I1 для интерфейса I2
  I2 := I1 as ISomeOtherInterface;      // Приведение к типу
                                        // другого интерфейса.
end;
```

Если объект, на который ссылается интерфейс I1, не поддерживает интерфейс ISomeOtherInterface, то оператор as породит исключение.

Одно дополнительное языковое правило, касающееся интерфейсов, заключается в следующем: переменные интерфейса совместимы по присвоению с классом Object Pascal, который реализует этот интерфейс. Например, рассмотрим следующие объявления интерфейса и класса:

```
type
  IFoo = interface
    // Определение IFoo
  end;

  IBar = interface(IFoo)
    // Определение IBar
  end;

  TBarClass = class(TObject, IBar)
    // Определение TBarClass
  end;
```

При таких объявлениях следующий код является вполне корректным:

```
var
  IB: IBar;
  TB: TBarClass;
begin
  TB := TBarClass.Create;
  try
    // Получить указатель на интерфейс IBar объекта TB:
    IB := TB;
    // Использование объекта TB и интерфейса IB
  finally
    IB := nil;  // Явное освобождение интерфейса IB
    TB.Free;
```

```
  end;
end;
```

Несмотря на то что такой подход, казалось бы, нарушает традиционные правила языка Object Pascal, связанные с совместимостью при выполнении операций присвоения, он делает интерфейсы более естественными и простыми в работе.

Существует важное, но не вполне очевидное следствие из этого правила — интерфейсы совместимы по присвоению только с теми классами, которые явно поддерживают данный интерфейс. Например, класс `TBarClass`, определенный выше, объявляет явную поддержку интерфейса `IBar`. Поскольку интерфейс `IBar` происходит от интерфейса `IFoo`, резонно было бы предположить, что интерфейс `TBarClass` также поддерживает интерфейс `IFoo`. Однако это вовсе не так, что и проиллюстрировано в приведенном ниже фрагменте.

```
var
  IF: IFoo;
  TB: TBarClass;
begin
  TB := TBarClass.Create;
  try
    // В следующей строке находится ошибка времени компиляции,
    // поскольку класс TBarClass не поддерживает
    // интерфейс IFoo явно
    IF := TB;
    // Использование объекта TB и интерфейса IF
  finally
    IF := nil;  // Явное освобождение интерфейса IF
    TB.Free;
  end;
end;
```

Интерфейсы и идентификаторы интерфейсов

Поскольку идентификатор интерфейса (ID) описывается как часть объявления интерфейса, компилятор Object Pascal знает о том, как получить его. Следовательно, можно передать тип интерфейса процедуре или функции, которой необходимы параметры типа `TIID` или `TGUID`. Предположим, существует функция, подобная следующей:

```
procedure TakesIID(const IID: TIID);
```

В этом случае приведенная ниже строка кода синтаксически правильна:

```
TakesIID(IUnknown);
```

Такая возможность предотвращает необходимость использования констант `IID_`*`ТипИнтерфейса`*, определенных для каждого типа интерфейса. Эти константы хорошо знакомы всем, кому приходилось иметь дело с разработкой или использованием объектов COM в языке C++.

Псевдоним метода

Иногда при реализации нескольких интерфейсов в одном классе появляется еще одна проблема, заключающаяся в конфликте имен методов в одном или нескольких интерфейсах. Рассмотрим следующий фрагмент:

```
type
  IIntf1 = interface
    procedure AProc;
  end;

  IIntf2 = interface
    procedure AProc;
  end;
```

Каждый из описываемых интерфейсов содержит метод `AProc()`. Как объявить в таком случае класс реализующий оба интерфейса? Оказывается, для этого можно использовать *псевдоним метода* (method aliasing). С помощью псевдонима метод интерфейса в классе можно поставить в соответствие методу с другим именем. Рассмотрим фрагмент, демонстрирующий объявление класса, в котором реализованы интерфейсы `IIntf1` и `IIntf2`:

```
type
  TNewClass = class(TInterfacedObject, IIntf1, IIntf2)
  protected
    procedure IIntf2.AProc = AProc2;
    procedure AProc;   // Связывает с IIntf1.AProc
    procedure AProc2;  // Связывает с IIntf2.AProc
  end;
```

В данном объявлении метод `AProc()` интерфейса `IIntf2` устанавливается в соответствие методу по имени `AProc2()`. Создание псевдонима в этом случае позволяет реализовать любой интерфейс в любом классе, избежав конфликта имен методов.

Тип возвращаемого значения HResult

Как можно заметить, метод `QueryInterface()` интерфейса `IUnknown` возвращает результат типа `HResult`. Это наиболее популярный тип значений, возвращаемых многими методами различных интерфейсов ActiveX и OLE, а также функциями API COM. Тип `HResult` определен в модуле `System` как тип `LongWord`. Возможные значения типа `HResult` перечислены в модуле `Windows` (если есть исходный код библиотеки VCL, то описания его значений можно найти под заголовком `{ HRESULT value definitions }`). Значение `S_OK` или `NOERROR (0)` типа `HResult` говорит об успешном выполнении. Если же старший бит значения типа `HResult` установлен равным единице, значит, при выполнении произошла ошибка. В модуле `Windows` есть две функции — `Succeeded()` и `Failed()`, — которые принимают значение типа `HResult` в качестве параметра и возвращают значение типа `BOOL`, означающее успешное выполнение или наличие ошибки. Синтаксис вызова этих методов имеет следующий вид:

```
if Succeeded(FunctionThatReturnsHResult) then
  \\ Нормальное продолжение
```

```
if Failed(FunctionThatReturnsHResult) then
  \\ Код обработки ошибки
```

Естественно, проверка значения, возвращаемого при каждом отдельном вызове функции, — занятие скучное. Кроме того, работа над ошибками, возвращаемыми функциями, относится к "компетенции" методов обработки исключений Delphi. А потому в модуле `ComObj` определена процедура `OleCheck()`, с помощью которой ошибки типа `HResult` преобразуются в исключения. Синтаксис вызова данного метода имеет следующий вид:

```
OleCheck(FunctionThatReturnsHResult);
```

Эта процедура удобна в применении, к тому же значительно упрощает код программы.

Объекты COM и фабрики классов

Кроме поддержки одного или нескольких интерфейсов, которые происходят от интерфейса `IUnknown`, и реализации счетчика ссылок для отслеживания своего срока существования, объекты COM имеют еще одну специфическую особенность: они создаются специальными объектами, называемыми *фабриками классов* (class factories). Каждый класс COM имеет соответствующую фабрику класса, которая отвечает за создание экземпляров объектов данного класса COM. Фабрика класса — это специальный объект COM, поддерживающий интерфейс `IClassFactory`. Данный интерфейс определен в модуле `ActiveX` следующим образом:

```
type
  IClassFactory = interface(IUnknown)
    ['{00000001-0000-0000-C000-000000000046}']
    function CreateInstance(const unkOuter: IUnknown;
                      const iid: TIID; out obj): HResult; stdcall;
    function LockServer(fLock: BOOL): HResult; stdcall;
  end;
```

Метод `CreateInstance()` вызывается для создания экземпляра объекта COM, связанного с данной фабрикой класса. Параметр `unkOuter` этого метода указывает на управляющий интерфейс `IUnknown`, если объект создается в качестве некоего агрегата (понятие агрегирования, или сборки разъясняется в настоящей главе далее). Параметр `iid` содержит идентификатор интерфейса (IID), с помощью которого можно управлять объектом. Параметр `obj` должен содержать указатель на интерфейс, определенный параметром `iid`.

Метод `LockServer()` вызывается для хранения сервера COM в памяти даже в том случае, если на сервер не ссылается ни один клиент. Если параметр `fLock` равен `True`, то счетчик блокировок сервера увеличивается на единицу. Если же параметр `fLock` равен `False`, то счетчик блокировок сервера уменьшается на единицу. Если в результате счетчик ссылок сервера окажется равен `0` (означая отсутствие использующих его клиентов), то данный сервер COM выгружается из памяти.

Классы TComObject и TComObjectFactory

В Delphi существует два класса, инкапсулирующие объекты COM и фабрики классов: `TComObject` и `TComObjectFactory`. Класс `TComObject` содержит инфраструктуру необходимую для поддержки интерфейса `IUnknown` и создания объектов с помощью класса `TComObjectFactory`. Подобным же образом класс `TComObjectFactory` поддерживает интерфейс `IClassFactory` и "умеет" создавать объекты класса `TComObject`. Проще всего создать объект COM с помощью мастера COM Object Wizard, пиктограмма которого расположена во вкладке ActiveX диалогового окна New Items. В листинге 15.1 приведен псевдокод модуля, созданного этим мастером. Данный псевдокод иллюстрирует отношения между упомянутыми классами.

Листинг 15.1. Псевдокод модуля сервера COM

```
unit ComDemo;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj;
type
  TSomeComObject = class(TComObject, поддерживаемые интерфейсы)
    Здесь объявляются методы класса и интерфейса
  end;

const
  Class_SomeObject: TGUID =
                    '{CB11BA07-735D-4937-885A-1CFB5312AEC8}';

implementation

uses ComServ;

Здесь находится реализация объекта TSomeComObject

initialization
  TComObjectFactory.Create(ComServer,
                           TSomeObject, Class_SomeObject,
                           'SomeObject', 'The SomeObject class',
                           ciMultiInstance, tmApartment);
end;
```

Класс, производный от класса `TComServer`, объявляется и реализуется подобно большинству других классов VCL. Параметры, переданные конструктору `Create()` класса `TComObjectFactory`, связывают этот потомок класса `TComServer` с соответствующим объектом `TComObjectFactory`. Первый параметр конструктора — это объект `TComServer`. В качестве такого параметра практически всегда передается гло-

бальный объект ComServer, объявленный в модуле ComServ. Вторым параметром является класс TComObject, который необходимо связать с фабрикой класса, а третий параметр — это идентификатор CLSID класса COM TComObject. Четвертый и пятый параметры — это строки имени и описания класса, используемые для регистрации класса COM в системном реестре. Шестой параметр указывает экземпляр объекта COM и, наконец, последний параметр указывает потоковую модель объекта.

Экземпляр класса TComObjectFactory создается в разделе инициализации модуля; в этом случае фабрика класса обязательно будет доступна для создания экземпляров объекта COM сразу после загрузки сервера COM. Выполнение загрузки сервера COM зависит от его типа, т.е., от того, является ли данный сервер *внутренним* (DLL) или *внешним* (приложение).

Внутренние серверы COM

Внутренний сервер (in-process server) COM представляет собой библиотеку DLL, которая может создавать объекты COM, предназначенные для использования в главном приложении. Этот тип сервера COM называется внутренним, потому что он, как и библиотека DLL, относится к тому же процессу, что и основное приложение. Внутренний сервер должен экспортировать (предоставлять) четыре стандартные функции:

```
function DllRegisterServer: HResult; stdcall;
function DllUnregisterServer: HResult; stdcall;
function DllGetClassObject (const CLSID,
                            IID: TGUID; var Obj): HResult; stdcall;
function DllCanUnloadNow: HResult; stdcall;
```

Каждая из таких функций уже реализована в модуле ComServ, поэтому остается лишь убедиться в том, что данные функции добавлены в раздел exports проекта.

НА ЗАМЕТКУ

Хороший пример применения внутренних серверов COM можно найти в главе 16, "Программирование для оболочки Windows". Этот пример демонстрирует создание расширений оболочки.

Функция DllRegisterServer()

Функция DllRegisterServer() вызывается для регистрации библиотеки DLL сервера COM в системном реестре. Если просто экспортировать данный метод из приложения Delphi, как описывалось выше, то подпрограммы библиотеки VCL последовательно опросят все объекты COM этого приложения и зарегистрирует их в системном реестре. Для каждого класса COM при регистрации сервера COM в системном реестре создается раздел в следующей ветви:

```
HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}
```

Здесь символы *x...x* заменяют реальный идентификатор CLSID этого класса COM. Для внутренних серверов создается также подраздел InProcServer32. Параметром по умолчанию в таком подразделе является полный путь к библиотеке DLL внутреннего сервера. На рис. 15.1 показан пример сервера COM, зарегистрированного в системном реестре.

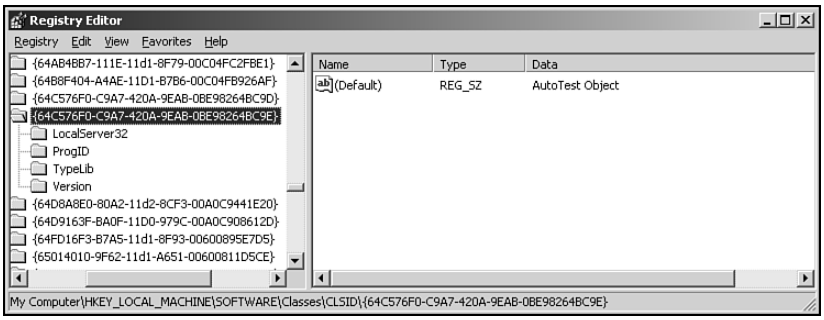

Рис. 15.1. *Сервер COM в окне редактора системного реестра*

Функция DllUnregsterServer()

Данная функция предназначена для отмены действий, выполненных функцией `DllRegisterServer()`. При вызове эта функция удаляет все разделы, подразделы и параметры, созданные функцией `DllRegisterServer()` в системном реестре.

Функция DllGetClassObject()

Функция `DllGetClassObject()` вызывается механизмом COM для получения фабрики конкретного класса COM. Параметр `CLSID` этого метода — не что иное, как идентификатор CLSID типа создаваемого класса COM. Параметр `IID` содержит значение идентификатора интерфейса (IID) — указателя на экземпляр, который необходимо получить для объекта фабрики классов (обычно здесь передается идентификатор интерфейса `IClassFactory`). При корректном завершении функции параметр `Obj` содержит указатель на интерфейс фабрики классов, обозначенный параметром `IID` и способный создавать объекты COM с типом класса, определенным параметром `CLSID`.

Функция DllCanUnloadNow()

Данная функция вызывается механизмом COM для определения возможности выгрузки из памяти библиотеки DLL данного сервера COM. Если внутри этой библиотеки DLL существуют указатели на какой-либо объект COM, то функция возвращает значение `S_FALSE`, указывающее на то, что библиотеку выгрузить нельзя. Если же в библиотеке DLL не используется в данный момент ни один из объектов COM, то функция возвращает значение `S_TRUE`.

СОВЕТ

Даже если все ссылки на объекты COM внутреннего сервера уже освобождены, иногда не нужно вызывать функцию `DllCanUnloadNow()` для инициирования процесса выгрузки из памяти библиотеки DLL внутреннего сервера. Если желательно гарантированно выгрузить все неиспользуемые библиотеки DLL сервера COM из памяти, вызовите функцию API COM `CoFreeUnusedLibraries()`, которая определена в модуле `ActiveX` следующим образом:

```
procedure CoFreeUnusedLibraries; stdcall;
```

Создание экземпляра внутреннего сервера COM

Для создания экземпляра сервера COM в среде Delphi необходимо использовать функцию `CreateComObject()`, определенную в модуле `ComObj` следующим образом:

```
function CreateComObject(const ClassID: TGUID): IUnknown;
```

Параметр `ClassID` содержит идентификатор CLSID, который определяет тип создаваемого объекта COM. Значение, возвращаемое этой функцией, — интерфейс `IUnknown` требуемого объекта COM. Если объект COM не может быть создан, то возникает исключение.

Функция `CreateComObject()` инкапсулирует функцию API COM `CoCreateInstance()`. Внутри функции `CoCreateInstance()`, для получения интерфейса `IClassFactory` заданного объекта COM, вызывается функция API COM `CoGetClassObject()`. Функция `CoCreateInstance()` выполняет этот вызов, осуществив поиск в системном реестре параметра `InProcServer32` заданного класса COM для определения пути к библиотеке DLL внутреннего сервера, вызова функции `LoadLibrary()` для загрузки этой библиотеки DLL и, наконец, вызова принадлежащей данной библиотеке DLL функции `DllGetClassObject()`. После получения указателя на интерфейс `IClassFactory`, для создания экземпляра объекта заданного класса COM функция `CoCreateInstance()` вызывает функцию `IClassFactory.CreateInstance()`.

СОВЕТ

Чтобы создать с помощью фабрики классов несколько объектов одного типа использовать функцию `CreateComObject()` неэффективно. Это объясняется тем, что после создания заданного объекта COM данная функция освобождает указатель на интерфейс `IClassFactory`, полученный от функции `CoGetClassObject()`. Если необходимо создать несколько экземпляров одного и того же объекта COM, вызовите непосредственно функцию `CoGetClassObject()`, а затем многократно обращайтесь к функции `IClassFactory.CreateInstance()` для создания необходимого количества объектов.

НА ЗАМЕТКУ

Перед использованием любой функции COM или API OLE нужно инициализировать библиотеку COM, вызвав функцию `CoInitialize()` с единственным параметром `nil`. Для правильного отключения библиотеки COM следует вызвать функцию `CoUninitialize()` (это же справедливо и в отношении библиотеки OLE). Данные вызовы подсчитываются системой, поэтому каждый вызов функции `CoInitialize()` в приложении должен сопровождаться вызовом функции `CoUninitialize()`.

В приложениях Delphi функция `CoInitialize()` вызывается автоматически из метода `Application.Initialize()`, а функция `CoUninitialize()` — при завершении работы модуля `ComObj`.

Нет необходимости вызывать эти функции из библиотек внутренних серверов, поскольку выполнение процедур инициализации и завершения возлагается на приложения-клиенты.

Внешние серверы COM

Внешние серверы (out-of-process server) COM представляют собой исполняемые файлы, которые могут создавать объекты COM для использования в других приложениях. Серверы этого типа называются внешними, поскольку они выполняются вне процесса клиента, а в контексте их собственного процесса.

Регистрация

Внешние серверы, как и их внутренние "родственники", должны быть зарегистрированы в системном реестре. Для каждого внешнего сервера должен быть создан раздел следующего вида:

```
HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}
```

Полный путь к исполняемому файлу внешнего сервера должен быть описан в параметре `LocalServer32` этого раздела.

Серверы COM приложений Delphi регистрируются в методе `Application.Initialize()`, который обычно вызывается в первой строке кода файла проекта приложения. Если при запуске приложения ему передается параметр командной строки `/regserver`, то метод `Application.Initialize()` зарегистрирует классы COM в системном реестре и немедленно прекратит выполнение приложения. Точно так же, если при запуске приложения указывается параметр командной строки `/unregserver`, то метод `Application.Initialize()` удалит описание классов COM из системного реестра и немедленно прекратит выполнение приложения. Если в командной строке не использован ни один из упомянутых параметров, то метод `Application.Initialize()` зарегистрирует классы COM в системном реестре и продолжит выполнение приложения в обычном режиме.

Создание экземпляра внешнего сервера COM

В первом приближении процесс создания экземпляров объектов COM из внешнего сервера напоминает аналогичную процедуру для внутреннего сервера — достаточно просто вызвать функцию `CreateComObject()`, определенную в модуле `ComObj`. Однако, по сути, эти процессы абсолютно не похожи. В данном случае функция `CoGetClassObject()` просматривает системный реестр в поисках параметра `LocalServer32` и вызывает соответствующее приложение, используя функцию API Win32 `CreateProcess()`. При запуске внешнего приложения-сервера последнее должно зарегистрировать свои фабрики классов, используя функцию API COM `CoRegisterClassObject()`. Данная функция добавляет указатель на интерфейс `IClassFactory` во внутреннюю таблицу зарегистрированных активных объектов классов COM. В результате функция `CoGetClassObject()` получит возможность выбрать из этой таблицы необходимый указатель `IClassFactory` для создания требуемого экземпляра объекта COM.

Агрегация

Итак, основными строительными блоками модели COM являются интерфейсы. Кроме того, возможно наследование интерфейсов. Однако интерфейсы представляют собой сущности, не имеющие реализации. Что же произойдет, если необходимо

организовать реализацию одного объекта COM внутри другого? Для ответа на данный вопрос в COM существует концепция *агрегации* (aggregation). Под агрегацией подразумевается, что содержащий (внешний) объект создает содержащийся (внутренний) объект как часть процесса своего создания, при этом интерфейсы внутреннего объекта предоставляются клиентам внешним объектом. Объект должен позволить работать с собой как с неким агрегатом, предоставив средства перенаправления всех вызовов его методов `IUnknown` содержащему его внешнему объекту. Примером агрегации объектов COM в контексте библиотеки VCL может служить класс `TaggregatedObject` в модуле `AxCtrls`.

Распределенная модель COM

Предложенная в Windows NT 4 модель *распределенной COM* (DCOM — Distributed COM) предоставляет средства доступа к объектам COM, расположенным на других компьютерах в сети. Кроме создания удаленных объектов, модель DCOM предоставляет также простые средства обеспечения безопасности, позволяющие серверам определять, какие клиенты имеют право создавать экземпляры серверов и какие операции они могут выполнять. Операционные системы Windows NT 4 и Windows 98 содержат встроенную поддержку модели DCOM, а для Windows 95 необходимо загрузить с Web-сайта корпорации *Microsoft* (`http://www.microsoft.com`) специальное дополнение, которое будет выполнять функции клиента DCOM.

Для создания удаленного объекта COM следует использовать функцию `CreateRemoteComObject()`, объявленную в модуле `ComObj` следующим образом:

```
function CreateRemoteComObject(const MachineName: WideString;
                               const ClassID: TGUID): IUnknown;
```

Первый параметр этой функции, `MachineName`, является строкой, определяющей сетевое имя компьютера, содержащего необходимый класс COM. Параметр `ClassID` задает идентификатор CLSID создаваемого объекта COM. Значение, возвращаемое этой функцией, представляет собой указатель на интерфейс `IUnknown` объекта COM, определенного параметром `ClassID`. Если создать объект невозможно, то передается исключение.

Функция `CreateRemoteComObject()` инкапсулирует функцию API COM `CoCreateInstanceEx()`, которая является расширенной версией функции `CoCreateInstance()` и предназначена для создания удаленных объектов.

Автоматизация

Автоматизация (automation), ранее известная как *OLE-автоматизация* (OLE automation), позволяет приложениям или библиотекам DLL предоставлять свои программируемые объекты для использования их другими приложениями. Приложения или библиотеки DLL, которые предоставляют свои программируемые объекты, называются *серверами автоматизации* (automation server). Приложения, которые получают доступ к управлению программируемыми объектами, содержащимися в серверах автоматизации, называются *контроллерами автоматизации* (automation controller). Контроллеры автоматизации способны программировать сервер автоматизации с помощью специального макроязыка, предлагаемого сервером.

Одно из основных преимуществ использования автоматизации в приложениях — независимость от языка программирования. Контроллеры автоматизации могут управлять сервером независимо от языка, использовавшегося при разработке этого компонента. Кроме того, поскольку автоматизация поддерживается на уровне операционной системы, теоретически, в будущем можно будет легко использовать все вновь появившиеся возможности этой технологии, начав применение автоматизации уже сегодня. Если данная тема интересна, давайте продолжим изучение автоматизации. Ниже в настоящей главе рассматриваются вопросы создания серверов и контроллеров автоматизации в Delphi.

СОВЕТ

Если существует проект Delphi 2, использующий автоматизацию и подлежащий переносу в текущую версию Delphi, то не забывайте, что применяемые для создания автоматизации методы коренным образом изменились, начиная с Delphi 3. Запомните, что нельзя смешивать подпрограммы модуля автоматизации `OleAuto` Delphi 2 с подпрограммами модулей `ComObj` и `ComServ`. Если необходимо скомпилировать проект автоматизации Delphi 2 в среде Delphi 6, то модуль `OleAuto` следует поместить в папку `\Delphi6\lib\Delphi2` для организации поддержки обратной совместимости.

Интерфейс IDispatch

Объекты автоматизации — это, в сущности, просто объекты COM, в которых реализован интерфейс `IDispatch`. Интерфейс `IDispatch` определен в модуле `System` следующим образом:

```
type
  IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): Integer;
                                                   stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
                         out TypeInfo): Integer; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
                           NameCount, LocaleID: Integer;
                           DispIDs: Pointer): Integer; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
                    LocaleID: Integer; Flags: Word;
                    var Params; VarResult, ExcepInfo,
                    ArgErr: Pointer): Integer;
  end;
```

Прежде всего следует отметить, что, для того чтобы воспользоваться преимуществами автоматизации в Delphi, вовсе не обязательно вникать во все детали интерфейса `IDispatch`. Поэтому не стоит “сушить мозги”, пытаясь разобраться во всех его сложностях. Вообще-то, напрямую с этим интерфейсом вряд ли придется работать, поскольку Delphi предлагает весьма элегантное решение — инкапсуляцию автоматизации. Описание интерфейса `IDispatch` в настоящем разделе рассматривается только в качестве хорошей основы для понимания сути автоматизации.

Основной функцией интерфейса IDispatch является метод Invoke(). Когда клиент получает указатель IDispatch на сервер автоматизации, он может вызвать метод Invoke() для выполнения определенных методов на сервере. Параметр DispID этого метода содержит число, называемое *диспетчерским идентификатором* (dispatch ID), который указывает, какой именно метод должен быть вызван на сервере. Параметр IID не используется. Параметр LocaleID содержит информацию о языке. Параметр Flags определяет, какого вида метод будет вызван: обычный или метод доступа к свойствам. Свойство Params содержит указатель на массив TDispParams, который содержит параметры, передаваемые этому методу. Параметр VarResult — это указатель на переменную типа OleVariant, в которую будет записано возвращаемое значение вызываемого метода. Параметр ExcepInfo является указателем на запись типа TExcepInfo, которая будет содержать информацию об ошибке, если метод Invoke() возвратит значение DISP_E_EXCEPTION. И, наконец, если метод Invoke() возвращает значение DISP_E_TYPEMISMATCH или DISP_E_PARAMNOTFOUND, то параметр ArgError (указатель на целое число) будет содержать индекс некорректного параметра в массиве Params.

Метод GetIDsOfNames() интерфейса IDispatch вызывается для получения диспетчерского идентификатора одного или нескольких имен методов, заданных в виде строк. Параметр IID этого метода не используется, параметр Names указывает на массив имен методов типа PWideChar. Параметр NameCount содержит количество строк в массиве Names, а параметр LocaleID — информацию о языке. Последний параметр, DispIDs, является указателем на массив целых чисел NameCount, который метод GetIDsOfNames() заполнит диспетчерскими идентификаторами для методов, перечисленных в параметре Names.

Метод GetTypeInfo() возвращает информацию о типе объектов автоматизации (описываемую в настоящей главе далее). Параметр Index определяет тип возвращаемой информации и обычно равен 0. Параметр LCID содержит информацию о языке. При успешном выполнении параметр TypeInfo будет содержать указатель ITypeInfo на информацию о типе объекта автоматизации.

Метод GetTypeInfoCount() в параметре Count возвращает количество типов информации об интерфейсах, поддерживаемых объектом автоматизации. В настоящее время параметр Count может содержать только одно из двух возможных значений: 0, если объект автоматизации не поддерживает информацию о типе, и 1, если он ее поддерживает.

Информация о типе

Затратив много времени и усилий на создание сервера автоматизации, было бы большим разочарованием обнаружить, что потенциальный пользователь не сможет полностью реализовать все его возможности из-за недостаточного описания свойств и методов сервера в прилагаемой документации. К счастью, автоматизация предлагает средство для решения этих проблем и позволяет разработчикам ассоциировать с объектом автоматизации информацию о его типе. Информация о типе хранится в так называемых *библиотеках типов* (type library). Библиотека типов сервера автоматизации может быть добавлена к приложению-серверу или к его библиотеке DLL как ресурс либо храниться во внешнем файле. Библиотеки типов содержат информацию о классах, интерфейсах, типах данных и других компонентах сервера. Эта информация предлагается клиентам сервера автоматизации наряду с другой информацией, необ-

ходимой для создания экземпляров каждого из классов и корректного вызова методов каждого интерфейса.

Delphi самостоятельно создает библиотеки типов при добавлении объектов автоматизации в разрабатываемое приложение или библиотеку. Кроме того, Delphi знает, каким образом преобразовать информацию из библиотеки типов в данные Object Pascal так, чтобы можно было легко управлять сервером автоматизации из приложения Delphi.

Позднее и раннее связывание

Элементы автоматизации, которые рассматривались до сих пор в настоящей главе, работают на основе подхода, который носит название *позднего связывания* (late binding). В данном случае для вызова необходимого метода используется метод `Invoke()` интерфейса `IDispatch`. Под поздним связыванием подразумевается, что вызов метода невозможен до момента запуска программы, поскольку необходимый адрес просто неизвестен. Во время компиляции вызов метода автоматизации имеет вид вызова метода `IDispatch.Invoke()` с соответствующими параметрами, и лишь во время выполнения программы метод `Invoke()` вызовет указанный метод автоматизации. При вызове метода автоматизации с помощью типа Delphi `Variant` или `OleVariant` также используется позднее связывание, поскольку транслятор Delphi сначала должен будет организовать вызов метод `IDispatch.GetIDsOfNames()` для преобразования имени заданного метода в его параметр `DispID`, а затем уже реализовать вызов указанного метода с помощью вызова метода `IDispatch.Invoke()` с полученным параметром `DispID`.

Оптимизация за счет раннего связывания заключается в разрешении параметров `DispID` для вызываемых методов еще во время компиляции, что позволит избежать во время выполнения приложения обращений к методу `GetIDsOfNames()`, перед вызовом того или иного метода. Такая оптимизация часто называется *связыванием ID* (ID binding), или *связыванием идентификаторов*, и обычно осуществляется при вызове методов с использованием типа `dispinterface` Delphi.

Раннее связывание (early binding) происходит, когда объект автоматизации предоставляет свои методы с помощью пользовательского интерфейса, производного от интерфейса `IDispatch`. В этом случае контроллеры автоматизации могут вызывать объекты автоматизации непосредственно с помощью виртуальных таблиц (vtable), т.е. без использования метода `IDispatch.Invoke()`. Поскольку вызов метода осуществляется напрямую, то доступ к методу происходит быстрее, чем при позднем связывании. Раннее связывание используется при вызове метода с помощью типа `interface` Delphi.

Об объекте автоматизации, который позволяет вызывать методы с помощью как метода `Invoke()`, так и потомков интерфейса `IDispatch`, говорят как о поддерживающем *двойной интерфейс* (dual interface). Объекты автоматизации, созданные в среде Delphi, всегда поддерживают двойной интерфейс, а контроллеры автоматизации Delphi позволяют вызывать методы как с помощью метода `Invoke()`, так и напрямую — через интерфейс.

Регистрация

Разумеется, объекты автоматизации должны создать такие же записи в системном реестре, как и обычные объекты COM:

```
HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}
```

Но все серверы автоматизации должны создавать и дополнительный параметр `ProgID`, представляющий собой строковый идентификатор класса автоматизации. Кроме того, создается параметр в ветви `HKEY_CLASSES_ROOT\(ProgID string)`, который содержит идентификатор CLSID класса автоматизации, позволяющий с помощью перекрестной ссылки вернуться к первой записи реестра в разделе `CLSID`.

Создание сервера автоматизации

Delphi существенно упрощает работу по созданию серверов автоматизации обоих типов — как внешнего, так и внутреннего. Процедура создания сервера автоматизации сводится к выполнению четырех действий.

1. Создайте приложение или библиотеку DLL, которые будут субъектом автоматизации. В качестве отправной точки можно даже использовать одно из уже существующих приложений, к которому будут добавлены средства поддержки автоматизации. Это единственное действие, где проявляется реальное различие между созданием внутреннего и внешнего серверов.
2. Создайте объект автоматизации и добавьте его в проект. Delphi поддерживает эксперт объектов автоматизации, который позволит пройти данный этап без особых затруднений.
3. Используя библиотеку типов, добавьте в объект автоматизации свойства и методы. Именно эти свойства и методы будут предоставлены контроллерам автоматизации.
4. Реализуйте методы, созданные Delphi по описанию, в библиотеке типов.

Создание внешнего сервера автоматизации

В настоящем разделе речь пойдет о создании простого внешнего сервера автоматизации. Откройте новый проект и поместите в главную форму компоненты `TShape` и `TEdit`, как показано на рис. 15.2. Сохраните проект под именем `Srv.dpr`.

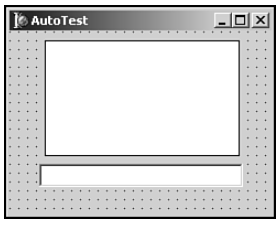

Рис. 15.2. *Главная форма проекта* `Srv`

Теперь добавим в проект объект автоматизации. Для этого необходимо в меню File выбрать пункт New. В раскрывшемся диалоговом окне New Items перейдите во вкладку ActiveX и дважды щелкните на пиктограмме Automation Object (рис. 15.3). Откроется диалоговое окно мастера Automation Object Wizard, показанное на рис. 15.4.

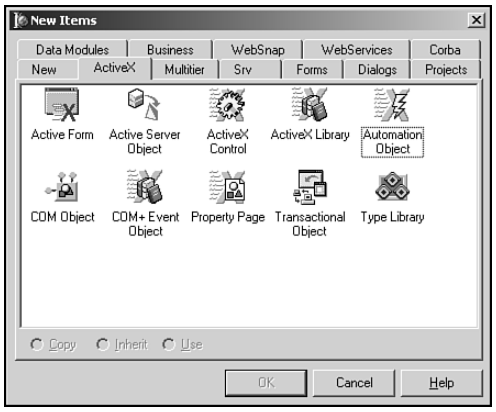

Рис. 15.3. *Добавление нового объекта автоматизации*

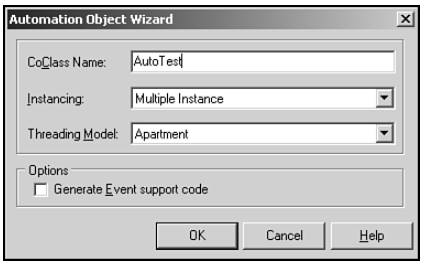

Рис. 15.4. *Диалоговое окно мастера Automation Object Wizard*

В поле CoClass Name диалогового окна Automation Object Wizard необходимо ввести имя класса COM для данного объекта автоматизации. Мастер автоматически добавит префикс `T` к имени класса, если создается класс Object Pascal для объекта автоматизации, или префикс `I`, если создается интерфейс объекта автоматизации. В раскрывающемся списке Instancing содержится три значения, представленных в табл. 15.1.

Таблица 15.1. Допустимые варианты использования объекта автоматизации

Значение	*Описание*
Internal	Этот объект OLE можно использовать только внутри приложения, и в системном реестре регистрировать его не нужно. Внешние процессы не смогут обращаться к внутреннему экземпляру такого сервера автоматизации
Single Instance	Каждый экземпляр сервера может экспортировать только один экземпляр объекта OLE. Если приложение-контроллер запрашивает другой экземпляр объекта OLE, Windows запускает новый экземпляр приложения-сервера
Multiple Instance	Каждый экземпляр сервера может создавать и экспортировать множество экземпляров объекта OLE. Внутренние серверы всегда поддерживают много экземпляров

После установки всех параметров в диалоговом окне Automation Object Wizard Delphi создаст новую библиотеку типов для проекта (если она еще не создана) и добавит в нее описание интерфейса и класса COM. Мастер также создаст в проекте новый модуль, который будет содержать реализацию интерфейса автоматизации для типа, добавленного к библиотеке. На рис. 15.5 показано окно редактора библиотеки типов сразу же после закрытия диалогового окна мастера, а в листинге 15.2 приведен код модуля реализации объекта автоматизации.

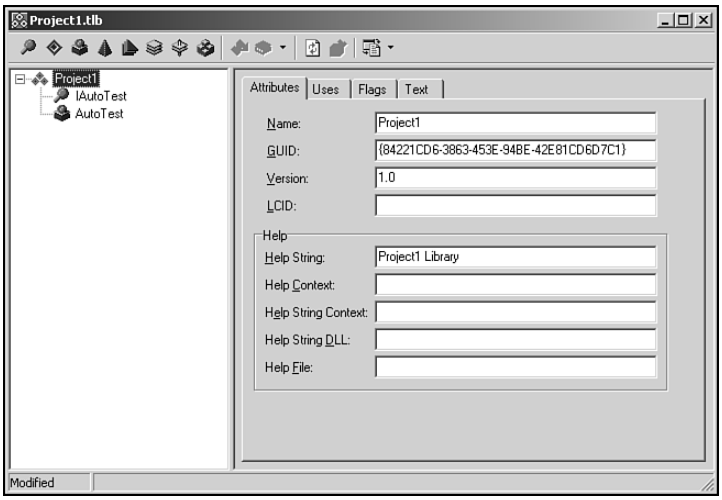

Рис. 15.5. *Новый проект автоматизации в окне редактора библиотеки типов*

Листинг 15.2. Модуль реализации объекта автоматизации

```
unit TestImpl;

interface

uses
  ComObj, ActiveX, Srv_TLB;

type
  TAutoTest = class(TAutoObject, IAutoTest)
  protected
    { Защищенные объявления }
  end;

implementation

uses ComServ;

initialization
  TAutoObjectFactory.Create(ComServer, TAutoTest, Class_AutoTest,
                            ciMultiInstance, tmApartment);
end.
```

Объект автоматизации `TAutoTest` является классом, производным от класса `TAutoObject`. Класс `TAutoObject` — это базовый класс для всех серверов автоматизации. При добавлении методов к интерфейсу с использованием редактора библиотеки типов в том же модуле будут созданы и заготовки новых методов, которые образуют фундамент объекта автоматизации.

СОВЕТ

И снова напомним, что не нужно путать класс `TAutoObject` Delphi 2 (модуль `OleAuto`) с классом `TAutoObject` Delphi 6 (модуль `ComObj`) — они не совместимы. Введенный в Delphi 2 спецификатор видимости `automated` устарел и в Delphi 6 практически не используется.

Добавив в проект объект автоматизации, необходимо, используя редактор библиотеки типов, добавить к исходному интерфейсу одно или несколько свойств и методов. В рассматриваемом проекте библиотека типов будет содержать свойства для чтения и установки формы, цвета и типа, а также текста в полях редактирования. Для визуального представления следует также добавить метод, который отображает текущее состояние этих свойств в диалоговом окне. На рис. 15.6 показана законченная библиотека типов проекта `Srv`. Обратите внимание на добавленное в библиотеку типов перечисление (его элементы отображены в левой панели), предназначенное для поддержки свойства `ShapeType`.

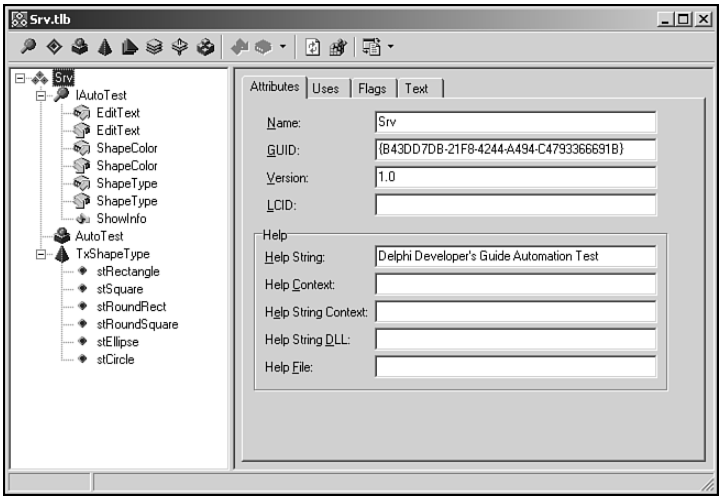

Рис. 15.6. *Законченная библиотека типов*

НА ЗАМЕТКУ

Запомните, что при добавлении к объектам автоматизации в библиотеке типов свойств и методов, используемые для этих свойств и методов параметры или возвращаемые значения должны принадлежать к типу, совместимому с автоматизацией. С автоматизацией совместимы следующие типы: `Byte`, `SmallInt`, `Integer`, `Single`, `Double`, `Currency`, `TDateTime`, `WideString`, `WordBool`, `PSafeArray`, `TDecimal`, `OleVariant`, `IUnknown` и `IDispatch`.

Завершая работу с библиотекой типов, не забудьте в каждом методе заменить кодом реализации те “заготовки”, которые были созданы редактором библиотеки типов. Текст модуля содержащего реализацию всех методов приведен в листинге 15.3.

Листинг 15.3. Законченный модуль реализации

```
unit TestImpl;

interface

uses
  ComObj, ActiveX, Srv_TLB, StdVcl;

type
  TAutoTest = class(TAutoObject, IAutoTest)
  protected
    function Get_EditText: WideString; safecall;
    function Get_ShapeColor: OLE_COLOR; safecall;
    procedure Set_EditText(const Value: WideString); safecall;
    procedure Set_ShapeColor(Value: OLE_COLOR); safecall;
    function Get_ShapeType: TxShapeType; safecall;
    procedure Set_ShapeType(Value: TxShapeType); safecall;
    procedure ShowInfo; safecall;
  end;

implementation

uses ComServ, SrvMain, TypInfo, ExtCtrls, Dialogs, SysUtils,
     Graphics;

function TAutoTest.Get_EditText: WideString;
begin
  Result := FrmAutoTest.Edit.Text;
end;

function TAutoTest.Get_ShapeColor: OLE_COLOR;
begin
  Result := ColorToRGB(FrmAutoTest.Shape.Brush.Color);
end;

procedure TAutoTest.Set_EditText(const Value: WideString);
begin
  FrmAutoTest.Edit.Text := Value;
end;

procedure TAutoTest.Set_ShapeColor(Value: OLE_COLOR);
begin
  FrmAutoTest.Shape.Brush.Color := Value;
end;

function TAutoTest.Get_ShapeType: TxShapeType;
begin
  Result := TxShapeType(FrmAutoTest.Shape.Shape);
end;

procedure TAutoTest.Set_ShapeType(Value: TxShapeType);
```

```
begin
  FrmAutoTest.Shape.Shape := TShapeType(Value);
end;

procedure TAutoTest.ShowInfo;
const
  SInfoStr = 'The Shape''s color is %s, and it''s shape is
%s.'#13#10 +
    'The Edit''s text is "%s."';
begin
  with FrmAutoTest do
    ShowMessage(Format(SInfoStr,
                [ColorToString(Shape.Brush.Color),
                GetEnumName(TypeInfo(TShapeType),
                Ord(Shape.Shape)), Edit.Text]));
end;

initialization
  TAutoObjectFactory.Create(ComServer, TAutoTest, Class_AutoTest,
                            ciMultiInstance, tmApartment);
end.
```

Раздел uses приведенного выше модуля содержит имя модуля Srv_TLB. Это предоставляет содержимое модуля Srv_TLB компилятору Object Pascal при трансляции библиотеки типов данного проекта. Исходный код данного модуля приведен в листинге 15.4.

Листинг 15.4. Модуль Srv_TLB — файл библиотеки типов сервера автоматизации

```
unit Srv_TLB;

// ************************************************************ //
// ВНИМАНИЕ
// --------
// Типы, объявленные в этом файле, были созданы из данных
// библиотеки типов. Если эта библиотека типов будет явно или
// косвенно (через другую библиотеку типов, ссылающуюся на эту
// библиотеку типов) импортирована повторно, или с помощью команды
// 'Refresh' в окне Type Library Editor активизирована во время
// редактирования библиотеки типа, то все содержимое такого файла
// будет создано повторно, а все изменения, внесенные
// пользователем вручную, будут утеряны.
// ************************************************************ //

// PASTLWTR : $Revision:   1.130.3.0.1.0  $
// Файл создан 10/1/2001 12:47:11 AM из библиотеки типов,
// описанной ниже.

// ************************************************************ //
// Type Lib: G:\Doc\D6DG\Source\Ch15\Automate\Srv.tlb (1)
// LIBID: {B43DD7DB-21F8-4244-A494-C4793366691B}
```

```
// LCID: 0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINNT\System32\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINNT\System32\stdvcl40.dll)
// ************************************************************ //
{$TYPEDADDRESS OFF} // Модуль должен быть скомпилирован без
                    // указателей, проверяемых на тип.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}

interface

uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;


// ************************************************************ //
// GUID объявлены в TypeLibrary. Используются следующие префиксы:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************ //
const
  // Главная и вспомогательные версии библиотеки типов
  SrvMajorVersion = 1;
  SrvMinorVersion = 0;

  LIBID_Srv: TGUID = '{B43DD7DB-21F8-4244-A494-C4793366691B}';

  IID_IAutoTest: TGUID = '{C16B6A4C-842C-417F-8BF2-2F306F6C6B59}';
  CLASS_AutoTest: TGUID = '{64C576F0-C9A7-420A-9EAB-0BE98264BC9E}';

// ************************************************************ //
// Объявление перечислений, определенных в библиотеке типов
// ************************************************************ //
// Константы для перечисления TxShapeType
type
  TxShapeType = TOleEnum;
const
  stRectangle = $00000000;
  stSquare = $00000001;
  stRoundRect = $00000002;
  stRoundSquare = $00000003;
  stEllipse = $00000004;
  stCircle = $00000005;

type

// ************************************************************ //
// Продолжение объявления типов, определенных в TypeLibrary
// ************************************************************ //
  IAutoTest = interface;
```

```
  IAutoTestDisp = dispinterface;

// ********************************************************** //
// Объявление компонентных классов, определенных в библиотеке типа
// (ПРИМЕЧАНИЕ. Здесь описывается каждый компонентный класс со
// своим стандартным интерфейсом)
// ********************************************************** //
  AutoTest = IAutoTest;

// ********************************************************** //
// Interface: IAutoTest
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {C16B6A4C-842C-417F-8BF2-2F306F6C6B59}
// ********************************************************** //
  IAutoTest = interface(IDispatch)
    ['{C16B6A4C-842C-417F-8BF2-2F306F6C6B59}']
    function Get_EditText: WideString; safecall;
    procedure Set_EditText(const Value: WideString); safecall;
    function Get_ShapeColor: OLE_COLOR; safecall;
    procedure Set_ShapeColor(Value: OLE_COLOR); safecall;
    function Get_ShapeType: TxShapeType; safecall;
    procedure Set_ShapeType(Value: TxShapeType); safecall;
    procedure ShowInfo; safecall;
    property EditText: WideString read Get_EditText
                                write Set_EditText;
    property ShapeColor: OLE_COLOR read Get_ShapeColor
                                 write Set_ShapeColor;
    property ShapeType: TxShapeType read Get_ShapeType
                                  write Set_ShapeType;
  end;

// ********************************************************** //
// DispIntf:  IAutoTestDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {C16B6A4C-842C-417F-8BF2-2F306F6C6B59}
// ********************************************************** //
  IAutoTestDisp = dispinterface
    ['{C16B6A4C-842C-417F-8BF2-2F306F6C6B59}']
    property EditText: WideString dispid 1;
    property ShapeColor: OLE_COLOR dispid 2;
    property ShapeType: TxShapeType dispid 3;
    procedure ShowInfo; dispid 4;
  end;

// ********************************************************** //
// Класс CoAutoTest предоставляет методы Create и CreateRemote
// для создания экземпляров стандартного интерфейса IAutoTest,
// предоставляемого компонентным классом AutoTest. Эти функции
// предназначены для использования клиентами, желающими
// автоматизировать объекты CoClass, предоставляемые сервером этой
// библиотеки типов
// ********************************************************** //
  CoAutoTest = class
```

```
    class function Create: IAutoTest;
    class function CreateRemote(const MachineName:
                                string): IAutoTest;
  end;

implementation

uses ComObj;

class function CoAutoTest.Create: IAutoTest;
begin
  Result := CreateComObject(CLASS_AutoTest) as IAutoTest;
end;

class function CoAutoTest.CreateRemote(const MachineName:
                                       string): IAutoTest;
begin
  Result := CreateRemoteComObject(MachineName,
                                  CLASS_AutoTest) as IAutoTest;
end;

end.
```

Рассмотрим подробнее этот модуль. Вначале объявляется версия библиотеки типов, а затем — уникальный идентификатор GUID для нее — `LIBID_Srv`. Это значение идентификатора GUID будет использовано при регистрации библиотеки типов в системном реестре. Затем приводятся значения для перечисления `TxShapeType`. Самое интересное в данном перечислении то, что его значения объявляются как константы, а не с использованием перечислимого типа Object Pascal. Эти перечисления подобны перечислениям в C/C++, которые, в отличие от перечислений в Object Pascal, не обязаны начинаться с нулевого элемента и представлять собой ряд последовательных значений.

Затем в модуле `Srv_TLB` объявляется интерфейс `IAutoTest`. В этом объявлении присутствуют свойства и методы, созданные в редакторе библиотеки типов. Кроме того, нетрудно заметить, что методы `Get_`*XXX* и `Set_`*XXX* создаются как методы чтения и записи для каждого свойства.

Соглашение о вызовах `Safecall`

Это — стандартное соглашение о вызовах для методов, введенных в редакторе библиотеки типов (см. приведенное выше объявление интерфейса `IAutoTest`). Однако ключевое слово `safecall` несет в себе несколько большую нагрузку, чем просто соглашение о вызове. Поясним, что имеется в виду. Во-первых, метод *должен* вызываться с использованием соглашения о вызове `safecall`. Во-вторых, метод будет инкапсулирован так, чтобы возвращать значение типа `HResult`. Например, предположим, что существует метод, который в Object Pascal выглядит следующим образом:

```
function Foo(W: WideString): Integer; safecall;
```

На самом деле этот метод компилируется в следующую строку программного кода:
`function Foo(W: WideString; out RetVal: Integer): HResult; stdcall;`
Преимущество соглашения `safecall` состоит в том, что все исключения перехватываются до того, как они будут переданы в вызывающий метод. При возникновении в

методе `safecall` необработанного исключения, оно обрабатывается неявной оболочкой с преобразованием в значение типа `HResult`, которое и возвращается вызывающему методу.

Далее в модуле `Srv_TLB` следует объявление `dispinterface` объекта автоматизации `IAutoTestDisp`. Ключевое слово `dispinterface` сообщает вызывающему методу о том, что методы автоматизации могут быть выполнены с помощью метода `Invoke()`, но при этом не имеется в виду пользовательский интерфейс, с помощью которого могут выполняться такие методы. Несмотря на то что интерфейс `IAutoTest` может быть использован как инструмент разработки, который поддерживает автоматизацию с применением раннего связывания, благодаря объявлению `dispinterface` объекта `IautoTestDisp` можно применять средства, поддерживающие позднее связывание.

Затем в модуле `Srv_TLB` объявляется класс `CoAutoTest`, который весьма облегчает процесс создания объекта автоматизации, в результате чего для создания экземпляра этого объекта достаточно вызвать метод `CoAutoTest.Create()`.

Наконец, в модуле `Srv_TLB` создается класс `TAutoTest`, превращающий сервер в компонент, который можно поместить в палитру компонентов. Эта возможность появилась лишь в Delphi 5 и относится скорее к импортируемым, чем к вновь создаваемым серверам автоматизации.

Как уже упоминалось в настоящей главе, первый раз созданное приложение запускается для выполнения регистрации в системном реестре. Далее в этом разделе речь пойдет о том, как использовать приложение-контроллер для управления сервером.

Создание внутреннего сервера автоматизации

Подобно тому как создание внешних серверов (out-of-process) начинается с построения отдельных приложений, создание внутренних серверов (in-process) начинается с построения библиотек DLL. Можно воспользоваться уже существующей библиотекой DLL или создать совершенно новую, дважды щелкнув на пиктограмме DLL в диалоговом окне New Items (меню File пункт New).

НА ЗАМЕТКУ

Более подробная информация о методах разработки библиотек DLL приведена в главе 6, “Динамически компонуемые библиотеки”. А в настоящем разделе предполагается, что читатель уже имеет представление о создании подобных приложений.

Как уже упоминалось, чтобы успешно работать в качестве внутреннего сервера автоматизации, библиотека DLL должна экспортировать четыре функции, определения которых присутствуют в модуле `ComServ`: `DllGetClassObject()`, `DllCanUnloadNow()`, `DllRegisterServer()` и `DllUnregisterServer()`. Добавьте эти четыре функции в раздел `exports` проекта так, как показано в листинге 15.5.

Листинг 15.5. `IPS.dpr` — файл проекта внутреннего сервера

```
library IPS;

uses
  ComServ,
  IPSMain in 'IPSMain.pas',
  IPS_TLB in 'IPS_TLB.pas';

exports
  DllRegisterServer,
  DllUnregisterServer,
  DllGetClassObject,
  DllCanUnloadNow;

{$R *.TLB}

begin
end.
```

Объект автоматизации добавляется в проект DLL точно так же, как и в проект исполняемого файла, — с помощью мастера объектов автоматизации Automation Object Wizard. Для данного проекта добавим лишь одно свойство и один метод, как показано на рис. 15.7. Текст библиотеки типов в версии Object Pascal приведен в листинге 15.6.

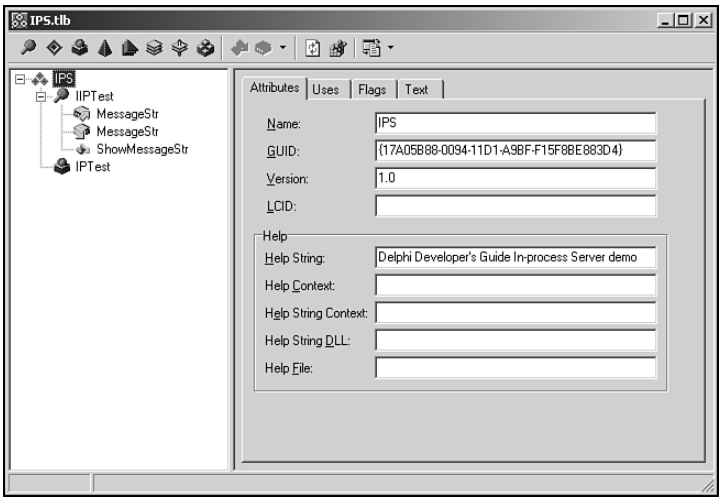

Рис. 15.7. *Проект `IPS` в окне редактора библиотеки типов*

Листинг 15.6. `IPS_TLB.pas` — файл импорта библиотеки типов для проекта внутреннего сервера

```
unit IPS_TLB;

// ************************************************************ //
// ВНИМАНИЕ
// --------
// Типы, объявленные в этом файле, были созданы из данных
// библиотеки типов. Если эта библиотека типов будет явно или
// косвенно (через другую библиотеку типов, ссылающуюся на эту
// библиотеку типа) импортирована повторно, или с помощью команды
// 'Refresh' в окне Type Library Editor активизирована во время
// редактирования библиотеки типа, то все содержимое этого файла
// будет создано повторно, а все изменения, внесенные
// пользователем вручную, будут утеряны.
// ************************************************************ //

// PASTLWTR : $Revision:   1.130.3.0.1.0  $
// Файл создан 10/1/2001 1:06:49 AM из библиотеки типов,
// описанной ниже.

// ************************************************************ //
// Type Lib: G:\Doc\D6DG\Source\Ch15\Automate\IPS.tlb (1)
// LIBID: {17A05B88-0094-11D1-A9BF-F15F8BE883D4}
// LCID: 0
// Helpfile:
// DepndLst:
//   (1) v1.0 stdole, (C:\WINNT\System32\stdole32.tlb)
//   (2) v2.0 StdType, (C:\WINNT\System32\olepro32.dll)
//   (3) v1.0 StdVCL, (C:\WINNT\System32\stdvcl32.dll)
// ************************************************************ //
{$TYPEDADDRESS OFF} // Модуль должен быть скомпилирован без
                    // указателей, проверяемых на тип.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}

interface

uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;


// ************************************************************ //
// GUID объявлены в TypeLibrary. Используются следующие префиксы:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************ //
const
  // Главная и вспомогательные версии библиотеки типов
  IPSMajorVersion = 1;
  IPSMinorVersion = 0;

  LIBID_IPS: TGUID = '{17A05B88-0094-11D1-A9BF-F15F8BE883D4}';
```

```
  IID_IIPTest: TGUID = '{17A05B89-0094-11D1-A9BF-F15F8BE883D4}';
  CLASS_IPTest: TGUID = '{17A05B8A-0094-11D1-A9BF-F15F8BE883D4}';
type

// ********************************************************* //
// Продолжение объявления типов, определенных в TypeLibrary
// ********************************************************* //
  IIPTest = interface;
  IIPTestDisp = dispinterface;

// ********************************************************* //
// Объявление компонентных классов, определенных в библиотеке типа
// (ПРИМЕЧАНИЕ. Здесь описывается каждый компонентный класс со
// своим стандартным интерфейсом)
// ********************************************************* //
  IPTest = IIPTest;

// ********************************************************* //
// Interface: IIPTest
// Flags:     (4432) Hidden Dual OleAutomation Dispatchable
// GUID:      {17A05B89-0094-11D1-A9BF-F15F8BE883D4}
// ********************************************************* //
  IIPTest = interface(IDispatch)
    ['{17A05B89-0094-11D1-A9BF-F15F8BE883D4}']
    function Get_MessageStr: WideString; safecall;
    procedure Set_MessageStr(const Value: WideString); safecall;
    function ShowMessageStr: Integer; safecall;
    property MessageStr: WideString read Get_MessageStr
                                  write Set_MessageStr;
  end;

// ********************************************************* //
// DispIntf:  IIPTestDisp
// Flags:     (4432) Hidden Dual OleAutomation Dispatchable
// GUID:      {17A05B89-0094-11D1-A9BF-F15F8BE883D4}
// ********************************************************* //
  IIPTestDisp = dispinterface
    ['{17A05B89-0094-11D1-A9BF-F15F8BE883D4}']
    property MessageStr: WideString dispid 1;
    function ShowMessageStr: Integer; dispid 2;
  end;

// ********************************************************* //
// Класс CoIPTest предоставляет методы Create и CreateRemote для
// создания экземпляров стандартного интерфейса IIPTest,
// предоставляемого компонентным классом IPTest. Эти функции
// предназначены для использования клиентами, желающими с помощью
// средств автоматизации использовать объекты компонентного
// класса, предоставляемые сервером этой библиотеки типов.
// ********************************************************* //
  CoIPTest = class
    class function Create: IIPTest;
    class function CreateRemote(const MachineName:
```

```
                                    string): IIPTest;
  end;

implementation

uses ComObj;

class function CoIPTest.Create: IIPTest;
begin
  Result := CreateComObject(CLASS_IPTest) as IIPTest;
end;

class function CoIPTest.CreateRemote(const MachineName:
                                       string): IIPTest;
begin
  Result := CreateRemoteComObject(MachineName,
                                  CLASS_IPTest) as IIPTest;
end;

end.
```

Очевидно, что выше приведен простейший пример сервера автоматизации, но он предназначен всего лишь для иллюстрации. Свойству `MessageStr` может быть присвоено значение, которое затем можно отобразить с помощью функции `ShowMessageStr()`. Реализация интерфейса `IIPTest` содержится в модуле `IPSMain.pas`, код которого приведен в листинге 15.7.

Листинг 15.7. `IPSMain.pas` — главный модуль проекта внутреннего сервера автоматизации

```
unit IPSMain;

interface

uses
  ComObj, IPS_TLB;

type
  TIPTest = class(TAutoObject, IIPTest)
  private
    MessageStr: string;
  protected
    function Get_MessageStr: WideString; safecall;
    procedure Set_MessageStr(const Value: WideString); safecall;
    function ShowMessageStr: Integer; safecall;
  end;

implementation

uses Windows, ComServ;
```

```
function TIPTest.Get_MessageStr: WideString;
begin
  Result := MessageStr;
end;

function TIPTest.ShowMessageStr: Integer;
begin
  MessageBox(0, PChar(MessageStr), 'Your string is...', MB_OK);
  Result := Length(MessageStr);
end;

procedure TIPTest.Set_MessageStr(const Value: WideString);
begin
  MessageStr := Value;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TIPTest, Class_IPTest,
                            ciMultiInstance, tmApartment);
end.
```

Как уже упоминалось в настоящей главе, внутренние серверы регистрируются не так, как внешние. Для регистрации в системном реестре внутреннего сервера вызывается функция `DllRegisterServer()`. В интегрированной среде разработки Delphi этот процесс выполняется очень просто, достаточно выбрать в меню Run пункт Register ActiveX Server.

Создание контроллеров автоматизации

В Delphi процессы управления серверами автоматизации из приложений максимально упрощены. Но, тем не менее, при этом обеспечивается высокая степень гибкости управления — благодаря возможности раннего (с помощью интерфейсов) или позднего (с использованием типов `dispinterface` или `OleVariant`) связывания.

Управление внешним сервером

Проект `Control` представляет собой контроллер автоматизации, с помощью которого демонстрируются все три типа поддержки функций автоматизации (интерфейсы, диспинтерфейсы и варианты). Проект `Control` — это контроллер, предназначенный для работы с приложением сервера автоматизации `Srv`, создание которого описано в настоящей главе ранее. Главная форма проекта показана на рис. 15.8.

При щелчке на кнопке Connect приложение `Control` подключается к серверу, как показано ниже.

```
FIntf := CoAutoTest.Create;
FDispintf := CreateComObject(Class_AutoTest) as IAutoTestDisp;
FVar := CreateOleObject('Srv.AutoTest');
```

В этом фрагменте кода представлены переменные типа interface, dispinterface и OleVariant, с помощью которых экземпляры серверов автоматизации создаются тремя различными способами. Интересно то, что эти способы практически абсолютно взаимозаменяемы. Например, следующий код также вполне допустим:

```
FIntf := CreateComObject(Class_AutoTest) as
IAutoTest;
FDispintf := CreateOleObject('Srv.AutoTest') as
IAutoTestDisp;
FVar := CoAutoTest.Create;
```

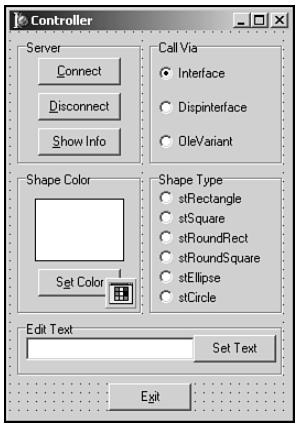

Рис. 15.8. *Главная форма проекта* Control

В листинге 15.8 приведен код модуля Ctrl, который содержит остальную часть кода контроллера автоматизации. Заметьте, что это приложение позволяет управлять сервером, используя любой тип переменной Delphi: interface, dispinterface или OleVariant.

Листинг 15.8. Ctrl.pas — главный модуль проекта контроллера для работы с внешним сервером

```
unit Ctrl;

interface

{$WARN SYMBOL_PLATFORM OFF}

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ColorGrd, ExtCtrls, Srv_TLB, Buttons;

type
  TControlForm = class(TForm)
    CallViaRG: TRadioGroup;
    ShapeTypeRG: TRadioGroup;
    GroupBox1: TGroupBox;
    GroupBox2: TGroupBox;
    Edit: TEdit;
    GroupBox3: TGroupBox;
    ConBtn: TButton;
    DisBtn: TButton;
    InfoBtn: TButton;
    ColorBtn: TButton;
    ColorDialog: TColorDialog;
    ColorShape: TShape;
    ExitBtn: TButton;
    TextBtn: TButton;
    procedure ConBtnClick(Sender: TObject);
    procedure DisBtnClick(Sender: TObject);
```

```
    procedure ColorBtnClick(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
    procedure TextBtnClick(Sender: TObject);
    procedure InfoBtnClick(Sender: TObject);
    procedure ShapeTypeRGClick(Sender: TObject);
  private
    { Закрытые объявления }
    FIntf: IAutoTest;
    FDispintf: IAutoTestDisp;
    FVar: OleVariant;
    procedure SetControls;
    procedure EnableControls(DoEnable: Boolean);
  public
    { Открытые объявления }
  end;

var
  ControlForm: TControlForm;

implementation

{$R *.DFM}

uses ComObj, Variants;

procedure TControlForm.SetControls;
// Инициализация элементов управления текущими значениями сервера
begin
  case CallViaRG.ItemIndex of
    0: begin
      ColorShape.Brush.Color := FIntf.ShapeColor;
      ShapeTypeRG.ItemIndex := FIntf.ShapeType;
      Edit.Text := FIntf.EditText;
    end;
    1: begin
      ColorShape.Brush.Color := FDispintf.ShapeColor;
      ShapeTypeRG.ItemIndex := FDispintf.ShapeType;
      Edit.Text := FDispintf.EditText;
    end;
    2: begin
      ColorShape.Brush.Color := FVar.ShapeColor;
      ShapeTypeRG.ItemIndex := FVar.ShapeType;
      Edit.Text := FVar.EditText;
    end;
  end;   // case
end;

procedure TControlForm.EnableControls(DoEnable: Boolean);
begin
  DisBtn.Enabled := DoEnable;
  InfoBtn.Enabled := DoEnable;
  ColorBtn.Enabled := DoEnable;
  ShapeTypeRG.Enabled := DoEnable;
```

```
  Edit.Enabled := DoEnable;
  TextBtn.Enabled := DoEnable;
end;

procedure TControlForm.ConBtnClick(Sender: TObject);
begin
  FIntf := CoAutoTest.Create;
  FDispintf := CreateComObject(Class_AutoTest) as IAutoTestDisp;
  FVar := CreateOleObject('Srv.AutoTest');
  EnableControls(True);
  SetControls;
end;

procedure TControlForm.DisBtnClick(Sender: TObject);
begin
  FIntf := nil;
  FDispintf := nil;
  FVar := Unassigned;
  EnableControls(False);
end;

procedure TControlForm.ColorBtnClick(Sender: TObject);
var
  NewColor: TColor;
begin
  if ColorDialog.Execute then begin
    NewColor := ColorDialog.Color;
    case CallViaRG.ItemIndex of
      0: FIntf.ShapeColor := NewColor;
      1: FDispintf.ShapeColor := NewColor;
      2: FVar.ShapeColor := NewColor;
    end; // case
    ColorShape.Brush.Color := NewColor;
  end;
end;

procedure TControlForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TControlForm.TextBtnClick(Sender: TObject);
begin
  case CallViaRG.ItemIndex of
    0: FIntf.EditText := Edit.Text;
    1: FDispintf.EditText := Edit.Text;
    2: FVar.EditText := Edit.Text;
  end;
end;

procedure TControlForm.InfoBtnClick(Sender: TObject);
begin
  case CallViaRG.ItemIndex of
```

```
    0: FIntf.ShowInfo;
    1: FDispintf.ShowInfo;
    2: FVar.ShowInfo;
  end;
end;

procedure TControlForm.ShapeTypeRGClick(Sender: TObject);
begin
  case CallViaRG.ItemIndex of
    0: FIntf.ShapeType := ShapeTypeRG.ItemIndex;
    1: FDispintf.ShapeType := ShapeTypeRG.ItemIndex;
    2: FVar.ShapeType := ShapeTypeRG.ItemIndex;
  end;
end;

end.
```

Еще одна интересная особенность, приведенная в этом листинге, — простота отключения от сервера автоматизации: переменные типа `interface` и `dispinterface` достаточно установить равными значению `nil`, а переменную типа `Variant` — равной значению `Unassigned`. Естественно, сервер автоматизации также освобождается, если приложение `Control` закрывается обычным способом.

СОВЕТ

Использование интерфейсов практически всегда эффективней использования типов `dispinterface` или `Variant`, поэтому по возможности всегда для управления серверами автоматизации пользуйтесь именно интерфейсами.
Тип `Variant` по части производительности хуже прочих методов, поскольку в этом случае во время выполнения приложения перед вызовом нужного метода с помощью обращения к методу `Invoke()` необходимо вызывать функцию `GetIDsOfNames()` для преобразования имени метода в его диспетчерский идентификатор (dispatch ID).
Производительность при использовании типа `dispinterface` можно оценить как промежуточную между использованием интерфейса и типа `Variant`. Но почему же существует различие в производительности при использовании типов `Variant` и `dispinterface`, если в обоих случаях применяется позднее связывание? Причина в следующем: при использовании типа `dispinterface` возможна оптимизация, называемая *связыванием идентификаторов* (ID binding). Имеется в виду следующее: диспетчерские идентификаторы (dispatch ID) используемых методов известны еще на стадии компиляции, поэтому компилятору не нужно организовывать динамический вызов метода `GetIDsOfNames()` перед вызовом метода `Invoke()`. Но наиболее заметное преимущество диспинтерфейсов перед типом `Variant` состоит в том, что тип `dispinterface` поддерживает использование утилиты CodeInsight, которая упрощает код программ, а при работе с типом `Variant` это невозможно.

На рис. 15.9 показана форма приложения `Control`, управляющего сервером `Srv`.

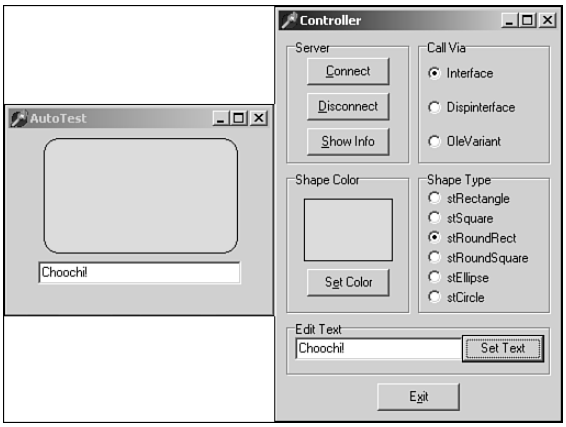

Рис. 15.9. *Контроллер и сервер автоматизации*

Управление внутренним сервером

Механизм управления внутренним сервером не отличается от механизма управления внешним сервером. Необходимо лишь помнить, что контроллер автоматизации в данном случае работает в пространстве процесса управляющего приложения. Это означает, что производительность будет несколько выше, чем при управлении внешним сервером, и то, что ошибка сервера автоматизации может привести к аварийному завершению приложения контроллера.

Рассмотрим приложение контроллера, предназначенного для управления внутренним сервером автоматизации, создание которого описано в настоящей главе выше. В данном случае для управления сервером используем только интерфейс. На рис. 15.10 показана главная форма проекта `IPCtrl`. Код главного модуля проекта `IPCtrl` (файл `IPCMain.pas`) приведен в листинге 15.9.

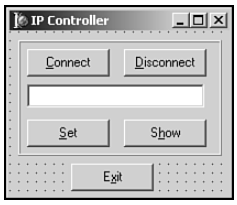

Рис. 15.10. *Главная форма проекта `IPCtrl`*

Листинг 15.9. `IPCMain.pas` — главный модуль контроллера для управления внутренним сервером

```
unit IPCMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, IPS_TLB;

type
  TIPCForm = class(TForm)
    ExitBtn: TButton;
    Panel1: TPanel;
```

```
    ConBtn: TButton;
    DisBtn: TButton;
    Edit: TEdit;
    SetBtn: TButton;
    ShowBtn: TButton;
    procedure ConBtnClick(Sender: TObject);
    procedure DisBtnClick(Sender: TObject);
    procedure SetBtnClick(Sender: TObject);
    procedure ShowBtnClick(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
  private
    { Закрытые объявления }
    IPTest: IIPTest;
    procedure EnableControls(DoEnable: Boolean);
  public
    { Открытые объявления }
  end;

var
  IPCForm: TIPCForm;

implementation

uses ComObj;

{$R *.DFM}

procedure TIPCForm.EnableControls(DoEnable: Boolean);
begin
  DisBtn.Enabled := DoEnable;
  Edit.Enabled := DoEnable;
  SetBtn.Enabled := DoEnable;
  ShowBtn.Enabled := DoEnable;
end;

procedure TIPCForm.ConBtnClick(Sender: TObject);
begin
  IPTest := CreateComObject(CLASS_IPTest) as IIPTest;
  EnableControls(True);
end;

procedure TIPCForm.DisBtnClick(Sender: TObject);
begin
  IPTest := nil;
  EnableControls(False);
end;

procedure TIPCForm.SetBtnClick(Sender: TObject);
begin
  IPTest.MessageStr := Edit.Text;
end;

procedure TIPCForm.ShowBtnClick(Sender: TObject);
```

```
begin
  IPTest.ShowMessageStr;
end;

procedure TIPCForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

end.
```

Не забудьте, что перед запуском проекта `IPCtrl` сервер должен быть зарегистрирован. Это можно сделать несколькими способами: воспользоваться меню Run пункт Register ActiveX Server, предварительно открыв проект `IPS` в среде разработки Delphi, прибегнуть к утилите Windows `RegSvr32.exe` или применить программу `RegSvr32.exe`, которая входит в комплект поставки Delphi. На рис. 15.11 показан запущенный проект `IPCtrl`, управляющий сервером `IPS`.

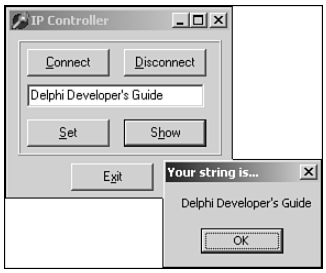

Рис. 15.11. *Контроллер `IPCtrl` управляет сервером `IPS`*

Усложнение технологий автоматизации

В настоящем разделе рассмотрим более интересные возможности автоматизации, о которых часто не знают даже признанные мастера Delphi. Познакомимся с событиями автоматизации, коллекциями, особенностями библиотеки типов и поддержкой языка низкого уровня для COM. Не будем тратить время на пространные рассуждения — лучше сразу приступим к делу!

События автоматизации

Программисты Delphi давно освоили работу с событиями, особенно в части их использования для создания обычных элементов управления. Перетащим кнопку в форму, дважды щелкаем на ее событии `OnClick` в окне инспектора объектов, пишем нужный программный код — и вся недолга. Даже при создании новых элементов управления с событиями особых проблем не возникает: создаем новый тип метода, добавляем в данный элемент управления поле и опубликовав свойство остаемся вполне довольны собой. Однако у разработчиков COM в среде Delphi одно только упоминание о событиях может вызвать нервную дрожь. Многие разработчики Delphi стараются избегать событий COM лишь потому, что у них, видите ли, "нет времени разбираться во всех этих штучках". Если к последней группе относитесь и вы, читатель,

то вам в очередной раз будет предоставлена возможность убедиться, что "не так страшен черт…", — благодаря прекрасной встроенной поддержке Delphi. И хотя все новые термины, связанные с событиями автоматизации, могут, казалось бы, еще больше усложнить общую картину, в настоящем разделе, думаю, нам удастся добраться до самой сути этих событий, и вскоре вы не сможете сдержать недоумения: "И всего делов-то?!".

Что такое события?

Попросту говоря, события обеспечивают серверу средство организации обратного вызова в приложении-клиенте для предоставления ему определенной информации. При использовании традиционной модели "клиент/сервер" клиент вызывает сервер для выполнения некоторого действия или получения некоторых данных, сервер выполняет необходимое действие или получает данные, после чего управление возвращается клиенту. В большинстве случаев эта модель прекрасно работает, но она оказывается бессильной, когда событие, в котором заинтересован клиент, асинхронно по своей природе или зависит от элементов пользовательского интерфейса. Например, когда клиент посылает серверу требование загрузить файл, и при этом он (клиент) не хочет бесполезно тратить время на ожидание окончания загрузки, после чего он смог бы продолжить свою работу (особенно при использовании достаточно медленной связи через модем). Для клиента предпочтительней было бы использовать такую модель, при которой сразу после отправки серверу некоторой инструкции можно было бы продолжать работу до тех пор, пока сервер не уведомит клиента о завершении загрузки файла. Аналогично, даже такой простейший способ работы с пользовательским интерфейсом, как щелчок на кнопке, служит хорошим примером ситуации, в которой серверу потребуется некоторый механизм уведомления клиента об имевшем место событий. Очевидно, что сам клиент не может вызвать в сервере метод, который будет ожидать щелчка на некоторой кнопке.

В общем случае сервер должен сам нести ответственность за перехват и генерацию событий, тогда как клиент будет отвечать за получение событий и подключение к серверу с целью их реализации. Конечно, такая вольная трактовка распределения ответственности может послужить прекрасной мишенью для придирок, тем не менее она работает. В соответствии с этим определением средствами Delphi и механизмами автоматизации предлагаются два различных подхода к понятию событий. Углубленное рассмотрение деталей каждой из этих моделей поможет правильно представить всю картину в перспективе.

События в Delphi

Delphi при подходе к событиям придерживается методологии KISS (*k*eep *i*t *s*imple, *s*tupid! — "чем проще, тем лучше"). События реализуются как указатели на методы — если назначить эти указатели определенным методам в приложении, то они могут быть вызваны с помощью этого указателя. В качестве иллюстрации рассмотрим сценарий разработки приложения, в задачу которого входит обработка события в компоненте. Если на данную ситуацию посмотреть абстрактно, то "сервер" в данном случае будет компонентом, определяющим и генерирующим событие. А роль "клиента" будет исполнять приложение, которое использует этот компонент, поскольку оно подключается к событию, назначая имя некоторого заданного метода указателю события.

Несмотря на то что именно эта простая модель генерации событий и делает среду разработки в Delphi элегантной и простой в применении, ради такой простоты определенно приходится жертвовать некоторыми возможностями. Например, не существует встроенного способа разрешить сразу нескольким клиентам "услышать" одно и то же событие — это называется *мультивещанием* (multicasting). Кроме того, невозможно динамически получить описание типа для события, не написав некоторого кода RTTI (который, возможно, и не пришлось бы использовать в приложении из-за привязки к конкретной версии).

События в автоматизации

Если о модели событий Delphi можно сказать, что она проста, но весьма ограниченна, то модель событий автоматизации отличается бо́льшими возможностями, и большей сложностью. Опытные программисты уже догадались, что в технологии автоматизации события реализуются с помощью интерфейсов. Но события определяются не на основе пары метод-событие, а только как часть интерфейса. Такой интерфейс часто называется *интерфейсом событий* (events interface), или *исходящим интерфейсом* (outgoing interface). Это название связано с тем, что он реализуется не самим сервером (как другие интерфейсы), а его клиентами, причем методы интерфейса будут вызываться со стороны сервера. Как и все интерфейсы, интерфейсы событий имеют соответствующие идентификаторы интерфейсов (IID), которые определяют их уникальным образом. Кроме того, описание интерфейса событий (подобно иным интерфейсам) содержится в библиотеке типов объекта автоматизации, связанной с компонентным классом этого объекта автоматизации.

В серверах, интерфейсы событий которых необходимо сделать доступными для клиентов, следует реализовать интерфейс `IConnectionPointContainer`. Этот интерфейс определен в модуле `ActiveX` следующим образом:

```
type
  IConnectionPointContainer = interface
    ['{B196B284-BAB4-101A-B69C-00AA00341D07}']
    function EnumConnectionPoints(
                out Enum: IEnumConnectionPoints): HResult; stdcall;
    function FindConnectionPoint(const iid: TIID;
                out cp: IConnectionPoint): HResult; stdcall;
  end;
```

В терминологии COM под *точкой подключения* (connection point) понимают некоторые средства, предоставляющие программный доступ к исходящему интерфейсу. Если клиенту нужно выяснить, поддерживает ли сервер события, то для этого достаточно вызвать функцию `QueryInterface` для интерфейса `IConnectionPointContainer`. Если такой интерфейс присутствует, то сервер может служить источником событий. Метод `EnumConnectionPoints()` интерфейса `IConnectionPointContainer` позволяет клиентам опросить все исходящие интерфейсы, поддерживаемые сервером. Для получения конкретного внешнего интерфейса клиенты могут использовать метод `FindConnectionPoint()`.

Нетрудно заметить, что метод `FindConnectionPoint()` обеспечивает клиента интерфейсом `IConnectionPoint`, который представляет нужный исходящий интерфейс. Интерфейс `IConnectionPoint` также определен в модуле `ActiveX`, и это объявление имеет следующий вид:

```
type
  IConnectionPoint = interface
    ['{B196B286-BAB4-101A-B69C-00AA00341D07}']
    function GetConnectionInterface(
            out iid: TIID): HResult; stdcall;
    function GetConnectionPointContainer(
            out cpc: IConnectionPointContainer): HResult; stdcall;
    function Advise(const unkSink: IUnknown;
            out dwCookie: Longint): HResult; stdcall;
    function Unadvise(dwCookie: Longint): HResult; stdcall;
    function EnumConnections(
            out Enum: IEnumConnections): HResult; stdcall;
  end;
```

Метод `GetConnectionInterface()` интерфейса `IConnectionPoint` предоставляет идентификатор (IID) исходящего интерфейса, поддерживаемый данной точкой подключения. А метод `GetConnectionPointContainer()` предоставляет интерфейс `IConnectionPointContainer` (описанный ранее), который управляет этой точкой подключения. Весьма любопытен метод `Advise`. Именно он воплощает в реальность магию связывания исходящих событий на сервере с интерфейсом `events`, реализованным клиентом. Первый параметр, передаваемый данному методу, представляет собой клиентскую реализацию интерфейса `events`, второй — это cookie, идентифицирующий это конкретное подключение. Функция `Unadvise()` предназначена для разрыва соединения, установленного ранее между клиентом и сервером с помощью функции `Advise()`. Функция `EnumConnections()` позволяет клиенту опросить все активные в данный момент подключения, т.е. все соединения, выполненные функцией `Advise()`.

Ввиду очевидной путаницы, которая может возникнуть, если описывать участников этих отношений просто как *клиент* и *сервер*, в технологии автоматизации было решено использовать несколько другие термины, позволяющие однозначно определить, "кто есть кто". Поэтому реализацию исходящего интерфейса, содержащегося внутри клиента, договорились называть *стоком* (sink), а объект сервера, который генерирует события для клиента, — *источником* (source).

Из всего вышесказанного, надеемся, вполне очевидно, что события автоматизации имеют два преимущества перед событиями Delphi. А именно: они могут быть мультивещательными, поскольку функцию `IConnectionPoint.Advise()` в этом случае можно вызывать более одного раза. И, кроме того, события автоматизации являются самоописательными (благодаря использованию библиотеки типов и методов перечисления), поэтому ими можно управлять динамически.

События автоматизации в Delphi

Итак, все эти рассуждения звучат прекрасно, но как на самом деле заставить события автоматизации работать в среде Delphi? С удовольствием ответим, а в качестве примера создадим приложение-сервер автоматизации, которое предоставляет некоторый внешний интерфейс, и приложение-клиент, которое реализует сток для этого интерфейса. Учтите, что вовсе не обязательно быть экспертом по части точек подключения, стоков, источников и прочих премудростей, чтобы добиться от Delphi желаемых результатов. Однако глубокое понимание того, что действительно происходит под покровами оболочек и инкапсулирующих функций Delphi, сослужит хорошую и, надеемся, полезную службу.

Сервер

Создание сервера начнем с создания нового приложения. В качестве примера создадим новое приложение, содержащее одну форму с полем текстового редактора (компонент TMemo), которое будет управляться клиентом — как показано на рис. 15.12.

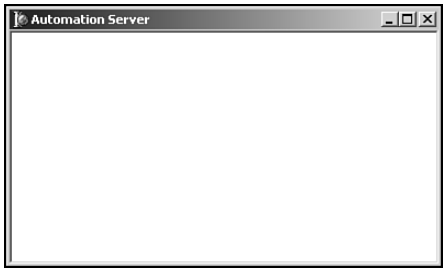

Рис. 15.12. *Сервер автоматизации со своей главной формой* Events

Затем к данному приложению добавим объект автоматизации. Для этого в меню Delphi File выберем пункты New, ActiveX, и Automation Object, в результате чего откроется окно мастера объектов автоматизации Automation Object Wizard (см. рис. 15.4).

Обратите внимание на флажок параметра Generate Event Support Code (Создавать код поддержки событий) в окне мастера Automation Object Wizard. Данный флажок нужно установить, поскольку лишь в этом случае будет создан код, необходимый для активизации исходящего интерфейса в объекте автоматизации. К тому же в библиотеку типов будет записана вся необходимая информация, связанная с исходящим интерфейсом. При щелчке на кнопке OK в диалоговом окне мастера открывается окно редактора библиотеки типов Type Library Editor. При этом в библиотеке типов уже присутствуют как интерфейс автоматизации, так и исходящий интерфейс (с именами IServerWithEvents и IServerWithEventsEvents соответственно). В интерфейс IServerWithEvents добавлены методы AddText() и Clear(), а в интерфейс IServerWithEventsEvents — методы OnTextChanged() и OnClear().

Нетрудно догадаться, что метод Clear() предназначен для очистки содержимого поля текстового редактора, а метод AddText() — для добавления в него еще одной строки текста. Событие OnTextChanged() происходит при изменении содержимого поля текстового редактора, а событие OnClear() — при очистке его поля. Заметьте также, что каждому из методов AddText() и OnTextChanged() передается один параметр типа WideString.

Первое, что нужно сделать, — это реализовать методы AddText() и Clear(). Реализация таких методов выглядит следующим образом:

```
procedure TServerWithEvents.AddText(const NewText: WideString);
begin
  MainForm.Memo.Lines.Add(NewText);
end;

procedure TServerWithEvents.Clear;
begin
  MainForm.Memo.Lines.Clear;
  if FEvents <> nil then FEvents.OnClear;
end;
```

Скорее всего, весь этот код уже знаком, за исключением, разве что, последней строки в определении метода `Clear()`. В данной строке кода с помощью проверки переменной `FEvents` на неравенство значению `nil` гарантируется существование стока клиента, готового к приему данного события, которое в этом случае создается простым вызовом метода `OnClear()`.

Чтобы установить событие `OnTextChanged()`, необходим обработчик события `OnChange` поля текстового редактора. Для этого добавим строку кода в метод `Initialized()` класса `TServerWithEvents`, связывающего данное событие с методом в объекте `TServerWithEvents`:

```
MainForm.Memo.OnChange := MemoChange;
```

Реализация метода `MemoChange()` имеет следующий вид:

```
procedure TServerWithEvents.MemoChange(Sender: TObject);
begin
  if FEvents <> nil then
    FEvents.OnTextChanged((Sender as TMemo).Text);
end;
```

При выполнении данного кода также проверяется готовность клиента к прослушиванию события. И если это событие ожидаемо (переменная `FEvents` не равна значению `nil`), то оно происходит, передавая в качестве параметра текст из поля текстового редактора.

Верите или нет, но на этом с реализацией сервера покончено! Теперь займемся клиентом.

Клиент

Приложение клиента состоит из одной формы, содержащей компоненты `TEdit`, `TMemo` и три компонента `TButton`, как показано на рис. 15.13.

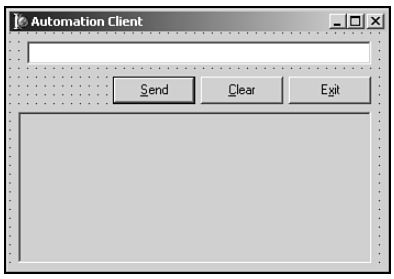

Рис. 15.13. *Главная форма клиента автоматизации*

В раздел `uses` модуля главной формы приложения клиента добавлено имя модуля `Server_TLB`, что обеспечило доступ ко всем типам и методам, содержащимся внутри него. Объект главной формы приложения клиента `TMainForm` будет содержать поле, которое ссылается на сервер типа `IServerWithEvents` по имени `FServer`. В конструкторе форм `TMainForm` создадим экземпляр этого сервера, используя вспомогательный класс, описанный в модуле `Server_TLB`:

```
FServer := CoServerWithEvents.Create;
```

Следующий этап заключается в реализации класса стока событий. Поскольку данный класс будет вызываться сервером через механизмы автоматизации, он должен реализовать

интерфейс IDispatch (и, разумеется, интерфейс IUnknown). Вот как выглядит объявление типа для этого класса:

```
type
  TEventSink = class(TObject, IUnknown, IDispatch)
  private
    FController: TMainForm;
    { IUnknown }
    function QueryInterface(const IID: TGUID;
                            out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    { IDispatch }
    function GetTypeInfoCount(out Count: Integer): HResult;
                                                   stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
                         out TypeInfo): HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
                           NameCount, LocaleID: Integer;
                           DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
                    LocaleID: Integer; Flags: Word;
                    var Params; VarResult, ExcepInfo,
                    ArgErr: Pointer): HResult; stdcall;
  public
    constructor Create(Controller: TMainForm);
  end;
```

Большинство методов интерфейсов IUnknown и IDispatch не реализованы, за исключением таких методов, как IUnknown.QueryInterface() и IDispatch.Invoke(). О них и пойдет речь далее.

Метод QueryInterface() класса TEventSink реализован следующим образом:

```
function TEventSink.QueryInterface(const IID: TGUID;
                                   out Obj): HResult;
begin
  // Сначала рассмотрим собственную реализацию интерфейса
  // (Я реализовал интерфейсы IUnknown и IDispatch).
  if GetInterface(IID, Obj) then
    Result := S_OK
  // Затем при поиске внешнего интерфейса выполняем рекурсивное,
  // обращение чтобы возвратить указатель на интерфейс IDispatch.
  else if IsEqualIID(IID, IServerWithEventsEvents) then
    Result := QueryInterface(IDispatch, Obj)
    // Во всех остальных случаях возвратить признак ошибки.
  else
    Result := E_NOINTERFACE;
end;
```

По сути, этот метод возвращает экземпляр только в том случае, если запрашиваемым интерфейсом является IUnknown, IDispatch или IServerWithEventsEvents.

Вот как выглядит метод Invoke класса TEventSink:

```
function TEventSink.Invoke(DispID: Integer; const IID: TGUID;
                           LocaleID: Integer; Flags: Word;
                           var Params; VarResult, ExcepInfo,
                           ArgErr: Pointer): HResult;
var
  V: OleVariant;
begin
  Result := S_OK;
  case DispID of
    1: begin
      // Первый параметр - новая строка
      V := OleVariant(TDispParams(Params).rgvarg^[0]);
      FController.OnServerMemoChanged(V);
    end;
    2: FController.OnClear;
  end;
end;
```

Метод `TEventSink.Invoke()` жестко привязан к методам, имеющим значения идентификаторов `DispID` 1 и 2. Эти значения выбраны в приложении сервера для методов `OnTextChanged()` и `OnClear()` соответственно. Метод `OnClear()` имеет самую простую реализацию: в ответ на это событие он просто вызывает метод `OnClear()` главной формы клиента. Событие `OnTextChanged()` несколько сложнее: оно извлекает параметр из массива `Params.rgvarg`, который передается данному методу в качестве параметра, и передает его методу `OnServerMemoChanged()` главной формы клиента. Заметьте, поскольку количество и тип параметров известны, можно существенно упростить исходный код. В принципе, можно реализовать функцию `Invoke()` и в общем виде, чтобы она вычисляла количество и типы параметров и помещала их в стек или в регистры до вызова соответствующей функции. Если эта тема интересна, то рассмотрите метод `TOleControl.InvokeEvent()` в модуле `OleCtrls`. Данный метод представляет логику приема событий для контейнера элементов управления ActiveX.

Методы `OnClear()` и `OnServerMemoChanged()` предназначены для манипулирования содержимым поля `memo` клиента. Вот как реализованы такие методы:

```
procedure TMainForm.OnServerMemoChanged(const NewText: string);
begin
  Memo.Text := NewText;
end;

procedure TMainForm.OnClear;
begin
  Memo.Clear;
end;
```

Теперь осталось лишь соединить сток событий с исходящим интерфейсом сервера-источника. Это легко осуществляется с помощью функции `InterfaceConnect()` (содержащейся в модуле `ComObj`), которая будет вызываться из конструктора главной формы:

```
InterfaceConnect(FServer, IServerWithEventsEvents,
                 FEventSink, FCookie);
```

Первый параметр, передаваемый этой функции, представляет собой ссылку на объект-источник. Второй параметр — это идентификатор исходящего интерфейса (IID). Третий — интерфейс стока событий. Четвертый (и последний) параметр задает cookie и является ссылкой — он будет заполнен вызывающей стороной.

"Приличия требуют", чтобы, наигравшись с событиями, клиент корректно завершил работу, вызвав функцию `InterfaceDisconnect()`. Эти действия реализуются в деструкторе главной формы:

```
InterfaceDisconnect(FEventSink, IServerWithEventsEvents, FCookie);
```

Пример

Теперь, когда приложения клиента и сервера написаны, можно увидеть их в действии. Прежде чем запускать приложение клиента, не забудьте один раз запустить и закрыть приложение сервера (или запустить его с параметром `/regserver`), чтобы зарегистрировать его как сервер. На рис. 15.14 показаны результаты взаимодействия между клиентом, сервером, источником и стоком.

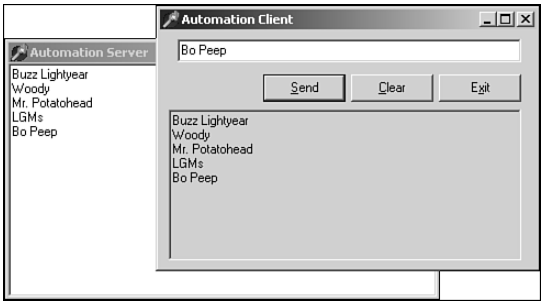

Рис. 15.14. *Клиент автоматизации, манипулирующий сервером и обрабатывающий события*

События с несколькими стоками

Хотя описанная выше методика прекрасно работает при генерации событий, предназначенных для одного клиента, она не так уж хороша при работе с несколькими клиентами. Однако ситуации подключения к серверу нескольких клиентов возникают довольно часто, и в этом случае необходимо генерировать события для всех клиентов. К счастью, для этого потребуется добавить лишь несколько строк кода. Чтобы создать события для нескольких клиентов, нужно написать код, опрашивающий все имеющиеся подключения и вызывающий соответствующий метод стока. Это можно реализовать, внеся несколько изменений в предыдущий пример.

Пойдем по порядку. Чтобы поддерживать несколько подключений, необходимо в качестве параметра `Kind` метода `TConnectionPoints.CreateConnectionPoint()` передать значение `ckMulti`. Этот метод вызывается из метода `Initialize()` объекта автоматизации:

```
FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
                                        ckMulti, EventConnect);
```

Перед построением перечня подключений нужно получить ссылку на интерфейс IConnectionPointContainer. Из интерфейса IConnectionPointContainer можно получить интерфейс IConnectionPoint, представляющий собой исходящий интерфейс, а с помощью метода IConnectionPoint.EnumConnections() можно получить интерфейс IEnumConnections, который будет использоваться для построения перечня подключений. Вся эта логика "укладывается" в следующем методе:

```
function TServerWithEvents.GetConnectionEnumerator: IEnumConnec-
tions;
var
  Container: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Result := nil;
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID,
                                         CP));
  CP.EnumConnections(Result);
end;
```

Получив перечень подключений интерфейса для каждого из клиентов, осуществляется вызов его стока последовательным перебором всех установленных подключений. Эта логика демонстрируется в следующем коде, который передает событие OnTextChanged():

```
procedure TServerWithEvents.MemoChange(Sender: TObject);
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  EC := GetConnectionEnumerator;
  if EC <> nil then begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then
        (ConnectData.pUnk as
         IServerWithEventsEvents).OnTextChanged(
         (Sender as TMemo).Text);
  end;
end;
```

Наконец, чтобы позволить клиентам подключиться к единственному объекту автоматизации, необходимо вызвать функцию API COM RegisterActiveObject(). Этой функции передаются следующие параметры: интерфейс IUnknown для объекта, идентификатор класса объекта (CLSID), флаг строгой регистрации (строгая регистрация подразумевает наличие у сервера метода AddRef, а нестрогая допускает его отсутствие) и дескриптор, который возвращается по ссылке:

```
RegisterActiveObject(Self as IUnknown, Class_ServerWithEvents,
                     ACTIVEOBJECT_WEAK, FObjRegHandle);
```

Все эти программные фрагменты связываются воедино в модуле ServAuto, полный исходный код которого представлен в листинге 15.10.

Листинг 15.10. Модуль `ServAuto.pas`

```
unit ServAuto;

interface

uses
  ComObj, ActiveX, AxCtrls, Server_TLB;

type
  TServerWithEvents = class(TAutoObject,
                            IConnectionPointContainer,
                            IServerWithEvents)
  private
    { Закрытые объявления }
    FConnectionPoints: TConnectionPoints;
    FObjRegHandle: Integer;
    procedure MemoChange(Sender: TObject);
  protected
    { Защищенные объявления }
    procedure AddText(const NewText: WideString); safecall;
    procedure Clear; safecall;
    function GetConnectionEnumerator: IEnumConnections;
    property ConnectionPoints: TConnectionPoints
                            read FConnectionPoints
                            implements IConnectionPointContainer;
  public
    destructor Destroy; override;
    procedure Initialize; override;
    end;

implementation

uses Windows, ComServ, ServMain, SysUtils, StdCtrls;

destructor TServerWithEvents.Destroy;
begin
  inherited Destroy;
  // Удаление объекта из таблицы ROT
  RevokeActiveObject(FObjRegHandle, nil);
end;

procedure TServerWithEvents.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
                                            ckMulti, EventConnect);
  // Направить событие OnChange поля memo главной
  // формы методу MemoChange:
  MainForm.Memo.OnChange := MemoChange;
  // Зарегистрировать этот объект с помощью
  // таблицы ROT (Running Object Table), чтобы другие клиенты
  // могли подключиться к этому экземпляру.
  RegisterActiveObject(Self as IUnknown, Class_ServerWithEvents,
                       ACTIVEOBJECT_WEAK, FObjRegHandle);
```

```
end;

procedure TServerWithEvents.Clear;
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  MainForm.Memo.Lines.Clear;
  EC := GetConnectionEnumerator;
  if EC <> nil then begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then
        (ConnectData.pUnk as IServerWithEventsEvents).OnClear;
  end;
end;

procedure TServerWithEvents.AddText(const NewText: WideString);
begin
  MainForm.Memo.Lines.Add(NewText);
end;

procedure TServerWithEvents.MemoChange(Sender: TObject);
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  EC := GetConnectionEnumerator;
  if EC <> nil then begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then
        (ConnectData.pUnk as
         IServerWithEventsEvents).OnTextChanged(((Sender as
         TMemo).Text);
  end;
end;

function TServerWithEvents.GetConnectionEnumerator:↳
                                              IEnumConnections;
var
  Container: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Result := nil;
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID,
           CP));
    CP.EnumConnections(Result);
end;

initialization
  TAutoObjectFactory.Create(ComServer, TServerWithEvents,
                            Class_ServerWithEvents,
                            ciMultiInstance, tmApartment);
end.
```

В программное обеспечение клиентов также нужно внести небольшие изменения, чтобы позволить клиентам подключаться к активному экземпляру, если он уже запущен. Эти действия реализованы с помощью функции API COM `GetActiveObject()`:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  ActiveObj: IUnknown;
begin
  // Получить активный объект, если он доступен, в противном
  // случае создать новый
  GetActiveObject(Class_ServerWithEvents, nil, ActiveObj);
  if ActiveObj <> nil then
    FServer := ActiveObj as IServerWithEvents
  else
    FServer := CoServerWithEvents.Create;
  FEventSink := TEventSink.Create(Self);
  InterfaceConnect(FServer, IServerWithEventsEvents,
                   FEventSink, FCookie);
end;
```

На рис. 15.15 показаны клиенты, принимающие события от одного-единственного сервера.

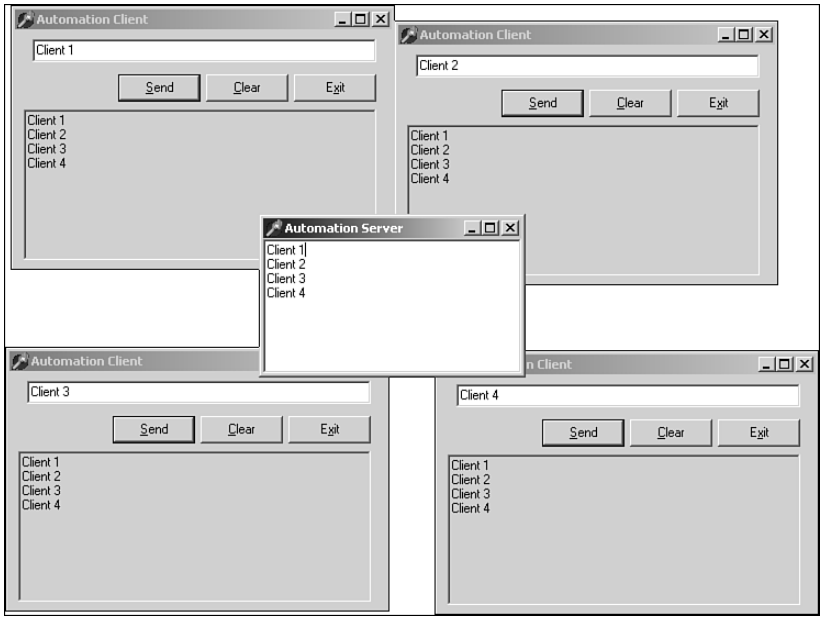

Рис. 15.15. *Несколько клиентов, манипулирующих одним и тем же сервером и принимают его события*

Коллекции автоматизации

Прямо скажем: нас, программистов, со всех сторон окружают программные объекты, которые используются в качестве контейнеров для других программных объектов. Задумайтесь, как велико их разнообразие — будь то массив, список (компонент `TList`), коллекция (компонент `TCollection`), класс контейнера шаблона C++ или вектор Java. Кажется, что мы постоянно только то и делаем, что подыскиваем оптимальный вариант пресловутой мышеловки для программных объектов, которая бы наилучшим образом справлялась с задачей хранения других программных объектов. Если оценить время, затраченное на создание идеального контейнерного класса, то станет ясно, что это одна из самых важных проблем, занимающих умы разработчиков. А почему бы и нет? Подобное логическое разделение контейнера и его содержимого помогает лучше организовать алгоритмы и создает вполне приемлемое соответствие реальному миру (в корзинке могут лежать яйца, в кармане — деньги, на стоянке можно спокойно оставлять автомобили и т.д.). При изучении нового языка или модели разработки всегда приходится знакомиться с "новым способом" управления группами некоторых элементов. Это и есть та самая мысль, к которой мы вас подводили: подобно любой другой модели разработки программных продуктов, модель COM также имеет свои способы управления собственными разновидностями групп элементов. Чтобы добиться эффективности в разработке приложений COM, необходимо знать, как обращаться с такими объектами.

При работе с интерфейсом `IDispatch` модель COM определяет два основных метода, с помощью которых представляется категория контейнера: массивы и коллекции. Те, кому уже приходилось иметь дело с технологией автоматизации или элементами ActiveX в Delphi, скорее всего, уже знакомы с массивами. В Delphi создание массивов автоматизации не представляет затруднений. Для этого достаточно добавить свойство массива в потомок интерфейса `IDispatch` или в диспинтерфейс, как показано в следующем примере:

```
type
  IMyDisp = interface(IDispatch)
    function GetProp(Index: Integer): Integer; safecall;
    procedure SetProp(Index, Value: Integer); safecall;
    property Prop[Index: Integer]: Integer read GetProp
                                           write SetProp;
  end;
```

Массивы полезны во многих ситуациях, но им свойственны некоторые ограничения. Например, массивы удобны, когда существуют логически объединенные данные, к которым можно получить доступ, используя фиксированный индекс, например при доступе к строкам в экземпляре типа `IStrings`. Но, если природа данных такова, что отдельные элементы часто удаляются, добавляются или перемещаются, массив оказывается не слишком удобным контейнером. Классический пример — группа активных окон. Поскольку окна постоянно создаются, удаляются и изменяют свой порядок, не существует надежного критерия для определения последовательности их размещения в массиве.

Для решения этой проблемы и были разработаны коллекции. Они позволяют манипулировать набором элементов таким способом, который не предполагает какого бы то ни было порядка следования или номеров элементов. Необычность коллекций заключается в том, что в действительности не существует объекта или интерфейса

коллекции (collection), вместо этого коллекция представляется как пользовательский интерфейс IDispatch, который соблюдает некоторый набор правил и принципов. Итак, чтобы интерфейс IDispatch можно было квалифицировать как коллекцию, необходимо придерживаться следующих правил.

- Коллекции должны содержать свойство _NewEnum, возвращающее интерфейс IUnknown объекту, который поддерживает интерфейс IEnumVARIANT. Интерфейс IEnumVARIANT будет использован для перечисления элементов в коллекции. Обратите внимание на то, что имя этого свойства должно начинаться с символа подчеркивания, а само свойство — быть отмечено в библиотеке типов как *restricted* (ограниченное). Диспетчерский идентификатор DispID для свойства _NewEnum должен быть равен значению DISPID_NEWENUM (-4), и определен он будет в редакторе библиотеки типов Delphi следующим образом:

```
function _NewEnum: IUnknown [propget, dispid $FFFFFFFC,
                             restricted]; safecall;
```

- Языки, поддерживающие конструкцию For..Each (например Visual Basic), будут использовать данный метод для получения интерфейса IEnumVARIANT, необходимого для перечисления элементов коллекции. (Более подробная информация по этой теме приведена в настоящей главе далее.)
- Коллекции должны обладать методом Item(), который на основании индекса возвращает из коллекции элемент. Идентификатор диспетчера DispID для этого метода должен быть равен 0, и его следует отметить флагом *стандартного элемента коллекции* (default collection element). Если бы понадобилось реализовать коллекцию указателей на интерфейсы IFoo, определение этого метода в редакторе библиотеки типов выглядело бы примерно так:

```
function Item(Index: Integer): IFoo [propget,
     dispid $00000000,defaultcollelem]; safecall;
```

  Обратите внимание: параметр Index вполне может иметь тип OleVariant, что позволяет индексировать элементы с помощью значения типа Integer, WideString или любого другого типа.

- Коллекции должны обладать свойством Count, которое возвращает количество элементов в коллекции. Этот метод обычно определяется в редакторе библиотеки типов следующим образом:

```
function Count: Integer [propget, dispid $00000001]; safecall;
```

Кроме вышеизложенных правил, при создании собственных объектов коллекций желательно придерживаться следующих рекомендаций.

- Свойство или метод, возвращающие коллекцию, должны иметь имя, представляющее собой форму множественного числа, образованного от имени элементов коллекции. Например если есть свойство, возвращающее коллекцию элементов списка, то в качестве имени этого свойства подошло бы слово Items (если элемент коллекции имеет имя Item). Аналогично элемент по имени Foot (нога) будет входить в свойство-коллекцию по имени Feet (ноги). В тех редких случаях, когда форма множественного числа для нужного слова совпадает с формой един-

ственного числа (например, для коллекции элементов `fish` или `deer`), имя свойства коллекции должно состоять из имени элемента, к которому присоединяется слово "`Collection`" (`FishCollection` или `DeerCollection`).

- Коллекции, которые допускают добавление элементов, должны осуществлять это с помощью метода `Add()`. Параметры для данного метода варьируются в зависимости от реализации, но имеет смысл предусмотреть передачу параметров, которые определяют исходную позицию нового элемента внутри коллекции. Метод `Add()` обычно возвращает ссылку на элемент, добавленный в коллекцию.
- Коллекции, которые допускают удаление элементов, должны осуществлять это с помощью метода `Remove()`. Данному методу достаточно передать один параметр, представляющий собой индекс удаляемого элемента, причем семантически формат этого индекса должен совпадать с методом `Item()`.

Реализация в Delphi

Тем, кому приходилось создавать элементы управления ActiveX в Delphi, вероятно известно, что в раскрывающемся списке мастера ActiveX Control Wizard перечислено меньше элементов управления, чем представлено в палитре компонентов IDE. Дело в том, что компания *Borland* не допускает отображения в списке мастера некоторых элементов управления благодаря использованию функции `RegisterNonActiveX()`. Примером элемента управления, который доступен в палитре компонентов, но не в окне мастера, может служить элемент `TListView`, расположенный во вкладке Win32 палитры. Причина "сокрытия" мастером элемента управления `TListView` состоит в том, что мастер "не знает", что ему делать со свойством `Items` данного компонента, которое имеет тип `TListItems`. Поскольку мастер не знает, какой контейнер необходим для этого типа свойства элемента управления ActiveX, то такой элемент просто исключается из списка мастера, чтобы не создавать совершенно бесполезную оболочку для элемента управления ActiveX.

Однако в случае с элементом `TListView` функция `RegisterNonActiveX()` вызывается с флагом `axrComponentOnly`, который означает, что потомок компонента `TListView` будет помещен в список мастера ActiveX Control Wizard. Предприняв некоторые дополнительные меры по созданию, казалось бы, бездействующего потомка компонента `TListView` по имени `TListView2` и добавив его в палитру, можно получить возможность создавать впоследствии элементы управления ActiveX, инкапсулирующие элемент управления `TListView`. Потом, конечно, все равно придется столкнуться с той же самой проблемой отказа мастера создавать оболочки для свойства `Items` и созданный элемент управления ActiveX придется признать бесполезным. К счастью, при создании элемента управления ActiveX никто не принуждает ограничиться кодом, созданным мастером, и в этом случае можно самостоятельно заняться свойством `Items`, чтобы сделать элемент управления полезным. Возможно, читатель уже начал догадываться, что идеальным способом для инкапсуляции свойства `Items` элемента `TListView` является коллекция.

Чтобы реализовать подобную коллекцию элементов списка, необходимо создать новые объекты, представляющие как отдельный элемент, так и всю коллекцию, а затем добавить новое свойство в стандартный интерфейс элемента управления ActiveX, который будет содержать данную коллекцию. Начнем с определения объекта, представляющего собой элемент списка. Присвоим ему имя `ListItem`. Первым шагом

разработки объекта ListItem будет создание нового объекта автоматизации с помощью соответствующей пиктограммы во вкладке ActiveX диалогового окна New Items. После создания этого объекта заполним его свойства и методы в редакторе библиотеки типов. В целях демонстрации добавим в элемент TListView свойства Caption, Index, Checked и SubItems. Аналогичным образом создадим второй новый объект автоматизации для самой коллекции и присвоим ему имя ListItems. Пополним его описанными выше методами _NewEnum, Item(), Count(), Add() и Remove(). И, наконец, в стандартный интерфейс элемента управления ActiveX добавим новое свойство по имени Items, которое будет возвращать коллекцию.

После того как интерфейсы IListItem и IListItems будут полностью определены в редакторе библиотеки типов, потребуется вручную внести небольшие изменения в файлы реализации, автоматически созданные для этих объектов. Стандартным родительским классом для нового объекта автоматизации является класс TAutoObject; но новые объекты будут создаваться только внутренне (т.е. без помощи фабрики класса), поэтому изменим вручную тип предка и укажем в качестве базового класс TAutoInfObject, который больше подходит для объектов автоматизации создаваемых внутренне. Кроме того, поскольку эти объекты не создаются с помощью фабрики класса, из модулей можно удалить код инициализации, реализующий ненужные (в данном случае) фабрики.

Теперь, когда вся инфраструктура приняла надлежащий вид, пора приняться за реализацию объектов ListItem и ListItems. Объект ListItem — самый простой, поскольку он представляет собой незатейливую оболочку вокруг элемента списка. Текст модуля, содержащего этот объект, представлен в листинге 15.11.

Листинг 15.11. Оболочка для элемента Listview

```
unit LVItem;

interface

uses
  ComObj, ActiveX, ComCtrls, LVCtrl_TLB, StdVcl, AxCtrls;

type
  TListItem = class(TAutoIntfObject, IListItem)
  private
    FListItem: ComCtrls.TListItem;
  protected
    function Get_Caption: WideString; safecall;
    function Get_Index: Integer; safecall;
    function Get_SubItems: IStrings; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    procedure Set_SubItems(const Value: IStrings); safecall;
    function Get_Checked: WordBool; safecall;
    procedure Set_Checked(Value: WordBool); safecall;
  public
    constructor Create(AOwner: ComCtrls.TListItem);
  end;

implementation
```

```
uses ComServ;

constructor TListItem.Create(AOwner: ComCtrls.TListItem);
begin
  inherited Create(ComServer.TypeLib, IListItem);
  FListItem := AOwner;
end;

function TListItem.Get_Caption: WideString;
begin
  Result := FListItem.Caption;
end;

function TListItem.Get_Index: Integer;
begin
  Result := FListItem.Index;
end;

function TListItem.Get_SubItems: IStrings;
begin
  GetOleStrings(FListItem.SubItems, Result);
end;

procedure TListItem.Set_Caption(const Value: WideString);
begin
  FListItem.Caption := Value;
end;

procedure TListItem.Set_SubItems(const Value: IStrings);
begin
  SetOleStrings(FListItem.SubItems, Value);
end;

function TListItem.Get_Checked: WordBool;
begin
  Result := FListItem.Checked;
end;

procedure TListItem.Set_Checked(Value: WordBool);
begin
  FListItem.Checked := Value;
end;

end.
```

Заметьте: в конструктор передается объект `ComCtrls.TListItem`, который будет служить в качестве элемента списка, контролируемого данным объектом автоматизации.

Реализация объекта коллекции `ListItems` сложнее, но не намного. Поскольку этот объект должен обеспечивать объектную поддержку типа `IEnumVARIANT`, чтобы реализовать свойство `_NewEnum`, тип `IEnumVARIANT` обеспечен прямо в этом объекте. Следо-

вательно, класс TListItems поддерживает оба интерфейса: IListItems и IEnumVARIANT. Интерфейс IEnumVARIANT содержит четыре метода описанных в табл. 15.2.

Таблица 15.2. Методы интерфейса IEnumVARIANT

Метод	*Назначение*
Next	Возвращает следующие n элементов коллекции
Skip	Пропускает n элементов коллекции
Reset	Переходит к первому элементу коллекции
Clone	Создает копию интерфейса IEnumVARIANT

Исходный код модуля, содержащего объект ListItems, представлен в листинге 15.12.

Листинг 15.12. Оболочка для элементов ListView

```
unit LVItems;

interface

uses
  ComObj, Windows, ActiveX, ComCtrls, LVCtrl_TLB, StdVcl;

type
  TListItems = class(TAutoIntfObject, IListItems, IEnumVARIANT)
  private
    FListItems: ComCtrls.TListItems;
    FEnumPos: Integer;
  protected
    { Методы IListItems }
    function Add: IListItem; safecall;
    function Get_Count: Integer; safecall;
    function Get_Item(Index: Integer): IListItem; safecall;
    procedure Remove(Index: Integer); safecall;
    function Get__NewEnum: IUnknown; safecall;
    { Методы IEnumVariant }
    function Next(celt: LongWord; var rgvar : OleVariant;
                  out pceltFetched: LongWord): HResult; stdcall;
    function Skip(celt: LongWord): HResult; stdcall;
    function Reset: HResult; stdcall;
    function Clone(out Enum: IEnumVariant): HResult; stdcall;
  public
    constructor Create(AOwner: ComCtrls.TListItems);
  end;

implementation

uses ComServ, LVItem;
```

```
{ TListItems }

constructor TListItems.Create(AOwner: ComCtrls.TListItems);
begin
  inherited Create(ComServer.TypeLib, IListItems);
  FListItems := AOwner;
end;

{ TListItems.IListItems }

function TListItems.Add: IListItem;
begin
  Result := LVItem.TListItem.Create(FListItems.Add);
end;

function TListItems.Get__NewEnum: IUnknown;
begin
  Result := Self;
end;

function TListItems.Get_Count: Integer;
begin
  Result := FListItems.Count;
end;

function TListItems.Get_Item(Index: Integer): IListItem;
begin
  Result := LVItem.TListItem.Create(FListItems[Index]);
end;

procedure TListItems.Remove(Index: Integer);
begin
  FListItems.Delete(Index);
end;

{ TListItems.IEnumVariant }

function TListItems.Clone(out Enum: IEnumVariant): HResult;
begin
  Enum := nil;
  Result := S_OK;
  try
    Enum := TListItems.Create(FListItems);
  except
    Result := E_OUTOFMEMORY;
  end;
end;

function TListItems.Next(celt: LongWord; var rgvar : OleVariant;
                 out pceltFetched: LongWord): HResult; stdcall;
var
  V: OleVariant;
  PV: PVariantArg;
```

```
  I: Integer;
begin
  Result := S_FALSE;
  try
    if @pceltFetched <> nil then pceltFetched := 0;
    PV := @rgvar;
    for I := 0 to celt - 1 do begin
      if FEnumPos >= FListItems.Count then Exit;
      V := Get_Item(FEnumPos);
      PV^ := TVariantArg(V);
      // Прием, позволяющий защитить вариант от "сбора мусора".
      // Поскольку вариант является частью массива elt, его нельзя
      // ликвидировать.
      TVarData(V).VType := varEmpty;
      TVarData(V).VInteger := 0;
      Inc(PV);
      Inc(FEnumPos);
      if @pceltFetched <> nil then Inc(pceltFetched);
    end;
  except
  end;
  if (@pceltFetched = nil) or ((@pceltFetched <> nil) and
     (pceltFetched = celt)) then Result := S_OK;
end;

function TListItems.Reset: HResult;
begin
  FEnumPos := 0;
  Result := S_OK;
end;

function TListItems.Skip(celt: LongWord): HResult;
begin
  Inc(FEnumPos, celt);
  Result := S_OK;
end;

end.
```

В этом модуле единственным методом с нетривиальной реализацией является метод `Next()`. Параметр `celt` метода `Next()` указывает, сколько элементов должно быть возвращено. Параметр `elt` содержит массив типа `TVarArgs`, содержащий по крайней мере `elt` элементов. При возврате параметр `pceltFetched` (если не равен значению `nil`) должен хранить реальное количество выбранных элементов. Этот метод возвращает значение `S_OK`, если количество возвращаемых элементов совпадает с числом затребованных; в противном случае возвращается значение `S_FALSE`. Логика данного метода состоит в просмотре массива `elt` и присвоении очередному его элементу объекта типа `TVarArg`, представляющего собой элемент коллекции. Обратите внимание на маленький трюк, который выполняется для очистки варианта типа `OleVariant` после присваивания его массиву. Это гарантирует, что массив не подвергнется автоматическому удалению из памяти (“уборке мусора”). Если бы это не было

сделано, то содержимое массива elt могло бы быть удалено из памяти, при завершении работы и освобождении объектов типа OleVariant, на которые ссылается переменная V.

Подобно классу TListItem, конструктору класса TListItems передается в качестве параметра объект ComCtrls.TListItems, который участвует в реализации различных методов этого класса.

И, наконец, для завершения реализации элемента управления ActiveX необходимо добавить механизм управления свойством Items. Прежде всего к объекту следует добавить поле для хранения коллекции:

```
type
  TListViewX = class(TActiveXControl, IListViewX)
  private
    ...
    FItems: IListItems;
    ...
  end;
```

Затем в методе InitializeControl() присваиваем переменной FItems новый экземпляр объекта класса TListItems:

```
FItems := LVItems.TListItems.Create(FDelphiControl.Items);
```

Наконец, метод Get_Items() можно реализовать с помощью простого возврата значения переменной FItems:

```
function TListViewX.Get_Items: IListItems;
begin
  Result := FItems;
end;
```

Для проверки реальной работоспособности этой коллекции можно загрузить элемент управления в среде Visual Basic 6 и попробовать использовать конструкцию For..Each для работы с этой коллекцией. На рис. 15.16 показан результат запуска простого тестового приложения в VB.

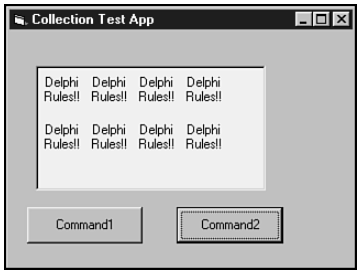

Рис. 15.16. *Приложение Visual Basic проверяет созданную коллекцию*

На рис. 15.16 представлены две кнопки. Кнопка Command1 предназначена для добавления элементов в коллекцию listview, а кнопка Command2 – для опроса всех

элементов коллекции с помощью конструкции `For..Each` и добавления восклицательных знаков к каждому из них. Вот как выглядит реализация этих методов:

```
Private Sub Command1_Click()
  ListViewX1.Items.Add.Caption = "Delphi"
End Sub

Private Sub Command2_Click()
  Dim Item As ListItem
  Set Items = ListViewX1.Items
  For Each Item In Items
    Item.Caption = Item.Caption + " Rules!!"
  Next

End Sub
```

Несмотря на чувства, питаемые некоторыми программистами Delphi к Visual Basic, мы должны помнить, что приложения Visual Basic — это главный потребитель разрабатываемых ими элементов управления ActiveX, и очень важно гарантировать надлежащее функционирование таких элементов управления в данной среде.

Коллекции обеспечивают обширные возможности, которые позволяют создаваемым элементам управления и серверам автоматизации более естественно функционировать в мире COM. Поскольку коллекции чрезвычайно трудны для реализации, поэтому стоит потратить время на их освоение — чтобы научиться использовать их в любых подходящих случаях. К сожалению, вполне возможно, что, освоив в совершенстве работу с коллекциями, вы узнаете, что нашелся какой-то умник, который придумал новый, более мощный объект контейнера для модели COM.

Новые типы интерфейсов в библиотеке типов

Как и подобает каждому приличному разработчику Delphi, для определения новых экземпляров объектов автоматизации до сих пор использовался редактор библиотек типов. Но можно легко попасть в ситуацию, когда один из методов для нового интерфейса включает в себя параметр типа интерфейса COM, который не поддерживается по умолчанию в редакторе библиотеки типов. А поскольку редактор библиотеки типов не позволяет работать с типами, о которых он не знает, то как же тогда завершить такое определение метода?

Прежде чем перейти к подробностям, важно понять причины подобного поведения редактора библиотеки типов. Если при создании нового метода в редакторе библиотеки типов просмотреть типы, доступные в разделе Type вкладки Parameters, то можно увидеть такие интерфейсы, как `IDataBroker`, `IDispatch`, `IEnumVARIANT`, `IFont`, `IPicture`, `IProvider`, `IStrings` и `IUnknown`. Почему же доступны только эти интерфейсы? Что делает их такими особенными? На самом деле в них нет ничего необычного — просто они являются типами, которые определены в библиотеках типов, используемых данной библиотекой типов. По умолчанию библиотека типов Delphi автоматически использует библиотеку типов Borland Standard VCL и библиотеку типов OLE Automation. Для определения конфигурации собственной библиотеки типов выделите корневой узел в изображении дерева на левой панели редактора библиотеки типов и перейдите во вкладку Uses на правой панели. Типы, содержа-

щиеся в библиотеках типов, используемых данной библиотекой, станут автоматически доступными в раскрывающемся списке редактора библиотеки типов.

Зная все это, нетрудно догадаться о том, что, если интерфейс, который необходимо использовать в качестве рассматриваемого параметра метода, определен в библиотеке типов, то можно просто использовать эту библиотеку типов — и проблема будет решена. Но как быть, если интерфейс в библиотеке типов не определен? Ведь существует довольно много интерфейсов COM, которые определяются только инструментальными средствами *комплекта разработчика программного обеспечения* (SDK — Software Development Kit) в файлах заголовков или IDL и не содержатся в библиотеках типов. В этом случае лучше всего определить параметр метода с помощью типа `IUnknown`. Интерфейс типа `IUnknown` может быть опрошен с помощью метода `QueryInterface` непосредственно в реализации метода, что позволит определить конкретный тип интерфейса, с которым предстоит работать. Необходимо также обязательно описать этот параметр метода как тип `IUnknown`, что должно обеспечить поддержку соответствующего интерфейса. Следующий фрагмент кода служит примером реализации такого метода:

```
procedure TSomeClass.SomeMethod(SomeParam: IUnknown);
var
  Intf: ISomeComInterface;
begin
  Intf := SomeParam as ISomeComInterface;
  // Остальная часть реализации метода
end;
```

Напомним, что интерфейс, к которому приводят тип `IUnknown`, должен быть интерфейсом, для которого модели COM известен способ передачи параметров (маршалинга). Это означает, что он должен быть либо определен в какой-нибудь библиотеке типов, либо должен быть типом, совместимым со стандартным маршалером автоматизации, либо рассматриваемый сервер COM должен предоставить библиотеку DLL с заместителем-заглушкой (proxy/stub), способной выполнить маршалинг интерфейса.

Обмен двоичными данными

Рано или поздно, но все же придется осуществлять обмен блоками двоичных данных между клиентом и сервером автоматизации. Поскольку модель COM не поддерживает операций обмена обычными указателями, решить задачу с помощью привычной операции передачи указателя не удастся. Поэтому воспользуемся другим, более сложным способом. Самый простой способ обмена двоичными данными между клиентами и серверами автоматизации состоит в использовании массива байтов типа `SafeArray`. Delphi успешно инкапсулирует массивы этого типа в переменных типа `OleVariant`. Достаточно изощренный пример подробных действий показан в листингах 15.13 и 15.14, содержащих код модулей клиента и сервера, в которых поля текстового редактора используются для демонстрации методов передачи двоичных данных, представленных в виде массивов байт типа `SafeArray`.

Листинг 15.13. Модуль сервера

```
unit ServObj;

interface

uses
  ComObj, ActiveX, Server_TLB;

type
  TBinaryData = class(TAutoObject, IBinaryData)
  protected
    function Get_Data: OleVariant; safecall;
    procedure Set_Data(Value: OleVariant); safecall;
  end;

implementation

uses ComServ, ServMain;

function TBinaryData.Get_Data: OleVariant;
var
  P: Pointer;
  L: Integer;
begin
  // Переместить данные из поля memo в массив
  L := Length(MainForm.Memo.Text);
  Result := VarArrayCreate([0, L - 1], varByte);
  P := VarArrayLock(Result);
  try
    Move(MainForm.Memo.Text[1], P^, L);
  finally
    VarArrayUnlock(Result);
  end;
end;

procedure TBinaryData.Set_Data(Value: OleVariant);
var
  P: Pointer;
  L: Integer;
  S: string;
begin
  // Переместить данные из массива в поле memo
  L := VarArrayHighBound(Value, 1) -
       VarArrayLowBound(Value, 1) + 1;
  SetLength(S, L);
  P := VarArrayLock(Value);
  try
    Move(P^, S[1], L);
  finally
    VarArrayUnlock(Value);
  end;
  MainForm.Memo.Text := S;
end;
```

```
initialization
  TAutoObjectFactory.Create(ComServer, TBinaryData,
                            Class_BinaryData,
                            ciSingleInstance, tmApartment);
end.
```

Листинг 15.14. Модуль клиента

```
unit CliMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, Server_TLB;

type
  TMainForm = class(TForm)
    Memo: TMemo;
    Panel1: TPanel;
    SetButton: TButton;
    GetButton: TButton;
    OpenButton: TButton;
    OpenDialog: TOpenDialog;
    procedure OpenButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SetButtonClick(Sender: TObject);
    procedure GetButtonClick(Sender: TObject);
  private
    FServer: IBinaryData;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FServer := CoBinaryData.Create;
end;

procedure TMainForm.OpenButtonClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    Memo.Lines.LoadFromFile(OpenDialog.FileName);
end;

procedure TMainForm.SetButtonClick(Sender: TObject);
```

```
var
  P: Pointer;
  L: Integer;
  V: OleVariant;
begin
  // Послать серверу данные поля memo
  L := Length(Memo.Text);
  V := VarArrayCreate([0, L - 1], varByte);
  P := VarArrayLock(V);
  try
    Move(Memo.Text[1], P^, L);
  finally
    VarArrayUnlock(V);
  end;
  FServer.Data := V;
end;

procedure TMainForm.GetButtonClick(Sender: TObject);
var
  P: Pointer;
  L: Integer;
  S: string;
  V: OleVariant;
begin
  // Получить от сервера данные поля memo
  V := FServer.Data;
  L := VarArrayHighBound(V, 1) - VarArrayLowBound(V, 1) + 1;
  SetLength(S, L);
  P := VarArrayLock(V);
  try
    Move(P^, S[1], L);
  finally
    VarArrayUnlock(V);
  end;
  Memo.Text := S;
end;

end.
```

За кулисами: языковая поддержка COM

В разговорах, касающихся разработки приложений COM в Delphi, часто можно услышать о сильной поддержке, предоставляемой языком Object Pascal модели COM. С этим трудно спорить, если учесть, что в язык встроена поддержка таких элементов, как интерфейсы, варианты и длинные строки. Но что же реально означает "поддержка, встроенная в язык"? Как работают эти средства и в чем природа их зависимости от функций API COM? В настоящем разделе рассмотрим, как все эти вещи объединяются на низком уровне для обеспечения поддержки COM в языке Object Pascal, и разберемся в некоторых деталях реализации этих языковых средств.

Как уже говорилось, все средства поддержки COM в языке Object Pascal можно разделить на три основные категории.

- Типы данных `Variant` и `OleVariant`, которые инкапсулируют в модели COM вариантные записи, массивы `SafeArray` и автоматизацию с поздним связыванием.
- Тип данных `WideString`, который инкапсулирует в модели COM строки `BSTR`.
- Типы `Interface` и `dispinterface`, которые инкапсулируют интерфейсы COM и автоматизацию с ранним связыванием, а также *автоматизацию со связыванием на уровне идентификаторов* (ID-bound Automation).

Разработчики OLE со стажем (со времен Delphi 2), вероятно, заметили, что зарезервированное слово `automated`, благодаря которому могли создаваться серверы автоматизации позднего связывания, практически игнорируется. Это произошло вследствие того, что эта функция была отодвинута на задний план средствами "настоящей" поддержки автоматизации, впервые введенной в Delphi 3. Теперь она осталась только для совместимости с прежними версиями, а потому здесь ее рассматривать не будем.

Варианты

Варианты представляют собой старейшую форму поддержки технологии COM в Delphi, впервые появившуюся в Delphi 2. По сути, тип `Variant` — это просто большая запись, которая используется для передачи некоторых данных одного из многочисленных допустимых типов. Эта запись определена в модуле `System` как тип `TVarData`:

```
type
  PVarData = ^TVarData;
  TVarData = record
    VType: Word;
    Reserved1, Reserved2, Reserved3: Word;
    case Integer of
      varSmallint: (VSmallint: Smallint);
      varInteger:  (VInteger: Integer);
      varSingle:   (VSingle: Single);
      varDouble:   (VDouble: Double);
      varCurrency: (VCurrency: Currency);
      varDate:     (VDate: Double);
      varOleStr:   (VOleStr: PWideChar);
      varDispatch: (VDispatch: Pointer);
      varError:    (VError: LongWord);
      varBoolean:  (VBoolean: WordBool);
      varUnknown:  (VUnknown: Pointer);
      varByte:     (VByte: Byte);
      varString:   (VString: Pointer);
      varAny:      (VAny: Pointer);
      varArray:    (VArray: PVarArray);
      varByRef:    (VPointer: Pointer);
  end;
```

Значение поля `VType` этой записи означает тип данных, содержащийся в типе `Variant`, и это может быть любое из обозначений типа варианта, приведенных в начале модуля `System` и перечисленных в разделе `variant` этой записи (внутри оператора `case`). Единственным различием между типами `Variant` и `OleVariant` является то, что тип `Variant` поддерживает все стандартные типы, а тип `OleVariant` — только те из них, которые совместимы с автоматизацией. Например, вполне приемлемо

присвоить стандартный для языка Pascal тип `String` (`varString`) типу `Variant`, но для присвоения той же строки типу `OleVariant` придется преобразовать ее в совместимый с автоматизацией тип `WideString` (`varOleStr`).

При работе с типами `Variant` и `OleVariant` компилятор в действительности обрабатывает и передает экземпляры записей типа `TVarData`. И в самом деле, всегда можно безопасно привести тип `Variant` или `OleVariant` к типу `TVarData`, если по некоторым причинам необходимо манипулировать внутренними составляющими такой записи (хотя делать этого не рекомендуется).

В суровом мире программирования COM на языках C и C++ (за рамками классов) варианты представлены структурой `VARIANT`, определенной в файле заголовка `oaidl.h`. При работе с вариантами в этой среде приходится вручную инициализировать и управлять ими с помощью функций API `VariantXXX()` из библиотеки `oleaut32.dll`. (Речь идет о функциях `VariantInit()`, `VariantCopy()`, `VariantClear()` и т.д.) Это делает работу с вариантами в языках C и C++ достаточно сложной задачей.

Поскольку поддержка вариантов встроена непосредственно в язык Object Pascal, компилятор организует необходимые обращения к процедурам поддержки вариантов автоматически, по мере использования экземпляров данных типа `Variant` или `OleVariant`. Но за комфорт в языке приходится расплачиваться необходимостью приобретения новых знаний. Если рассмотреть *таблицу импорта* (import table) "ничего не делающего" исполняемого файла `EXE` Delphi с помощью такого инструмента, как утилита *Borland* `TDUMP.EXE` или утилита *Microsoft* `DUMPBIN.EXE`, то можно заметить несколько подозрительных операций импорта из библиотеки `oleaut32.dll`: `VariantChangeTypeEx()`, `VariantCopyInd()` и `VariantClear()`. Это означает, что даже в приложении, в котором типы `Variant` или `OleVariant` не используются явно, исполняемый файл Delphi все равно зависит от функций API COM из библиотеки `oleaut32.dll`.

Массивы вариантов

Массивы вариантов в Delphi разработаны для инкапсуляции массивов COM типа `SafeArray`, которые представляют собой тип записи, используемой для инкапсуляции массива данных в автоматизации. Они названы *безопасными* (safe), поскольку способны описать сами себя. Помимо данных массива, эта запись содержит информацию о количестве измерений массива, размере и количестве его элементов. Массивы `Variant` создаются и управляются в Delphi с помощью функций и процедур `VarArray`*`XXX`*`()`, определенных в модуле `System` и описанных в интерактивной справочной системе. Эти функции и процедуры являются, по сути, оболочками для функций API `SafeArray`*`XXX`*`()`. Если переменная типа `Variant` содержит массив `Variant`, то для доступа к элементам такого массива используется стандартный синтаксис индексации массива. Сравнивая эти средства с возможностями языков C и C++ для массивов `safearray`, где все приходится делать вручную, можно заметить, что использованная в языке Object Pascal инкапсуляция отличается ясностью, компактностью и надежностью.

Автоматизация с поздним связыванием

Как уже упоминалось в настоящей главе, типы `Variant` и `OleVariant` позволяют писать приложения-клиенты, использующие автоматизацию с поздним связыванием (позднее связывание означает, что функции вызываются во время выполнения с помощью метода `Invoke` интерфейса `IDispatch`). Все это легко можно принять за чистую

монету, но вот вопрос: "Где же та магическая связь между вызовом метода сервера автоматизации из переменной типа Variant в программе и самим методом IDispatch.Invoke(), каким-то образом вызванным с правильными параметрами?". Ответ находится на уровне более низком, чем можно было бы ожидать.

После вызова метода объектами типа Variant или OleVariant, содержащими интерфейс IDispatch, компилятор просто генерирует обращение к вспомогательной функции _DispInvoke (объявленной в модуле System), которая передает управление по значению указателя на функцию, имя которого VarDispProc. По умолчанию указатель VarDispProc указывает на метод, который при вызове просто возвращает ошибку. Но, если в директиву uses включить модуль ComObj, то в разделе initialization модуля ComObj указатель VarDispProc будет перенаправлен на другой метод с помощью следующего оператора:

```
VarDispProc := @VarDispInvoke;
```

VarDispInvoke — это процедура в модуле ComObj, имеющая следующее объявление:

```
procedure VarDispInvoke(Result: PVariant; const Instance: Variant;
                        CallDesc: PCallDesc; Params: Pointer); cdecl;
```

Реализация этой процедуры обеспечивает вызов метода IDispatch.GetIDsOfNames() для получения диспетчерского идентификатора (DispID) на основе имени метода, корректно устанавливает требуемые параметры и выполняет обращение к методу IDispatch.Invoke(). Самое интересное заключается в том, что компилятор в данном случае не обладает никакими внутренними "знаниями" об интерфейсе IDispatch или о том, как осуществляется вызов метода Invoke(). Он просто передает управление в соответствии со значением указателя на функцию. Также интересно то, что благодаря подобной архитектуре вполне возможно перенаправить этот указатель функции на свою собственную процедуру, если необходимо самостоятельно обрабатывать все вызовы автоматизации через типы Variant и OleVariant. При этом следует позаботиться лишь о том, чтобы объявление пользовательской функции совпадало с объявлением процедуры VarDispInvoke. Безусловно, это — задача не для новичков, но полезно знать, что при необходимости вполне можно воспользоваться и таким гибким подходом.

Тип данных WideString

Тип данных WideString был введен в Delphi 3 с двойной целью: для поддержки двухбайтовых символов Unicode и для поддержки символьных строк, совместимых со строками BSTR COM. Тип WideString отличается от близкого типа AnsiString по нескольким основным параметрам.

- Все символы, входящие в строку типа WideString, имеют размер, равный двум байтам.
- Для типов WideString память всегда выделяется с помощью функции SysAllocStringLen(), в следствии чего они полностью совместимы со строками BSTR.
- Для типов WideString никогда не ведется подсчет ссылок, поэтому при присвоении значения переменных этого типа всегда копируются.

Подобно вариантам, работа со строками BSTR при использовании стандартных функций API отличается громоздкостью, поэтому встроенная в язык Object Pascal

поддержка типа WideString вносит заметное упрощение. Однако, поскольку эти строки требуют двойного объема памяти и не поддерживают ссылок, они менее эффективны по сравнению со строками типа AnsiString. С учетом вышесказанного, прежде чем их использовать, хорошо взвесьте все за и против.

Подобно типу Pascal Variant, наличие типа WideString приводит к автоматическому импортированию некоторых функций из библиотеки oleaut32.dll даже в том случае, если программист и не использует этот тип. При изучении таблицы импорта "ничего не делающего" приложения Delphi оказывается, что функции SysStringLen(), SysFreeString(), SysReAllocStringLen() и SysAllocStringLen() задействованы библиотекой RTL Delphi для обеспечения поддержки типа WideString.

Интерфейсы

Возможно, наиболее важным элементом реализации модели COM в языке Object Pascal является встроенная поддержка интерфейсов. Ирония судьбы состоит в том, что если менее масштабные средства (имеются в виду типы Variant и WideString) реализуются непосредственным использованием функций интерфейса API COM, то при реализации интерфейсов в языке Object Pascal функции интерфейса API COM вообще не нужны. То есть в языке Object Pascal имеется абсолютно самодостаточная реализация интерфейсов, которая полностью соответствует спецификации COM, но при этом не использует ни одной функции интерфейса API COM.

В плане соответствия спецификации COM все интерфейсы в Delphi косвенно происходят от интерфейса IUnknown. А интерфейс IUnknown, как известно, обеспечивает средства определения типа и поддержку учета ссылок, что является основой основ модели COM. Это означает, что знание особенностей интерфейса IUnknown встроено непосредственно в компилятор, а сам интерфейс IUnknown определен в модуле System. Сделав интерфейс IUnknown полноправным членом языка программирования, среда Delphi приобрела способность организовывать автоматический подсчет ссылок, обязав компилятор генерировать обращения к функциям IUnknown.AddRef() и IUnknown.Release() в соответствующие моменты времени. Кроме того, оператор as может быть использован в качестве ускоренного варианта определения типа интерфейса, обычно реализуемого с помощью метода QueryInterface(). Но встроенная поддержка интерфейса IUnknown оказывается просто незначительным фрагментом, если рассмотреть весь объем низкоуровневой поддержки, обеспечиваемой языком и компилятором для интерфейсов в целом.

На рис. 15.17 показана упрощенная схема внутренней поддержки интерфейсов со стороны классов. В действительности объект Delphi — это ссылка, которая указывает на физический экземпляр. Первых четыре байта экземпляра объекта представляют собой указатель на *таблицу виртуальных методов* объекта (VMT — Virtual Method Table). При положительном смещении от значения VMT находятся все виртуальные методы объекта. С отрицательным смещением размещаются те указатели на методы и данные, которые важны для внутреннего функционирования объекта. В частности, на уровне смещения -72 от значения VMT содержится указатель на таблицу интерфейсов объекта. Таблица интерфейсов представляет собой список записей типа PInterfaceEntry (определенных в модуле System), которые, по сути, содержат идентификаторы интерфейса IID и информацию о том, где найти указатель vtable для данного идентификатора интерфейса IID.

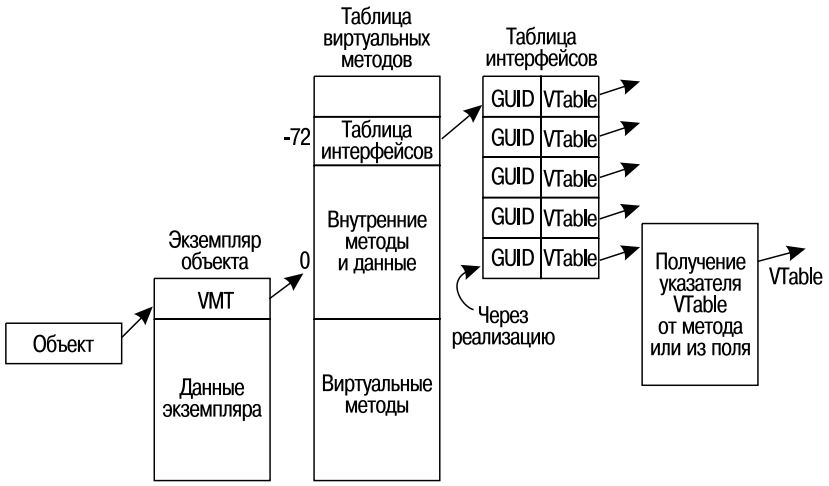

Рис. 15.17. *Поддержка интерфейсов внутренними средствами языка Object Pascal*

Рассмотрев показанную на рис. 15.17 схему, можно понять, как увязаны друг с другом отдельные элементы. Например, метод `QueryInterface()` обычно реализуется в объектах Object Pascal с помощью вызова метода `TObject.GetInterface()`. Метод `GetInterface()` просматривает таблицу интерфейсов в надежде найти нужный идентификатор интерфейса IID и возвращает указатель виртуальной таблицы (указатель `vtable`) для этого интерфейса. Теперь понятно, почему новые типы интерфейсов должны быть определены с помощью уникального идентификатора GUID — ведь в противном случае метод `GetInterface()` не сможет найти их при просмотре таблицы интерфейсов, и, следовательно, получение интерфейса с помощью метода `QueryInterface()` будет невозможным. Приведение типов интерфейсов с помощью оператора `as` просто создает обращение к методу `QueryInterface()`, поэтому и здесь применяются те же самые правила.

Последняя запись в таблице интерфейсов (см. рис. 15.17) представляет собой внутреннюю реализацию интерфейса на основе применения директивы `implements`. Вместо прямого указателя для виртуальной таблицы (указателя `vtable`) запись таблицы интерфейсов содержит адрес небольшой функции, создаваемой компилятором, которая возвращает виртуальную таблицу интерфейса из свойства, для которого была использована директива `implements`.

Диспинтерфейсы

Диспинтерфейс обеспечивает инкапсуляцию недвойственного (non-dual) интерфейса `IDispatch`, т.е. интерфейса `IDispatch`, в котором методы могут быть вызваны только через метод `Invoke()`, но не через виртуальную таблицу. В этом отношении диспинтерфейс аналогичен автоматизации с вариантами. Однако диспинтерфейсы чуть более эффективны, чем варианты, поскольку объявления `dispinterface` содержат диспетчерский идентификатор DispID для каждого поддерживаемого свойства или метода. Это означает, что метод `IDispatch.Invoke()` можно вызвать на-

прямую, без предварительного вызова метода IDispatch.GetIDsOfNames(), как это происходит в случае с вариантами. В остальном же механизм работы диспинтерфейсов аналогичен механизму работы вариантов: при вызове метода через диспинтерфейс компилятор генерирует обращение к функции _IntfDispCall из модуля System. Данный метод передает управление указателю DispCallByIDProc, который по умолчанию возвращает только ошибку. Но при включении в раздел uses модуля ComObj указатель DispCallByIDProc инициализируется адресом процедуры DispCallByID(), которая объявлена в модуле ComObj следующим образом:

```
procedure DispCallByID(Result: Pointer; const Dispatch: IDispatch;
                       DispDesc: PDispDesc; Params: Pointer); cdecl;
```

Класс TOleContainer

Теперь, рассмотрев основы технологий ActiveX и OLE, ознакомимся с классом Delphi TOleContainer. Класс TOleContainer определен в модуле OleCntrs и инкапсулирует подробности работы с контейнерами документов OLE и ActiveX в простом и удобном компоненте библиотеки VCL.

НА ЗАМЕТКУ

Тем, кто уже знаком с использованием компонента TOleContainer в Delphi 1, не следует пропускать этот раздел, поскольку в Delphi 2 этот компонент был полностью переделан и все знания о разработке 16-разрядных приложений в Delphi будут практически неприменимы при разработке 32-разрядных приложений. Но не стоит расстраиваться: 32-разрядная версия этого компонента проще в применении, а код, который необходимо написать для поддержки объекта в новой версии, гораздо короче того, который приходилось создавать ранее.

Пример простого приложения

Теперь приступим к созданию приложения обладающего контейнером OLE. Создайте новый проект, щелкните на вкладке System в палитре компонентов и поместите в форму компонент TOleContainer. Щелкните правой кнопкой мыши на созданном объекте в окне конструктора форм и в появившемся контекстном меню выберите пункт Insert Object. Раскроется диалоговое окно Insert Object (Вставка объекта), показанное на рис. 15.18.

Внедрение нового объекта OLE

По умолчанию в диалоговом окне Insert Object содержатся имена приложений OLE-серверов, зарегистрированных в Windows. Для внедрения нового объекта OLE необходимо из списка Object Type (Тип объекта) выбрать приложение-сервер. Это запустит OLE-сервер, что позволит создать новый объект OLE, который будет вставлен в объект TOleContainer. При закрытии приложения-сервера объект TOleContainer будет обновлен, и в нем появится изображение внедренного объекта. Например, можно создать новый документ MS Word 2000, как показано на рис. 15.19.

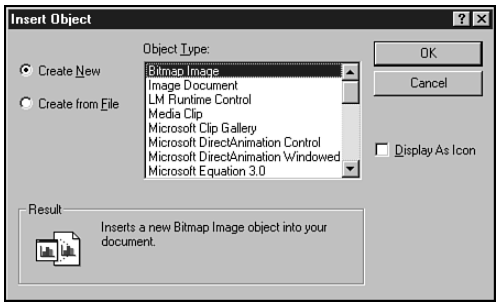

Рис. 15.18. *Диалоговое окно* Insert Object

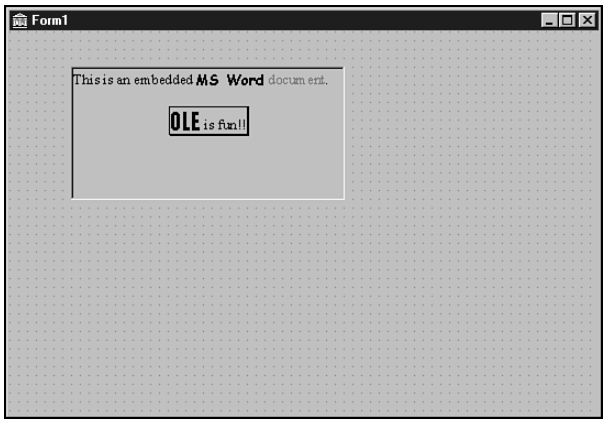

Рис. 15.19. *Внедренный документ MS Word 2000*

НА ЗАМЕТКУ

Объект OLE не может быть активизирован во время разработки. Объекта `TOleContainer` работает только во время выполнения.

Чтобы открыть диалоговое окно Insert Object во время выполнения, можно вызвать метод `InsertObjectDialog()` класса `TOleContainer`, который определен следующим образом:

```
function InsertObjectDialog: Boolean;
```

Эта функция возвращает значение `True`, если новый тип объекта OLE был успешно выбран из списка диалогового окна Insert Object.

Внедрение или связывание существующего файла OLE

Для внедрения существующего файла OLE в объект `TOleContainer` установите переключатель в диалоговом окне Insert Object в положение Create From File (Создать

из файла), а затем выберите существующий файл, как показано на рис. 15.20. Выбранный файл будет вести себя как новый объект OLE.

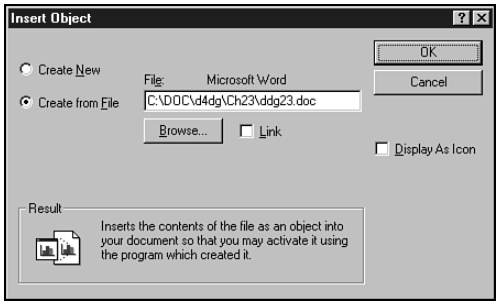

Рис. 15.20. *Вставка объекта из файла*

Для внедрения файла во время выполнения необходимо вызвать метод `CreateObjectFromFile()` класса `TOleContainer`, который определен следующим образом:

```
procedure CreateObjectFromFile(const FileName: string;
                               Iconic: Boolean);
```

Для связывания (но не внедрения) объекта OLE просто установите флажок Link (Связь) в диалоговом окне Insert Object (см. рис. 15.20). Как уже было сказано, это действие приводит к созданию связи между приложением и файлом OLE, благодаря которой можно будет просматривать и редактировать один и тот же связанный объект сразу из нескольких приложений.

Для связывания файла с приложением во время выполнения вызовите метод `CreateLinkToFile()` класса `TOleContainer`, который определен следующим образом:

```
procedure CreateLinkToFile(const FileName: string;
                           Iconic: Boolean);
```

Пример более сложного приложения

Познакомившись с основами технологии OLE и классом `TOleContainer`, можно приступить к рассмотрению примера, который по-настоящему продемонстрирует возможности использования OLE в реальных приложениях.

Начнем с создания нового проекта, по шаблону приложения MDI (Multiple-Document Interface — многодокументный интерфейс). После применения стандартного шаблона MDI главная форма этого приложения была немного модифицирована (рис. 15.21).

Дочерняя форма MDI показана на рис. 15.22. Это пример формы стиля `fsMDIChild` с компонентом `TOleContainer`, помещенным в форму с параметром выравнивания `alClient` (Занять всю клиентскую область).

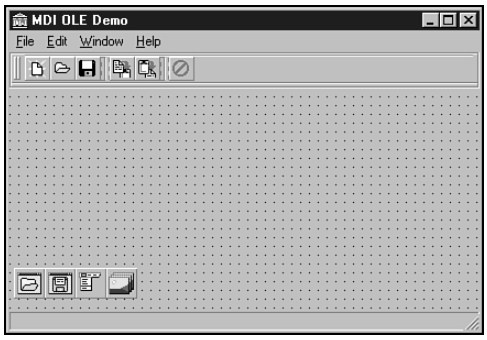

Рис. 15.21. *Главное окно демонстрационного приложения MDI с объектом OLE*

Рис. 15.22. *Дочернее окно демонстрационного приложения MDI с объектом OLE*

В листинге 15.15 приведен исходный код модуля дочерней формы приложения MDI (`ChildWin.pas`). Заметьте, этот модуль имеет стандартный вид, за исключением добавленного свойства `OLEFileName`, связанного с ним метода и закрытой переменной экземпляра. Свойство `OLEFileName` предназначено для хранения пути и имени файла OLE, а соответствующий метод доступа к свойству устанавливает заголовок дочерней формы равным имени файла.

Листинг 15.15. Исходный код модуля `ChildWin.pas`

```
unit Childwin;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls,
OleCtnrs;

type
  TMDIChild = class(TForm)
    OleContainer: TOleContainer;
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
  private
    FOLEFilename: string;
```

```
    procedure SetOLEFileName(const Value: string);
  public
    property OLEFileName: string read FOLEFileName
                                write SetOLEFileName;
  end;

implementation

{$R *.DFM}

uses Main, SysUtils;

procedure TMDIChild.SetOLEFileName(const Value: string);
begin
  if Value <> FOLEFileName then begin
    FOLEFileName := Value;
    Caption := ExtractFileName(FOLEFileName);
  end;
end;

procedure TMDIChild.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  Action := caFree;
end;

end.
```

Создание дочерней формы

При создании новой дочерней формы приложения MDI (меню File пункт New) с помощью вызова описанного выше метода `InsertObjectDialog()` открывается диалоговое окно Insert Object. Заголовок дочерней формы MDI устанавливается с помощью глобальной переменной `NumChildren`, обеспечивающей уникальный номер. В приведенном ниже фрагменте кода показан метод `CreateMDIChild()` главной формы:

```
procedure TMainForm.FileNewItemClick(Sender: TObject);
begin
  inc(NumChildren);
  { Создайть новое дочернее окно MDI }
  with TMDIChild.Create(Application) do begin
    Caption := 'Untitled' + IntToStr(NumChildren);
    { Открыть диалог объекта OLE и осуществить вставку в
      дочернее окно }
    OleContainer.InsertObjectDialog;
  end;

end;
```

Чтение и запись в файл

Как уже отмечалось в этой главе, объекты OLE позволяют записывать и считывать информацию из потоков, а следовательно, и из файлов. Компонент `TOleContainer`

обладает методами SaveToStream(), LoadFromStream(), SaveToFile() и LoadFromFile(), которые упрощают сохранение объектов OLE в файле или потоке.

Главная форма приложения MDIOLE содержит методы, позволяющие открывать и сохранять файлы объектов OLE. В приведенном ниже фрагменте кода показан метод FileOpenItemClick(), который вызывается при выборе в меню File главной формы пункта Open. Кроме загрузки из файла объекта OLE, заданного свойством OpenDialog.FileName, данный метод также присваивает полю OleFileName экземпляра TMDIChild имя этого файла. Если при загрузке файла возникает ошибка, то экземпляр формы будет освобожден:

```
procedure TMainForm.FileOpenItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    with TMDIChild.Create(Application) do begin
      try
        OleFileName := OpenDialog.FileName;
        OleContainer.LoadFromFile(OleFileName);
        Show;
      except
        Release;  // При ошибке освободить форму
        raise;    // Передать исключение дальше
      end;
    end;
end;
```

В следующем фрагменте приведен код, выполняемый при выборе в меню File пункта Save As или Save. Заметьте, что метод FileSaveItemClick() вызывает метод FileSaveAsItemClick(), если активной дочерней форме MDI еще не присвоено имя.

```
procedure TMainForm.FileSaveAsItemClick(Sender: TObject);
begin
  if (ActiveMDIChild <> Nil) and (SaveDialog.Execute) then
    with TMDIChild(ActiveMDIChild) do begin
      OleFileName := SaveDialog.FileName;
      OleContainer.SaveToFile(OleFileName);
    end;
end;

procedure TMainForm.FileSaveItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> Nil then
    { Если имя не присвоено, выполнить "save as", }
    if TMDIChild(ActiveMDIChild).OLEFileName = '' then
      FileSaveAsItemClick(Sender)
    { в противном случае сохранить под текущим именем. }
    else
      with TMDIChild(ActiveMDIChild) do
        OleContainer.SaveToFile(OLEFileName);
end;
```

Использование буфера обмена для копирования и вставки

Благодаря универсальному механизму передачи данных, описанному ранее в этой главе, для передачи объектов OLE можно также использовать буфер обмена Windows. И вновь при решении таких задач не обойтись без компонента `TOleContainer`.

Копирование объекта OLE из компонента `TOleContainer` в буфер обмена — тривиальная задача. Для этого достаточно вызвать метод `Copy()`, как показано в следующем фрагменте кода:

```
procedure TMainForm.CopyItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> Nil then
    TMDIChild(ActiveMDIChild).OleContainer.Copy;
end;
```

После помещения объекта OLE в буфер обмена необходим только один дополнительный шаг для его правильного считывания в компонент `TOleContainer`. Перед вставкой содержимого буфера обмена в объект `TOleContainer` следует сначала проверить значение свойства `CanPaste`, чтобы удостовериться, что данные, содержащиеся в буфере обмена, представляют собой допустимый объект OLE. После этого можно открыть диалоговое окно Paste Special (Специальная вставка) для вставки объекта в компонент `TOleContainer`, вызвав метод `PasteSpecialDialog()`, как показано в приведенном ниже фрагменте кода. Диалоговое окно Paste Special показано на рис. 15.23.

```
procedure TMainForm.PasteItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    with TMDIChild(ActiveMDIChild).OleContainer do
      { Перед вызовом диалогового окна удостоверьтесь, что в
        буфере обмена находится допустимый объект OLE. }
      if CanPaste then PasteSpecialDialog;
end;
```

Когда приложение будет запущено, сервер, управляющий объектом OLE в активной дочерней форме MDI, объединит меню и панели инструментов приложений. На рис. 15.24 и 15.25 показаны средства активизации объекта OLE — приложение MDI OLE управляется двумя различными серверами OLE.

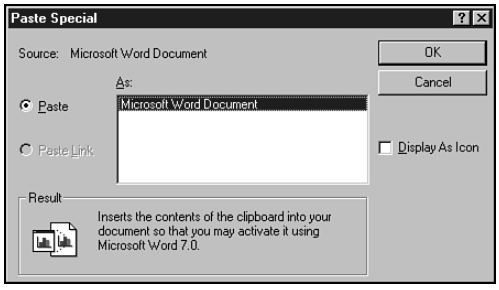

Рис. 15.23. *Диалоговое окно* Paste Special

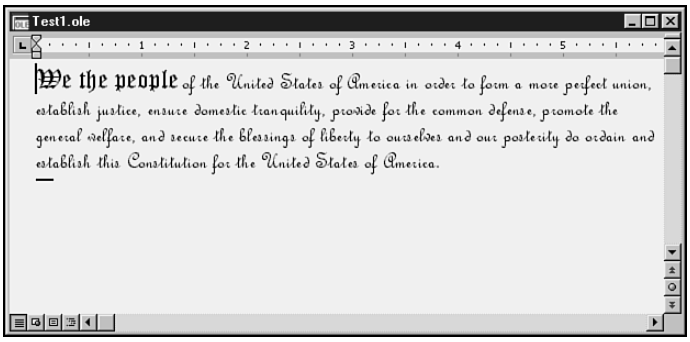

Рис. 15.24. *Редактирование внедренного документа MS Word 2000*

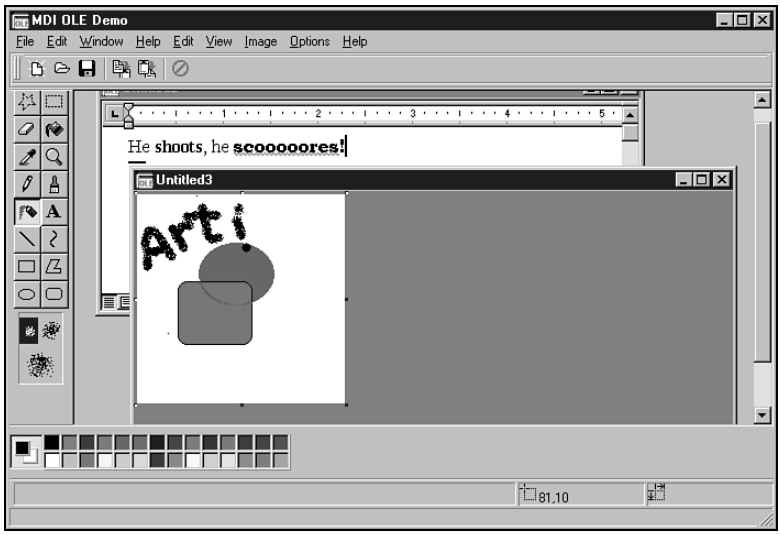

Рис. 15.25. *Редактирование внедренного рисунка Paint*

Полный исходный код главного модуля приложения MDI OLE (`Main.pas`) приведен в листинге 15.16.

Листинг 15.16. Исходный код модуля `Main.pas`

```
unit Main;

interface

uses WinTypes, WinProcs, SysUtils, Classes, Graphics, Forms,
Controls, Menus, StdCtrls, Dialogs, Buttons, Messages, ExtCtrls,
ChildWin, ComCtrls, ToolWin, ImgList;

type
  TMainForm = class(TForm)
```

```
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    FileNewItem: TMenuItem;
    FileOpenItem: TMenuItem;
    FileCloseItem: TMenuItem;
    Window1: TMenuItem;
    Help1: TMenuItem;
    N1: TMenuItem;
    FileExitItem: TMenuItem;
    WindowCascadeItem: TMenuItem;
    WindowTileItem: TMenuItem;
    WindowArrangeItem: TMenuItem;
    HelpAboutItem: TMenuItem;
    OpenDialog: TOpenDialog;
    FileSaveItem: TMenuItem;
    FileSaveAsItem: TMenuItem;
    Edit1: TMenuItem;
    PasteItem: TMenuItem;
    WindowMinimizeItem: TMenuItem;
    SaveDialog: TSaveDialog;
    CopyItem: TMenuItem;
    CloseAll1: TMenuItem;
    StatusBar: TStatusBar;
    CoolBar1: TCoolBar;
    ToolBar1: TToolBar;
    OpenBtn: TToolButton;
    SaveBtn: TToolButton;
    ToolButton3: TToolButton;
    CopyBtn: TToolButton;
    PasteBtn: TToolButton;
    ToolButton6: TToolButton;
    ExitBtn: TToolButton;
    ImageList1: TImageList;
    NewBtn: TToolButton;
    procedure FormCreate(Sender: TObject);
    procedure FileNewItemClick(Sender: TObject);
    procedure WindowCascadeItemClick(Sender: TObject);
    procedure UpdateMenuItems(Sender: TObject);
    procedure WindowTileItemClick(Sender: TObject);
    procedure WindowArrangeItemClick(Sender: TObject);
    procedure FileCloseItemClick(Sender: TObject);
    procedure FileOpenItemClick(Sender: TObject);
    procedure FileExitItemClick(Sender: TObject);
    procedure FileSaveItemClick(Sender: TObject);
    procedure FileSaveAsItemClick(Sender: TObject);
    procedure PasteItemClick(Sender: TObject);
    procedure WindowMinimizeItemClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure HelpAboutItemClick(Sender: TObject);
    procedure CopyItemClick(Sender: TObject);
    procedure CloseAll1Click(Sender: TObject);
  private
    procedure ShowHint(Sender: TObject);
```

```
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses About;

var
  NumChildren: Cardinal = 0;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := ShowHint;
  Screen.OnActiveFormChange := UpdateMenuItems;
end;

procedure TMainForm.ShowHint(Sender: TObject);
begin
  { Отображать подсказки на строке состояния. }
  StatusBar.SimpleText := Application.Hint;
end;

procedure TMainForm.FileNewItemClick(Sender: TObject);
begin
  inc(NumChildren);
  { Создать новое дочернее окно MDI }
  with TMDIChild.Create(Application) do begin
    Caption := 'Untitled' + IntToStr(NumChildren);
    { Открыть диалог объекта OLE и осуществить вставку
      в дочернее окно. }
    OleContainer.InsertObjectDialog;
  end;
end;

procedure TMainForm.FileOpenItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    with TMDIChild.Create(Application) do begin
      try
        OleFileName := OpenDialog.FileName;
        OleContainer.LoadFromFile(OleFileName);
        Show;
      except
        Release;  // При ошибке освободить форму
        raise;    // Передать исключение дальше
      end;
    end;
end;
```

```
procedure TMainForm.FileCloseItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.FileSaveAsItemClick(Sender: TObject);
begin
  if (ActiveMDIChild <> nil) and (SaveDialog.Execute) then
    with TMDIChild(ActiveMDIChild) do begin
      OleFileName := SaveDialog.FileName;
      OleContainer.SaveToFile(OleFileName);
    end;
end;

procedure TMainForm.FileSaveItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    { Если имя не присвоено, выполнить "save as", }
    if TMDIChild(ActiveMDIChild).OLEFileName = '' then
      FileSaveAsItemClick(Sender)
    { в противном случае сохранить под текущим именем. }
    else
      with TMDIChild(ActiveMDIChild) do
        OleContainer.SaveToFile(OLEFileName);
end;

procedure TMainForm.FileExitItemClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.PasteItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    with TMDIChild(ActiveMDIChild).OleContainer do
      { Перед вызовом диалогового окна удостоверьтесь, что в
        буфере обмена находится допустимый объект OLE. }
      if CanPaste then PasteSpecialDialog;
end;

procedure TMainForm.WindowCascadeItemClick(Sender: TObject);
begin
  Cascade;
end;

procedure TMainForm.WindowTileItemClick(Sender: TObject);
begin
  Tile;
end;

procedure TMainForm.WindowArrangeItemClick(Sender: TObject);
begin
```

```
  ArrangeIcons;
end;

procedure TMainForm.WindowMinimizeItemClick(Sender: TObject);
var
  I: Integer;
begin
  { Обратный проход массива MDIChildren. }
  for I := MDIChildCount - 1 downto 0 do
    MDIChildren[I].WindowState := wsMinimized;
end;

procedure TMainForm.UpdateMenuItems(Sender: TObject);
var
  DoIt: Boolean;
begin
  DoIt := MDIChildCount > 0;
  { Только допустимые параметры, если это активное дочернее окно }
  FileCloseItem.Enabled := DoIt;
  FileSaveItem.Enabled := DoIt;
  CloseAll1.Enabled := DoIt;
  FileSaveAsItem.Enabled := DoIt;
  CopyItem.Enabled := DoIt;
  PasteItem.Enabled := DoIt;
  CopyBtn.Enabled := DoIt;
  SaveBtn.Enabled := DoIt;
  PasteBtn.Enabled := DoIt;
  WindowCascadeItem.Enabled := DoIt;
  WindowTileItem.Enabled := DoIt;
  WindowArrangeItem.Enabled := DoIt;
  WindowMinimizeItem.Enabled := DoIt;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  Screen.OnActiveFormChange := nil;
end;

procedure TMainForm.HelpAboutItemClick(Sender: TObject);
begin
  with TAboutBox.Create(Self) do begin
    ShowModal;
    Free;
  end;
end;

procedure TMainForm.CopyItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    TMDIChild(ActiveMDIChild).OleContainer.Copy;
end;

procedure TMainForm.CloseAll1Click(Sender: TObject);
```

```
begin
  while ActiveMDIChild <> nil do begin
    // Использовать метод Release, а не Free!
    ActiveMDIChild.Release;
    // Пусть Windows примет необходимые меры
    Application.ProcessMessages;
  end;
end;

end.
```

Резюме

В этой большой по объему главе рассматривались основы технологий COM, OLE и ActiveX. Знание этих основ поможет понять процессы, происходящие при выполнении приложений, созданных с использованием данных технологий. Кроме того, было дано некоторое представление о различных типах COM-ориентированных клиентов и серверов, а также разновидностях технологий автоматизации в Delphi. Помимо глубокого рассмотрения технологий COM и автоматизации, желающие могли познакомиться с работой компонента библиотеки VCL `TOleContainer`.

За дополнительной информацией о технологиях COM и ActiveX обращайтесь к другим главам этой книги. В главе 16, “Программирование для оболочки Windows”, описаны примеры создания реального сервера COM, а в главе 18, “Транзакционные методы разработки с применением COM+ и MTS”— транзакционные методы при разработке приложений COM+ в Delphi.

ГЛАВА

16

Программирование для оболочки Windows

В ЭТОЙ ГЛАВЕ...

Впервые представленная в Windows 95 оболочка Windows поддерживается во всех последующих версиях (NT 3.51, 4.0, 5.0 и более поздних версиях Windows 98, 2000, Me и XP). Сильно отличаясь от диспетчера программ (Program Manager) Windows 3.x, оболочка Windows содержит ряд средств, существенно расширяющих ее возможности и позволяющих удовлетворить самые разные запросы пользователей. Однако многие из этих средств, к сожалению, очень плохо документированы. Настоящая глава должна восполнить данный пробел: здесь предлагаются информация и примеры, которые позволят эффективно использовать такие средства, как индикаторы панели задач, панели инструментов приложений, ярлыки и расширения оболочки.

Вывод пиктограммы на панель задач

В настоящем разделе рассматривается методика инкапсуляции *индикатора панели задач* (tray-notification icon) в компонент Delphi. В процессе построения компонента `TTrayNotifyIcon` будет продемонстрировано, что именно необходимо для создания индикатора панели задач с точки зрения API, и то, как решаются некоторые "неразрешимые" проблемы, возникающие при попытке обеспечить функции индикаторов в рамках компонента. Для тех, кто не знаком с понятием "индикатор панели задач", напоминаем, что это небольшие значки, расположенные с правой стороны панели задач Windows (рис. 16.1). Предполагается, что панель задач находится в нижней части экрана.

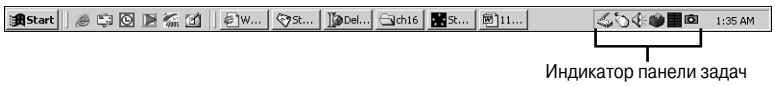

Рис. 16.1. *Индикаторы панели задач находятся в правом нижнем углу экрана*

Интерфейс API

Хотите верьте, хотите нет, но в процессах создания, изменения и удаления индикаторов панели задач выполняется обращение только к одной функции API Win32 `Shell_NotifyIcon()`. Эта и другие функции, имеющие отношение к оболочке Windows, находятся в модуле `ShellAPI`. Функция `Shell_NotifyIcon()` определена следующим образом:

```
function Shell_NotifyIcon(dwMessage: DWORD;
                          lpData: PNotifyIconData): BOOL; stdcall;
```

Параметр `dwMessage` описывает действие, выполняемое над индикатором, и может принимать одно из значений, приведенных в табл. 16.1.

Таблица 16.1. Значения параметра `dwMessage`

Константа	*Значение*	*Назначение*
`NIM_ADD`	`0`	Добавляет индикатор на панель задач
`NIM_MODIFY`	`1`	Модифицирует свойства существующего индикатора
`NIM_DELETE`	`2`	Удаляет индикатор из панели задач

Параметр lpData является указателем на запись типа TNotifyIconData. Эта запись определена следующим образом:

```
type
  TNotifyIconData = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

В поле cbSize хранится размер записи; это поле инициализируется с помощью функции SizeOf(TNotifyIconData).

В параметре Wnd указывается дескриптор окна, которому посылаются "обратные" (callback) сообщения от индикаторов панели задач. (Хотя здесь используется слово *обратный*, тем не менее реально никакого обратного вызова не происходит; однако в документации по Win32 для сообщений, посылаемых окну от имени индикатора панели задач, используется именно такой термин.)

В поле uID задается определяемый программистом уникальный идентификатор (ID) пиктограммы индикатора. Если в приложении для индикатора используется несколько пиктограмм, то нужно идентифицировать каждую из них — для этого и предназначено поле uID.

Значение в поле uFlags определяет, какие из полей записи TNotifyIconData должны учитываться функцией Shell_NotifyIcon(), а следовательно, к каким из свойств пиктограммы будут применены действия, определяемые параметром dwMessage. Этот параметр может быть любой комбинацией флагов, приведенных в табл. 16.2 (для объединения флагов воспользуйтесь ключевым словом or).

Таблица 16.2. Возможные значения флагов поля uFlags

Константа	*Значение*	*Назначение*
NIF_MESSAGE	0	Поле uCallbackMessage используется
NIF_ICON	2	Поле hIcon используется
NIF_TIP	4	Поле szTip используется

Поле uCallbackMessage содержит номер сообщения Windows. Именно это сообщение будет послано окну, определяемому полем Wnd. Для получения значения данного поля обычно вызывается функция RegisterWindowMessage() (или же используется смещение от значения WM_USER). Параметр lParam посылаемого сообщения будет иметь то же значение, что и поле uID, а параметр wParam — содержать сообщение от мыши, посланное при помещении ее указателя на индикатор.

В поле hIcon задается дескриптор пиктограммы, помещаемой на панель задач.

Поле szTip должно включать в себя строку подсказки, оканчивающуюся нулевым символом. Эта строка будет выводиться на экран при подведении указателя мыши к индикатору.

Компонент TTrayNotifyIcon инкапсулирует функцию Shell_NotifyIcon() в методе SendTrayMessage(), код которого приведен ниже.

```
procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD;
                                          Flags: UINT);
{ Этот метод содержит вызов функции API Shell_NotifyIcon. }
begin
  { Заполнить поля записи соответствующими значениями. }
  with Tnd do begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := Tray_Callback;
    hIcon  := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;
```

В данном методе в поле szTip копируется значение закрытого поля строкового типа FHint.

Параметр uID используется для хранения ссылки на параметр Self. Поскольку эти данные будут внесены во все последующие сообщения индикаторов панели задач, распознавание сообщений от каждого отдельного компонента среди нескольких сообщений от других индикаторов не составит никакого труда.

Параметру Wnd присваивается значение свойства HWindow глобальной переменной IconMgr типа TIconMgr. Реализация этого объекта приведена далее в настоящей главе, но пока важно знать, что все сообщения индикаторов панели задач отправляются именно через данный компонент.

Параметру uCallbackMessage присваивается значение сообщения DDGM_TRAYICON, определенное с помощью вызова функции API RegisterWindowMessage(). Это гарантирует, что сообщение DDGM_TRAYICON будет иметь уникальный *идентификатор сообщения* (message ID) в масштабе всей системы. Данная задача решается с помощью следующего фрагмента кода:

```
const
  { Строка для идентификации зарегистрированного
    сообщения Windows. }
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';

initialization
  { Получить уникальный идентификатор сообщения Windows для
    обратного вызова из панели задач. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
```

В параметр hIcon помещается значение, возвращаемое методом ActiveIconHandle(). Этот метод возвращает дескриптор пиктограммы, выбранной в данный момент в свойстве Icon рассматриваемого компонента.

Обработка сообщений

Ранее уже упоминалось, что все сообщения индикаторной области панели задач пересылаются окну, поддерживаемому глобальным объектом `IconMgr`. Такой объект создается и освобождается в разделах `initialization` и `finalization` модуля данного компонента, как показано в приведенном ниже фрагменте кода.

```
initialization
  { Получить уникальный идентификатор сообщения Windows для
    обратного вызова из панели задач. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
  IconMgr := TIconManager.Create;
finalization
  IconMgr.Free;
```

Этот относительно небольшой объект определен следующим образом:

```
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;
```

Окно, которому будут отсылаться сообщения панели задач, создается в конструкторе данного объекта с помощью функции `AllocateHWnd()`:

```
constructor TIconManager.Create;
begin
  FHWindow := AllocateHWnd(TrayWndProc);
end;
```

Метод `TrayWndProc()` выступает в роли процедуры окна для окна, создаваемого в конструкторе. Более подробная информация об этом методе приведена далее в настоящей главе.

Пиктограммы и подсказки

Проще всего отобразить пиктограмму или подсказку с помощью свойств самого компонента. Кроме того, свойство `Icon` имеет тип `TIcon`, а это означает, что при определении его значения можно воспользоваться преимуществами встроенного редактора свойств Delphi, предназначенного для пиктограмм. Поскольку пиктограмма индикатора панели задач видима даже во время разработки, необходимо убедиться в том, что пиктограмма и подсказка могут изменяться динамически. Для этого не потребует особых усилий: нужно лишь убедиться в том, что метод `SendTrayMessage()` вызывается (с помощью сообщения `NIM_MODIFY`) в методе `write` свойств `Hint` и `Icon`.

Ниже приведена реализация методов `write` для этих свойств:

```
procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Метод Write для свойства Icon. }
begin
  FIcon.Assign(Value);  // Установить новую пиктограмму
  if FIconVisible then
    { Изменить пиктограмму на панели задач. }
    SendTrayMessage(NIM_MODIFY, NIF_ICON);
end;

procedure TTrayNotifyIcon.SetHint(Value: String);
{ Метод установки для свойства Hint. }
begin
  if FHint <> Value then begin
    FHint := Value;
    if FIconVisible then
      { Изменить подсказку пиктограммы на панели задач. }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
  end;
end;
```

Обработка щелчков мышью

При работе с индикаторами одной из наиболее сложных задач является обеспечение правильной обработки щелчков мышью. Как можно заметить, многие индикаторы панели задач реагируют на щелчки мышью тремя различными способами:

- открывают окно при одинарном щелчке;
- открывают другое окно (обычно окно свойств) при двойном щелчке;
- вызывают контекстное меню при щелчке правой кнопкой.

Проблема заключается в создании события, представляющего двойной щелчок мыши без предварительного события одиночного щелчка.

В системе сообщений Windows, при выполнении пользователем двойного щелчка левой кнопкой мыши, обладающее фокусом окно получит сразу два сообщения: `WM_LBUTTONDOWN` (нажата левая кнопка) и `WM_LBUTTONDBLCLK` (двойной щелчок на левой кнопке). Для того чтобы позволить программе обработать только двойной щелчок мышью, необходимо предусмотреть механизм задержки обработки сообщения `WM_LBUTTONDOWN` на период времени, достаточный, чтобы убедиться в отсутст-

на период времени, достаточный, чтобы убедиться в отсутствии сообщения о выполнении двойного щелчка.

Нетрудно определить период ожидания, необходимый для того, чтобы убедиться в отсутствии сообщения `WM_LBUTTONDBLCLK` после сообщения `WM_LBUTTONDOWN`. Для этого используется функция API `GetDoubleClickTime()`, которая не имеет никаких параметров и возвращает максимальный промежуток времени (в миллисекундах) между двумя щелчками двойного щелчка мышью, допускаемый панелью управления (Control Panel). Для построения механизма ожидания, устанавливаемого на основании периода ожидания в миллисекундах, возвращаемого функцией `GetDoubleClickTime()`, применяется компонент `TTimer`. Он создается и инициируется в конструкторе компонента `TTrayNotifyIcon` следующим образом:

```
FTimer := TTimer.Create(Self);
with FTimer do begin
  Enabled := False;
  Interval := GetDoubleClickTime;
  OnTimer := OnButtonTimer;
end;
```

По истечении периода ожидания вызывается метод `OnButtonTimer()`, Более подробная информация об этом методе приведена далее в настоящей главе.

Как уже говорилось, сообщения панели задач фильтруются методом `TrayWndProc()` глобального объекта `IconMgr`. Рассмотрим код данного метода.

```
procedure TIconManager.TrayWndProc(var Message: TMessage);
{ Позволяет обрабатывать все "обратные" сообщения панели задач в
контексте компонента. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
  with Message do begin
    { Если это обратное сообщение от панели задач. }
    if (Msg = DDGM_TRAYICON) then begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { Запустить таймер при первом щелчке мышью. Событие
          OnClick будет передано методом OnTimer, гарантируя, что
          двойного щелчка не было. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
        { Сбросить флаг двойного щелчка. Это предотвратит
          одиночный щелчок. }
        WM_LBUTTONDBLCLK: begin
          TheIcon.FNoShowClick := True;
          if Assigned(TheIcon.FOnDblClick) then
            TheIcon.FOnDblClick(Self);
        end;
        WM_RBUTTONDOWN: begin
          if Assigned(TheIcon.FPopupMenu) then begin
          { Вызвать функцию SetForegroundWindow, необходимую API }
            SetForegroundWindow(IconMgr.HWindow);
            { Открыть контекстное меню в позиции курсора. }
            GetCursorPos(Pt);
            TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
            { Отправить сообщение, необходимое API для
              переключения задач. }
            PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
          end;
        end;
      end;   // case
    end      // if
    else
      { Если сообщение не относится к панели задач, то вызвать
        процедуру DefWindowProc. }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
```

```
  end;      // with
end;
```

В этом фрагменте кода задана реакция на сообщения, передаваемые различными событиями: при одинарном щелчке просто запускается таймер, при двойном — устанавливается флаг двойного щелчка перед передачей события OnDblClick, а при щелчке правой кнопкой мыши вызывается контекстное меню, определяемое свойством PopupMenu. Теперь рассмотрим метод OnButtonTimer(), код которого приведен ниже.

```
procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
begin
  { Отключить таймер, ведь он нужен только при первом щелчке. }
  FTimer.Enabled := False;
  { Если второго щелчка так и не последовало, то передать
    одиночный щелчок. }
  if (not FNoShowClick) and Assigned(FOnClick) then
    FOnClick(Self);
  FNoShowClick := False;   // Сбросить флаг
end;
```

В первую очередь этот метод отключает таймер, гарантируя, что щелчок мыши создаст только одно событие. Затем проверяется состояние флага FNoShowClick. Помните, что такой флаг устанавливается в методе OwnerWndProc() при обработке сообщения, вызванного двойным щелчком мыши. Таким образом, событие OnClick будет создано только при условии отсутствия события OnDblClk.

Сокрытие приложения

Еще одним аспектом приложений, связанных с областью индикаторов панели задач, является то, что они не отображаются на панели задач в виде кнопок. Чтобы обеспечить приложение подобной возможностью, в компонент TTrayNotifyIcon введено свойство HideTask, позволяющее пользователю самому решить, отображать или нет приложение на панели задач в виде кнопки. Код метода write для такого свойства приведен ниже. Главную роль здесь играет строка, содержащая вызов процедуры API Win32 ShowWindow(), которой передается свойство Handle объекта Application, а также константа, определяющая, будет ли приложение отображаться на панели задач в виде кнопки.

```
procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ Метод Write для свойства HideTask. }
const
  { Флаги, определяющие, будет приложение отображено или скрыто. }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then begin
    FHideTask := Value;
    { В режиме разработки не делать ничего. }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;
```

В листинге 16.1 приведен код модуля TrayIcon.pas, представляющего собой реализацию компонента TTrayNotifyIcon.

Листинг 16.1. TrayIcon.pas — исходный код компонента TTrayNotifyIcon

```
unit TrayIcon;

interface

uses Windows, SysUtils, Messages, ShellAPI, Classes, Graphics,
Forms, Menus, StdCtrls, ExtCtrls;

type
  ENotifyIconError = class(Exception);

  TTrayNotifyIcon = class(TComponent)
  private
    FDefaultIcon: THandle;
    FIcon: TIcon;
    FHideTask: Boolean;
    FHint: string;
    FIconVisible: Boolean;
    FPopupMenu: TPopupMenu;
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FNoShowClick: Boolean;
    FTimer: TTimer;
    Tnd: TNotifyIconData;
    procedure SetIcon(Value: TIcon);
    procedure SetHideTask(Value: Boolean);
    procedure SetHint(Value: string);
    procedure SetIconVisible(Value: Boolean);
    procedure SetPopupMenu(Value: TPopupMenu);
    procedure SendTrayMessage(Msg: DWORD; Flags: UINT);
    function ActiveIconHandle: THandle;
    procedure OnButtonTimer(Sender: TObject);
  protected
    procedure Loaded; override;
    procedure LoadDefaultIcon; virtual;
    procedure Notification(AComponent: TComponent;
                           Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Icon: TIcon read FIcon write SetIcon;
    property HideTask: Boolean read FHideTask
                               write SetHideTask default False;
    property Hint: String read FHint write SetHint;
    property IconVisible: Boolean read FIconVisible
                          write SetIconVisible default False;
    property PopupMenu: TPopupMenu read FPopupMenu
```

```
                                    write SetPopupMenu;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick
                                      write FOnDblClick;
  end;

implementation

{ TIconManager }
{ Этот класс создает скрытое окно, которое перехватывает и
перенаправляет сообщения индикаторов панели задач. }
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;

var
  IconMgr: TIconManager;
  DDGM_TRAYICON: Cardinal;

constructor TIconManager.Create;
begin
  FHWindow := Classes.AllocateHWnd(TrayWndProc);
end;

destructor TIconManager.Destroy;
begin
  if FHWindow <> 0 then Classes.DeallocateHWnd(FHWindow);
  inherited Destroy;
end;

procedure TIconManager.TrayWndProc(var Message: TMessage);
{ Позволяет обрабатывать все "обратные" сообщения панели задач в
контексте компонента. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
  with Message do begin
    { Если это обратное сообщение от панели задач. }
    if (Msg = DDGM_TRAYICON) then begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { Запустить таймер при первом щелчке мышью.
          Событие OnClick будет передано методом OnTimer,
          гарантируя, что двойного щелчка не было. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
```

```
        { Сбросить флаг двойного щелчка. Это предотвратит
          одиночный щелчок. }
        WM_LBUTTONDBLCLK: begin
          TheIcon.FNoShowClick := True;
          if Assigned(TheIcon.FOnDblClick) then
            TheIcon.FOnDblClick(Self);
        end;
        WM_RBUTTONDOWN: begin
          if Assigned(TheIcon.FPopupMenu) then begin
          { Вызвать функцию SetForegroundWindow, необходимую API }
            SetForegroundWindow(IconMgr.HWindow);
            { Открыть контекстное меню в позиции курсора. }
            GetCursorPos(Pt);
            TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
            { Отправить сообщение, необходимое API для
              переключения задач. }
            PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
          end;
        end;
      end;
    end
    else
      { Если сообщение не относится к панели задач, то вызвать
        процедуру DefWindowProc. }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
  end;
end;

{ TTrayNotifyIcon }

constructor TTrayNotifyIcon.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FIcon := TIcon.Create;
  FTimer := TTimer.Create(Self);
  with FTimer do begin
    Enabled := False;
    Interval := GetDoubleClickTime;
    OnTimer := OnButtonTimer;
  end;
  { Загрузить стандартную пиктограмму окна... }
  LoadDefaultIcon;
end;

destructor TTrayNotifyIcon.Destroy;
begin
  // Удалить пиктограмму
  if FIconVisible then SetIconVisible(False);
  FIcon.Free;                                  // Освободить ресурс
  FTimer.Free;
  inherited Destroy;
end;
```

```
function TTrayNotifyIcon.ActiveIconHandle: THandle;
{ Возвращает дескриптор активного индикатора. }
begin
  { Если ни одна из пиктограмм не загружена, возвратить
    стандартную пиктограмму. }
  if (FIcon.Handle <> 0) then
    Result := FIcon.Handle
  else
    Result := FDefaultIcon;
end;

procedure TTrayNotifyIcon.LoadDefaultIcon;
{ Загружает стандартную пиктограмму окна, которая всегда "под
рукой". Это позволит компоненту использовать логотип Windows в
качестве стандартной пиктограммы, если в свойстве Icon не выбрана
ни одна из пиктограмм. }
begin
  FDefaultIcon := LoadIcon(0, IDI_WINLOGO);
end;

procedure TTrayNotifyIcon.Loaded;
{ Вызывается после загрузки компонента из потока. }
begin
  inherited Loaded;
  { Если предполагается, что пиктограмма видима, создать ее. }
  if FIconVisible then
    SendTrayMessage(NIM_ADD, NIF_MESSAGE or NIF_ICON or NIF_TIP);
end;

procedure TTrayNotifyIcon.Notification(AComponent: TComponent;
                                       Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = PopupMenu) then
    PopupMenu := nil;
end;

procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
{ Таймер используется для отслеживания времени между двумя щелчками
двойного щелчка. Реакция на первый щелчок задерживается на время,
достаточное для того, чтобы удостовериться в отсутствии второго
щелчка. Смысл всех этих манипуляций состоит лишь в одном:
обеспечить независимость событий OnClicks и OnDblClicks. }
begin
  { Отключить таймер, ведь он нужен только при первом щелчке. }
  FTimer.Enabled := False;
  { Если второго щелчка так и не последовало, то передать
    одиночный щелчок. }
  if (not FNoShowClick) and Assigned(FOnClick) then
    FOnClick(Self);
  FNoShowClick := False;   // Сбросить флаг
end;
```

```
procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD;
                                          Flags: UINT);
{ Этот метод содержит вызов функции API Shell_NotifyIcon. }
begin
  { Заполнить поля записи соответствующими значениями. }
  with Tnd do begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := DDGM_TRAYICON;
    hIcon  := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;

procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ Метод Write для свойства HideTask. }
const
  { Флаги, определяющие, будет приложение отображено или скрыто. }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then begin
    FHideTask := Value;
    { В режиме разработки не делать ничего. }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;

procedure TTrayNotifyIcon.SetHint(Value: string);
{ Метод установки для свойства Hint. }
begin
  if FHint <> Value then begin
    FHint := Value;
    if FIconVisible then
      { Изменить подсказку индикатора на панели задач. }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
  end;
end;

procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Метод Write для свойства Icon. }
begin
  FIcon.Assign(Value);  // Установить новую пиктограмму
  { Изменить индикатор на панели задач. }
  if FIconVisible then SendTrayMessage(NIM_MODIFY, NIF_ICON);
end;

procedure TTrayNotifyIcon.SetIconVisible(Value: Boolean);
{ Метод записи для свойства IconVisible }
const
```

```
  { Флаги, для добавления или удаления индикатора панели задач }
  MsgArray: array[Boolean] of DWORD = (NIM_DELETE, NIM_ADD);
begin
  if FIconVisible <> Value then begin
    FIconVisible := Value;
    { Установить соответствующий индикатор. }
    SendTrayMessage(MsgArray[Value],
                    NIF_MESSAGE or NIF_ICON or NIF_TIP);
  end;
end;

procedure TTrayNotifyIcon.SetPopupMenu(Value: TPopupMenu);
{ Метод записи для свойства PopupMenu }
begin
  FPopupMenu := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

const
  { Строка для идентификации зарегистрированного
    сообщения Windows. }
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';

initialization
  { Получить уникальный идентификатор сообщения Windows
    для обратного вызова из панели задач. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
  IconMgr := TIconManager.Create;
finalization
  IconMgr.Free;
end.
```

На рис. 16.2 показан внешний вид индикатора, созданного на панели задач компонентом TTrayNotifyIcon.

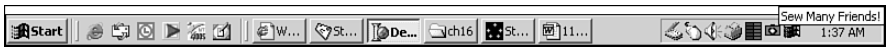

Рис. 16.2. *Компонент TTrayNotifyIcon в действии*

Кстати, так как индикатор инициализируется внутри конструктора компонента, а конструкторы выполняются и во время разработки, то компонент отображает предназначенную для индикатора пиктограмму даже во время разработки приложения!

Пример приложения

Чтобы лучше ознакомиться с компонентом TTrayNotifyIcon, рассмотрим его работу в контексте приложения. На рис. 16.3 изображено главное окно, а в листинге 16.2 – код главного модуля этого приложения.

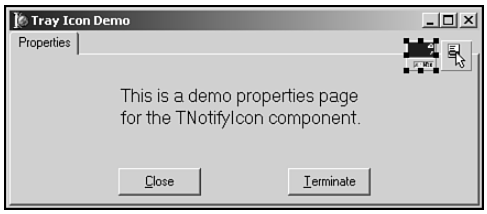

Рис. 16.3. *Приложение индикатора панели задач*

Листинг 16.2. `Main.pas` — главный модуль приложения индикатора

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ShellAPI, TrayIcon, Menus, ComCtrls;

type
  TMainForm = class(TForm)
    pmiPopup: TPopupMenu;
    pgclPageCtl: TPageControl;
    TabSheet1: TTabSheet;
    btnClose: TButton;
    btnTerm: TButton;
    Terminate1: TMenuItem;
    Label1: TLabel;
    N1: TMenuItem;
    Propeties1: TMenuItem;
    TrayNotifyIcon1: TTrayNotifyIcon;
    procedure NotifyIcon1Click(Sender: TObject);
    procedure NotifyIcon1DblClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure btnTermClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.NotifyIcon1Click(Sender: TObject);
begin
  ShowMessage('Single click');
end;
```

```
procedure TMainForm.NotifyIcon1DblClick(Sender: TObject);
begin
  Show;
end;

procedure TMainForm.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  Action := caNone;
  Hide;
end;

procedure TMainForm.btnTermClick(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TMainForm.btnCloseClick(Sender: TObject);
begin
  Hide;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  TrayNotifyIcon1.IconVisible := True;
end;

end.
```

Панели инструментов рабочего стола

Панели инструментов рабочего стола приложений (AppBar — Application Desktop Toolbar) представляют собой окна рабочего стола, которые могут быть *закреплены* у одной из границ экрана (docking). Хотя сам этот термин может показаться и не знакомым, но с объектами, обладающими таким свойством, приходится встречаться каждый раз, работая с компьютером. Пример окна AppBar — панель задач Windows. Как показано на рис. 16.4, панель задач на самом деле даже несколько больше, чем просто окно AppBar, поскольку оно содержит кнопку Start (Пуск), область индикаторов и другие элементы управления.

Рис. 16.4. *Панель задач оболочки Windows*

Помимо свойства закрепления по границам экрана, окна типа AppBar обладают рядом других свойств, присущих панели задач (например, автоматическое сокрытие или возможность перетаскивания с помощью мыши). Но что по-настоящему удивляет — так это размеры интерфейса API окон данного типа — всего лишь одна функция! Следствием подобной компактности, естественно, является ограниченность реали-

зуемых интерфейсом возможностей. Роль интерфейса здесь скорее консультативная, нежели функциональная, т.е. вместо того чтобы управлять окном AppBar с помощью команд типа "сделай это" или "сделай то", окну AppBar передаются запросы наподобие "можно ли сделать это?" и "можно ли сделать то?".

Интерфейс API

Подобно индикаторам области панели задач, для панелей инструментов рабочего стола предусмотрена только одна функция API — `SHAppBarMessage()`. Она определена в модуле `ShellAPI` следующим образом:

```
function SHAppBarMessage(dwMessage: DWORD;
                         var pData: TAppBarData): UINT; stdcall;
```

Первый параметр `dwMessage` может принимать только одно из значений, приведенных в табл. 16.3.

Таблица 16.3. Сообщения `AppBar`

Константа	*Значение*	*Назначение*
ABM_NEW	$0	Регистрирует новое окно AppBar и определяет новое "обратное" сообщение
ABM_REMOVE	$1	Отменяет регистрацию существующего окна AppBar
ABM_QUERYPOS	$2	Запрашивает новую позицию и размер окна AppBar
ABM_SETPOS	$3	Устанавливает новую позицию и размер окна AppBar
ABM_GETSTATE	$4	Возвращает состояния "автосокрытие" и "расположить поверх всех" панели инструментов оболочки
ABM_GETTASKBARPOS	$5	Возвращает позицию панели инструментов оболочки
ABM_ACTIVATE	$6	Уведомляет Windows о создании нового окна AppBar
ABM_GETAUTOHIDEBAR	$7	Возвращает дескриптор скрытого окна AppBar, прикрепленного к одной из границ экрана
ABM_SETAUTOHIDEBAR	$8	Регистрирует скрытое окно AppBar на определенной границе экрана
ABM_WINDOWPOSCHANGED	$9	Информирует Windows об изменении позиции окна AppBar

Параметр `pData` функции `SHAppBarMessage()` представляет собой запись типа `TAppBarData`, которая определена в модуле `ShellAPI` следующим образом:

```
type
  PAppBarData = ^TAppBarData;
  TAppBarData = record
    cbSize: DWORD;
    hWnd: HWND;
    uCallbackMessage: UINT;
    uEdge: UINT;
    rc: TRect;
    lParam: LPARAM; { Специфическая информация сообщения }
  end;
```

В этой записи в поле `cbSize` хранится размер записи; в поле `hWnd` — дескриптор заданного окна AppBar; в поле `uCallbackMessage` — значение сообщения, которое будет послано окну AppBar вместе с уведомляющими сообщениями; в поле `rc` — границы опрашиваемого прямоугольника окна AppBar; в поле `lParam` — дополнительная информация, специфичная для данного сообщения.

СОВЕТ

Более подробную информацию о функции API `SHAppBarMessage()` и о записи типа `TAppBarData` можно найти в интерактивной справочной системе Win32.

Класс TAppBar: форма окна AppBar

Учитывая небольшие размеры функции API окна AppBar, ее нетрудно встроить в форму библиотеки VCL. В настоящем разделе речь пойдет об инкапсуляции функции API AppBar в элементе управления, производном от класса `TCustomForm`. Поскольку компонент `TCustomForm` представляет собой форму, то и в окне конструктора форм (Form Designer) работать с ним будем как с формой верхнего уровня, а не как с отдельным компонентом в составе другой формы.

Бо́льшая часть работы в окне AppBar выполняется с помощью передачи оболочке Windows записи `TAppBarData`, что осуществляется благодаря вызову функции API `SHAppBarMessage()`. Компонент `TAppBar` поддерживает внутреннюю запись `TAppBarData` по имени `FABD`. В конструкторе компонента и методе `CreateWnd()` запись `FABD` настраивается на вызов функции `SendAppBarMsg()` для создания окна AppBar. В частности, здесь инициализируется поле `cbSize`, поле `uCallbackMessage` получает значение, возвращаемое функцией API `RegisterWindowMessage()`, а поле `hWnd` получает дескриптор текущего окна формы. Функция `SendAppBarMessage()` инкапсулирует функцию `SHAppBarMessage()` и определяется следующим образом:

```
function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  Result := SHAppBarMessage(Msg, FABD);
end;
```

При успешном создании окна AppBar вызывается метод `SetAppBarEdge()`, используемый для помещения окна AppBar в исходное положение. Этот метод, в свою очередь, вызывает метод `SetAppBarPos()`, передавая флаг, указывающий на требуемую границу экрана, у которой должно разместиться окно AppBar. Флаги `ABE_TOP`, `ABE_BOTTOM`, `ABE_LEFT` и `ABE_RIGHT` представляют соответственно верхнюю, ниж-

нюю, левую и правую границы экрана. Реализация данного метода показана в приведенном ниже фрагменте кода:

```
procedure TAppBar.SetAppBarPos(Edge: UINT);
begin
  if csDesigning in ComponentState then Exit;
  FABD.uEdge := Edge;       // Установить границу окна
  with FABD.rc do begin
    // Установить координаты полного экрана
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    // Послать сообщение ABM_QUERYPOS для получения размеров
    // соответствующего прямоугольника у границы экрана.
    SendAppBarMsg(ABM_QUERYPOS);
    // Перестроить прямоугольник на основании полученных данных
    case Edge of
      ABE_LEFT: Right := Left + FDockedWidth;
      ABE_RIGHT: Left := Right - FDockedWidth;
      ABE_TOP: Bottom := Top + FDockedHeight;
      ABE_BOTTOM: Top := Bottom - FDockedHeight;
    end;
    // Установить позицию окна AppBar.
    SendAppBarMsg(ABM_SETPOS);
  end;
  // Согласовать свойство BoundsRect с ограничивающим
  // прямоугольником, передаваемым системе.
  BoundsRect := FABD.rc;
end;
```

Сначала в данном методе поле uEdge записи FABD устанавливается равным значению параметра Edge. Затем поле rc заполняется значениями координат полного экрана и посылается сообщение ABM_QUERYPOS. Это сообщение переустанавливает размеры прямоугольника, определяемого полем rc, которое теперь содержит координаты прямоугольника, состыкованного с краем, определяемым полем uEdge. После этого поле rc перестраивается еще раз — до нужных размеров подстраиваются высота и ширина. Теперь поле rc содержит окончательный размер ограничивающей рамки для окна AppBar. Затем оболочке Windows посылается сообщение ABM_SETPOS, уведомляющее о создании нового прямоугольника, и, наконец, сам прямоугольник устанавливается с помощью свойства BoundsRect элемента управления.

Как уже упоминалось в настоящей главе, уведомляющие сообщения будут посылаться окну, определяемому значением FABD.hWnd, с использованием идентификатора сообщения, содержащегося в поле FABD.uCallbackMessage. Уведомляющие сообщения обрабатывает метод WndProc(), код которого приведен ниже.

```
procedure TAppBar.WndProc(var M: TMessage);
var
  State: UINT;
  WndPos: HWnd;
begin
  if M.Msg = AppBarMsg then begin
```

```
    case M.WParam of
      // Рассылается при изменении свойств "всегда наверху" или
      // "автосокрытие".
      ABN_STATECHANGE: begin
        // Проверка флага ABS_ALWAYSONTOP ("всегда наверху")
        State := SendAppBarMsg(ABM_GETSTATE);
        if ABS_ALWAYSONTOP and State = 0 then
          SetTopMost(False)
        else
          SetTopMost(True);
      end;
      // Запущено (или закрыто последнее) полноэкранное
      // приложение.
      ABN_FULLSCREENAPP: begin
        // Установить обратный порядок сортировки окон AppBar.
        State := SendAppBarMsg(ABM_GETSTATE);
        if M.lParam <> 0 then begin
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost(False)
          else
            SetTopMost(True);
        end
        else
          if State and ABS_ALWAYSONTOP <> 0 then
            SetTopMost(True);
      end;
      // Рассылается при любом изменении положения окна AppBar.
      ABN_POSCHANGED: begin
        // Панель задач или какая-нибудь другая панель изменили
        // размер или расположение.
        SetAppBarPos(FABD.uEdge);
      end;
    end;   // case
  end
  else
    inherited WndProc(M);
end;
```

Этот метод обрабатывает несколько уведомляющих сообщений, которые позволяют окну AppBar реагировать на изменения, происходящие в оболочке во время работы приложения. Остальная часть кода компонента TAppBar приведена в листинге 16.3.

Листинг 16.3. AppBars.pas — модуль, содержащий базовый класс для поддержки окна AppBar

```
unit AppBars;

interface

uses Windows, Messages, SysUtils, Forms, ShellAPI, Classes, Controls;

type
  TAppBarEdge = (abeTop, abeBottom, abeLeft, abeRight);
```

```
EAppBarError = class(Exception);

TAppBar = class(TCustomForm)
private
  FABD: TAppBarData;
  FDockedHeight: Integer;
  FDockedWidth: Integer;
  FEdge: TAppBarEdge;
  FOnEdgeChanged: TNotifyEvent;
  FTopMost: Boolean;
  procedure WMActivate(var M: TMessage); message WM_ACTIVATE;
  procedure WMWindowPosChanged(var M: TMessage);
                            message WM_WINDOWPOSCHANGED;
  function SendAppBarMsg(Msg: DWORD): UINT;
  procedure SetAppBarEdge(Value: TAppBarEdge);
  procedure SetAppBarPos(Edge: UINT);
  procedure SetTopMost(Value: Boolean);
  procedure SetDockedHeight(const Value: Integer);
  procedure SetDockedWidth(const Value: Integer);
protected
  procedure CreateParams(var Params: TCreateParams); override;
  procedure CreateWnd; override;
  procedure DestroyWnd; override;
  procedure WndProc(var M: TMessage); override;
public
  constructor CreateNew(AOwner: TComponent;
                        Dummy: Integer = 0); override;
  property DockManager;
published
  property Action;
  property ActiveControl;
  property AutoScroll;
  property AutoSize;
  property BiDiMode;
  property BorderWidth;
  property Color;
  property Ctl3D;
  property DockedHeight: Integer read FDockedHeight
                              write SetDockedHeight default 35;
  property DockedWidth: Integer read FDockedWidth
                              write SetDockedWidth default 40;
  property UseDockManager;
  property DockSite;
  property DragKind;
  property DragMode;
  property Edge: TAppBarEdge read Fedge
                            write SetAppBarEdge default abeTop;
  property Enabled;
  property ParentFont default False;
  property Font;
  property HelpFile;
  property HorzScrollBar;
```

```
    property Icon;
    property KeyPreview;
    property ObjectMenuItem;
    property ParentBiDiMode;
    property PixelsPerInch;
    property PopupMenu;
    property PrintScale;
    property Scaled;
    property ShowHint;
    property TopMost: Boolean read FTopMost
                            write SetTopMost default False;
    property VertScrollBar;
    property Visible;
    property OnActivate;
    property OnCanResize;
    property OnClick;
    property OnClose;
    property OnCloseQuery;
    property OnConstrainedResize;
    property OnCreate;
    property OnDblClick;
    property OnDestroy;
    property OnDeactivate;
    property OnDockDrop;
    property OnDockOver;
    property OnDragDrop;
    property OnDragOver;
    property OnEdgeChanged: TNotifyEvent read FOnEdgeChanged
                                       write FOnEdgeChanged;
    property OnEndDock;
    property OnGetSiteInfo;
    property OnHide;
    property OnHelp;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnMouseWheel;
    property OnMouseWheelDown;
    property OnMouseWheelUp;
    property OnPaint;
    property OnResize;
    property OnShortCut;
    property OnShow;
    property OnStartDock;
    property OnUnDock;
  end;

implementation

var
```

```
  AppBarMsg: UINT;

constructor TAppBar.CreateNew(AOwner: TComponent; Dummy: Integer);
begin
  FDockedHeight := 35;
  FDockedWidth := 40;
  inherited CreateNew(AOwner, Dummy);
  ClientHeight := 35;
  Width := 100;
  BorderStyle := bsNone;
  BorderIcons := [];
  // Инициализация записи TAppBarData
  FABD.cbSize := SizeOf(FABD);
  FABD.uCallbackMessage := AppBarMsg;
end;

procedure TAppBar.WMWindowPosChanged(var M: TMessage);
begin
  inherited;
  // Проинформировать оболочку об изменении положения окна AppBar.
  SendAppBarMsg(ABM_WINDOWPOSCHANGED);
end;

procedure TAppBar.WMActivate(var M: TMessage);
begin
  inherited;
  // Проинформировать оболочку об активизации окна AppBar
  SendAppBarMsg(ABM_ACTIVATE);
end;

procedure TAppBar.WndProc(var M: TMessage);
var
  State: UINT;
begin
  if M.Msg = AppBarMsg then begin
    case M.WParam of
      // Рассылается при изменении свойств "всегда наверху" или
      // "автосокрытие".
      ABN_STATECHANGE: begin
        // Проверка флага ABS_ALWAYSONTOP ("всегда наверху")
        State := SendAppBarMsg(ABM_GETSTATE);
        if ABS_ALWAYSONTOP and State = 0 then
          SetTopMost(False)
        else
          SetTopMost(True);
      end;
      // Запущено (или закрыто последнее) полноэкранное
      // приложение.
      ABN_FULLSCREENAPP: begin
        // Установить обратный порядок сортировки окон AppBar.
        State := SendAppBarMsg(ABM_GETSTATE);
        if M.lParam <> 0 then begin
          if ABS_ALWAYSONTOP and State = 0 then
```

```
            SetTopMost(False)
          else
            SetTopMost(True);
        end
        else
          if State and ABS_ALWAYSONTOP <> 0 then
            SetTopMost(True);
      end;
      // Рассылается при любом изменении положения окна AppBar.
      ABN_POSCHANGED:
        // Панель задач или какая-нибудь другая панель изменили
        // размер или расположение.
        SetAppBarEdge(FEdge);
    end;
  end
  else
    inherited WndProc(M);
end;

function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  // Не создавать AppBar во время разработки.
  if csDesigning in ComponentState then Result := 0
  else Result := SHAppBarMessage(Msg, FABD);
end;

procedure TAppBar.SetAppBarPos(Edge: UINT);
begin
  if csDesigning in ComponentState then Exit;
  FABD.uEdge := Edge;       // Установить границу экрана
  with FABD.rc do begin
    // Установить координаты в соответствии с размером
    // полного экрана
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    // Послать сообщение ABM_QUERYPOS для получения прямоугольника
    // соответствующих размеров у края экрана.
    SendAppBarMsg(ABM_QUERYPOS);
    // Перестроить прямоугольник в соответствии с информацией,
    // полученной в ответ на сообщение ABM_QUERYPOS.
    case Edge of
      ABE_LEFT: Right := Left + FDockedWidth;
      ABE_RIGHT: Left := Right - FDockedWidth;
      ABE_TOP: Bottom := Top + FDockedHeight;
      ABE_BOTTOM: Top := Bottom - FDockedHeight;
    end;
    // Установить позицию панели.
    SendAppBarMsg(ABM_SETPOS);
  end;
  // Установить свойство BoundsRect согласно с ограничивающим
  // прямоугольником, переданным системе.
```

```
  BoundsRect := FABD.rc;
end;

procedure TAppBar.SetTopMost(Value: Boolean);
const
  WndPosArray: array[Boolean] of HWND = (HWND_BOTTOM,
                                         HWND_TOPMOST);
begin
  if FTopMost <> Value then begin
    FTopMost := Value;
    if not (csDesigning in ComponentState) then
      SetWindowPos(Handle, WndPosArray[Value], 0, 0, 0, 0,
                   SWP_NOMOVE or SWP_NOSIZE or SWP_NOACTIVATE);
  end;
end;

procedure TAppBar.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  if not (csDesigning in ComponentState) then begin
    Params.ExStyle := Params.ExStyle or
                      WS_EX_TOPMOST or WS_EX_WINDOWEDGE;
    Params.Style := Params.Style or WS_DLGFRAME;
  end;
end;

procedure TAppBar.CreateWnd;
begin
  inherited CreateWnd;
  FABD.hWnd := Handle;
  if not (csDesigning in ComponentState) then begin
    if SendAppBarMsg(ABM_NEW) = 0 then
      raise EAppBarError.Create('Failed to create AppBar');
    // Инициализировать позицию
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.DestroyWnd;
begin
  // Проинформировать оболочку об удалении панели AppBar
  SendAppBarMsg(ABM_REMOVE);
  inherited DestroyWnd;
end;

procedure TAppBar.SetAppBarEdge(Value: TAppBarEdge);
const
  EdgeArray: array[TAppBarEdge] of UINT =
    (ABE_TOP, ABE_BOTTOM, ABE_LEFT, ABE_RIGHT);
begin
  SetAppBarPos(EdgeArray[Value]);
  FEdge := Value;
  if Assigned(FOnEdgeChanged) then FOnEdgeChanged(Self);
```

```
end;

procedure TAppBar.SetDockedHeight(const Value: Integer);
begin
  if FDockedHeight <> Value then begin
    FDockedHeight := Value;
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.SetDockedWidth(const Value: Integer);
begin
  if FDockedWidth <> Value then begin
    FDockedWidth := Value;
    SetAppBarEdge(FEdge);
  end;
end;

initialization
  AppBarMsg := RegisterWindowMessage('DDG AppBar Message');
end.
```

Использование компонента TAppBar

Если установить программный продукт с компакт-диска, прилагаемого к этой книге, то использование компонента TAppBar не вызовет никаких трудностей: достаточно выбрать в меню File пункт New dialog, а затем во вкладке DDG диалогового окна параметр AppBar — и вызванный мастер немедленно создаст модуль, содержащий компонент TAppBar.

НА ЗАМЕТКУ

В главе 17, "Применение интерфейса API Open Tools", будет показано, как создать мастер, который автоматически генерирует компонент TAppBar. Для задач, решаемых в настоящей главе, можно вполне обойтись и без рассмотрения реализации мастера. Просто примите сейчас к сведению то, что часть работы по созданию формы и соответствующего модуля выполняется "за кулисами".

Ниже приведен код небольшого приложения, использующего компонент TAppBar для создания панели инструментов на рабочем столе приложения с кнопками для выполнения различных команд редактирования и работы с файлами: Open, Save, Cut, Copy и Paste. С помощью этих кнопок можно управлять компонентом TMemo, расположенным в главной форме. Код главного модуля приложения приведен в листинге 16.4, а на рис. 16.5 данное приложение показано в действии — создана панель AppBar, которая закреплена на верхней границе экрана.

Листинг 16.4. ApBarFrm.pas — главный модуль приложения, демонстрирующего использование окна AppBar

```
unit ApBarFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, AppBars, Menus, Buttons;

type
  TAppBarForm = class(TAppBar)
    sbOpen: TSpeedButton;
    sbSave: TSpeedButton;
    sbCut: TSpeedButton;
    sbCopy: TSpeedButton;
    sbPaste: TSpeedButton;
    OpenDialog: TOpenDialog;
    pmPopup: TPopupMenu;
    Top1: TMenuItem;
    Bottom1: TMenuItem;
    Left1: TMenuItem;
    Right1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure Right1Click(Sender: TObject);
    procedure sbOpenClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbCutClick(Sender: TObject);
    procedure sbCopyClick(Sender: TObject);
    procedure sbPasteClick(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormEdgeChanged(Sender: TObject);
  private
    FLastChecked: TMenuItem;
    procedure MoveButtons;
  end;

var
  AppBarForm: TAppBarForm;

implementation

uses Main;

{$R *.DFM}

{ TAppBarForm }

procedure TAppBarForm.MoveButtons;
{ Этот метод выглядит сложным, но на самом деле он просто размещает
кнопки в зависимости от того, к какой стороне экрана прикреплена
панель AppBar. }
var
```

```
  DeltaCenter, NewPos: Integer;
begin
  if Edge in [abeTop, abeBottom] then begin
    DeltaCenter := (ClientHeight - sbOpen.Height) div 2;
    sbOpen.SetBounds(10, DeltaCenter, sbOpen.Width,
                     sbOpen.Height);
    NewPos := sbOpen.Width + 20;
    sbSave.SetBounds(NewPos, DeltaCenter, sbOpen.Width,
                     sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCut.SetBounds(NewPos, DeltaCenter, sbOpen.Width,
                    sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCopy.SetBounds(NewPos, DeltaCenter, sbOpen.Width,
                     sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbPaste.SetBounds(NewPos, DeltaCenter, sbOpen.Width,
                      sbOpen.Height);
  end else begin
    DeltaCenter := (ClientWidth - sbOpen.Width) div 2;
    sbOpen.SetBounds(DeltaCenter, 10, sbOpen.Width,
                     sbOpen.Height);
    NewPos := sbOpen.Height + 20;
    sbSave.SetBounds(DeltaCenter, NewPos, sbOpen.Width,
                     sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCut.SetBounds(DeltaCenter, NewPos, sbOpen.Width,
                    sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCopy.SetBounds(DeltaCenter, NewPos, sbOpen.Width,
                     sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbPaste.SetBounds(DeltaCenter, NewPos, sbOpen.Width,
                      sbOpen.Height);
  end;
end;

procedure TAppBarForm.Right1Click(Sender: TObject);
begin
  FLastChecked.Checked := False;
  (Sender as TMenuItem).Checked := True;
  case TMenuItem(Sender).Caption[2] of
    'T': Edge := abeTop;
    'B': Edge := abeBottom;
    'L': Edge := abeLeft;
    'R': Edge := abeRight;
  end;
  FLastChecked := TMenuItem(Sender);
end;

procedure TAppBarForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
```

```
    MainForm.FileName := OpenDialog.FileName;
end;

procedure TAppBarForm.sbSaveClick(Sender: TObject);
begin
  MainForm.memEditor.Lines.SaveToFile(MainForm.FileName);
end;

procedure TAppBarForm.sbCutClick(Sender: TObject);
begin
  MainForm.memEditor.CutToClipboard;
end;

procedure TAppBarForm.sbCopyClick(Sender: TObject);
begin
  MainForm.memEditor.CopyToClipboard;
end;

procedure TAppBarForm.sbPasteClick(Sender: TObject);
begin
  MainForm.memEditor.PasteFromClipboard;
end;

procedure TAppBarForm.Exit1Click(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TAppBarForm.FormCreate(Sender: TObject);
begin
  FLastChecked := Top1;
end;

procedure TAppBarForm.FormEdgeChanged(Sender: TObject);
begin
  MoveButtons;
end;

end.
```

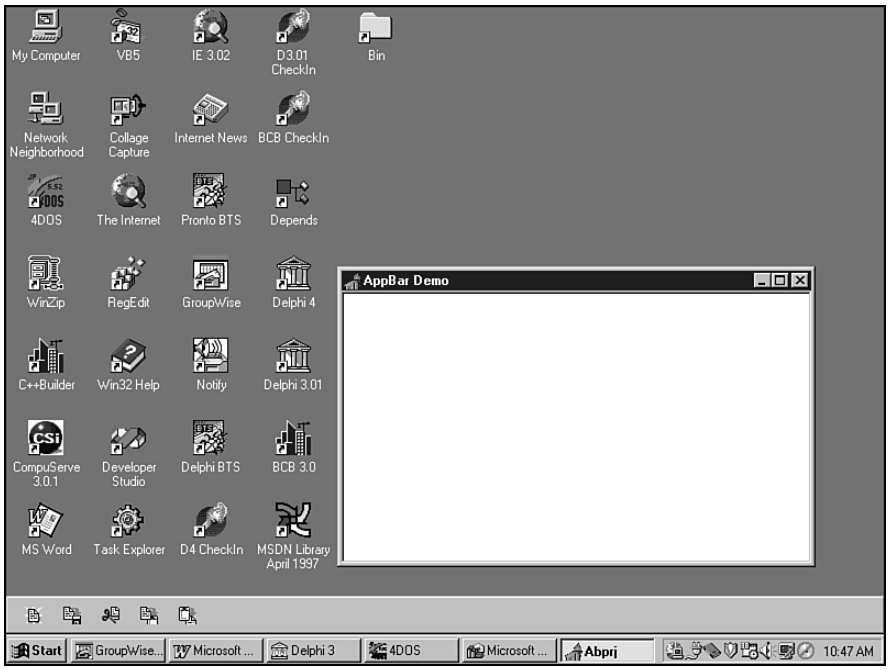

Рис. 16.5. *Компонент `TAppBar` в действии*

Ярлыки Windows

В оболочке Windows предусмотрен ряд интерфейсов, которые можно использовать для управления различными ее свойствами. Эти интерфейсы определены в модуле `ShlObj`. Подробного обсуждения всех объектов такого модуля хватило бы на целую книгу, но в данном разделе остановимся лишь на одном из самых полезных (и чаще всего используемых) — интерфейсе `IShellLink`.

Интерфейс `IShellLink` позволяет создавать в приложениях ярлыки Windows и управлять ими. При работе на компьютере с такими объектами приходится встречаться постоянно: большинство пиктограмм на рабочем столе — это именно ярлыки. Кроме того, такие пункты меню, как Send To (Отправить) или Documents (Документы), — это тоже ярлыки. Интерфейс `IShellLink` определен следующим образом:

```
const

type
  IShellLink = interface(IUnknown)
    ['{000214EE-0000-0000-C000-000000000046}']
    function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
    var pfd: TWin32FindData; fFlags: DWORD): HResult; stdcall;
    function GetIDList(var ppidl: PItemIDList): HResult; stdcall;
    function SetIDList(pidl: PItemIDList): HResult; stdcall;
    function GetDescription(pszName: PAnsiChar;
                            cchMaxName: Integer): HResult; stdcall;
```

```
    function SetDescription(pszName: PAnsiChar): HResult; stdcall;
    function GetWorkingDirectory(pszDir: PAnsiChar;
                          cchMaxPath: Integer): HResult; stdcall;
    function SetWorkingDirectory(pszDir: PAnsiChar): HResult;
                                                    stdcall;
    function GetArguments(pszArgs: PAnsiChar;
                          cchMaxPath: Integer): HResult; stdcall;
    function SetArguments(pszArgs: PAnsiChar): HResult; stdcall;
    function GetHotkey(var pwHotkey: Word): HResult; stdcall;
    function SetHotkey(wHotkey: Word): HResult; stdcall;
    function GetShowCmd(out piShowCmd: Integer): HResult; stdcall;
    function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
    function GetIconLocation(pszIconPath: PAnsiChar;
                           cchIconPath: Integer;
                           out piIcon: Integer): HResult; stdcall;
    function SetIconLocation(pszIconPath: PAnsiChar;
                             iIcon: Integer): HResult; stdcall;
    function SetRelativePath(pszPathRel: PAnsiChar;
                             dwReserved: DWORD): HResult; stdcall;
    function Resolve(Wnd: HWND; fFlags: DWORD): HResult; stdcall;
    function SetPath(pszFile: PAnsiChar): HResult; stdcall;
  end;
```

НА ЗАМЕТКУ

Интерфейс `IShellLink` и все его методы подробно описаны в интерактивной справочной системе Win32 и здесь не приводятся.

Создание экземпляра интерфейса IShellLink

В отличие от расширений оболочки, речь о которых пойдет далее в настоящей главе, при работе с интерфейсом `IShellLink` не нужно реализовывать интерфейс — он уже реализован в самой оболочке Windows, и остается лишь создать его экземпляр. Для этого служит функция API COM `CoCreateInstance()`. Предлагаем рассмотреть пример создания подобного экземпляра.

```
var
  SL: IShellLink;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil,
           CLSCTX_INPROC_SERVER, IShellLink, SL));
  // Здесь используется экземпляр интерфейса SL
end;
```

НА ЗАМЕТКУ

Не забудьте, что, прежде чем использовать какую-либо из функций OLE, необходимо с помощью функции API `CoInitialize()` инициализировать библиотеку COM. По окончании работы для освобождения ресурсов следует вызвать функцию API `CoUninitialize()`. Если приложение использует модуль `ComObj` и содержит вызов функции `Application.Initialize()`, то упомянутые функции будут вызваны автоматически. В противном случае их придется вызывать самостоятельно.

Использование интерфейса IShellLink

Иногда кажется, что ярлыки Windows обладают просто магическими возможностями: щелкнув правой кнопкой мыши на поверхности рабочего стола можно выбирать в контекстном меню пункт создания нового ярлыка, и после этого происходит *нечто*, сопровождающееся появлением пиктограммы на поверхности рабочего стола. Но стоит узнать, что именно происходит в действительности — и от загадочности не остается и следа. *Ярлык* (shell link) на самом деле представляет собой файл с расширением `.lnk`, расположенный в определенной папке (shell folder). При запуске Windows в известных папках выполняется поиск файлов `.lnk`. К числу этих специальных папок (папок системной оболочки или системных папок) относятся такие, как `Network Neighborhood`, `Send To`, `Startup`, `Desktop` и т.д. Windows хранит отношения "ярлык–папка" в системном реестре, главным образом в следующей ветви:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\
↳Explorer\Shell Folders
```

Для того чтобы создать в некоторой папке ярлык, достаточно просто поместить в эту папку файл `.lnk`. Чтобы узнать пути к системным папкам можно, конечно, обратиться к системному реестру, однако гораздо эффективнее использовать функцию API Win32 `SHGetSpecialFolderPath()`, которая определена следующим образом:

```
function SHGetSpecialFolderPath(hwndOwner: HWND; lpszPath: PChar;
                 nFolder: Integer; fCreate: BOOL): BOOL; stdcall;
```

Параметр `hwndOwner` содержит дескриптор окна, которое будет владельцем всех диалоговых окон, вызываемых этой функцией.

Параметр `lpszPath` является указателем на буфер, куда будет передан возвращаемый путь. Размер данного буфера должен быть не меньше значения `MAX_PATH`.

В параметре `nFolder` указано имя специальной папки, путь к которой необходимо получить. В табл. 16.4 приведены возможные значения этого параметра.

С помощью параметра `fCreate` определяется, будет ли создана папка в случае, если таковая отсутствует.

Таблица 16.4. Возможные значения параметра `nFolder`

Флаг	*Описание*
`CSIDL_ALTSTARTUP`	Папка, соответствующая нелокализованной группе Startup (Автозагрузка) конкретного пользователя
`CSIDL_APPDATA`	Папка, выделенная как общее место хранения данных приложений
`CSIDL_BITBUCKET`	Папка, содержащая файловые объекты корзины пользователя (`Recycle Bin`). Расположение этой папки не зафиксировано в системном реестре; чтобы не допустить ее перемещения или удаления, она помечена атрибутами "скрытый" и "системный"

Продолжение табл. 16.4.

Флаг	*Описание*
CSIDL_COMMON_ALTSTARTUP	Папка, соответствующая нелокализованной группе Startup всех пользователей
CSIDL_COMMON_DESKTOPDIRECTORY	Папка, содержащая файлы и папки, которые отображаются на рабочем столе для всех пользователей Desktop (Рабочий стол)
CSIDL_COMMON_FAVORITES	Папка Favorite (Избранное) для всех пользователей
CSIDL_COMMON_PROGRAMS	Папка, содержащая программы из папки Start (Пуск) для всех пользователей (Programs (Программы))
CSIDL_COMMON_STARTMENU	Папка, содержащая папки групп программ из меню Start (для всех пользователей)
CSIDL_COMMON_STARTUP	Папка, содержащая программы и папки, отображаемые в меню Start (для всех пользователей)
CSIDL_CONTROLS	Виртуальная папка, содержащая индикаторы приложений панели управления
CSIDL_COOKIES	Папка для хранения файлов Internet типа cookies
CSIDL_DESKTOP	Виртуальная папка рабочего стола (Windows Desktop) для хранения корневого пространства имен
CSIDL_DESKTOPDIRECTORY	Папка, где физически расположены файловые объекты рабочего стола (не путать с папкой Desktop!)
CSIDL_DRIVES	Папка My Computer (Мой компьютер), содержащая все объекты на локальном компьютере: дисковые устройства, принтеры и панель управления. Эта папка может также содержать подключенные сетевые устройства
CSIDL_FAVORITES	Папка, в которой хранится “Избранное” конкретного пользователя
CSIDL_FONTS	Виртуальная папка, содержащая шрифты
CSIDL_HISTORY	Папка, где хранятся ссылки на адреса Internet, посещенные пользователем
CSIDL_INTERNET	Виртуальная папка, представляющая Internet
CSIDL_INTERNET_CACHE	Папка общего хранилища временных файлов Internet
CSIDL_NETHOOD	Папка, содержащая объекты сетевого окружения (Network Neighborhood)
CSIDL_NETWORK	Виртуальная папка Network Neighborhood, представляющая собой высший уровень сетевой иерархии

Окончание табл. 16.4.

Флаг	*Описание*
CSIDL_PERSONAL	Папка, используемая как общее хранилище для документов
CSIDL_PRINTERS	Виртуальная папка, содержащая информацию обо всех установленных принтерах
CSIDL_PRINTHOOD	Папка, используемая как общее хранилище для ярлыков принтеров
CSIDL_PROGRAMS	Папка, содержащая группы программ (которые также являются папками) конкретного пользователя
CSIDL_RECENT	Папка, содержащая ссылки на документы, с которыми пользователь работал в последнее время
CSIDL_SENDTO	Папка, которая содержит объекты, появляющиеся в меню Send To
CSIDL_STARTMENU	Папка, содержащая пункты меню Start
CSIDL_STARTUP	Папка, соответствующая группе программ Startup (для конкретного пользователя). Программы из этой папки автоматически запускаются каждый раз при регистрации в Windows NT либо при запуске Windows 95 или Windows 98
CSIDL_TEMPLATES	Папка для хранения шаблонов документов

Создание ярлыка

Интерфейс IShellLink инкапсулирует объект ярлыка, но не определяет, как он может быть прочитан или записан на диск. Поэтому реализация данного интерфейса требует дополнительной поддержки интерфейса IPersistFile, который используется для обеспечения доступа к файлам. Интерфейс IPersistFile содержит методы чтения и записи данных на диск и определен следующим образом:

```
type
  IPersistFile = interface(IPersist)
    ['{0000010B-0000-0000-C000-000000000046}']
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr;
                  dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr;
                  fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult;
                                                  stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult;
                                                   stdcall;
  end;
```

НА ЗАМЕТКУ

Более подробное описание интерфейса `IPersistFile` и всех его методов можно найти в интерактивной справочной системе Win32.

Поскольку класс, реализующий интерфейс `IShellLink`, всегда содержит реализацию интерфейса `IPersistFile`, экземпляр интерфейса `IShellLink` можно использовать для получения экземпляра интерфейса `IPersistFile` с помощью операции преобразования типа `as`, как показано ниже.

```
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil,
           CLSCTX_INPROC_SERVER, IShellLink, SL));
  PF := SL as IPersistFile;
  // Использовать экземпляры интерфейсов PF и SL

end;
```

Как уже упоминалось в настоящей главе, использование объектов интерфейсов COM не отличается от использования обычных объектов Object Pascal. В следующем фрагменте кода приведен пример создания ярлыка для приложения Notepad (Блокнот), который помещается на поверхность рабочего стола:

```
procedure MakeNotepad;
const
  // Предположим, что Notepad расположен в папке:
  AppName = 'c:\windows\notepad.exe';
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil,
           CLSCTX_INPROC_SERVER, IShellLink, SL));
  { При реализации интерфейса IShellLink обязательно реализуется
    и интерфейс IPersistFile }
  PF := SL as IPersistFile;
  OleCheck(SL.SetPath(PChar(AppName)));
  // Установить путь к необходимому файлу:
  { Указать размещение и имя файла создаваемого ярлыка: }
  LnkName := GetFolderLocation('Desktop') + '\' +
             ChangeFileExt(ExtractFileName(AppName), '.lnk');
  PF.Save(PWideChar(LnkName), True);      // Сохранить файл ярлыка
end;
```

В этой процедуре метод `SetPath()` интерфейса `IShellLink` используется для указания пути к исполняемому файлу или документу, для которого создается ярлык (в данном случае это программа Notepad). Затем определяется полный путь и имя файла ярлыка, для чего используется информация, возвращаемая функцией `GetFolderLo-`

cation('Desktop') (описанной выше в этом разделе). Кроме того, для изменения расширения файла Notepad.exe с .exe на .lnk используется функция ChangeFileExt(). Новое имя файла сохраняется в переменной LnkName. Затем с помощью метода Save() новый ярлык сохраняется в файле на диске. Как уже было сказано, при завершении приведенной выше процедуры и выходе экземпляров интерфейсов SL и PF за пределы области видимости соответствующие ссылки на них освобождаются.

Установка и возвращение информации ярлыка

Как видно из определения интерфейса IShellLink, он содержит несколько методов вида Get*XXX*() и Set*XXX*(), позволяющих возвращать и устанавливать параметры, которые определяют различные аспекты ярлыка. Рассмотрим следующее объявление записи, поля которой можно просматривать и изменять:

```
type
  TShellLinkInfo = record
    PathName: string;
    Arguments: string;
    Description: string;
    WorkingDirectory: string;
    IconLocation: string;
    IconIndex: Integer;
    ShowCmd: Integer;
    HotKey: Word;
  end;
```

Располагая подобной записью, можно создать функции для работы с параметрами ярлыка (для считывания значений параметров в поля этой записи или для установки параметров ярлыка в соответствии с содержимым полей записи). Исходный код таких функций содержится в модуле WinShell.pas, приведенном в листинге 16.5.

Листинг 16.5. Модуль WinShell.pas содержит функции доступа к параметрам ярлыка

```
unit WinShell;

interface

uses SysUtils, Windows, Registry, ActiveX, ShlObj;

type
  EShellOleError = class(Exception);

  TShellLinkInfo = record
    PathName: string;
    Arguments: string;
    Description: string;
    WorkingDirectory: string;
    IconLocation: string;
    IconIndex: integer;
    ShowCmd: integer;
    HotKey: word;
```

```
  end;

  TSpecialFolderInfo = record
    Name: string;
    ID: Integer;
  end;

const
  SpecialFolders: array[0..29] of TSpecialFolderInfo = (
    (Name: 'Alt Startup'; ID: CSIDL_ALTSTARTUP),
    (Name: 'Application Data'; ID: CSIDL_APPDATA),
    (Name: 'Recycle Bin'; ID: CSIDL_BITBUCKET),
    (Name: 'Common Alt Startup'; ID: CSIDL_COMMON_ALTSTARTUP),
    (Name: 'Common Desktop'; ID: CSIDL_COMMON_DESKTOPDIRECTORY),
    (Name: 'Common Favorites'; ID: CSIDL_COMMON_FAVORITES),
    (Name: 'Common Programs'; ID: CSIDL_COMMON_PROGRAMS),
    (Name: 'Common Start Menu'; ID: CSIDL_COMMON_STARTMENU),
    (Name: 'Common Startup'; ID: CSIDL_COMMON_STARTUP),
    (Name: 'Controls'; ID: CSIDL_CONTROLS),
    (Name: 'Cookies'; ID: CSIDL_COOKIES),
    (Name: 'Desktop'; ID: CSIDL_DESKTOP),
    (Name: 'Desktop Directory'; ID: CSIDL_DESKTOPDIRECTORY),
    (Name: 'Drives'; ID: CSIDL_DRIVES),
    (Name: 'Favorites'; ID: CSIDL_FAVORITES),
    (Name: 'Fonts'; ID: CSIDL_FONTS),
    (Name: 'History'; ID: CSIDL_HISTORY),
    (Name: 'Internet'; ID: CSIDL_INTERNET),
    (Name: 'Internet Cache'; ID: CSIDL_INTERNET_CACHE),
    (Name: 'Network Neighborhood'; ID: CSIDL_NETHOOD),
    (Name: 'Network Top'; ID: CSIDL_NETWORK),
    (Name: 'Personal'; ID: CSIDL_PERSONAL),
    (Name: 'Printers'; ID: CSIDL_PRINTERS),
    (Name: 'Printer Links'; ID: CSIDL_PRINTHOOD),
    (Name: 'Programs'; ID: CSIDL_PROGRAMS),
    (Name: 'Recent Documents'; ID: CSIDL_RECENT),
    (Name: 'Send To'; ID: CSIDL_SENDTO),
    (Name: 'Start Menu'; ID: CSIDL_STARTMENU),
    (Name: 'Startup'; ID: CSIDL_STARTUP),
    (Name: 'Templates'; ID: CSIDL_TEMPLATES));

function CreateShellLink(const AppName, Desc: string;
                         Dest: Integer): string;
function GetSpecialFolderPath(Folder: Integer;
                         CanCreate: Boolean): string;
procedure GetShellLinkInfo(const LinkFile: WideString;
                         var SLI: TShellLinkInfo);
procedure SetShellLinkInfo(const LinkFile: WideString;
                         const SLI: TShellLinkInfo);

implementation

uses ComObj;
```

```
function GetSpecialFolderPath(Folder: Integer;
                              CanCreate: Boolean): string;
var
  FilePath: array[0..MAX_PATH] of char;
begin
  { Получить путь выбранного местоположения. }
  SHGetSpecialFolderPath(0, FilePath, Folder, CanCreate);
  Result := FilePath;
end;

function CreateShellLink(const AppName, Desc: string;
                         Dest: Integer): string;
{ Создает ярлык для приложения или документа, определяемого
константой AppName и описанием в строке Desc. Ярлык будет помещен в
папку, заданную параметром Dest и указанную одной из строковых
констант массива, который определен в заголовке этого модуля.
Возвращает полное имя файла ярлыка. }
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil,
           CLSCTX_INPROC_SERVER, IShellLink, SL));
  { Реализация интерфейса IShellLink всегда поддерживает
    интерфейс PersistFile. Получить указатель на этот интерфейс. }
  PF := SL as IPersistFile;
  // Установить путь к необходимому файлу:
  OleCheck(SL.SetPath(PChar(AppName)));
  if Desc <> '' then   // Установить описание
    OleCheck(SL.SetDescription(PChar(Desc)));
  { Указать размещение и имя файла создаваемого ярлыка: }
  LnkName := GetSpecialFolderPath(Dest, True) + '\' +
             ChangeFileExt(AppName, 'lnk');
  PF.Save(PWideChar(LnkName), True);     // Сохранить файл ярлыка
  Result := LnkName;
end;

procedure GetShellLinkInfo(const LinkFile: WideString;
                           var SLI: TShellLinkInfo);
{ Получить информацию о существующем ярлыке. }
var
  SL: IShellLink;
  PF: IPersistFile;
  FindData: TWin32FindData;
  AStr: array[0..MAX_PATH] of char;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil,
           CLSCTX_INPROC_SERVER, IShellLink, SL));
  { Реализация интерфейса IShellLink всегда поддерживает
    интерфейс PersistFile. Получить указатель на этот интерфейс. }
  PF := SL as IPersistFile;
  { Загрузить файл в объект IPersistFile. }
```

```
  OleCheck(PF.Load(PWideChar(LinkFile), STGM_READ));
  { Найти ярлык с помощью функции Resolve интерфейса. }
  OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_NO_UI));
  { Получить всю информацию о ярлыке! }
  with SLI do begin
    OleCheck(SL.GetPath(AStr, MAX_PATH, FindData,
             SLGP_SHORTPATH));
    PathName := AStr;
    OleCheck(SL.GetArguments(AStr, MAX_PATH));
    Arguments := AStr;
    OleCheck(SL.GetDescription(AStr, MAX_PATH));
    Description := AStr;
    OleCheck(SL.GetWorkingDirectory(AStr, MAX_PATH));
    WorkingDirectory := AStr;
    OleCheck(SL.GetIconLocation(AStr, MAX_PATH, IconIndex));
    IconLocation := AStr;
    OleCheck(SL.GetShowCmd(ShowCmd));
    OleCheck(SL.GetHotKey(HotKey));
  end;
end;

procedure SetShellLinkInfo(const LinkFile: WideString;
                           const SLI: TShellLinkInfo);
{ Установить информацию о существующем ярлыке. }
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil,
                           CLSCTX_INPROC_SERVER, IShellLink, SL));
  { Реализация интерфейса IShellLink всегда поддерживает
    интерфейс PersistFile. Получить указатель на этот интерфейс. }
  PF := SL as IPersistFile;
  { Загрузить файл в объект IPersistFile. }
  OleCheck(PF.Load(PWideChar(LinkFile), STGM_SHARE_DENY_WRITE));
  { Найти ярлык с помощью функции Resolve интерфейса. }
  OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_UPDATE
                      or SLR_NO_UI));
  { Установить всю информацию о ярлыке! }
  with SLI, SL do begin
    OleCheck(SetPath(PChar(PathName)));
    OleCheck(SetArguments(PChar(Arguments)));
    OleCheck(SetDescription(PChar(Description)));
    OleCheck(SetWorkingDirectory(PChar(WorkingDirectory)));
    OleCheck(SetIconLocation(PChar(IconLocation), IconIndex));
    OleCheck(SetShowCmd(ShowCmd));
    OleCheck(SetHotKey(HotKey));
  end;
  PF.Save(PWideChar(LinkFile), True);   // Сохранить файл
end;

end.
```

Из используемых в данном модуле методов интерфейса IShellLink лишь метод Resolve() нуждается в некоторых разъяснениях. Его следует вызывать после того, как с помощью интерфейса IPersistFile объекта IShellLink будет загружен файл ярлыка. Этот метод ищет заданный файл ярлыка и заполняет объект IShellLink значениями, содержащимися в найденном файле.

СОВЕТ

Просматривая код функции GetShellLinkInfo() (см. листинг 16.5), обратите внимание на то, что для возвращаемых значений используется локальный массив AStr. Для этого можно было бы воспользоваться функцией SetLength(), выделяющей память для хранения строк. Однако предпочтение было отдано массиву AStr, поскольку применение функции SetLength() для такого большого количества строк приводит к значительной фрагментации распределяемой памяти приложения. Использование массива AStr, как промежуточного звена, препятствует данному явлению. Более того, поскольку длина строк устанавливается лишь один раз, использование массива AStr несколько ускоряет процесс.

Пример приложения

Проще всего ознакомиться с возможностями рассмотренных в предыдущем разделе функций и интерфейсов — это включить их в какое-нибудь приложение. В качестве примера рассмотрим проект ShellLink. Главная форма данного проекта показана на рис. 16.6.

Код главного модуля приложения приведен в листинге 16.6 (модуль Main.pas). Кроме модуля Main.pas в проект входят два дополнительных модуля — NewLinkU.pas (листинг 16.7) и PickU.pas (листинг 16.8).

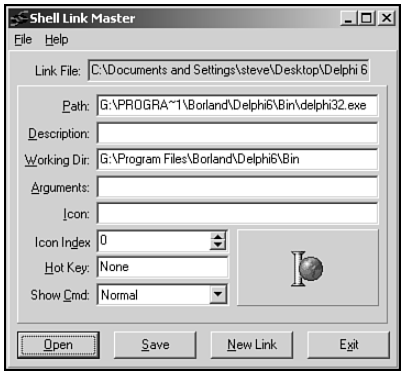

Рис. 16.6. *Главная форма ShellLink, в которой отображена информация об одном из ярлыков на рабочем столе*

Листинг 16.6. Main.pas — главный модуль проекта

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
```

```
  Dialogs, StdCtrls, ComCtrls, ExtCtrls, Spin, WinShell, Menus;

type
  TMainForm = class(TForm)
    Panel1: TPanel;
    btnOpen: TButton;
    edLink: TEdit;
    btnNew: TButton;
    btnSave: TButton;
    Label3: TLabel;
    Panel2: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    edIcon: TEdit;
    edDesc: TEdit;
    edWorkDir: TEdit;
    edArg: TEdit;
    cbShowCmd: TComboBox;
    hkHotKey: THotKey;
    speIcnIdx: TSpinEdit;
    pnlIconPanel: TPanel;
    imgIconImage: TImage;
    btnExit: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
    NewLInk1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    edPath: TEdit;
    procedure btnOpenClick(Sender: TObject);
    procedure btnNewClick(Sender: TObject);
    procedure edIconChange(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
    procedure About1Click(Sender: TObject);
  private
    procedure GetControls(var SLI: TShellLinkInfo);
    procedure SetControls(const SLI: TShellLinkInfo);
    procedure ShowIcon;
    procedure OpenLinkFile(const LinkFileName: String);
  end;

var
```

```
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses PickU, NewLinkU, AboutU, CommCtrl, ShellAPI;

type
  THotKeyRec = record
    Char, ModCode: Byte;
  end;

procedure TMainForm.SetControls(const SLI: TShellLinkInfo);
{ Установить значения элементов управления пользовательского
интерфейса на основании содержимого параметра SLI. }
var
  Mods: THKModifiers;
begin
  with SLI do begin
    edPath.Text := PathName;
    edIcon.Text := IconLocation;
    { Если путь пиктограммы отсутствует, а ярлык указывает на
      исполняемый файл, то для пути пиктограммы используется имя
      исполняемого файла. Это необходимо для того, чтобы можно
      было игнорировать индекс пиктограммы в случае, если не
      заполнено поле пути пиктограммы, а исполняемый файл
      содержит несколько пиктограмм. }
    if (IconLocation = '') and
       (CompareText(ExtractFileExt(PathName), 'EXE') = 0) then
      edIcon.Text := PathName;
    edWorkDir.Text := WorkingDirectory;
    edArg.Text := Arguments;
    speIcnIdx.Value := IconIndex;
    edDesc.Text := Description;
    { Константы SW_* начинаются с 1 }
    cbShowCmd.ItemIndex := ShowCmd - 1;
    { Код горячей клавиши в младшем байте }
    hkHotKey.HotKey := Lo(HotKey);
    { Информация о флагах модификаторов в старшем байте. }
    Mods := [];
    if (HOTKEYF_ALT and Hi(HotKey)) <> 0 then
      include(Mods, hkAlt);
    if (HOTKEYF_CONTROL and Hi(HotKey)) <> 0 then
      include(Mods, hkCtrl);
    if (HOTKEYF_EXT and Hi(HotKey)) <> 0 then
      include(Mods, hkExt);
    if (HOTKEYF_SHIFT and Hi(HotKey)) <> 0 then
      include(Mods, hkShift);
    { Установить набор модификаторов. }
    hkHotKey.Modifiers := Mods;
  end;
  ShowIcon;
```

```
end;

procedure TMainForm.GetControls(var SLI: TShellLinkInfo);
{ Получить значения элементов управления пользовательского
интерфейса и использовать их для установки значений записи SLI. }
var
  CtlMods: THKModifiers;
  HR: THotKeyRec;
begin
  with SLI do begin
    PathName := edPath.Text;
    IconLocation := edIcon.Text;
    WorkingDirectory := edWorkDir.Text;
    Arguments := edArg.Text;
    IconIndex := speIcnIdx.Value;
    Description := edDesc.Text;
    { Константы SW_* начинаются с 1 }
    ShowCmd := cbShowCmd.ItemIndex + 1;
    { Получить код горячей клавиши }
    word(HR) := hkHotKey.HotKey;
    { Получить информацию об используемых модификаторах. }
    CtlMods := hkHotKey.Modifiers;
    with HR do begin
      ModCode := 0;
      if (hkAlt in CtlMods) then
        ModCode := ModCode or HOTKEYF_ALT;
      if (hkCtrl in CtlMods) then
        ModCode := ModCode or HOTKEYF_CONTROL;
      if (hkExt in CtlMods) then
        ModCode := ModCode or HOTKEYF_EXT;
      if (hkShift in CtlMods) then
        ModCode := ModCode or HOTKEYF_SHIFT;
    end;
    HotKey := word(HR);
  end;
end;

procedure TMainForm.ShowIcon;
{ Извлечь пиктограмму из соответствующего файла и отобразить ее в
объекте IconImage. }
var
  HI: THandle;
  IcnFile: string;
  IconIndex: word;
begin
  { Получить имя файла пиктограммы. }
  IcnFile := edIcon.Text;
  { Если это поле пустое, использовать имя исполняемого файла. }
  if IcnFile = '' then
    IcnFile := edPath.Text;
  { Удостовериться в существовании файла. }
  if FileExists(IcnFile) then begin
    IconIndex := speIcnIdx.Value;
```

```
    { Извлечь пиктограмму из файла. }
    HI := ExtractAssociatedIcon(hInstance, PChar(IcnFile),
                                IconIndex);
    { Присвоить дескриптор пиктограммы объекту IconImage. }
    imgIconImage.Picture.Icon.Handle := HI;
  end;
end;

procedure TMainForm.OpenLinkFile(const LinkFileName: string);
{ Открыть файл ярлыка, выбрать информацию и отобразить ее с помощью
элементов управления пользовательского интерфейса. }
var
  SLI: TShellLinkInfo;
begin
  edLink.Text := LinkFileName;
  try
    GetShellLinkInfo(LinkFileName, SLI);
  except
    on EShellOleError do
      MessageDlg('Error occurred while opening link',
                 mtError, [mbOk], 0);
  end;
  SetControls(SLI);
end;

procedure TMainForm.btnOpenClick(Sender: TObject);
{ Обработчик события OnClick для кнопки btnOpen. }
var
  LinkFile: String;
begin
  if GetLinkFile(LinkFile) then
    OpenLinkFile(LinkFile);
end;

procedure TMainForm.btnNewClick(Sender: TObject);
{ Обработчик события OnClick для кнопки NewBtn }
var
  FileName: string;
  Dest: Integer;
begin
  if GetNewLinkName(FileName, Dest) then
    OpenLinkFile(CreateShellLink(FileName, '', Dest));
end;

procedure TMainForm.edIconChange(Sender: TObject);
{ Обработчик события OnChange для строк редактирования IconEd и
                                                        IcnIdxEd. }
begin
  ShowIcon;
end;

procedure TMainForm.btnSaveClick(Sender: TObject);
{ Обработчик события OnClick для кнопки SaveBtn }
```

```
var
  SLI: TShellLinkInfo;
begin
  GetControls(SLI);
  try
    SetShellLinkInfo(edLink.Text, SLI);
  except
    on EShellOleError do
      MessageDlg('Error occurred while setting info',
                 mtError, [mbOk], 0);
  end;
end;

procedure TMainForm.btnExitClick(Sender: TObject);
{ Обработчик события OnClick для кнопки ExitBtn }
begin
  Close;
end;

procedure TMainForm.About1Click(Sender: TObject);
{ Обработчик события OnClick для пункта меню Help|About. }
begin
  AboutBox;
end;

end.
```

Листинг 16.7. `NewLinkU.pas` — модуль формы, предназначенной для создания нового ярлыка

```
unit NewLinkU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Buttons, StdCtrls;

type
  TNewLinkForm = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    edLinkTo: TEdit;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    sbOpen: TSpeedButton;
    OpenDialog: TOpenDialog;
    procedure sbOpenClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;
```

```
function GetNewLinkName(var LinkTo: string;
                        var Dest: Integer): Boolean;

implementation

uses WinShell;

{$R *.DFM}

function GetNewLinkName(var LinkTo: string;
                        var Dest: Integer): Boolean;
{ Получить имя файла и папку назначения для хранения нового ярлыка.
Функция модифицирует параметры только в том случае, если Result =
True. }
begin
  with TNewLinkForm.Create(Application) do
  try
    cbLocation.ItemIndex := 0;
    Result := ShowModal = mrOk;
    if Result then begin
      LinkTo := edLinkTo.Text;
      Dest := cbLocation.ItemIndex;
    end;
  finally
    Free;
  end;
end;

procedure TNewLinkForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    edLinkTo.Text := OpenDialog.FileName;
end;

procedure TNewLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```

Листинг 16.8. `PickU.pas` — модуль формы, разрешающей пользователю выбрать местоположение ярлыка

```
unit PickU;

interface
```

```
{$WARN UNIT_PLATFORM OFF}

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, FileCtrl;

type
  TLinkForm = class(TForm)
    lbLinkFiles: TFileListBox;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    Label1: TLabel;
    procedure lbLinkFilesDblClick(Sender: TObject);
    procedure cbLocationChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

function GetLinkFile(var S: String): Boolean;

implementation

{$R *.DFM}

uses WinShell, ShlObj;

function GetLinkFile(var S: String): Boolean;
{ Имя файла ярлыка возвращается в параметр S. Параметр S
модифицируется только в том случае, если Result = True. }
begin
  with TLinkForm.Create(Application) do
    try
      { Удостовериться, что место выбрано. }
      cbLocation.ItemIndex := 0;
      { Получить путь выбранного местоположения. }
      cbLocationChange(nil);
      Result := ShowModal = mrOk;
      { Получить полный путь для файла ярлыка }
      if Result then
        S := lbLinkFiles.Directory + '\' +
             lbLinkFiles.Items[lbLinkFiles.ItemIndex];
    finally
      Free;
    end;
end;

procedure TLinkForm.lbLinkFilesDblClick(Sender: TObject);
begin
  ModalResult := mrOk;
end;

procedure TLinkForm.cbLocationChange(Sender: TObject);
var
```

```
  Folder: Integer;
begin
  { Получить путь выбранного местоположения. }
  Folder := SpecialFolders[cbLocation.ItemIndex].ID;
  lbLinkFiles.Directory := GetSpecialFolderPath(Folder, False);
end;

procedure TLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```

Расширения оболочки

Для получения предельной расширяемости в оболочке Windows предусмотрены средства, позволяющие создать код, который выполняется в рамках процесса самой оболочки и в ее пространстве имен. *Расширения оболочки* (shell extensions) реализуются в виде внутренних серверов COM, которые создаются и используются оболочкой Windows.

НА ЗАМЕТКУ

Поскольку расширения оболочки являются серверами COM, более подробная информация о них приведена в главе 15, “Разработка приложений COM”.

Существует несколько типов расширений оболочки, связанных с различными аспектами ее деятельности. Любое расширение оболочки (другое название — *обработчик* (handler)) должно реализовывать по крайней мере один интерфейс COM. Оболочка поддерживает следующие типы расширений.

- **Обработчик копирования** (copy hook handler) реализуют интерфейс `ICopyHook`. Эти расширения позволяют получать уведомления о копировании, перемещении, удалении или переименовании папок, а также предотвращать выполнение всех этих операций.
- **Обработчик контекстного меню** (context menu handler) реализует интерфейсы `IContextMenu` и `IShellExtInit`. Эти расширения позволяют добавлять команды в контекстное меню определенного файлового объекта оболочки.
- **Обработчик перетаскивания** (drag-and-drop handler) также реализуют интерфейсы `IContextMenu` и `IShellExtInit`. Эти расширения практически эквивалентны реализации обработчиков контекстного меню, за исключением того, что они вызываются в том случае, когда пользователь перетаскивает объект в новое место.

- **Обработчик пиктограмм** (icon handler) реализует интерфейсы `IExtractIcon` и `IPersistFile`. Эти обработчики позволяют присваивать пиктограммы различным экземплярам файлового объекта одного типа.
- **Обработчик вкладок** (property sheet handler) реализует интерфейсы `IShellPropSheetExt` и `IShellExtInit`. Они позволяют добавлять вкладки к диалоговым окнам свойств, ассоциированным с данным типом файла.
- **Обработчик наложения** (drop target handler) реализует интерфейсы `IDropTarget` и `IPersistFile`. Эти расширения позволяют определить действия оболочки при перетаскивании и опускании одного объекта оболочки на другой.
- **Обработчик объектов данных** (data object handler) реализует интерфейсы `IDataObject` и `IPersistFile`. Эти расширения предоставляют объекты данных, которые используются при перетаскивании с помощью мыши, копировании (команда Copy) и вставке (команда Paste) файлов.

Отладка расширений оболочки

Прежде чем углубляться в детали создания расширений оболочки, остановимся на теме их отладки. Поскольку расширения оболочки выполняются в пределах собственного процесса оболочки Windows, возникает закономерный вопрос: "Как же "внедриться" (hook into) в оболочку для отладки ее расширений?".

Решение проблемы основано на том, что оболочка представляет собой исполняемый файл (не слишком отличающийся от любого другого приложения), который называется `explorer.exe`. Как и любой другой исполняемый файл, это приложение имеет свойства. Но именно здесь и заключается его отличие от других исполняемых файлов: оболочку вызывает лишь первый экземпляр `explorer.exe`. Все последующие экземпляры просто вызывают дополнительные окна Explorer (Проводник).

Используя один малоизвестный трюк, можно закрыть оболочку, не выходя из Windows. Для отладки расширений в среде Delphi можно воспользоваться следующим алгоритмом.

1. Выбрав в меню Run пункт Parameter, сделайте файл `explorer.exe` основным приложением для расширения оболочки. Убедитесь, что задан полный путь к файлу (например `c:\windows\explorer.exe`).
2. В меню кнопки Start (Пуск) оболочки Windows выберите пункт Shut Down (Завершение работы). При этом откроется соответствующее диалоговое окно.
3. В диалоговом окне Shut Down Windows (Завершение работы Windows) щелкните на кнопке No (Отмена), удерживая при этом нажатой комбинацию клавиш <Ctrl+Alt+Shift>. Оболочка закроется, но среда Windows продолжит работу.
4. Используя комбинацию клавиш <Alt+Tab>, переключитесь в Delphi и запустите расширение оболочки. При этом под управлением отладчика Delphi будет вызвана новая копия оболочки. Теперь можно установить в нужных местах программы точки останова и отлаживать расширение оболочки как обычное приложение.
5. Если необходимо выйти из Windows, эта задача вполне осуществима и без использования сервиса оболочки: с помощью комбинации клавиш <Ctrl+Esc> вызовите меню Tasks (Задачи) и выберите в нем пункты Windows и Shut Down.

В последующих разделах настоящей главы вышеописанные расширения оболочки рассматриваются более подробно. Обсудим использование обработчиков копирования, контекстного меню и пиктограмм.

Мастер объектов COM

Прежде чем приступать к обсуждению каждой из библиотек, поддерживающих то или иное расширение оболочки, рассмотрим процедуру их создания. Поскольку расширения оболочки являются внутренними серверами COM, можно позволить интегрированной среде разработки Delphi выполнить бо́льшую часть рутинной работы по созданию исходного кода. Для каждого из расширений работа начинается со следующих двух действий:

1. Во вкладке ActiveX диалогового окна New Item дважды щелкните на пиктограмме ActiveX Library. При этом создается новая библиотека DLL сервера COM, в которую можно будет добавлять новые объекты COM.
2. Во вкладке ActiveX диалогового окна New Item дважды щелкните на пиктограмме COM Object. Вызывается мастер создания сервера COM. В диалоговом окне мастера введите имя и описание для создаваемого расширения оболочки и выберите вариант модели работы с потоками Apartment. По щелчку на кнопке OK будет создан новый модуль, содержащий код для разрабатываемого объекта COM.

Обработчики копирования

Как уже упоминалось, расширения оболочки, связанные с копированием, позволяют устанавливать обработчики, которые будут получать уведомления всякий раз при выполнении операции копирования, удаления, перемещения или переименования. После получения такого уведомления обработчик может пресечь выполнение соответствующей процедуры. Учтите, что этот обработчик вызывается только для объектов папок и принтеров и не вызывается для файлов и других объектов.

Создание обработчика копирования начинается с создания класса, производного от класса `TComObject`, в котором следует реализовать интерфейс `ICopyHook`. Этот интерфейс определен в модуле `ShlObj` следующим образом:

```
type
  ICopyHook = interface(IUnknown)
    ['{000214EF-0000-0000-C000-000000000046}']
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
                          pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
                          pszDestFile: PAnsiChar;
                          dwDestAttribs: DWORD): UINT; stdcall;
  end;
```

Метод CopyCallback()

Как видите, `ICopyHook` — достаточно простой интерфейс; в нем реализована лишь одна функция `CopyCallback()`. Данная функция вызывается при любых манипуляциях с папкой. Ниже описаны ее параметры.

Параметр `Wnd` — это дескриптор окна, которое будет использовано обработчиком копирования в качестве родительского для отображаемых им окон.

Параметр `wFunc` указывает на выполняемую операцию; данный параметр может принимать одно из значений, приведенных в табл. 16.5.

Таблица 16.5. Значения параметра `wFunc` функции `CopyCallback()`

Константа	*Значение*	*Назначение*
FO_COPY	$2	Копирует файл, задаваемый параметром pszSrcFile, в новое место, заданное параметром pszDestFile
FO_DELETE	$3	Удаляет файл, определяемый параметром pszSrcFile
FO_MOVE	$1	Перемещает файл, определяемый параметром pszSrcFile, в новое место, заданное параметром pszDestFile
FO_RENAME	$4	Переименовывает файл, определяемый параметром pszSrcFile
PO_DELETE	$13	Удаляет принтер, задаваемый параметром pszSrcFile
PO_PORTCHANGE	$20	Изменяет порт принтера. Параметры pszSrcFile и pszDestFile содержат списки, состоящие из строк, которые завершаются двумя нулевыми символами. Каждый список содержит имя принтера, за которым следует имя порта. Имя порта из параметра pszSrcFile является именем порта текущего принтера, а имя порта из параметра pszDestFile — именем порта нового принтера
PO_RENAME	$14	Переименовывает принтер, задаваемый параметром pszSrcFile
PO_REN_PORT	$34	Комбинация параметров PO_RENAME и PO_PORTCHANGE

Параметр wFlags содержит флаги, управляющие операцией. Этот параметр может быть комбинацией значений, представленных в табл. 16.6.

Таблица 16.6. Значения параметра wFlags функции CopyCallback()

Константа	*Значение*	*Назначение*
FOF_ALLOWUNDO	$40	Сохраняет информацию для отмены действия (если это возможно) (undo)
FOF_MULTIDESTFILES	$1	Вместо одной папки для хранения всех исходных файлов функция SHFileOperation() определяет множество конечных файлов (по одному для каждого исходного файла). Обработчик копирования обычно игнорирует это значение

Окончание табл. 16.6.

Константа	*Значение*	*Назначение*
FOF_NOCONFIRMATION	$10	В любом диалоговом окне ответом является вариант Yes to All (Да, для всех)
FOF_NOCONFIRMMKDIR	$200	Если операция требует создания новой папки, то подтверждающее сообщение отображаться не будет
FOF_RENAMEONCOLLISION	$8	Предлагает для файла новое имя (например Copy #1 of... (Копия #1...)) при операциях копирования, перемещения или переименования, если файл назначения с указанным именем уже существует
FOF_SILENT	$4	Не отображает индикатор процесса
FOF_SIMPLEPROGRESS	$100	Отображает индикатор процесса, но не показывает имена файлов

Используются также следующие параметры: `pszSourceFile` — имя исходной папки; `dwSrcAttribs` — атрибуты исходной папки; `pszDestFile` — имя папки назначения; `dwDestAttribs` — атрибуты папки назначения.

В отличие от большинства методов, данный интерфейс не возвращает результирующего кода OLE. Вместо этого он возвращает одно из значений, определенных в модуле `Windows` (они перечислены в табл. 16.7).

Таблица 16.7. Значения параметра `wFlags`, возвращаемые функцией `CopyCallback()`

Константа	*Значение*	*Назначение*
IDYES	6	Разрешает операцию
IDNO	7	Препятствует выполнению операции над текущим файлом, однако продолжает работу с остальными операциями (например в случае копирования с использованием командного файла)
IDCANCEL	2	Препятствует выполнению текущей операции и отменяет все остальные операции

Реализация TCopyHook

Поскольку класс `TCopyHook` реализует интерфейс лишь с одним методом, его описание выглядит достаточно компактно:

```
type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
```

```
                        pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
                        pszDestFile: PAnsiChar;
                        dwDestAttribs: DWORD): UINT; stdcall;
  end;
```

Реализация метода `CopyCallback()` также невелика по объему. Для подтверждения выполнения какой-либо операции вызывается функция API Win32 `MessageBox()`. Кстати, значение, возвращаемое методом `CopyCallback()`, аналогично значению, возвращаемому функцией `MessageBox()`:

```
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
                      pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
                      pszDestFile: PAnsiChar;
                      dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = 'Are you sure you want to mess with "%s"?';
begin
  // Подтвердить операцию
  Result := MessageBox(Wnd, PChar(Format(MyMessage,
                       [pszSrcFile])), 'DDG Shell Extension',
                       MB_YESNO);
end;
```

СОВЕТ

Обратите внимание: для вывода сообщений вместо функции Delphi `MessageDlg()` или `ShowMessage()` используется функция API Win32 `MessageBox()`. Причина этого проста: она заключается в размере кода и эффективности его выполнения. Вызов любой функции модуля `Dialogs` или `Forms` привел бы к тому, что к создаваемой библиотеке DLL была бы подсоединена значительная часть кода библиотеки VCL. Отказавшись от использования функций из таких двух модулей, можно существенно сократить размер созданной библиотеки DLL. Примерно на 70 Кбайт.

Возможно, в это трудно поверить, но об объекте `TCopyHook` сказано все, что нужно. Да, вот еще одна деталь: прежде чем использовать любое расширение оболочки, оно должно быть зарегистрировано в системном реестре.

Регистрация

Кроме обычной регистрации, необходимой любому серверу COM, обработчик копирования должен иметь дополнительную точку входа в системный реестр в следующей ветви:

```
HKEY_CLASSES_ROOT\directory\shellex\CopyHookHandlers
```

Более того, в Windows NT требуется, чтобы все расширения оболочки были зарегистрированы как утвержденные (approved). В этом случае регистрация осуществляется и в дополнительной ветви:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
↳ShellExtensions\Approved
```

Для регистрации расширений оболочки можно использовать несколько способов. Они могут быть зарегистрированы с помощью файла `.reg` или программы установки. И, наконец, библиотека DLL с расширением оболочки может быть саморегистрирующейся. Последнее решение, хотя и требует несколько бо́льших усилий, является все же наилучшим, поскольку при этом расширение является самодостаточным пакетом, размещенным в единственном файле.

В главе 15, "Разработка приложений COM", речь шла о том, что объекты COM всегда создаются из фабрик классов. В среде VCL объекты фабрики класса ответственны также и за регистрацию создаваемого объекта COM. Если объект COM требует точки входа в системный реестр (как в случае с расширением оболочки), то для их установки достаточно переопределить стандартный метод `UpdateRegistry()` фабрики класса. В листинге 16.9 приведен код модуля `CopyMain`, содержащего специализированную фабрику класса для выполнения регистрации.

Листинг 16.9. `CopyMain.pas` — главный модуль реализации обработчика копирования

```
unit CopyMain;

interface

uses Windows, ComObj, ShlObj;

type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
              pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
              pszDestFile: PAnsiChar;
              dwDestAttribs: DWORD): UINT; stdcall;
  end;

  TCopyHookFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean;
                                    const ClsID: string); virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, Registry;

{ TCopyHook }

// Этот метод вызывается оболочкой для операций с папками
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
                    pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
                    pszDestFile: PAnsiChar;
```

```
                    dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = 'Are you sure you want to mess with "%s"?';
begin
  // Подтвердить операцию
  Result := MessageBox(Wnd, PChar(Format(MyMessage,
                       [pszSrcFile])), 'DDG Shell Extension',
                       MB_YESNO);
end;

{ TCopyHookFactory }

function TCopyHookFactory.GetProgID: string;
begin
  // Идентификатор программы (ProgID) для расширения оболочки
  // не нужен.
  Result := '';
end;

procedure TCopyHookFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
    // Добавить идентификатор класса (clsid) расширения оболочки в
    // раздел CopyHookHandlers системного реестра.
    CreateRegKey('directory\shellex\CopyHookHandlers\' +
                 ClassName, '', ClsID)
  else
    DeleteRegKey('directory\shellex\CopyHookHandlers\' +
                 ClassName);
end;

procedure TCopyHookFactory.ApproveShellExtension(Register:
                                    Boolean; const ClsID: string);
// Этот элемент системного реестра необходим для корректной работы
// расширения под управлением Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\
↳Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
```

```
end;

const
  CLSID_CopyHook:TGUID = '{66CD5F60-A044-11D0-A9BF-00A024E3867F}';

initialization
  TCopyHookFactory.Create(ComServer, TCopyHook, CLSID_CopyHook,
       'DDG_CopyHook', 'DDG Copy Hook Shell Extension Example',
       ciMultiInstance, tmApartment);
end.
```

Движущей силой, заставляющей фабрику класса `TCopyHookFactory` работать, является тот факт, что именно ее экземпляр, а не обычный объект `TComObjectFactory`, создается в разделе инициализации (`initialization`) этого модуля. На рис. 16.7 показано, что происходит при попытке переименовать папку в системной оболочке после установки библиотеки DLL с расширением копирования.

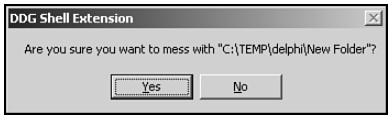

Рис. 16.7. *Расширение копирования в действии*

Обработчики контекстных меню

Обработчики контекстных меню позволяют добавлять новые пункты в контекстные меню, связанные с файловыми объектами оболочки. Пример контекстного меню для исполняемого файла показан на рис. 16.8.

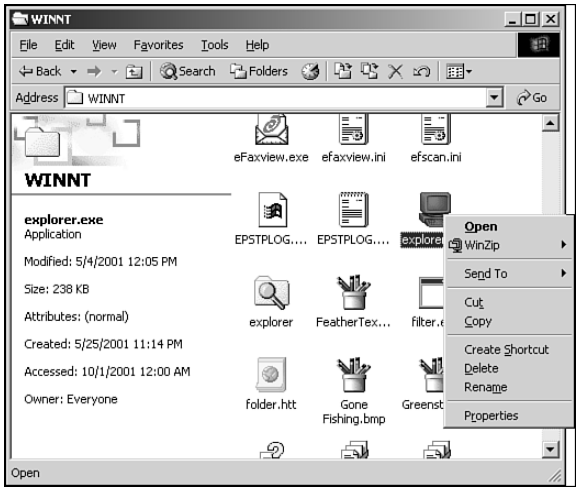

Рис. 16.8. *Контекстное меню исполняемого файла*

Действие расширений оболочки, связанных с контекстными меню, основано на реализации интерфейсов IShellExtInit и IContextMenu. В данном примере эти интерфейсы реализованы для создания обработчика контекстного меню для файлов BPL (Borland Package Library). В контекстное меню для файлов данного типа будет добавлена команда, позволяющая получить информацию о содержащемся в файле пакете. Объект обработчика контекстного меню назовем TContextMenu и, подобно обработчику копирования, сделаем класс TContextMenu производным от класса TComObject.

Интерфейс IShellExtInit

Для инициализации расширения оболочки используется интерфейс IShellExtInit, который определен в модуле ShlObj следующим образом:

```
type
  IShellExtInit = interface(IUnknown)
    ['{000214E8-0000-0000-C000-000000000046}']
    function Initialize(pidlFolder: PItemIDList;
                        lpdobj: IDataObject;
                        hKeyProgID: HKEY): HResult; stdcall;
    end;
```

В этом интерфейсе использован единственный метод — Initialize() — который и инициализирует обработчик контекстного меню. Ниже описаны параметры данного метода.

Параметр pidlFolder является указателем на структуру PItemIDList (список идентификаторов элементов) для папки, содержащей элемент, к которому относится отображаемое контекстное меню. Параметр lpdobj содержит объект интерфейса IDataObject, используемый для получения объектов, над которыми выполняется действие. Параметр hKeyProgID содержит ключ системного реестра для объекта файлового типа или для папки.

Реализация метода Initialize() приведена ниже. На первый взгляд этот код кажется довольно сложным, однако на самом деле все сводится к трем действиям: вызову функции lpobj.GetData() для получения данных из интерфейса IDataObject и двум вызовам функции DragQueryFile() (один — для получения количества файлов, а другой — для получения имени файла). Имя файла сохраняется в поле FFileName объекта lpdobj.

```
function TContextMenu.Initialize(pidlFolder: PItemIDList;
                  lpdobj: IDataObject; hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    // Аварийный выход из функции, если указатель на объект lpdobj
    // равен значению nil.
    if lpdobj = nil then begin
      Result := E_FAIL;
      Exit;
    end;
    with FE do begin
```

```
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
    // Вернуть данные, на которые ссылается указатель
    // типа IDataObject, в формате CF_HDROP
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
    // Если выбран только один файл, считать его имя и сохранить в
    // поле szFile. В противном случае аварийное завершение
    // работы функции.
    if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
    begin
      DragQueryFile(Medium.hGlobal, 0, FFileName,
                    SizeOf(FFileName));
      Result := NOERROR;
    end
    else
      Result := E_FAIL;
    finally
      ReleaseStgMedium(medium);
    end;
    except
      Result := E_UNEXPECTED;
    end;
end;
```

Интерфейс IContextMenu

Интерфейс `IContextMenu` используется для управления контекстным меню, ассоциированным с данным типом файлов в системной оболочке. Этот интерфейс также определен в модуле `ShlObj` следующим образом:

```
type
  IContextMenu = interface(IUnknown)
    ['{000214E4-0000-0000-C000-000000000046}']
    function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
                        idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici:
                          TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT;
                              pwReserved: PUINT; pszName: LPSTR;
                              cchMax: UINT): HResult; stdcall;
  end;
```

После инициализации обработчика, через интерфейс `IShellExtInit`, вызывается метод `IContextMenu.QueryContextMenu()`. В список параметров, передаваемых методу, входит обработчик меню, индекс, соответствующий первому пункту меню, минимальное и максимальное значения идентификаторов пунктов меню, а также флаги атрибутов меню. Ниже приведена реализация метода в объекте `TContextMenu`.

Этот метод используется для добавления пункта меню Package info в обработчик меню, передаваемый с помощью параметра Menu (заметьте, что значение, возвращаемое функцией QueryContextMenu(), на единицу больше индекса пункта, который был введен последним).

```
function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu,
                  idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIdx := indexMenu;
  // Добавить в контекстное меню один пункт.
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION,
              idCmdFirst, 'Package Info...');
  // Вернуть индекс последнего добавленного элемента + 1
  Result := FMenuIdx + 1;
end;
```

Следующий вызываемый оболочкой метод (GetCommandString()) предназначен для получения независимой от языка командной строки или справочной строки для конкретного пункта меню. В список параметров, передаваемых методу, входит номер пункта меню, флаги, указывающие на тип получаемой информации, зарезервированный параметр, строковый буфер и его размер. Приведенная ниже реализация этого метода в объекте TContextMenu отображает справочную строку для данного пункта меню.

```
function TContextMenu.GetCommandString(idCmd, uType: UINT;
       pwReserved: PUINT; pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
    // Удостовериться, что индекс меню правильный и оболочка
    // запросила строку подсказки
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
      // Возвратить строку подсказки для данного пункта меню
      StrLCopy(pszName,
               'Get information for the selected package.',
               cchMax)
    else
      Result := E_INVALIDARG;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

Метод InvokeCommand() вызывается каждый раз при щелчке на новом пункте меню. В качестве параметра этому методу передается запись TCMInvokeCommandInfo, которая определена в модуле ShlObj следующим образом:

```
type
  PCMInvokeCommandInfo = ^TCMInvokeCommandInfo;
  TCMInvokeCommandInfo = packed record
    { Должно иметь значение SizeOf(TCMInvokeCommandInfo). }
    cbSize: DWORD;
    { Любая комбинация CMIC_MASK_* }
```

```
    fMask: DWORD;
    { При отсутствии окна-владельца принимает значение NULL. }
    hwnd: HWND;
    { Любая строка AKEINTRESOURCE(idOffset). }
    lpVerb: LPCSTR;
    { При отсутствии параметра принимает значение NULL }
    lpParameters: LPCSTR;
    { При отсутствии заданной папки принимает значение NULL. }
    lpDirectory: LPCSTR;
    { Одно из значений SW_ функции API ShowWindow(). }
    nShow: Integer;
    dwHotKey: DWORD;
    hIcon: THandle;
  end;
```

Младшее слово (два байта) поля `lpVerb` будет содержать индекс выбранного пункта меню. Ниже приведена реализация этого метода.

```
function TContextMenu.InvokeCommand(var lpici:
                                    TCMInvokeCommandInfo): HResult;
begin
  Result := S_OK;
  try
    // Удостовериться, что вызов осуществляется не приложением
    if HiWord(Integer(lpici.lpVerb)) <> 0 then begin
      Result := E_FAIL;
      Exit;
    end;
    // Выполнить команду, определенную в lpici.lpVerb. Если
    // передан недопустимый номер аргумента, то возвратить
    // статус E_INVALIDARG.
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, 'Error obtaining package information.',
               'Error', MB_OK or MB_ICONERROR);
    Result := E_FAIL;
  end;
end;
```

Если все проходит успешно, то с помощью функции `ExecutePackInfoApp()` вызывается приложение `PackInfo.exe`, отображающее разного рода информацию о пакете. Здесь данное приложение не рассматривается, но информация по этой теме приведена в главе 13, "Дополнительный инструментарий разработчика", предыдущего издания *Delphi 5 Руководство разработчика*, находящегося на прилагаемом CD.

Регистрация

В системном реестре обработчики контекстного меню должны регистрироваться в следующей ветви:

`HKEY_CLASSES_ROOT\<тип файла>\shellex\ContextMenuHandlers`

Как и в случае обработчика копирования, возможность регистрации DLL контекстного меню реализована с помощью потомка объекта `TComObject`. Код модуля, содержащего объект `TContextMenu`, приведен в листинге 16.10. На рис. 16.9 показано контекстное меню для файла `.bpl` с новым пунктом, а на рис. 16.10 представлен вид окна программы `PackInfo.exe`, вызываемой обработчиком контекстного меню.

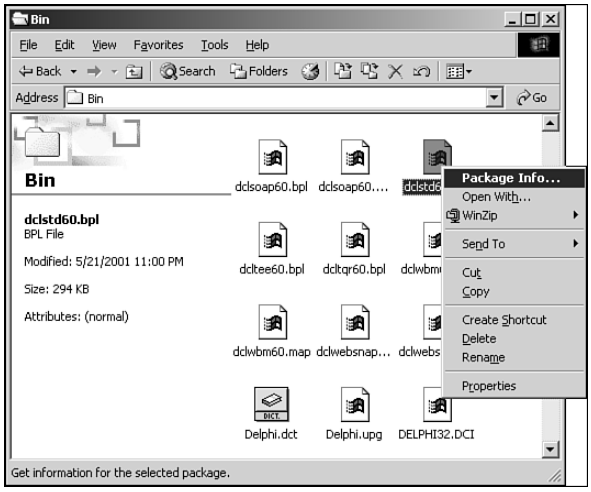

Рис. 16.9. *Обработчик контекстного меню в действии*

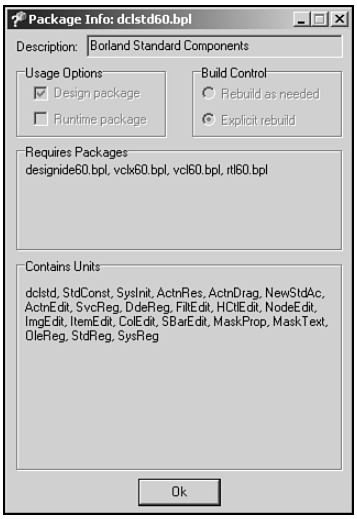

Рис. 16.10. *Получение информации о пакете с помощью обработчика контекстного меню*

Листинг 16.10. `ContMain.pas` — главный модуль реализации обработчика контекстного меню

```
unit ContMain;

interface

uses Windows, ComObj, ShlObj, ActiveX;

type
  TContextMenu = class(TComObject, IContextMenu, IShellExtInit)
  private
    FFileName: array[0..MAX_PATH] of char;
    FMenuIdx: UINT;
  protected
    // Методы интерфейса IContextMenu
    function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
                        idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici:
                        TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT;
                        pwReserved: PUINT; pszName: LPSTR;
                        cchMax: UINT): HResult; stdcall;
    // Методы интерфейса IShellExtInit
    function Initialize(pidlFolder: PItemIDList;
             lpdobj: IDataObject; hKeyProgID: HKEY): HResult;
             reintroduce; stdcall;
  end;

  TContextMenuFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean;
                                    const ClsID: string); virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, ShellAPI, Registry, Math;

procedure ExecutePackInfoApp(const FileName: string;
                             ParentWnd: HWND);
const
  SPackInfoApp = '%sPackInfo.exe';
  SCmdLine = '"%s" %s';
  SErrorStr = 'Failed to execute PackInfo:'#13#10#13#10;
var
  PI: TProcessInformation;
  SI: TStartupInfo;
  ExeName, ExeCmdLine: string;
```

```
  Buffer: array[0..MAX_PATH] of char;
begin
  // Получить папку данной DLL. Предполагаем, что исполняемый
  // файл (EXE) находится в этой же папке.
  GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
  ExeName := Format(SPackInfoApp, [ExtractFilePath(Buffer)]);
  ExeCmdLine := Format(SCmdLine, [ExeName, FileName]);
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  if not CreateProcess(PChar(ExeName), PChar(ExeCmdLine), nil,
                       nil, False, 0, nil, nil, SI, PI) then
    MessageBox(ParentWnd, PChar(SErrorStr +
               SysErrorMessage(GetLastError)), 'Error',
               MB_OK or MB_ICONERROR);
end;

{ TContextMenu }

{ TContextMenu.IContextMenu }

function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu,
                  idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIdx := indexMenu;
  // Добавить в контекстное меню один пункт.
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION,
              idCmdFirst, 'Package Info...');
  // Вернуть индекс последнего добавленного элемента + 1
  Result := FMenuIdx + 1;
end;

function TContextMenu.InvokeCommand(var lpici:
                                    TCMInvokeCommandInfo): HResult;
begin
  Result := S_OK;
  try
    // Удостовериться, что вызов осуществляется не приложением
    if HiWord(Integer(lpici.lpVerb)) <> 0 then begin
      Result := E_FAIL;
      Exit;
    end;
    // Выполнить команду, определенную в lpici.lpVerb. Если
    // передан недопустимый номер аргумента, то возвратить
    // статус E_INVALIDARG.
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, 'Error obtaining package information.',
               'Error', MB_OK or MB_ICONERROR);
    Result := E_FAIL;
  end;
```

```
end;

function TContextMenu.GetCommandString(idCmd, uType: UINT;
                            pwReserved: PUINT; pszName: LPSTR;
                            cchMax: UINT): HRESULT;
const
  SCmdStrA: String = 'Get information for the selected package.';
  SCmdStrW: WideString =
                     'Get information for the selected package.';
begin
  Result := S_OK;
  try
    // Удостовериться, что индекс меню правильный, и оболочка
    // запросила строку подсказки
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
    begin  // Возвратить строку подсказки для данного пункта меню
      if Win32MajorVersion >= 5 then
        // Обработать как unicode для Win2k или выше
        Move(SCmdStrW[1], pszName^,
          Min(cchMax, Length(SCmdStrW) + 1) * SizeOf(WideChar))
      else                      // В противном случае как ANSI
        StrLCopy(pszName, PChar(SCmdStrA), Min(cchMax,
                 Length(SCmdStrA) + 1));
    end
    else
      Result := E_INVALIDARG;
  except
    Result := E_UNEXPECTED;
  end;
end;

{ TContextMenu.IShellExtInit }

function TContextMenu.Initialize(pidlFolder: PItemIDList;
                  lpdobj: IDataObject; hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    // Аварийный выход из функции, если указатель на объект lpdobj
    // равен значению nil.
    if lpdobj = nil then begin
      Result := E_FAIL;
      Exit;
    end;
    with FE do begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
```

```
    // Вернуть данные, на которые ссылается указатель
    // типа IDataObject, в формате CF_HDROP.
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
    // Если выбран только один файл, считать его имя и сохранить в
    // поле szFile. В противном случае аварийное завершение
    // работы функции.
      if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        DragQueryFile(Medium.hGlobal, 0, FFileName,
                      SizeOf(FFileName));
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
    finally
      ReleaseStgMedium(medium);
    end;
  except
    Result := E_UNEXPECTED;
  end;
end;

{ TContextMenuFactory }

function TContextMenuFactory.GetProgID: string;
begin
  // Для расширения оболочки, управляющего работой контекстного
  // меню, идентификатор программы ProgID не нужен.
  Result := '';
end;

procedure TContextMenuFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then begin
    // Необходимо зарегистрировать тип файла .bpl
    CreateRegKey('.bpl', '', 'BorlandPackageLibrary');
    // Регистрация данной DLL в качестве обработчика контекстного
    // меню для файлов .bpl.
    CreateRegKey(
            'BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
             ClassName, '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey(
            'BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
```

```
            ClassName);
  end;
end;

procedure TContextMenuFactory.ApproveShellExtension(
                    Register: Boolean; const ClsID: string);
// Этот элемент системного реестра необходим для корректной работы
// расширения под управлением Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\
↳Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_CopyHook: TGUID ='{7C5E74A0-D5E0-11D0-A9BF-E886A83B9BE5}';

initialization
  TContextMenuFactory.Create(ComServer, TContextMenu,
                      CLSID_CopyHook, 'DDG_ContextMenu',
                      'DDG Context Menu Shell Extension Example',
                      ciMultiInstance, tmApartment);
end.
```

Обработчики пиктограмм

Обработчики пиктограмм позволяют применять пиктограммы для различных экземпляров файлов одного типа. В данном примере описан объект обработчика пиктограмм `TIconHandler`, который обеспечивает различные пиктограммы для разных типов пакетных файлов Borland Package (BPL). В зависимости от типа пакета — времени выполнения, времени разработки, универсального или иного — при отображении этих файлов в папке оболочки будут использоваться различные пиктограммы.

Флаги пакета

Прежде чем представить интерфейсы, необходимые для реализации данного расширения оболочки, рассмотрим метод, используемый для определения типа файла пакета. Этот метод возвращает объект типа `TPackType`, который определен следующим образом:

```
TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);
```

Приведем код метода:

```
function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: Integer;
begin
  // Поскольку необходимо получить доступ только к ресурсам
  // пакета, используем функцию LoadLibraryEx с
  // параметром LOAD_LIBRARY_AS_DATAFILE, обеспечивающим высокую
  // скорость загрузки пакета.
  PackMod := LoadLibraryEx(PChar(FFileName), 0,
                           LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
  // Отфильтровать по маске все флаги, кроме design и run,
  // а результат возвратить.
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;
```

Задача этого метода заключается в вызове метода `GetPackageInfo()` из модуля `SysUtils` для получения флагов пакета. Отметим, что для оптимизации производительности вместо встроенной процедуры Delphi `LoadPackage()` для загрузки библиотеки пакета вызывается функция API `LoadLibraryEx()`. Внутри функции `LoadPackage()` содержится вызов функции API `LoadLibrary()`, которая загружает библиотеку BPL. После загрузки библиотеки вызывается функция `InitializePackage()`, выполняющая инициализацию каждого модуля в пакете. Но поскольку в данном случае нужно лишь получить флаги пакета, которые хранятся в файле ресурсов, связанном с библиотекой BPL, можно вполне обойтись загрузкой пакета с помощью функции `LoadLibraryEx()` с флагом `LOAD_LIBRARY_AS_DATAFILE` в качестве параметра.

Интерфейсы обработчика пиктограмм

Как уже упоминалось в настоящей главе, обработчики пиктограмм должны поддерживать оба интерфейса — и `IExtractIcon` (определенный в модуле `ShlObj`), и `IPersistFile` (определенный в модуле `ActiveX`). Определения этих интерфейсов имеют следующий вид:

```
type
  IExtractIcon = interface(IUnknown)
    ['{000214EB-0000-0000-C000-000000000046}']
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
                             cchMax: UINT; out piIndex: Integer;
                             out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT;
                             out phiconLarge, phiconSmall: HICON;
                             nIconSize: UINT): HResult; stdcall;
  end;

  IPersistFile = interface(IPersist)
    ['{0000010B-0000-0000-C000-000000000046}']
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr;
                  dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr;
                  fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult;
                                                  stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult;
                                                  stdcall;
end;
```

Хотя на первый взгляд кажется, что данные интерфейсы выполняют значительный объем работ, однако это впечатление обманчиво: на самом деле только два из приведенных выше методов должны быть реализованы. Первый из них — метод `IPersistFileLoad()`. Данный метод вызывается для инициализации расширения оболочки, и внутри него нужно предусмотреть сохранение имени файла, передаваемого с помощью параметра `pszFileName`. Ниже приведена реализация такого метода в объекте `TExtractIcon`:

```
function TIconHandler.Load(pszFileName: POleStr;
                           dwMode: Longint): HResult;
begin
  // Этот метод вызывается для инициализированного обработчика
  // пиктограмм. Необходимо сохранить имя файла, которое передано
  // в параметре pszFileName.
  FFileName := pszFileName;
  Result := S_OK;
end;
```

Второй метод, подлежащий обязательной реализации, — это `IExtractIcon.GetIconLocation()`. Передаваемые ему параметры рассматриваются далее в настоящей главе.

Параметр `uFlags` указывает на тип отображаемой пиктограммы. Этот параметр может принимать значение `0`, `GIL_FORSHELL` или `GIL_OPENICON`. Значение `GIL_FORSHELL` свидетельствует о том, что пиктограмма будет отображаться в папке системной оболочки. Значение `GIL_OPENICON` указывает, что пиктограмма должна быть в "открытом" состоянии, если доступны изображения как для открытого, так и для закрытого состояний. Если данный флаг не задан, то пиктограмма должна быть в нормальном, т.е. "закрытом", состоянии. Этот флаг обычно используется для объектов папки.

Параметр szIconFile представляет собой буфер, в который передается информация о расположении пиктограммы. Параметр cchMax содержит размер такого буфера. Параметр piIndex — переменная целого типа — получает индекс пиктограммы, уточняющий ее расположение.

Параметр pwFlags может принимать нулевое значение либо одно из значений, приведенных в табл. 16.8.

Таблица 16.8. Значения параметра pwFlags функции GetIconLocation()

Флаг	*Назначение*
GIL_DONTCACHE	Битовая карта этой пиктограммы не подлежит кэшированию вызывающей процедурой. Это необходимо, поскольку в последующих версиях оболочки может быть введен флаг GIL_DONTCACHELOCATION
GIL_NOTFILENAME	Местонахождение пиктограммы не может быть описано парой значений "имя файла/индекс". Для получения изображения пиктограммы вызывающие процедуры должны обратиться к методу IExtractIcon.Extract()
GIL_PERCLASS	Все объекты данного класса имеют одну и ту же пиктограмму. Этот флаг используется внутренними средствами оболочки. Обычные реализации интерфейса IExtractIcon не требуют данного флага, поскольку для случаев, когда одному объекту соответствует одна пиктограмма, обработчик пиктограммы не требуется. Для реализации пиктограмм типа "одна пиктограмма на один класс" рекомендуется просто зарегистрировать стандартную пиктограмму для данного класса
GIL_PERINSTANCE	Каждый объект этого класса имеет свою собственную пиктограмму. Данный флаг используется оболочкой в случаях применения файла установки setup.exe. В таких случаях оболочка может "знать" о нескольких объектах, имеющих идентичные имена, но использовать различные пиктограммы. Обычные реализации интерфейса IExtractIcon не требуют этого флага
GIL_SIMULATEDOC	Вызывающая процедура должна создать пиктограмму документа, используя указанную пиктограмму

Реализация метода GetIconLocation() в объекте TIconHandler выглядит следующим образом:

```
function TIconHandler.GetIconLocation(uFlags: UINT;
                szIconFile: PAnsiChar; cchMax: UINT;
                out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;
  try
    // Возвращает DLL по имени модуля искомой пиктограммы
    GetModuleFileName(HInstance, szIconFile, cchMax);
    // Сообщает оболочке о том, что изображение не подлежит
```

```
    // кэшированию при изменении пиктограммы, а также о том, что
    // каждый экземпляр может иметь свою собственную пиктограмму.
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
    // Индекс пиктограммы согласуется с типом TPackType
    piIndex := Ord(GetPackageType);
  except
    // В случае ошибки используем пиктограмму по умолчанию.
    piIndex := Ord(ptNone);
  end;
end;
```

Пиктограммы связываются с библиотекой DLL расширения оболочки в виде файла ресурсов, поэтому имя текущего файла, возвращаемое функцией `GetModuleFileName()`, записывается в буфер `szIconFile`. Пиктограммы организованы таким образом, чтобы индекс пиктограммы для типа пакета соответствовал индексу типа пакета в перечне типов `TPackType`. Поэтому значение, возвращаемое функцией `GetPackageType()`, присваивается параметру `piIndex`.

Регистрация

Обработчики пиктограмм должны быть зарегистрированы в системном реестре в следующем виде:

`HKEY_CLASSES_ROOT\<тип файла>\shellex\IconHandler`

Как и в случае с другими расширениями, для регистрации обработчика пиктограмм создается потомок класса `TComObjectFactory`. Код модуля обработчика пиктограмм, включающий в себя код объекта `TComObjectFactory`, приведен в листинге 16.11, а на рис. 16.11 показана папка системной оболочки, содержащая пакеты различных типов. Обратите внимание: для отображения различных типов пакетов используются различные пиктограммы.

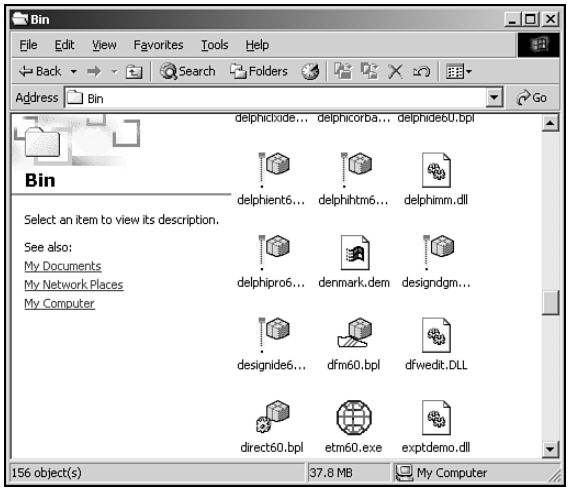

Рис. 16.11. *Результат использования обработчика пиктограмм*

Листинг 16.11. `IconMain.pas` — главный модуль реализации обработчика пиктограмм

```
unit IconMain;

interface

uses Windows, ActiveX, ComObj, ShlObj;

type
  TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);

  TIconHandler = class(TComObject, IExtractIcon, IPersistFile)
  private
    FFileName: string;
    function GetPackageType: TPackType;
  protected
    // Методы интерфейса IExtractIcon
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
                             cchMax: UINT; out piIndex: Integer;
                             out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT;
                             out phiconLarge, phiconSmall: HICON;
                             nIconSize: UINT): HResult; stdcall;
    // Методы интерфейса IPersist
    function GetClassID(out classID: TCLSID): HResult; stdcall;
    // Методы интерфейса IPersistFile
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr;
                  dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr;
                  fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult;
                                                    stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult;
                                                    stdcall;
  end;

  TIconHandlerFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean;
                                    const ClsID: string); virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses SysUtils, ComServ, Registry;

{ TIconHandler }
```

```
procedure PackInfoProc(const Name: string; NameType: TNameType;
                       Flags: Byte; Param: Pointer);
begin
  // В реализации данной процедуры нет необходимости, поскольку
  // нужны только флаги пакета, а не модули.
end;

function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: Integer;
begin
  // Поскольку необходимо получить доступ только к ресурсам
  // пакета, используем функцию LoadLibraryEx с
  // параметром LOAD_LIBRARY_AS_DATAFILE, обеспечивающим высокую
  // скорость загрузки пакета.
  PackMod := LoadLibraryEx(PChar(FFileName), 0,
                           LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
  // Отфильтровать по маске все флаги, кроме design и run,
  // а результат возвратить.
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;

{ TIconHandler.IExtractIcon }

function TIconHandler.GetIconLocation(uFlags: UINT;
                szIconFile: PAnsiChar; cchMax: UINT;
                out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;
  try
    // Возвращает DLL по имени модуля искомой пиктограммы
    GetModuleFileName(HInstance, szIconFile, cchMax);
    // Сообщить оболочке о том, что изображение не подлежит
    // кэшированию при изменении пиктограммы, а также о том, что
    // каждый экземпляр может иметь свою собственную пиктограмму.
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
```

```
    // Индекс пиктограммы согласуется с типом TPackType
    piIndex := Ord(GetPackageType);
  except
    // В случае ошибки используем пиктограмму по умолчанию.
    piIndex := Ord(ptNone);
  end;
end;

function TIconHandler.Extract(pszFile: PAnsiChar;
            nIconIndex: UINT; out phiconLarge, phiconSmall: HICON;
            nIconSize: UINT): HResult;
begin
  // Данный метод нужно реализовывать только в том случае, если
  // пиктограмма сохранена в каком-то пользовательском формате.
  // Поскольку данная пиктограмма содержится в DLL,
  // возвращается значение S_FALSE.
  Result := S_FALSE;
end;

{ TIconHandler.IPersist }

function TIconHandler.GetClassID(out classID: TCLSID): HResult;
begin
  // Для обработчиков пиктограмм данный метод не вызывается.
  Result := E_NOTIMPL;
end;

{ TIconHandler.IPersistFile }

function TIconHandler.IsDirty: HResult;
begin
  // Для обработчиков пиктограмм этот метод не вызывается.
  Result := S_FALSE;
end;

function TIconHandler.Load(pszFileName: POleStr;
                           dwMode: Longint): HResult;
begin
  // Этот метод вызывается для инициализированного обработчика
  // пиктограмм. Необходимо сохранить имя файла, которое передано
  // в параметре pszFileName.
  FFileName := pszFileName;
  Result := S_OK;
end;

function TIconHandler.Save(pszFileName: POleStr;
                           fRemember: BOOL): HResult;
begin
  // Для обработчиков пиктограмм этот метод не вызывается.
  Result := E_NOTIMPL;
end;

function TIconHandler.SaveCompleted(pszFileName: POleStr): HResult;
```

```
begin
  // Для обработчиков пиктограмм этот метод не вызывается.
  Result := E_NOTIMPL;
end;

function TIconHandler.GetCurFile(out pszFileName: POleStr): HRe-
sult;
begin
  // Для обработчиков пиктограмм этот метод не вызывается.
  Result := E_NOTIMPL;
end;

{ TIconHandlerFactory }

function TIconHandlerFactory.GetProgID: string;
begin
  // Для расширений контекстного меню ProgID не нужен.
  Result := '';
end;

procedure TIconHandlerFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then begin
    // Необходимо зарегистрировать .bpl как новый тип файла.
    CreateRegKey('.bpl', '', 'BorlandPackageLibrary');
    // Зарегистрировать эту библиотеку DLL как обработчик
    // пиктограмм для файлов .bpl.
    CreateRegKey('BorlandPackageLibrary\shellex\IconHandler',
                 '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey('BorlandPackageLibrary\shellex\IconHandler');
  end;
end;

procedure TIconHandlerFactory.ApproveShellExtension(Register: Boo-
lean;
  const ClsID: string);
// Этот элемент системного реестра необходим для корректной работы
// расширения под управлением Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\
↳Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
```

```
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_IconHandler: TGUID =
                       '{ED6D2F60-DA7C-11D0-A9BF-90D146FC32B3}';

initialization
  TIconHandlerFactory.Create(ComServer, TIconHandler,
                  CLSID_IconHandler, 'DDG_IconHandler',
                  'DDG Icon Handler Shell Extension Example',
                  ciMultiInstance, tmApartment);
end.
```

Обработчики контекстной подсказки

Контекстная подсказка (InfoTip) впервые появилась в оболочке Windows 2000. Обработчики контекстной подсказки обеспечивают специальные всплывающие контекстные информационные окна (известные в Delphi как ToolTip), которые возникают в оболочке при помещении мыши поверх пиктограммы, представляющей файл. Стандартная контекстная подсказка, отображаемая оболочкой Windows, содержит имя, тип (определенный на основании его расширения) и размер файла. Обработчики контекстной подсказки применяются тогда, когда необходимо отобразить несколько больше информации, чем предоставляет стандартный встроенный механизм.

Наибольший интерес для разработчиков Delphi представляют файлы пакетов. Несмотря на то, что все знают, что файлы пакетов могут содержать один или несколько модулей, но не известно наверняка, какие именно модули он содержит. Ранее в настоящей главе был представлен обработчик контекстного меню, способный предоставить подобную информацию. Для этого после выбора соответствующего пункта контекстного меню запускалось внешнее приложение, которое и предоставляло соответствующую информацию. Теперь рассмотрим, как получить такую информацию еще проще, и без использования внешней программы.

Интерфейсы обработчика контекстной подсказки

Обработчики контекстной подсказки должны реализовать интерфейсы `IPersistFile` и `IQueryInfo`. Интерфейс `IPersistFile` рассматривался ранее, при обсуждении ярлыков и обработчиков пиктограмм. Здесь он используется для того, чтобы получить имя рассматриваемого файла. Интерфейс `IQueryInfo` относительно прост — он содержит два метода — и определен в модуле `ShlObj` следующим образом:

```
type
  IQueryInfo = interface(IUnknown)
  [SID_IQueryInfo]
    function GetInfoTip(dwFlags: DWORD;
```

```
                    var ppwszTip: PWideChar): HResult; stdcall;
    function GetInfoFlags(out pdwFlags: DWORD): HResult; stdcall;
end;
```

Оболочка обращается к методу `GetInfoTip()`, чтобы вызвать контекстную подсказку для данного файла. Параметр `DwFlags` в настоящее время не используется. Параметр `ppwszTip` возвращает строку контекстной подсказки.

НА ЗАМЕТКУ

Параметр `PpwszTip` представляет собой указатель на символьную строку типа `WideChar`. Память для этой строки следует выделять внутри обработчика контекстной подсказки с помощью системы выделения памяти оболочки. Оболочка сама несет ответственность за освобождение этой памяти.

Реализация

Подобно другим расширениям оболочки, обработчик контекстной подсказки реализован в качестве DLL простого сервера COM. Объект COM содержит в своей реализации методы `IQueryInfo` и `IPersistFile`. В листинге 16.12 приведено содержимое файла `InfoMain.pas`, представляющего собой главный модуль проекта `DDGInfoTip`, который содержит реализацию обработчика контекстной подсказки.

Листинг 16.12. `InfoMain.pas` — главный модуль реализации обработчика контекстной подсказки

```
unit InfoMain;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, ShlObj;

type
  TInfoTipHandler = class(TComObject, IQueryInfo, IPersistFile)
  private
    FFileName: string;
    FMalloc: IMalloc;
  protected
    { IQUeryInfo }
    function GetInfoTip(dwFlags: DWORD;
                        var ppwszTip: PWideChar): HResult; stdcall;
    function GetInfoFlags(out pdwFlags: DWORD): HResult; stdcall;
    {IPersist}
    function GetClassID(out classID: TCLSID): HResult; stdcall;
    { IPersistFile }
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr;
                  dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr;
```

```
                  fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult;
                                                  stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult;
                                                  stdcall;
  public
    procedure Initialize; override;
  end;

  TInfoTipFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean;
                                    const ClsID: string); virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

const
  Class_InfoTipHandler: TGUID =
                        '{5E08F28D-A5B1-4996-BDF1-5D32108DB5E5}';

implementation

uses ComServ, SysUtils, Registry;

const
  TipBufLen = 1024;

procedure PackageInfoCallback(const Name: string;
              NameType: TNameType; Flags: Byte; Param: Pointer);
var
  S: string;
begin
  // При передаче имени содержащегося модуля добавить его к
  // списку модулей, переданных в параметре Param.
  if NameType = ntContainsUnit then begin
    S := Name;
    if PChar(Param)^ <> #0 then S := ', ' + S;
    StrLCat(PChar(Param), PChar(S), TipBufLen);
  end;
end;

function TInfoTipHandler.GetClassID(out classID: TCLSID): HResult;
begin
  classID := Class_InfoTipHandler;
  Result := S_OK;
end;

function TInfoTipHandler.GetCurFile(out pszFileName:
                                    POleStr): HResult;
begin
  Result := E_NOTIMPL;
```

```
end;

function TInfoTipHandler.GetInfoFlags(out pdwFlags:
                                      DWORD): HResult;
begin
  Result := E_NOTIMPL;
end;

function TInfoTipHandler.GetInfoTip(dwFlags: DWORD;
                           var ppwszTip: PWideChar): HResult;
var
  PackMod: HModule;
  TipStr: PChar;
  Size, Flags, TipStrLen: Integer;
begin
  Result := S_OK;
  if (CompareText(ExtractFileExt(FFileName), '.bpl') = 0) and
     Assigned(FMalloc) then begin
  // Поскольку необходимо получить доступ только к ресурсам
  // пакета, используем функцию LoadLibraryEx с
  // параметром LOAD_LIBRARY_AS_DATAFILE, обеспечивающим высокую
  // скорость загрузки пакета.
    PackMod := LoadLibraryEx(PChar(FFileName), 0,
               LOAD_LIBRARY_AS_DATAFILE);
    if PackMod <> 0 then
      try
        TipStr := StrAlloc(TipBufLen);
        try
          // Заполнить нулями область памяти, занимаемую строкой
          FillChar(TipStr^, TipBufLen, 0);
          // Заполнить строку подсказки содержащимися модулями
          GetPackageInfo(PackMod, TipStr, Flags,
                         PackageInfoCallback);
          TipStrLen := StrLen(TipStr) + 1;
          Size := (TipStrLen + 1) * SizeOf(WideChar);
          // использовать механизм выделения память оболочки
          ppwszTip := FMalloc.Alloc(Size);
          // Копировать PAnsiChar в PWideChar
          MultiByteToWideChar(0, 0, TipStr, TipStrLen,
                              ppwszTip, Size);
        finally
          StrDispose(TipStr);
        end;
      finally
        FreeLibrary(PackMod);
      end;
  end;
end;

procedure TInfoTipHandler.Initialize;
begin
  inherited;
  // применить систему выделения памяти оболочки
  // и сохранить результат
```

```
  SHGetMalloc(FMalloc);
end;

function TInfoTipHandler.IsDirty: HResult;
begin
  Result := E_NOTIMPL;
end;

function TInfoTipHandler.Load(pszFileName: POleStr;
                              dwMode: Integer): HResult;
begin
  // Это единственный метод интерфейса IPersistFile, необходимый
  // для хранения имени файла
  FFileName := pszFileName;
  Result := S_OK;
end;

function TInfoTipHandler.Save(pszFileName: POleStr;
                              fRemember: BOOL): HResult;
begin
  Result := E_NOTIMPL;
end;

function TInfoTipHandler.SaveCompleted(pszFileName:
                                       POleStr): HResult;
begin
  Result := E_NOTIMPL;
end;

{ TInfoTipFactory }

function TInfoTipFactory.GetProgID: string;
begin
  // Идентификатор программы (ProgID) для расширения контекстной
  // подсказки оболочки не нужен.
  Result := '';
end;

procedure TInfoTipFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then begin
    // Зарегистрировать эту DLL как обработчик контекстной
    // подсказки для файлов .bpl
    CreateRegKey('.bpl\shellex\{00021500-0000-0000-C000-
↳000000000046}', '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl\shellex\{00021500-0000-0000-C000-
↳000000000046}');
```

```
  end;
end;

procedure TInfoTipFactory.ApproveShellExtension(Register: Boolean;
                                                const ClsID: string);
// Этот элемент системного реестра необходим для корректной работы
// расширения под управлением Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\
↳Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

initialization
  TInfoTipFactory.Create(ComServer, TInfoTipHandler,
                         Class_InfoTipHandler, 'InfoTipHandler',
                        'DDG sample InfoTip handler',
                         ciMultiInstance, tmApartment);
end.
```

Эта реализация содержит два интересных момента. Обратите внимание, что метод `Initialize()` не только получает, но и сохраняет экземпляр системы выделения памяти оболочки. Впоследствии он используется для выделения памяти для строки контекстной подсказки в методе `GetInfoTip()`. Метод `Load()` передает имя рассматриваемого файла обработчику. Основную работу осуществляет метод `GetInfoTip()`, который получает информацию о пакете с помощью функции `GetPackageInfo()`, рассмотренной ранее в настоящей главе. Функция обратного вызова `PackageInfoCallback()` неоднократно вызывается внутри функции `GetPackageInfo()`, чтобы один за другим добавлять имена файлов в стороку контекстной подсказки.

Регистрация

Как можно заметить в листинге 16.12, методика регистрации DLL сервера COM практически аналогична регистрации других расширений оболочки, приведенных в настоящей главе. Главным различием является ключ системного реестра, в которым регистрируются обработчики контекстной подсказки; для них установлено следующее местоположение:

```
HKEY_CLASSES_ROOT\<расширение файла>\shellex\{00021500-0000-0000-
↳C000-000000000046}
```

Где *<расширение файла>* представляет собой расширение файла, включая в себя предшествующую точку. Ниже (рис. 16.12) приведен пример обработчика контекстной подсказки в действии.

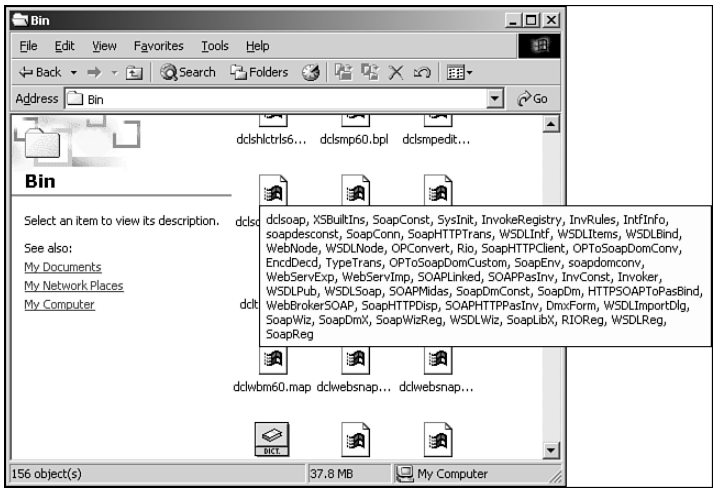

Рис. 16.12. *Обработчик контекстной подсказки в действии*

Резюме

В этой главе были рассмотрены различные аспекты расширения оболочки Windows: помещение пиктограмм в область панели задач, создание панели инструментов рабочего стола системы (AppBars), работа с ярлыками и разработка расширений оболочки Windows различных типов. Материал данной главы основан на знаниях, содержащихся в предыдущем разделе, — прежде всего об объектах COM. В главе 17, "Применение интерфейса API Open Tools", содержится более подробная информация о применении интерфейсов при компонент-ориентированной разработке.

ГЛАВА

17

Применение интерфейса API Open Tools

В ЭТОЙ ГЛАВЕ...

Наверное, у многих не раз возникал вопрос: Delphi — великолепный инструмент, но нельзя ли дополнить его интегрированную среду разработки еще одной небольшой функцией? Ответ: конечно же, можно! Для этого следует использовать *интерфейс API Open Tools*. Данный интерфейс позволяет создавать дополнительные инструменты, которые можно применять в интегрированной среде разработки Delphi. В настоящей главе речь пойдет о различных интерфейсах, которые создаются с помощью API Open Tools, об их применении, а также о том, как воспользоваться полученными знаниями при создании полнофункционального мастера.

Интерфейсы Open Tools

Интерфейс API Open Tools состоит из четырнадцати модулей, каждый из которых содержит один или несколько объектов, обеспечивающих взаимодействие со многими средствами интегрированной среды разработки. С помощью этих интерфейсов можно создавать собственные мастера Delphi, диспетчеры управления версиями, а также компоненты и редакторы свойств. Такие дополнения позволяют значительно расширить возможности среды разработки Delphi.

За исключением интерфейсов, разработанных для компонентов и редакторов свойств, объекты интерфейса Open Tools обеспечивают полностью виртуальный интерфейс с внешним миром. Это означает, что можно использовать лишь виртуальные функции таких объектов. Ни к полям данных подобного объекта, ни к его свойствам или статическим функциям нельзя получить доступ, поскольку объекты интерфейса Open Tools создаются на основе стандарта COM (см. главу 15, "Разработка приложений COM"). После небольшой доработки эти интерфейсы могут использоваться с любым языком программирования, поддерживающим технологию COM. В настоящей главе рассматривается лишь Delphi, но помните, что можно использовать и другие языки программирования.

НА ЗАМЕТКУ

Реализовать в полной мере возможности, предоставляемые интерфейсом API Open Tools, можно лишь в версиях Delphi Professional и Enterprise Edition. В версии Standard возможно применение дополнений, созданных с помощью интерфейса API Open Tools, но их невозможно создавать, поскольку в этой версии содержатся лишь модули для разработки компонентов и редакторов свойств. Исходный код интерфейсов Open Tools размещен в каталоге `\Delphi 6\Source\ToolsAPI`.

В табл. 17.1 приведены модули, которые, в сущности, и составляют интерфейс API Open Tools. Модули табл. 17.2 сохранены лишь для обеспечения совместимости с предыдущими версиями Delphi. Термин *интерфейс* в данном случае не означает встроенный в Delphi тип `interface`. Поскольку API Open Tools обеспечивает поддержку интерфейсов Delphi, в качестве замены "настоящих" интерфейсов он использует обычные классы Delphi с виртуальными абстрактными методами. Использование стандартных интерфейсов в API Open Tools Delphi в каждой новой версии увеличивалось, и текущая версия основывается практически только на них.

Таблица 17.1. Модули интерфейса API Open Tools

Имя модуля	*Назначение*
`ToolsAPI`	Содержит новейшие элементы интерфейса API Open Tools. Этот модуль, по существу, заместил абстрактные классы, применяемые в предыдущих версиях Delphi для управления дополнениями меню, системы уведомления, файловой системы, редактора и мастеров. В нем содержатся также новые интерфейсы для управления отладчиком, комбинациями клавиш интегрированной среды разработки, проектами, группами проектов, пакетами и списком To Do
`VCSIntf`	Определяет класс `TIVCSClient`, обеспечивающий взаимодействие интегрированной среды разработки Delphi с программным обеспечением управления версиями
`DesignConst`	Содержит строки, используемые API Open Tools
`DesignEditors`	Обеспечивает поддержку редактора свойств
`DesignIntf`	Заменяет модуль `DsgnIntf` предыдущих версий и обеспечивает базовую поддержку интерфейсов IDE времени разработки. IDE использует интерфейс `IProperty` для редактирования свойств. Интерфейс `IDesignerSelections` применяется для манипулирования объектами, выбранными в списке конструктора форм (заменяет `TDesignerSelectionList`, использовавшийся в предыдущих версиях Delphi). `IDesigner` — это один из первичных интерфейсов, который применяется мастерами для общих служб IDE. `IDesignNotification` поддерживает такие уведомления событий конструктора, как вставка, удаление и модификация элементов. Интерфейс `IComponentEditor` реализуется редакторами компонентов для обеспечения редактирования компонента во время разработки, а `ISelectionEditor` обеспечивает те же возможности для группы выбранных компонентов. Класс `TbaseComponentEditor` является базовым классом для всех редакторов компонентов. Интерфейс `ICustomModule` и класс `TBaseCustomModule` обеспечивают возможность установки модулей, которые могут быть отредактированы в конструкторе форм IDE
`DesignMenus`	Содержит интерфейсы `IMenuItems`, `IMenuItem`, а также другие подобные интерфейсы, предназначенные для манипулирования меню IDE во время разработки
`DesignWindows`	Содержит объявление класса `TDesignWindow`, являющегося базовым классом для всех новых окон проекта, которые добавляются в IDE

Окончание табл. 17.1.

Имя модуля	*Назначение*
PropertyCategories	Содержит классы, обеспечивающие категоризацию свойств специальных компонентов. Используется инспектором объектов для представления свойств по категориям
TreeIntf	Поддерживает класс TSprig и связанные с ним классы, а также интерфейсы, обеспечивающие узлы и ветвления в объекте TreeView IDE
VCLSprigs	Содержит реализацию ветвления компонентов VCL
VCLEditors	Содержит объявления базовых интерфейсов ICustomPropertyDrawing и ICustomPropertyListDrawing, обеспечивающих специальное представление свойств и списков свойств в инспекторе объектов IDE. Содержит также объявления специальных объектов для представления свойств VCL
ClxDesignWindows	Содержит объявление класса TClxDesignWindow, являющегося эквивалентом CLX класса TDesignWindow
ClxEditors	Эквивалент CLX модуля VCLEditors, содержащего редакторы свойств для компонентов CLX
ClxSprigs	Реализация ветвления для компонентов CLX

Таблица 17.2. Устаревшие модули API Open Tools

Имя модуля	*Назначение*
FileIntf	Определяет класс TIVirtualFileSystem, используемый IDE Delphi для работы с файлами. Мастера, диспетчеры управления версиями, а также редакторы свойств и компоненты могут использовать этот интерфейс для выполнения в Delphi специальных файловых операций
EditIntf	Определяет классы, необходимые для управления редактором кода и конструктором форм. Класс TIEditReader предоставляет доступ к буферу редактора "для чтения", а класс TIEditWriter — "для записи". Класс TIEditView предоставляет возможность просмотра буфера редактирования. Класс TIEditInterface — это базовый интерфейс редактора, который может использоваться для доступа к уже упоминавшимся интерфейсам. Класс TIComponentInterface — это интерфейс с отдельным компонентом, помещенным в форму в режиме разработки. Класс TIFormInterface — основной интерфейс с формой или модулем данных в режиме разработки. Класс TIResourceEntry обеспечивает взаимодействие с данными в файле ресурсов (*.res) проекта. Класс TIResourceFile опреде-

Окончание табл. 17.2.

Имя модуля	*Назначение*
	ляет высокоуровневый интерфейс с файлом ресурсов проекта. Класс `TIModuleNotifier` определяет сообщения, используемые при появлении в конкретном модуле различных событий. И, наконец, класс `TIModuleInterface` обеспечивает взаимодействие с любым файлом или модулем, открытым в интегрированной среде разработки
`ExptIntf`	Определяет абстрактный класс `TIExpert`, от которого происходят все мастера
`VirtIntf`	Определяет базовый класс `TInterface`, от которого происходят другие интерфейсы. В этом модуле определен также класс `TIStream`, который инкапсулирует класс VCL `TStream`
`IStreams`	Определяет классы `TIMemoryStream`, `TIFileStream` и `TIVirtualStream`, которые являются потомками класса `TIStream`. Эти интерфейсы могут использоваться для включения в IDE своего собственного механизма работы с потоками
`ToolIntf`	Определяет классы `TIMenuItemIntf` и `TIMainMenuIntf`, которые позволяют разработчику интерфейса Open Tools создавать и модифицировать меню IDE Delphi. В данном модуле определен также класс `TIAddInNotifier`, который позволяет создаваемым инструментам-дополнениям получать уведомления о возникновении в среде разработки определенных событий. Следует также отметить, что в этом модуле имеется класс `TIToolServices`, который обеспечивает взаимодействие между различными элементами IDE Delphi (такими как редактор, библиотека компонентов, редактор кода, конструктор форм и файловая система)

НА ЗАМЕТКУ

Может возникнуть закономерный вопрос: а где в Delphi содержится документация по всем этим мастерам? На этот счет можно сказать лишь следующее: все документировано, но поиск необходимой информации представляет собой непростую задачу. В каждом из таких модулей содержится полная документация по определенным в нем классам и методам. Для получения исчерпывающей информации проанализируйте все модули самостоятельно, поскольку нельзя сказать наверняка, в каком из модулей какие данные содержатся.

Использование интерфейса API Open Tools

Теперь, ознакомившись с интерфейсом API Open Tools, можно приступить к созданию и анализу работы реальной программы. В настоящем разделе основное внимание уделено процессу создания мастера с помощью интерфейса API Open Tools. Построение системы управления версиями не рассматривается, поскольку использование интереса в этой области довольно ограничено. Примеры разработки компонентов и редакторов свойств можно найти в главах 11, "Разработка компонентов VCL", и 12, "Создание расширенного компонента VCL".

Мастер Dumb

Для начала создадим очень простой мастер по имени "Dumb" ("глухой" или "бесполезный"). Для этого достаточно создать класс, реализующий интерфейс `IOTAWizard`. Напомним, что интерфейс `IOTAWizard` определен в модуле `ToolsAPI` следующим образом:

```
type
  IOTAWizard = interface(IOTANotifier)
    ['{B75C0CE0-EEA6-11D1-9504-00608CCBF153}']
    { Строки пользовательского интерфейса мастера }
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    { Запуск мастера }
    procedure Execute;
  end;
```

В основном такой интерфейс состоит из нескольких функций вида `Get`*`XXX`*`()`, которые подлежат переопределению в производных классах для обеспечения доступа к информации, специфической для каждого конкретного мастера. Метод `IOTAWizard.Execute()` вызывается интегрированной средой разработки при выборе пользователем определенного мастера в главном меню или диалоговом окне New Items. Поэтому в данном методе следует обеспечить создание и вызов соответствующего мастера.

Если провести углубленный анализ, то можно заметить, что интерфейс `IOTAWizard` является производным от другого интерфейса, `IOTANotifier`. Этот интерфейс также определен в модуле `ToolsAPI` и содержит методы, используемые интегрированной средой разработки для уведомления мастера о возникновении различных событий. Интерфейс `IOTANotifier` определен следующим образом:

```
type
  IOTANotifier = interface(IUnknown)
    ['{F17A7BCF-E07D-11D1-AB0B-00C04FB16FB3}']
    { Данная процедура вызывается сразу же после того, как элемент
      будет успешно сохранен. Для интерфейсов IOTAWizard такой
      вызов не выполняется. }
    procedure AfterSave;
    { Данная процедура вызывается непосредственно перед
```

```
      сохранением элемента. Для интерфейсов IOTAWizard такой вызов
      не выполняется. }
    procedure BeforeSave;
    { Начало удаления указанного элемента, поэтому все ссылки на
      него следует аннулировать. Исключения игнорируются. }
    procedure Destroyed;
    { Эта процедура вызывается при любой модификации указанного
      элемента. Для интерфейсов IOTAWizard такой вызов не
      выполняется. }
    procedure Modified;
  end;
```

Как видно из комментариев, содержащихся в коде, большинство методов для простых мастеров IOTAWizard не вызывается. Поэтому в модуле ToolsAPI имеется класс TNotifierObject, представляющий собой пустую реализацию методов интерфейса IOTANotifier. Если разрабатываемый мастер происходит от такого класса, то в распоряжении разработчика сразу окажутся все его методы.

При создании мастера естественно предположить, что он будет запускаться. Проще всего осуществить это с помощью выбора пункта меню. Если команду запуска мастера поместить в главное меню Delphi, то достаточно просто реализовать интерфейс IOTAMenuWizard, определенный в модуле ToolsAPI следующим образом:

```
type
  IOTAMenuWizard = interface(IOTAWizard)
    ['{B75C0CE2-EEA6-11D1-9504-00608CCBF153}']
    function GetMenuText: string;
  end;
```

Нетрудно заметить, что интерфейс IOTAMenuWizard является потомком интерфейса IOTAWizard, обладающим лишь одним дополнительным методом, который возвращает текстовую строку меню.

Все вышесказанное обобщено в листинге 17.1. В нем приведен код модуля DumbWiz.pas, содержащего класс TDumbWizard.

Листинг 17.1. DumbWiz.pas — реализация простейшего мастера

```
unit DumbWiz;

interface

uses
  ShareMem, SysUtils, Windows, ToolsAPI;

type
  TDumbWizard = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    // Методы класса IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // Методы класса IOTAMenuWizard
    function GetMenuText: string;
```

```
  end;

procedure Register;

implementation

uses Dialogs;

function TDumbWizard.GetName: string;
begin
  Result := 'Dumb Wizard';
end;

function TDumbWizard.GetState: TWizardState;
begin
  Result := [wsEnabled];
end;

function TDumbWizard.GetIDString: String;
begin
  Result := 'DDG.DumbWizard';
end;

procedure TDumbWizard.Execute;
begin
  MessageDlg('This is a dumb wizard.', mtInformation, [mbOk], 0);
end;

function TDumbWizard.GetMenuText: string;
begin
  Result := 'Dumb Wizard';
end;

procedure Register;
begin
  RegisterPackageWizard(TDumbWizard.Create);
end;

end.
```

Функция `IOTAWizard.GetName()` должна возвращать уникальное имя мастера.

Функция `IOTAWizard.GetState()` возвращает сведения о состоянии мастера `wsStandard` в главном меню. Возвращаемое значение этой функции представляет собой множество, в котором могут содержаться значения `wsEnabled` и/или `wsChecked`, в зависимости от того, как элемент меню будет представляться в интегрированной среде разработки. Данная функция вызывается для корректного отображения меню при каждом выводе мастера на экран.

Функция `IOTAWizard.GetIDString()` возвращает глобальный идентификатор мастера, представляющий собой уникальную строку. В соответствии с используемыми соглашениями эта строка должна иметь такой формат:

```
CompanyName.WizardName
```

Функция `IOTAWizard.Execute()` запускает мастер. Как видно из листинга 17.1, метод `Execute()` класса `TDumbWizard` не выполняет никаких действий. Но далее в настоящей главе будут показаны некоторые мастера, выполняющие реальные действия.

Функция `IOTAWizard.GetMenuText()` возвращает строку текста, которая должна появиться в главном меню. Эта функция вызывается каждый раз при выборе пункта меню Help. Таким образом, при запуске мастера можно динамически изменять текст меню.

Обратите внимание на вызов функции `RegisterPackageWizard()` внутри процедуры `Register()`. Заметим, что такой вызов очень напоминает синтаксис вызовов, используемых для регистрации компонентов, редакторов компонентов или редакторов свойств при включении в библиотеку компонентов, как описывалось в главах 11, "Разработка компонентов VCL", и 12, "Создание расширенного компонента VCL". Это связано с тем, что данный тип мастеров хранится в пакете, который является частью библиотеки компонентов. Как будет показано в следующем примере, мастер можно сохранить также в отдельной библиотеке DLL.

Мастер устанавливается точно так же, как и компонент. Выберите в главном меню Components пункт Install Component, а затем добавьте модуль к новому или существующему пакету. После установки в меню Help появится пункт запуска мастера Dumb (рис. 17.1). Работа этого мастера показана на рис. 17.2.

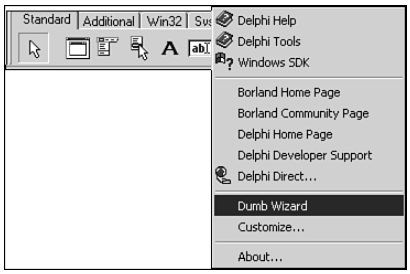

Рис. 17.1. *Пункт мастера Dumb в главном меню*

Рис. 17.2. *Мастер Dumb в действии*

Мастер Wizard

Процедура создания мастера в виде библиотеки DLL (DLL-based wizard) (далее — *мастер DLL*) несколько более трудоемка, чем процедура создания мастера в библиотеке компонентов. Кроме создания мастера в виде библиотеки DLL, на примере мастера "Wizard" иллюстрируется нескольких менее очевидных моментов. В частности, ниже будет показано, как мастер DLL связан с системным реестром и как использовать один и тот же исходный код для создания мастеров и в виде библиотек DLL, и в виде исполняемых файлов EXE.

НА ЗАМЕТКУ

Если операции ввода и вывода DLL Windows вызывают затруднения, то более подробная информация по этой теме приведена в главе 9, "Динамически компонуемые библиотеки", предыдущего издания *Delphi 5 Руководство разработчика*, находящегося на прилагаемом CD.

СОВЕТ

Не существует строго сформулированных правил, определяющих, где должен быть расположен мастер: в пакете библиотеки компонентов или в библиотеке DLL. С точки зрения пользователя, основное различие между этими двумя способами заключается в том, что для установки мастеров в библиотеке компонентов достаточно выполнить простую процедуру установки пакета, тогда как для установки мастеров в библиотеке DLL необходимо ввести соответствующие записи в системный реестр. К тому же, для активизации внесенных изменений, нужно выйти из среды Delphi, а затем запустить ее повторно. С точки зрения разработчика, проще иметь дело с мастерами, размещаемыми в пакетах, причем сразу по нескольким причинам. В частности, исключения будут автоматически передаваться между мастером и интегрированной средой разработки; для управления памятью нет необходимости использовать файл `sharemem.dll`; не нужно выполнять никаких специальных действий для инициализации переменных библиотеки DLL; будут правильно работать все всплывающие подсказки и сообщения мыши.
Принимая во внимание все вышесказанное, можно сделать вывод, что помещать мастер в библиотеку DLL целесообразно только в том случае, если усилия конечного пользователя по его установке должны быть минимальными.

Для того чтобы системой Delphi был распознан мастер, размещаемый в библиотеке DLL, соответствующая запись должна содержаться в следующей ветви системного реестра:

```
HKEY_CURRENT_USER\Software\Borland\Delphi\6.0\Experts
```

На рис. 17.3 эта ветвь показана в окне редактора системного реестра.

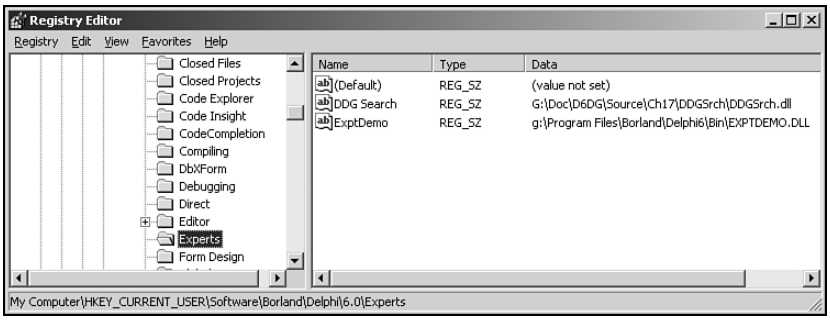

Рис. 17.3. *Записи о мастерах Delphi в системном реестре*

Интерфейс мастера

Мастер Wizard позволяет без использования редактора реестра добавлять, модифицировать и удалять из системного реестра записи о мастерах, размещенных в библиотеках DLL. Сначала рассмотрим модуль `InitWiz.pas`, содержащий класс мастера (листинг 17.2).

Листинг 17.2. Модуль `InitWiz.pas`, содержащий класс мастера DLL

```
unit InitWiz;

interface

uses Windows, ToolsAPI;

type
  TWizardWizard = class(TNotifierObject, IOTAWizard,
                        IOTAMenuWizard)
    // Методы класса IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // Методы класса IOTAMenuWizard
    function GetMenuText: string;
  end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
          RegisterProc: TWizardRegisterProc;
          var Terminate: TWizardTerminateProc): Boolean stdcall;

implementation

uses SysUtils, Forms, Controls, Main;

function TWizardWizard.GetName: string;
{ Возвращает имя мастера }
begin
  Result := 'WizardWizard';
end;

function TWizardWizard.GetState: TWizardState;
{ Этот мастер всегда доступен }
begin
  Result := [wsEnabled];
end;

function TWizardWizard.GetIDString: String;
{ "Vendor.AppName" строка идентификатора мастера }
begin
  Result := 'DDG.WizardWizard';
end;

function TWizardWizard.GetMenuText: string;
{ Строка меню мастера }
begin
  Result := 'Wizard Wizard';
end;
```

```
procedure TWizardWizard.Execute;
{ Вызывается при выборе мастера в главном меню. Эта процедура
создает, отображает и освобождает главную форму мастера. }
begin
  MainForm := TMainForm.Create(Application);
  try
    MainForm.ShowModal;
  finally
    MainForm.Free;
  end;
end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
           RegisterProc: TWizardRegisterProc;
           var Terminate: TWizardTerminateProc): Boolean stdcall;
var
  Svcs: IOTAServices;
begin
  Result := BorlandIDEServices <> nil;
  if Result then begin
    Svcs := BorlandIDEServices as IOTAServices;
    ToolsAPI.BorlandIDEServices := BorlandIDEServices;
    Application.Handle := Svcs.GetParentHandle;
    SDelphiKey := Svcs.GetBaseRegistryKey + '\Experts';
    RegisterProc(TWizardWizard.Create);
  end;
end;

end.
```

Между этим модулем и модулем мастера Dumb существует несколько различий. Самым важным из них является то, что функция инициализации типа `TWizardInitProc` должна быть точкой входа подпрограмм интегрированной среды разработки в библиотеке DLL. В данном случае такая функция называется `InitWizard()`. Она выполняет следующие задачи инициализации мастера.

- Получает через параметр `BorlandIDEServices` указатель на интерфейс `IOTAServices`.
- Сохраняет указатель `BorlandIDEServices` для дальнейшего использования.
- Присваивает переменной `Application` библиотеки DLL дескриптор, возвращаемый методом `IOTAServices.GetParentHandle()`. Этот метод возвращает дескриптор окна, которое должно быть родительским по отношению ко всем остальным окнам верхнего уровня, создаваемым мастером.
- Передает созданный экземпляр мастера процедуре `RegisterProc()` для его регистрации в интегрированной среде разработки. Процедура `RegisterProc()` вызывается один раз для каждого экземпляра мастера DLL, регистрируемого в интегрированной среде разработки.
- Кроме того, функция `InitWizard()` может назначать параметру `Terminate` процедуру типа `TWizardTerminateProc`, используемую в качестве процедуры

выхода. Эта процедура будет вызвана перед выгрузкой мастера из среды разработки. В ней можно выполнять освобождение необходимых ресурсов. Первоначально данный параметр равен `nil`, и если при выходе из мастера выполнять какие-либо специфические действия не требуется, то такое значение изменять не нужно.

СОВЕТ

В методе инициализации мастера необходимо использовать соглашение о вызовах `stdcall`.

СОВЕТ

В разделе `uses` любого размещаемого в DLL мастера, в котором вызываются функции интерфейса API Open Tools со строковыми параметрами, должен быть указан модуль `ShareMem`. В противном случае при освобождении экземпляра мастера произойдет ошибка, вызванная отказом в доступе (access violation).

Пользовательский интерфейс мастера

Метод `Execute()` немного сложнее методов, рассматривавшихся ранее в настоящей главе. В нем создается, а затем освобождается экземпляр мастера — форма `MainForm`, отображаемая в модальном режиме (рис. 17.4). В листинге 17.3 приведен модуль `Main.pas`, содержащий исходный код формы `MainForm`.

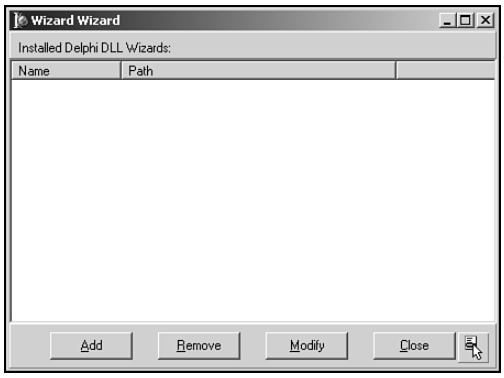

Рис. 17.4. *Главная форма мастера Wizard*

Листинг 17.3. `Main.pas` — главный модуль мастера Wizard

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, Registry, AddModU, ComCtrls, Menus;
```

```
type
  TMainForm = class(TForm)
    TopPanel: TPanel;
    Label1: TLabel;
    BottomPanel: TPanel;
    WizList: TListView;
    PopupMenu1: TPopupMenu;
    Add1: TMenuItem;
    Remove1: TMenuItem;
    Modify1: TMenuItem;
    AddBtn: TButton;
    RemoveBtn: TButton;
    ModifyBtn: TButton;
    CloseBtn: TButton;
    procedure RemoveBtnClick(Sender: TObject);
    procedure CloseBtnClick(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure ModifyBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure DoAddMod(Action: TAddModAction);
    procedure RefreshReg;
  end;

var
  MainForm: TMainForm;
  { Ключ системного реестра, в котором Delphi 6 регистрирует
    мастера. Версия EXE использует значение по умолчанию, а
    версия DLL получает ключ с помощью
    метода ToolServices.GetBaseRegistryKey. }
  SDelphiKey: string = '\Software\Borland\Delphi\6.0\Experts';

implementation

{$ifndef BUILD_EXE}
uses InitWiz;
{$endif}

{$R *.DFM}

var
  DelReg: TRegistry;

procedure TMainForm.RemoveBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Remove. Выделенный элемент удаляется
из системного реестра. }
var
  Item: TListItem;
begin
  Item := WizList.Selected;
  if Item <> nil then begin
    if MessageDlg(Format('Remove item "%s"', [Item.Caption]),
                  mtConfirmation, [mbYes, mbNo], 0) = mrYes then
```

```
      DelReg.DeleteValue(Item.Caption);
    RefreshReg;
  end;
end;

procedure TMainForm.CloseBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Close. Завершение приложения. }
begin
  Close;
end;

procedure TMainForm.DoAddMod(Action: TAddModAction);
{ Добавление в реестр записи для нового мастера или модификация уже
существующей записи. }
var
  OrigName, ExpName, ExpPath: String;
  Item: TListItem;
begin
  if Action = amaModify then begin // При модификации...
    Item := WizList.Selected;
    if Item = nil then Exit; // удостовериться, что элемент выбран
    ExpName := Item.Caption; // инициализировать переменные
    if Item.SubItems.Count > 0 then
      ExpPath := Item.SubItems[0];
    OrigName := ExpName;     // сохранить исходное имя.
  end;
  { Вызов диалога, позволяющего добавить или отредактировать
    запись }
  if AddModWiz(Action, ExpName, ExpPath) then begin
    { Обработка ситуации, когда выбрано действие Modify и изменено
      имя параметра. }
    if (Action = amaModify) and (OrigName <> ExpName) then
      DelReg.RenameValue(OrigName, ExpName);
    DelReg.WriteString(ExpName, ExpPath); // Запись нов. значения
  end;
  RefreshReg;                             // Обновить список
end;

procedure TMainForm.AddBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Add }
begin
  DoAddMod(amaAdd);
end;

procedure TMainForm.ModifyBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Modify }
begin
  DoAddMod(amaModify);
end;

procedure TMainForm.RefreshReg;
{ Обновление списка содержимым системного реестра }
var
```

```
  i: integer;
  TempList: TStringList;
  Item: TListItem;
begin
  WizList.Items.Clear;
  TempList := TStringList.Create;
  try
    { Получить имена мастеров из системного реестра }
    DelReg.GetValueNames(TempList);
    { Получить путь для каждого имени мастера }
    for i := 0 to TempList.Count - 1 do begin
      Item := WizList.Items.Add;
      Item.Caption := TempList[i];
      Item.SubItems.Add(DelReg.ReadString(TempList[i]));
    end;
  finally
    TempList.Free;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  RefreshReg;
end;

initialization
  DelReg := TRegistry.Create;          // Создать объект реестра
  DelReg.RootKey := HKEY_CURRENT_USER; // Установить корневой ключ
                                // Открыть/создать ключ мастера Delphi
  DelReg.OpenKey(SDelphiKey, True);
finalization
  Delreg.Free;                  // Удалить объект системного реестра
end.
```

Этот модуль обеспечивает пользовательский интерфейс мастера, помещенного в библиотеку DLL. Он предназначен для добавления, удаления и модификации записей в системном реестре. В разделе `initialization` данного модуля создается объект `DelReg`, имеющий тип `TRegistry`. Свойству `RootKey` объекта `DelReg` присваивается значение ключа `HKEY_CURRENT_USER`. Затем с помощью метода `OpenKey()` этот ключ верхнего уровня разворачивается в ключ `\Software\Borland\Delphi\6.0\Experts`, который содержит записи о мастерах Delphi, сохраняемых в библиотеках DLL.

При первом запуске мастера компонент `TListView` по имени `ExptList` заполняется параметрами и значениями из рассмотренного выше ключа системного реестра. Сначала вызов метода `DelReg.GetValueNames()` возвращает имена параметров и помещает их в объект `TStringList`. Затем для каждого элемента списка `TStringList` к компоненту `ExptList` добавляется компонент `TListItem`. Для получения значения каждого параметра объекта `TListItem`, помещаемого в список `SubItems`, используется метод `DelReg.ReadString()`.

Работа с реестром осуществляется в методах `RemoveBtnClick()` и `DoAddMod()`. Метод `RemoveBtnClick()` отвечает за удаление из системного реестра записи, соответст-

вующей текущему выделенному мастеру. Прежде всего в методе проверяется наличие некоторого выделенного элемента, а затем отображается диалоговое окно для подтверждения выполнения операции удаления. Затем вызывается метод `DelReg.DeleteValue()`, которому в качестве параметра передается значение `CurrentItem`.

Методу `DoAddMod()` может передаваться параметр типа `TAddModAction`. Этот тип определен следующим образом:

```
type
  TAddModAction = (amaAdd, amaModify);
```

Как следует из приведенных значений, параметр указывает, будет ли в реестр добавлен новый элемент или модифицирован уже существующий. Сначала функция проверяет наличие выделенного в данный момент элемента. Если его нет, то функция дополнительно проверяет, имеет ли переданный ей параметр `Action` значение `amaAdd`. Если в параметре `Action` имеется значение `amaModify`, то параметр и значение, которые соответствуют существующему мастеру, присваиваются локальным переменным `ExpName` и `ExpPath`. Затем эти значения передаются функции `AddModExpert()`, определенной в модуле `AddModU` (листинг 17.4). В этой функции создается диалоговое окно, в котором можно ввести новое либо изменить существующее имя или путь к мастеру (рис. 17.5). Функция `AddModExpert()` возвращает значение `True`, если ее диалоговое окно закрыто с помощью щелчка на кнопке OK. В таком случае существующий параметр системного реестра модифицируется с помощью метода `DelReg.RenameValue()`, а новое или модифицируемое значение записывается функцией `DelReg.WriteString()`.

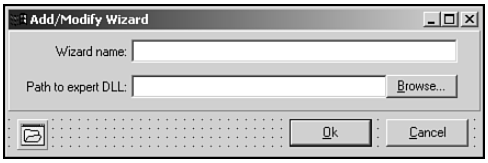

Рис. 17.5. *Форма `AddModForm` мастера Wizard*

Листинг 17.4. `AddModU.pas` — модуль, добавляющий и модифицирующий записи о мастере в системном реестре

```
unit AddModU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TAddModAction = (amaAdd, amaModify);

  TAddModForm = class(TForm)
    OkBtn: TButton;
    CancelBtn: TButton;
    OpenDialog: TOpenDialog;
    Panel1: TPanel;
```

```
    Label1: TLabel;
    Label2: TLabel;
    PathEd: TEdit;
    NameEd: TEdit;
    BrowseBtn: TButton;
    procedure BrowseBtnClick(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

function AddModWiz(AAction: TAddModAction; var WizName,
                   WizPath: String): Boolean;

implementation

{$R *.DFM}

function AddModWiz(AAction: TAddModAction; var WizName,
                   WizPath: String): Boolean;
{ Применяется для вызова диалогового окна, добавления или
модификации параметров системного реестра }
const
  CaptionArray: array[TAddModAction] of string[31] =
                   ('Add newexpert', 'Modify expert');
begin
  with TAddModForm.Create(Application) do begin  // Создать диалог
    Caption := CaptionArray[AAction];      // Установить заголовок
    if AAction = amaModify then begin      // При модификации...
      NameEd.Text := WizName;              // инициализировать имя
      PathEd.Text := WizPath;              // и путь.
    end;
    Result := ShowModal = mrOk;            // Показать диалог
    if Result then begin                   // Если Ok...
      WizName := NameEd.Text;              // установить имя
      WizPath := PathEd.Text;              // и путь.
    end;
    Free;
  end;
end;

procedure TAddModForm.BrowseBtnClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    PathEd.Text := OpenDialog.FileName;
end;

end.
```

Одним выстрелом — двух зайцев: EXE и DLL

Как уже упоминалось, один набор модулей с исходным кодом можно применять для создания мастера и в виде библиотеки DLL, и в виде отдельного исполняемого файла EXE. Это оказывается возможным, если в файле проекта использовались директивы компилятора. В листинге 17.5 содержится исходный код файла проекта `WizWiz.dpr`, применяемого для создания мастера Wizard.

Листинг 17.5. `WizWiz.dpr` — основной файл проекта мастера Wizard

```
{$ifdef BUILD_EXE}
1   program WizWiz;     // Создать как EXE
{$else}
  library WizWiz;     // Создать как DLL
{$endif}

uses
  {$ifndef BUILD_EXE}
    ShareMem,                 // Для DLL необходим модуль ShareMem
    InitWiz in 'InitWiz.pas', // Содержимое мастера Wizard
    ToolsAPI,
  {$endif}
  Forms,
  Main in 'Main.pas' {MainForm},
  AddModU in 'AddModU.pas' {AddModForm};

{$ifdef BUILD_EXE}
  {$R *.RES}                          // Необходим для EXE
{$else}
  exports                             // Необходим для DLL
    InitWizard name WizardEntryPoint;  // Необходимая точка входа
{$endif}

begin
{$ifdef BUILD_EXE}                    // Необходим для EXE...
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
{$endif}
end.
```

Как видно из приведенного кода, при построении этого проекта будет создан исполняемый файл, если определено условие компиляции `BUILD_EXE`. В противном случае будет построен мастер DLL. Условие компиляции задается в списке Conditional Defines, расположенном во вкладке Directories/Conditionals диалогового окна Project Options (рис. 17.6).

Еще одно, заключительное, замечание по материалам рассмотренного проекта относится к функции `InitWizard()` из модуля `IntWiz`, экспортируемой в разделе `exports` файла проекта. Эту функцию *необходимо* экспортировать под именем `WizardEntryPoint`, определенным в модуле `ToolsAPI`.

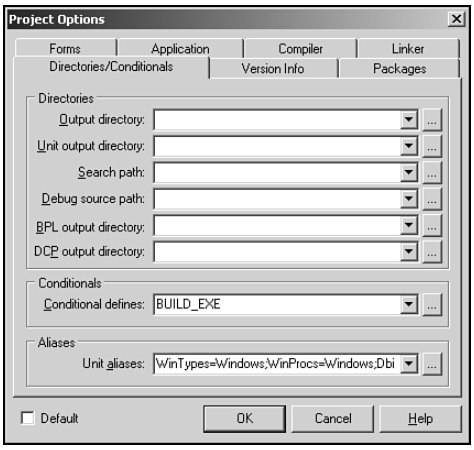

Рис. 17.6. *Диалоговое окно* Project Options

СОВЕТ

Компания *Borland* не внесла файл `ToolsAPI.dcu` в комплект поставки. Это означает, что мастера DLL и EXE, содержащие в операторе `uses` ссылки на модуль `ToolsAPI`, могут быть построены лишь с помощью *пакетов*. В настоящее время без пакетов мастер построить нельзя.

Мастер DDG Search

Помните небольшое изящное приложение для поиска (Search), разработка которого была описана в главе 5, "Создание многопоточных приложений"? В настоящем разделе изложено, как преобразовать эту утилиту в еще более полезный мастер Delphi с минимальными изменениями кода. Этот мастер называется DDG Search.

Сначала рассмотрим модуль `InitWiz.pas` (листинг 17.6), который обеспечивает взаимодействие мастера DDG Search с интегрированной средой разработки. Нетрудно заметить, что данный модуль очень похож на модуль из предыдущего примера с тем же именем. И это неслучайно. Модуль `InitWiz.pas` является всего лишь копией модуля из предыдущего примера, в которую внесены некоторые необходимые изменения в имени мастера и методе `Execute()`. Копирование и вставка — это именно то, что можно назвать "старым добрым" наследованием. Зачем набирать код вручную, если он уже есть?

Листинг 17.6. `InitWiz.pas` — содержит логику мастера `DDGSrch`

```
unit InitWiz;

interface

uses
  Windows, ToolsAPI;

type
```

```
  TSearchWizard = class(TNotifierObject, IOTAWizard,
                        IOTAMenuWizard)
    // Методы класса IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // Методы класса IOTAMenuWizard
    function GetMenuText: string;
  end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
          RegisterProc: TWizardRegisterProc;
          var Terminate: TWizardTerminateProc): Boolean stdcall;

var
  ActionSvc: IOTAActionServices;

implementation

uses SysUtils, Dialogs, Forms, Controls, Main, PriU;

function TSearchWizard.GetName: string;
{ Возвращает имя мастера }
begin
  Result := 'DDG Search';
end;

function TSearchWizard.GetState: TWizardState;
{ Этот мастер всегда доступен в меню }
begin
  Result := [wsEnabled];
end;

function TSearchWizard.GetIDString: String;
{ Возвращает уникальное имя мастера в формате Vendor.Product }
begin
  Result := 'DDG.DDGSearch';
end;

function TSearchWizard.GetMenuText: string;
{ Возвращает строку текста для помещения в меню Help }
begin
  Result := 'DDG Search Expert';
end;

procedure TSearchWizard.Execute;
{ Вызывается, если имя мастера выбрано в меню Help среды
разработки. В этой функции запускается мастер. }
begin
  // Если форма не создана, то создать и отобразить ее
  if MainForm = nil then begin
    MainForm := TMainForm.Create(Application);
```

```
    ThreadPriWin := TThreadPriWin.Create(Application);
    MainForm.Show;
  end
  else
  // Если форма уже создана, то восстановить и отобразить ее
    with MainForm do begin
      if not Visible then Show;
      if WindowState = wsMinimized then WindowState := wsNormal;
      SetFocus;
    end;
end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
            RegisterProc: TWizardRegisterProc;
            var Terminate: TWizardTerminateProc): Boolean stdcall;
var
  Svcs: IOTAServices;
begin
  Result := BorlandIDEServices <> nil;
  if Result then begin
    Svcs := BorlandIDEServices as IOTAServices;
    ActionSvc := BorlandIDEServices as IOTAActionServices;
    ToolsAPI.BorlandIDEServices := BorlandIDEServices;
    Application.Handle := Svcs.GetParentHandle;
    RegisterProc(TSearchWizard.Create);
  end;
end;

end.
```

Функция `Execute()` мастера немного отличается от аналогичной функции, рассмотренной выше. Главная форма мастера `MainForm` отображается в немодальном режиме. Конечно, это требует написания дополнительного кода, поскольку заранее нельзя определить, когда форма действительно создана, а когда переменная формы содержит некорректное значение. Для этого необходимо, чтобы переменная `MainForm` имела значение `nil`, когда мастер неактивен. Более подробная информация по данной теме приведена далее в настоящей главе.

Еще одним существенным отличием этого проекта от примера, приведенного в главе 5, "Создание многопоточных приложений", является то, что файл проекта теперь называется `DDGSrch.dpr`. Он приведен в листинге 17.7.

Листинг 17.7. `DDGSrch.dpr` — файл проекта `DDGSrch`

```
{$IFDEF BUILD_EXE}
  program DDGSrch;
{$ELSE}
  library DDGSrch;
{$ENDIF}

uses
```

```
{$IFDEF BUILD_EXE}
  Forms,
{$ELSE}
  ShareMem,
  ToolsAPI,
  InitWiz in 'InitWiz.pas',
{$ENDIF}
  Main in 'MAIN.PAS' {MainForm},
  SrchIni in 'SrchIni.pas',
  SrchU in 'SrchU.pas',
  PriU in 'PriU.pas' {ThreadPriWin},
  MemMap in '..\..\Utils\MemMap.pas',
  DDGStrUtils in '..\..\Utils\DDGStrUtils.pas';

{$R *.RES}

{$IFNDEF BUILD_EXE}
exports
  { Точка входа, используемая средой разработки Delphi }
  InitWizard name WizardEntryPoint;
{$ENDIF}

begin
{$IFDEF BUILD_EXE}
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
{$ENDIF}
end.
```

Как видно из кода, данный файл имеет небольшой размер. Обратите внимание на два важных момента. Во-первых, заголовок `library` показывает, что будет создан мастер DLL. А, во-вторых, для инициализации мастера интегрированной средой разработки Delphi экспортируется функция `InitExpert()`.

В модуль `Main` этого проекта также было внесено несколько изменений. Как уже говорилось, если мастер неактивен, то переменная `MainForm` должна содержать значение `nil`. Как уже упоминалось в главе 2, "Язык программирования Object Pascal", при запуске приложения переменная `MainForm` автоматически принимает значение `nil`. Кроме того, в обработчике события `OnClick` экземпляр формы уничтожается, а глобальная переменная `MainForm` принимает значение `nil`. Этот метод имеет следующий вид:

```
procedure TMainForm.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  Action := caFree;
  Application.OnShowHint := FOldShowHint;
  MainForm := nil;
end;
```

И, наконец, с помощью рассматриваемого мастера можно, дважды щелкнув в списке главной формы, переносить файлы в редактор кода среды разработки. Такая возможность реализована в методе `FileLBDblClick()` следующим образом:

```
procedure TMainForm.FileLBDblClick(Sender: TObject);
{ Вызывается, если пользователь дважды щелкнул в списке. Файл
загружается в интегрированную среду разработки. }
var
  FileName: string;
  Len: Integer;
begin
  { Удостовериться, что пользователь щелкнул на файле... }
  if Integer(FileLB.Items.Objects[FileLB.ItemIndex]) > 0 then
  begin
    FileName := FileLB.Items[FileLB.ItemIndex];
    { Удалить из строк фрагменты "File " и ":". }
    FileName := Copy(FileName, 6, Length(FileName));
    Len := Length(FileName);
    if FileName[Len] = ':' then SetLength(FileName, Len - 1);
    { Открыть проект или файл }
{$IFNDEF BUILD_EXE}
    if CompareText(ExtractFileExt(FileName), '.DPR') = 0 then
      ActionSvc.OpenProject(FileName, True)
    else
      ActionSvc.OpenFile(FileName);
{$ELSE}
    ShellExecute(0, 'open', PChar(FileName), nil,
                 nil, SW_SHOWNORMAL);
{$ENDIF}
  end;
end;
```

При компиляции в качестве мастера эта процедура использует методы `OpenFile()` и `OpenProject()` класса `IOTAActionServices`, позволяющие открыть соответствующий файл. При компиляции в качестве автономного исполняемого файла EXE, данный метод вызывает функцию API `ShellExecute()`, что также позволяет открыть файл, но используя для этого приложение, ассоциированное с файлом данного расширения.

В листинге 17.8 приведен полный код модуля `Main` проекта `DDGSrch`, а на рис. 17.7 показан мастер DDG Search, работающий в интегрированной среде.

Листинг 17.8. `Main.pas` — главный модуль проекта `DDGSrch`

```
unit Main;

interface

{$WARN UNIT_PLATFORM OFF}

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls, Menus,
```

```
  SrchIni, SrchU, ComCtrls;

type
  TMainForm = class(TForm)
    FileLB: TListBox;
    PopupMenu1: TPopupMenu;
    Font1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    FontDialog1: TFontDialog;
    StatusBar: TStatusBar;
    AlignPanel: TPanel;
    ControlPanel: TPanel;
    ParamsGB: TGroupBox;
    LFileSpec: TLabel;
    LToken: TLabel;
    lPathName: TLabel;
    EFileSpec: TEdit;
    EToken: TEdit;
    PathButton: TButton;
    OptionsGB: TGroupBox;
    cbCaseSensitive: TCheckBox;
    cbFileNamesOnly: TCheckBox;
    cbRecurse: TCheckBox;
    SearchButton: TBitBtn;
    CloseButton: TBitBtn;
    PrintButton: TBitBtn;
    PriorityButton: TBitBtn;
    View1: TMenuItem;
    EPathName: TEdit;
    procedure SearchButtonClick(Sender: TObject);
    procedure PathButtonClick(Sender: TObject);
    procedure FileLBDrawItem(Control: TWinControl; Index: Integer;
                             Rect: TRect; State: TOwnerDrawState);
    procedure Font1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure PrintButtonClick(Sender: TObject);
    procedure CloseButtonClick(Sender: TObject);
    procedure FileLBDblClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure PriorityButtonClick(Sender: TObject);
    procedure ETokenChange(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
  private
    FOldShowHint: TShowHintEvent;
    procedure ReadIni;
    procedure WriteIni;
    procedure DoShowHint(var HintStr: string;
                  var CanShow: Boolean; var HintInfo: THintInfo);
  protected
    procedure WndProc(var Message: TMessage); override;
```

```
  public
    Running: Boolean;
    SearchPri: integer;
    SearchThread: TSearchThread;
    procedure EnableSearchControls(Enable: Boolean);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses
{$IFNDEF BUILD_EXE}
  InitWiz,
{$ENDIF}
  Printers, ShellAPI, MemMap, FileCtrl, PriU;

procedure PrintStrings(Strings: TStrings);
{ Эта процедура печатает все строки из параметра Strings }
var
  Prn: TextFile;
  i: word;
begin
  if Strings.Count = 0 then begin        // Строки есть?
    MessageDlg('No text to print!', mtInformation, [mbOk], 0);
    Exit;
  end;
  AssignPrn(Prn);                        // Назначить принтеру Prn
  try
    Rewrite(Prn);                        // Открыть принтер
    try
      for i := 0 to Strings.Count - 1 do // Цикл по всем строкам
        WriteLn(Prn, Strings.Strings[i]); // Запись на принтер
    finally
      CloseFile(Prn);                    // Закрыть принтер
    end;
  except
    on EInOutError do
      MessageDlg('Error Printing text.', mtError, [mbOk], 0);
  end;
end;

procedure TMainForm.EnableSearchControls(Enable: Boolean);
{ Разрешает или запрещает использование определенных элементов
управления, в зависимости от возможности модифицировать параметры
во время выполнения поиска. }
begin
  // Разрешить/запретить соответствующие элементы управления
  SearchButton.Enabled := Enable;
  cbRecurse.Enabled := Enable;
```

```
  cbFileNamesOnly.Enabled := Enable;
  cbCaseSensitive.Enabled := Enable;
  PathButton.Enabled := Enable;
  EPathName.Enabled := Enable;
  EFileSpec.Enabled := Enable;
  EToken.Enabled := Enable;
  Running := not Enable;  // Установить флаг Running
  ETokenChange(nil);
  with CloseButton do begin
    if Enable then begin  // Установить свойства кнопки Close/Stop
      Caption := '&Close';
      Hint := 'Close Application';
    end
    else begin
      Caption := '&Stop';
      Hint := 'Stop Searching';
    end;
  end;
end;

procedure TMainForm.SearchButtonClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Search. Создает поток поиска. }
begin
  EnableSearchControls(False);    // Отключить элементы управления
  FileLB.Clear;                   // Очистить список
  { запуск потока }
  SearchThread := TSearchThread.Create(cbCaseSensitive.Checked,
       cbFileNamesOnly.Checked, cbRecurse.Checked, EToken.Text,
       EPathName.Text, EFileSpec.Text, Handle);
end;

procedure TMainForm.ETokenChange(Sender: TObject);
begin
  SearchButton.Enabled := not Running and (EToken.Text <> '');
end;

procedure TMainForm.PathButtonClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Path. Позволяет выбрать новый
путь. }
var
  ShowDir: string;
begin
  ShowDir := EPathName.Text;
  if SelectDirectory(ShowDir, [], 0) then
    EPathName.Text := ShowDir;
end;

procedure TMainForm.FileLBDblClick(Sender: TObject);
{ Вызывается, если пользователь дважды щелкнул в списке. Файл
загружается в интегрированную среду разработки. }
var
  FileName: string;
  Len: Integer;
```

```
begin
  { Удостовериться, что пользователь щелкнул на файле... }
  if Integer(FileLB.Items.Objects[FileLB.ItemIndex]) > 0 then
  begin
    FileName := FileLB.Items[FileLB.ItemIndex];
    { Удалить из строк фрагменты "File " и ":". }
    FileName := Copy(FileName, 6, Length(FileName));
    Len := Length(FileName);
    if FileName[Len] = ':' then SetLength(FileName, Len - 1);
    { Открыть проект или файл }
{$IFNDEF BUILD_EXE}
    if CompareText(ExtractFileExt(FileName), '.DPR') = 0 then
      ActionSvc.OpenProject(FileName, True)
    else
      ActionSvc.OpenFile(FileName);
{$ELSE}
    ShellExecute(0, 'open', PChar(FileName), nil,
                 nil, SW_SHOWNORMAL);
{$ENDIF}
  end;
end;

procedure TMainForm.FileLBDrawItem(Control: TWinControl;
            Index: Integer; Rect: TRect; State: TOwnerDrawState);
{ Вызывается для перерисовки списка. }
var
  CurStr: string;
begin
  with FileLB do begin
    CurStr := Items.Strings[Index];
    Canvas.FillRect(Rect);             // Очистка прямоугольника
    // Если не только имя файла ...
    if not cbFileNamesOnly.Checked then begin
      { если текущая строка - имя файла ... }
      if Integer(Items.Objects[Index]) > 0 then
        Canvas.Font.Style := [fsBold];      // Полужирный шрифт
    end
    else
      Rect.Left := Rect.Left + 15;   // в противном случае отступ
    DrawText(Canvas.Handle, PChar(CurStr), Length(CurStr),
            Rect, dt_SingleLine);
  end;
end;

procedure TMainForm.Font1Click(Sender: TObject);
{ Выбор нового шрифта для списка }
begin
  { Выбор нового шрифта }
  if FontDialog1.Execute then
    FileLB.Font := FontDialog1.Font;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
```

```
{ Обработчик события формы OnDestroy }
begin
  WriteIni;
end;

procedure TMainForm.FormCreate(Sender: TObject);
{ Обработчик события формы OnCreate }
begin
  Application.HintPause := 0; // Отображать подсказки без задержки
  FOldShowHint := Application.OnShowHint; // Установка подсказок
  Application.OnShowHint := DoShowHint;
  ReadIni;                                // Чтение INI-файла
end;

procedure TMainForm.DoShowHint(var HintStr: string;
                  var CanShow: Boolean; var HintInfo: THintInfo);
{ Обработчик события приложения OnHint }
begin
  { Отображение подсказок в строке состояния }
  StatusBar.Panels[0].Text := HintStr;
  { Не отображать всплывающие подсказки, если курсор расположен
    над пользовательским элементом управления }
  if (HintInfo.HintControl <> nil) and
    (HintInfo.HintControl.Parent <> nil) and
     ((HintInfo.HintControl.Parent = ParamsGB) or
    (HintInfo.HintControl.Parent = OptionsGB) or
    (HintInfo.HintControl.Parent = ControlPanel)) then
    CanShow := False;
  if Assigned(FOldShowHint) then
    FOldShowHint(HintStr, CanSHow, HintInfo);
end;

procedure TMainForm.PrintButtonClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Print. }
begin
  if MessageDlg('Send search results to printer?', mtConfirmation,
                [mbYes, mbNo], 0) = mrYes then
    PrintStrings(FileLB.Items);
end;

procedure TMainForm.CloseButtonClick(Sender: TObject);
{ Вызывается для остановки потока или завершения приложения }
begin
  // Если поток запущен, завершить его
  if Running then SearchThread.Terminate
  // В противном случае завершить приложение
  else Close;
end;

procedure TMainForm.FormResize(Sender: TObject);
{ Обработчик события OnResize. Центрирование элементов управления в
форме. }
begin
```

```
 { Разделить строку состояния на две панели в соотношении 2/3 }
  with StatusBar do begin
    Panels[0].Width := Width div 3;
    Panels[1].Width := Width * 2 div 3;
  end;
  { Центрирование элементов управления в середине формы }
  ControlPanel.Left := (AlignPanel.Width div 2) -
                       (ControlPanel.Width div 2);
end;

procedure TMainForm.PriorityButtonClick(Sender: TObject);
{ Отображение формы приоритета потока }
begin
  ThreadPriWin.Show;
end;

procedure TMainForm.ReadIni;
{ Считывание из системного реестра значений по умолчанию }
begin
  with SrchIniFile do begin
    EPathName.Text := ReadString('Defaults', 'LastPath', 'C:\');
    EFileSpec.Text := ReadString('Defaults',
                                 'LastFileSpec', '*.*');
    EToken.Text := ReadString('Defaults', 'LastToken', '');
    cbFileNamesOnly.Checked := ReadBool('Defaults',
                                        'FNamesOnly', False);
    cbCaseSensitive.Checked := ReadBool('Defaults',
                                        'CaseSens', False);
    cbRecurse.Checked := ReadBool('Defaults', 'Recurse', False);
    Left := ReadInteger('Position', 'Left', 100);
    Top := ReadInteger('Position', 'Top', 50);
    Width := ReadInteger('Position', 'Width', 510);
    Height := ReadInteger('Position', 'Height', 370);
  end;
end;

procedure TMainForm.WriteIni;
{ Запись текущих значений обратно в системный реестр }
begin
  with SrchIniFile do begin
    WriteString('Defaults', 'LastPath', EPathName.Text);
    WriteString('Defaults', 'LastFileSpec', EFileSpec.Text);
    WriteString('Defaults', 'LastToken', EToken.Text);
    WriteBool('Defaults', 'CaseSens', cbCaseSensitive.Checked);
    WriteBool('Defaults', 'FNamesOnly', cbFileNamesOnly.Checked);
    WriteBool('Defaults', 'Recurse', cbRecurse.Checked);
    WriteInteger('Position', 'Left', Left);
    WriteInteger('Position', 'Top', Top);
    WriteInteger('Position', 'Width', Width);
    WriteInteger('Position', 'Height', Height);
  end;
end;
```

```
procedure TMainForm.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  Action := caFree;
  Application.OnShowHint := FOldShowHint;
  MainForm := nil;
end;

procedure TMainForm.WndProc(var Message: TMessage);
begin
  if Message.Msg = DDGM_ADDSTR then begin
    FileLB.Items.AddObject(PChar(Message.WParam),
                           TObject(Message.LParam));
    StrDispose(PChar(Message.WParam));
  end
  else
    inherited WndProc(Message);
end;

end.
```

СОВЕТ

Обратите внимание на следующую строку листинга 17.8:

```
{$WARN UNIT_PLATFORM OFF}
```

Эта директива компилятора используется для отключения предупреждений времени компиляции, которые возникают в связи с использованием в `Main.pas` модуля `FileCtrl`, который специфичен только для платформы Windows. Кроме того, применение модуля `FileCtrl` отмечено директивами, зависимыми от платформы.

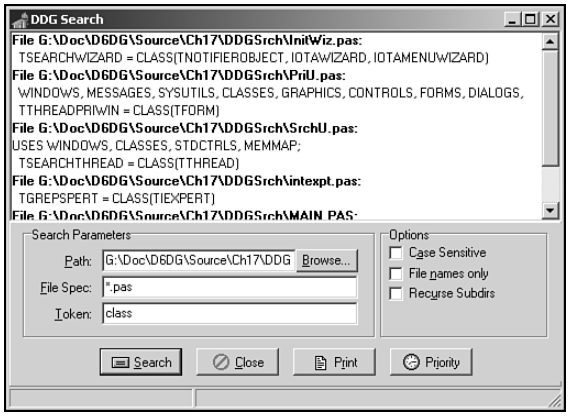

Рис. 17.7. *Мастер DDG Search в действии*

Мастера форм

Интерфейсом API Open Tools поддерживается еще один тип мастера — мастер форм. После установки мастер этого типа можно выбрать в диалоговом окне New Items. С его помощью можно генерировать новые формы и модули. В главе 16, "Программирование для оболочки Windows", данный тип мастера использовался для создания новых форм AppBar. Однако тогда не был показан код, который, собственно, и "оживлял" мастер.

Создать мастер форм очень просто, хотя при этом и требуется реализовать множество методов интерфейсов. Процесс создания мастера форм можно разделить на пять основных этапов.

1. Создайте класс, производный от класса TCustomForm, TDataModule или любого другого класса, производного от класса TWinControl, — он будет использоваться в качестве базового класса формы. Обычно такой класс находится в отдельном модуле. В данном случае в качестве базового будет использоваться класс TAppBar.
2. Создайте потомок класса TNotifierObject, реализующий следующие интерфейсы: IOTAWizard, IOTARepositoryWizard, IOTAFormWizard, IOTACreator и IOTAModuleCreator.
3. Обычно для получения нового имени модуля и класса мастера в методе IOTAWizard.Execute() необходимо вызвать метод IOTAModuleServices.GetNewModuleAndClassName(). Для того чтобы сообщить среде разработки о необходимости создания нового модуля, вызовите метод IOTAModuleServices.CreateModule().
4. Реализация большинства методов вышеупомянутых интерфейсов занимает лишь одну строку. К менее тривиальным относятся методы NewFormFile() и NewImplFile() интерфейса IOTAModuleCreator, возвращающие код для формы и модуля соответственно. Более сложной может оказаться реализация метода IOTACreator.GetOwner(), однако следующий пример является хорошей иллюстрацией того, как можно добавить модуль в текущий проект.
5. Наконец процедура Register() мастера регистрирует обработчик класса новой формы, что осуществляется с помощью процедуры RegisterCustomModule() модуля DsgnIntf. Создайте мастер, вызвав процедуру RegisterPackageWizard() модуля ToolsAPI.

В листинге 17.9 приведен исходный код модуля ABWizard.pas, в котором реализован мастер AppBar.

Листинг 17.9. Модуль ABWizard.pas, содержащий реализацию мастера AppBar

```
unit ABWizard;

interface

uses Windows, Classes, ToolsAPI;
```

```
type
  TAppBarWizard = class(TNotifierObject, IOTAWizard,
                        IOTARepositoryWizard, IOTAFormWizard,
                        IOTACreator, IOTAModuleCreator)
  private
    FUnitIdent: string;
    FClassName: string;
    FFileName: string;
  protected
    // Методы класса IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // Методы классов IOTARepositoryWizard и IOTAFormWizard
    function GetAuthor: string;
    function GetComment: string;
    function GetPage: string;
    function GetGlyph: HICON;
    // Методы класса IOTACreator
    function GetCreatorType: string;
    function GetExisting: Boolean;
    function GetFileSystem: string;
    function GetOwner: IOTAModule;
    function GetUnnamed: Boolean;
    // Методы класса IOTAModuleCreator
    function GetAncestorName: string;
    function GetImplFileName: string;
    function GetIntfFileName: string;
    function GetFormName: string;
    function GetMainForm: Boolean;
    function GetShowForm: Boolean;
    function GetShowSource: Boolean;
    function NewFormFile(const FormIdent,
                         AncestorIdent: string): IOTAFile;
    function NewImplSource(const ModuleIdent, FormIdent,
                         AncestorIdent: string): IOTAFile;
    function NewIntfSource(const ModuleIdent, FormIdent,
                         AncestorIdent: string): IOTAFile;
    procedure FormCreated(const FormEditor: IOTAFormEditor);
  end;

implementation

uses Forms, AppBars, SysUtils, DsgnIntf;

{$R CodeGen.res}

type
  TBaseFile = class(TInterfacedObject)
  private
    FModuleName: string;
    FFormName: string;
```

```
    FAncestorName: string;
  public
    constructor Create(const ModuleName, FormName,
                       AncestorName: string);
  end;

  TUnitFile = class(TBaseFile, IOTAFile)
  protected
    function GetSource: string;
    function GetAge: TDateTime;
  end;

  TFormFile = class(TBaseFile, IOTAFile)
  protected
    function GetSource: string;
    function GetAge: TDateTime;
  end;

{ TBaseFile }

constructor TBaseFile.Create(const ModuleName, FormName,
                             AncestorName: string);
begin
  inherited Create;
  FModuleName := ModuleName;
  FFormName := FormName;
  FAncestorName := AncestorName;
end;

{ TUnitFile }

function TUnitFile.GetSource: string;
var
  Text: string;
  ResInstance: THandle;
  HRes: HRSRC;
begin
  ResInstance := FindResourceHInstance(HInstance);
  HRes := FindResource(ResInstance, 'CODEGEN', RT_RCDATA);
  Text := PChar(LockResource(LoadResource(ResInstance, HRes)));
  SetLength(Text, SizeOfResource(ResInstance, HRes));
  Result := Format(Text, [FModuleName, FFormName, FAncestorName]);
end;

function TUnitFile.GetAge: TDateTime;
begin
  Result := -1;
end;

{ TFormFile }

function TFormFile.GetSource: string;
const
```

```
  FormText =
    'object %0:s: T%0:s'#13#10'end';
begin
  Result := Format(FormText, [FFormName]);
end;

function TFormFile.GetAge: TDateTime;
begin
  Result := -1;
end;

{ TAppBarWizard }

{ TAppBarWizard.IOTAWizard }

function TAppBarWizard.GetIDString: string;
begin
  Result := 'DDG.AppBarWizard';
end;

function TAppBarWizard.GetName: string;
begin
  Result := 'DDG AppBar Wizard';
end;

function TAppBarWizard.GetState: TWizardState;
begin
  Result := [wsEnabled];
end;

procedure TAppBarWizard.Execute;
begin
  (BorlandIDEServices as
   IOTAModuleServices).GetNewModuleAndClassName(
    'AppBar', FUnitIdent, FClassName, FFileName);
  (BorlandIDEServices as IOTAModuleServices).CreateModule(Self);
end;

{ TAppBarWizard.IOTARepositoryWizard/TAppBarWizard.IOTAFormWizard }

function TAppBarWizard.GetGlyph: HICON;
begin
  Result := 0;  // Использовать стандартную пиктограмму
end;

function TAppBarWizard.GetPage: string;
begin
  Result := 'DDG';
end;

function TAppBarWizard.GetAuthor: string;
begin
```

```
  Result := 'Delphi 5 Developer''s Guide';
end;

function TAppBarWizard.GetComment: string;
begin
  Result := 'Creates a new AppBar form.'
end;

{ TAppBarWizard.IOTACreator }

function TAppBarWizard.GetCreatorType: string;
begin
  Result := '';
end;

function TAppBarWizard.GetExisting: Boolean;
begin
  Result := False;
end;

function TAppBarWizard.GetFileSystem: string;
begin
  Result := '';
end;

function TAppBarWizard.GetOwner: IOTAModule;
var
  I: Integer;
  ModServ: IOTAModuleServices;
  Module: IOTAModule;
  ProjGrp: IOTAProjectGroup;
begin
  Result := nil;
  ModServ := BorlandIDEServices as IOTAModuleServices;
  for I := 0 to ModServ.ModuleCount - 1 do begin
    Module := ModSErv.Modules[I];
    // Поиск текущей группы проекта
    if CompareText(ExtractFileExt(Module.FileName),
                   '.bpg') = 0 then
      if Module.QueryInterface(IOTAProjectGroup,
                ProjGrp) = S_OK then begin
        // Возвратить активный проект группы
        Result := ProjGrp.GetActiveProject;
        Exit;
      end;
  end;
end;

function TAppBarWizard.GetUnnamed: Boolean;
begin
  Result := True;
end;
```

```
{ TAppBarWizard.IOTAModuleCreator }

function TAppBarWizard.GetAncestorName: string;
begin
  Result := 'TAppBar';
end;

function TAppBarWizard.GetImplFileName: string;
var
  CurrDir: array[0..MAX_PATH] of char;
begin
  // Примечание: необходим полный путь!
  GetCurrentDirectory(SizeOf(CurrDir), CurrDir);
  Result := Format('%s\%s.pas', [CurrDir, FUnitIdent, '.pas']);
end;

function TAppBarWizard.GetIntfFileName: string;
begin
  Result := '';
end;

function TAppBarWizard.GetFormName: string;
begin
  Result := FClassName;
end;

function TAppBarWizard.GetMainForm: Boolean;
begin
  Result := False;
end;

function TAppBarWizard.GetShowForm: Boolean;
begin
  Result := True;
end;

function TAppBarWizard.GetShowSource: Boolean;
begin
  Result := True;
end;

function TAppBarWizard.NewFormFile(const FormIdent,
                      AncestorIdent: string): IOTAFile;
begin
  Result := TFormFile.Create('', FormIdent, AncestorIdent);
end;

function TAppBarWizard.NewImplSource(const ModuleIdent, FormIdent,
                                AncestorIdent: string): IOTAFile;
begin
  Result := TUnitFile.Create(ModuleIdent, FormIdent,
                             AncestorIdent);
end;
```

```
function TAppBarWizard.NewIntfSource(const ModuleIdent, FormIdent,
                                     AncestorIdent: string): IOTAFile;
begin
  Result := nil;
end;

procedure TAppBarWizard.FormCreated(const FormEditor:
                                       IOTAFormEditor);
begin
  // Не делает ничего
end;

end.
```

В данном модуле для генерации исходного кода используется интересный прием: неформатированный исходный код сохраняется в файле `.res`, который подключается с помощью директивы `$R`. Это очень гибкий способ хранения исходного кода мастера, позволяющий легко его модифицировать. Файл ресурсов (`.res`) создается при подключении текстового файла и файла ресурсов `RCDATA` в файл `.RC`, который затем компилируется с помощью компилятора `BRCC32`. В листингах 17.10 и 17.11 приведено содержимое файлов `CodeGen.txt` и `CodeGen.rc`.

Листинг 17.10. `CodeGen.txt` — шаблон ресурсов мастера `AppBar`

```
unit %0:s;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, AppBars;

type
  T%1:s = class(%2:s)
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  %1:s: T%1:s;

implementation

{$R *.DFM}

end.
```

Листинг 17.11. CODEGEN.RC

```
CODEGEN RCDATA CODEGEN.TXT
```

Регистрация пользовательского модуля и мастера происходит внутри процедуры `Register()` в пакете разработки, содержащем этот модуль, с помощью следующих двух строк:

```
RegisterCustomModule(TAppBar, TCustomModule);
RegisterPackageWizard(TAppBarWizard.Create);
```

Резюме

Прочитав настоящую главу, можно гораздо лучше понять принципы работы различных модулей и интерфейсов, предоставляемых Delphi интерфейсом API Open Tools. В частности, здесь изложено, как создавать мастера, встраиваемые в интегрированную среду разработки. Эта глава завершает часть IV, "Компонент-ориентированная разработка" данной книги. В следующей части V, "Разработка корпоративных приложений" изложены методы построения приложений корпоративного уровня на основании технологий COM+ и MTS.

ЧАСТЬ

V

Разработка корпоративных приложений

В ЭТОЙ ЧАСТИ...

ГЛАВА

18

Транзакционные методы разработки с применением COM+ и MTS

В ЭТОЙ ГЛАВЕ...

Выход в свет операционной системы Windows 2000 стал, возможно, самым значительным событием в сфере использования технологии COM. В этой системе реализован стандарт OLE 2.0, основой для которого послужила новая спецификация COM+, являющаяся последней версией COM, входящей в состав операционных систем Windows 2000 и Windows XP. В данной главе будут рассмотрены различные аспекты COM+ и способы применения этой технологии в приложениях Delphi.

Что такое COM+?

Перед тем как приступить к непосредственному изучению COM+, следует отметить, что для этой спецификации характерны почти все свойства технологии COM. Данной технологии посвящено немало изданий, потому в настоящей книге будут рассмотрены лишь наиболее характерные особенности COM+. Сразу же следует отметить, что COM+ — это, фактически, обычная спецификация COM, внутри которой реализованы некоторые дополнительные COM-ориентированные службы *Microsoft*. Другими словами, COM+ — это COM с некоторыми дополнениями, связанными с *сервером транзакций Microsoft* (MTS — Microsoft Transaction Server) и *очередью сообщений Microsoft* (MSMQ — Microsoft Message Queue).

Поскольку спецификация COM+ полностью совместима с COM, работа с ней в Delphi нисколько не сложнее работы с COM. При этом для разработки оптимизированных компонентов COM+ в Delphi 6 было реализовано несколько дополнений, речь о которых пойдет несколько позже. В любом случае, все возможности COM+ доступны разработчикам Delphi в полном объеме.

Почему COM?

Почему в корпорации *Microsoft* вместо того, чтобы создать нечто совершенно новое, решили избрать в качестве основы для COM+ именно технологию COM? Это — хороший вопрос, особенно в свете некоторых негативных статей относительно COM, в которых были выявлены недостатки данной технологии по сравнению с конкурирующими технологиями CORBA и *Enterprise Java Beans* (EJB). Все дело в том, что COM является не только хорошим базисом для развития новых технологий, но и по следующим причинам:

- COM не зависит от языка программирования;
- COM поддерживается всеми основными средствами разработки в среде Windows;
- по данным *Microsoft*, около 150 миллионов пользователей 32-битных систем Windows уже используют COM;
- по данным группы Giga Information Group, технология COM занимает сегмент рынка объемом в 670 миллионов долларов (не учитывая *Microsoft*).

Вероятно, самым большим недостатком COM является сложность масштабирования, когда много пользователей открывают большое количество транзакций. Как всегда в таких случаях, при разработке COM+ в *Microsoft* решили не затрачивать силы, создавая что-то новое, а доработать уже существующие средства.

Все средства COM+ можно разделить на три категории: средства администрирования, службы и средства времени исполнения. Функции администрирования, главным образом, выполняет специальное средство Component Services, которое будет рассмотрено несколько позже, а в начале данной главы речь пойдет о службах и средствах времени исполнения. Службы в COM+ имеют много новых особенностей и потому будут рассмотрены в первую очередь.

Службы

Службы являются нововведением COM+. Например, технологии, которые используются в MTS и MSMQ, являются основой для соответствующих служб COM+. Можно сказать, что службы — это надстройки COM+, внедренные *Microsoft* для расширения возможностей компонентной разработки. Как уже упоминалось выше, службы обработки транзакций и очередей были основаны на уже существующих технологиях, поэтому тот, кто знаком с этими базовыми технологиями, сможет без труда освоить и разработку приложений с использованием COM+. Тем не менее в COM+ существуют и новые службы, с которыми читателю раньше сталкиваться не приходилось (например, организация пула объектов и обработка динамически связанных событий).

Транзакции

При работе с COM+ транзакции выполняют ключевую роль, так как в этой технологии реализована транзакционная модель сервера MTS (данный вопрос будет подробно рассмотрен в настоящей главе несколько позже). Без поддержки транзакций объекты из коллекций не смогли бы взаимодействовать со сложными коммерческими приложениями. Например, в обработке транзакции, которая открывается при интерактивной покупке какого-либо товара, принимают участие несколько объектов, связанных с одной или несколькими базами данных. Эти объекты отвечают за прием запроса, проверку наличия товара, определение остатка денег на кредитной карточке, обновление данных по счету и оформление заказа. Причем все указанные действия должны быть согласованы. В противном случае может возникнуть сбой в процессе покупки, в результате чего все объекты и данные должны быть возвращены в состояние, которое они имели до начала транзакции. Если при этом, кроме всего прочего, объекты находятся на множестве различных компьютеров, то процесс управления транзакциями значительно усложняется.

Централизованное управление транзакциями выполняет специальное средство, называемое *координатором распределенных транзакций* (DTC — Distributed Transaction Coordinator). При открытии транзакции приложением COM+ активизируется координатор DTC, который вовлекает в процесс обработки транзакции другие необходимые программные средства, включая администраторы транзакций, диспетчеры ресурсов и распределители ресурсов. На каждом компьютере, принимающем участие в обработке транзакций, используется собственный администратор транзакций. Однако администраторы транзакций не обрабатывают хранимую информацию, такую как записи баз данных или сообщения из очереди сообщений. Управление подобными данными, обрабатываемыми в транзакциях, осуществляется диспетчером ресурсов. Распределители ресурсов отвечают за управление динамической информацией (например о подключении к базе данных). Таким образом, координатор DTC выпол-

няет общее управление ресурсами с использованием описанных выше специализированных средств.

Система безопасности

В наше время, когда смена новых программных технологий набрала устрашающие темпы, иногда с теплотой вспоминаешь те старые добрые времена, когда приложения состояли из одного файла с расширением `.com` или `.exe`, а общие файлы данных размещались только в локальной сети. Теперь современные коммерческие приложения зачастую используют разнотипные пользовательские интерфейсы (для Windows, Web, Java и т.д.). При этом обмен данными осуществляется между распределенными по сети программными компонентами, каждый из которых, в свою очередь, взаимодействует с одним или несколькими серверами баз данных. В наше время успех разработки приложения зависит не только от способности связывать несопоставимые элементы, но и от использования средств, обеспечивающих защищенный обмен данными. Это подразумевает построение в распределенных приложениях систем безопасности, которые должны отвечать за распознавание компонентами друг друга, определение доступных таким компонентам служб и организацию безопасного обмена данными между ними.

В этом случае вопрос организации системы безопасности — это действительно важный вопрос, так как большая часть данных должна быть защищена. Например, не все сотрудники организации могут получить доступ к базе данных отдела кадров, данные о продажах не должны быть доступны для конкурентов и т.д. Подобным же образом должны быть защищены и методы компонентов. Например, право доступа к определенным объектам могут иметь только администраторы, а доступ к средствам управления бизнес-правилами — только начальники отделов. Тем не менее на практике для организации подобной защиты в распределенных приложениях может понадобиться очень много времени, и потому ею часто пренебрегают, уделяя основное внимание реализации главных функций.

Технология COM+ содержит ряд надежных средств обеспечения безопасности, благодаря которым организация защиты приложений из сферы программирования переходит в сферу администрирования. Такой подход позволяет уделять больше времени разработке основных функций приложений, не отвлекаясь на детали организации безопасности. Настройка системы безопасности при работе с COM+ осуществляется при помощи средства администрирования Component Services. Его настройка выполняется только один раз, после чего построение системы безопасности в приложении может вообще не понадобиться. В то же время, если этого окажется недостаточно, то можно построить собственную систему безопасности с использованием специальных функций API. Далее речь пойдет о структуре системы безопасности приложений сервера COM+, а также о способах использования возможностей такой системы в приложениях.

Ролевая система безопасности

Структура системы безопасности COM+ основана на использовании ролей. Вместо управления учетными записями отдельных пользователей в приложениях COM+ используются категории или группы пользователей, которые называют *ролями*. Роли тесно взаимосвязаны с системой безопасности операционной системы, поскольку

система безопасности сервера или домена Windows рассматривает роль как набор привилегий и прав. Роль можно легко создать при помощи средства администрирования Component Services. Для этого нужно щелкнуть правой кнопкой мыши в приложении COM+ на узле Roles древовидной структуры, расположенной в левой части окна Component Services. После создания новой роли к ней можно добавить пользователей, щелкнув на ней правой кнопкой мыши. Процесс добавления пользователей роли представлен на рис. 18.1.

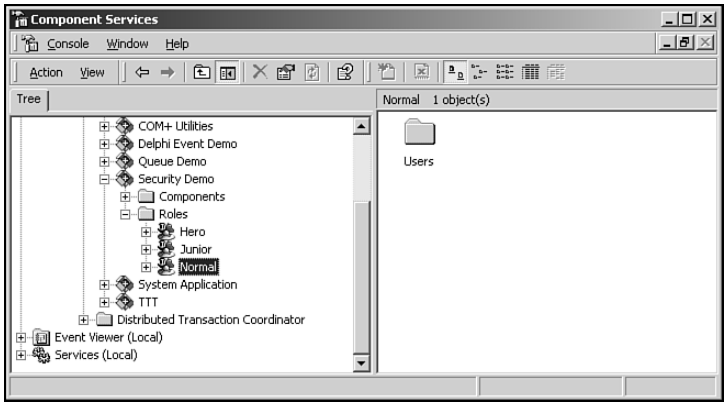

Рис. 18.1. *Использование администратора служб компонентов для настройки роли*

Как можно заметить на рис. 18.1, в данном примере приложение COM+ имеет три роли: Junior, Normal и Hero. Это — не более чем названия трех различных групп пользователей, которым в приложении будут предоставлены разные возможности. Заслуживает внимания тот факт, что собственно идентификация выполняется операционной системой автоматически, а система безопасности COM+ выступает в качестве надстройки соответствующих системных служб.

Настройка ролевой системы безопасности

Ролевая система безопасности COM+ замечательна тем, что она может быть реализована не только на уровне приложения, но и на уровне компонента, интерфейса и даже метода. Это позволяет использовать роль для контроля доступа к отдельным методам без единой строки кода, написанного вручную.

Вначале система безопасности приложения COM+ организуется именно на уровне приложения. Для этого модифицируются свойства самого приложения при помощи вкладки Security средства администрирования (консоли MMC) Component Services (рис. 18.2).

Система безопасности приложения активизируется при установке флажка Enforce access checks for this application (Установить проверку доступа для этого приложения). Кроме того, в данном диалоговом окне можно выбрать уровень защиты (только на уровне процесса или на уровне процесса и компонента).

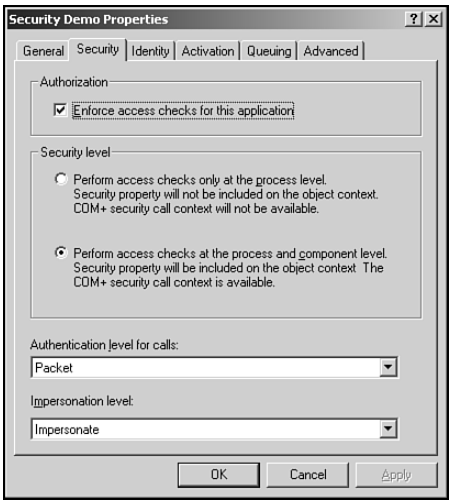

Рис. 18.2. *Настройка системы безопасности приложения COM+*

При организации защиты только на уровне процесса, доступ к приложению происходит через своего рода "двери", "ключ" от которых есть у каждого члена ролевой группы. При выборе такого режима защиты не выполняется никаких проверок на уровне компонентов, интерфейсов и методов, а информация о состоянии безопасности объектами приложения не обрабатывается. Такой подход можно использовать в тех случаях, когда не требуется строгая защита, а достаточно простого разграничения доступа к приложению различных групп пользователей. Кроме того, при таком типе защиты повышается производительность приложения, так как средствами COM+ не выполняется дополнительных проверок.

Организация защиты на уровне процесса и компонентов подразумевает выполнение ролевых проверок на уровне компонентов, интерфейсов и методов. В этом случае информация о состоянии безопасности обрабатывается объектами приложения. Такой подход более гибок и обеспечивает максимальную степень безопасности, однако значительно снижает производительность приложений из-за выполнения средствами COM+ дополнительных проверок.

В диалоговом окне редактирования свойств системы безопасности, представленном на рис. 18.2, также можно настроить уровень аутентификации приложения COM+. Этот уровень определяет момент, до которого выполняется аутентификация при обращении к приложению. Возможные значения уровня аутентификации представлены в табл. 18.1, причем каждому последующему значению соответствует более высокая степень защиты.

Таблица 18.1. Уровни аутентификации COM+

Уровень	*Описание*
None	Аутентификация не выполняется
Connect	Аутентификация выполняется только в момент подключения
Call	Аутентификация выполняется перед каждым обращением
Packet	При аутентификации проверяется получение всех данных. Это значение уровня аутентификации для серверных приложений COM+ установлено по умолчанию
Packet Integrity	При аутентификации проверяется целостность передаваемых данных
Packet Privacy	При аутентификации выполняется шифрование пакета, включая данные, идентификатор отправителя и его подпись (signature)

Обратите внимание, что в процессе аутентификации участвуют и клиент, и сервер. Средства COM+ проверяют уровень аутентификации и клиента, и сервера, а затем выбирают наибольший из них. Уровень аутентификации клиента может быть установлен одним из следующих способов:

- На уровне машины, при помощи средства администрирования Component Services (или средства DCOMCNFG для систем, отличных от Windows 2000/XP).
- На уровне приложения, при помощи средства администрирования Component Services (или средства DCOMCNFG для систем, отличных от Windows 2000/XP).
- На уровне процесса, используя программный вызов функции API COM `CoInitializeSecurity()`.
- Оперативно, при помощи программного вызова функции API `CoSetProxyBlanket()`.

И, наконец, при помощи диалогового окна свойств системы безопасности (см. рис. 18.2) можно настроить уровень *заимствования* (impersonation) прав доступа. Данный уровень определяет, в какой степени серверное приложение может заимствовать права клиента, для доступа к ресурсам от имени этого клиента. Возможные значения уровня заимствования прав доступа представлены в табл. 18.2.

Таблица 18.2. Уровни заимствования прав доступа COM+

Уровень	*Описание*
Anonymous	Клиент для сервера анонимен
Identify	Сервер может получить идентификатор клиента и выступать от его имени только при проверке прав доступа

Окончание табл. 18.2.

Уровень	*Описание*
Impersonate	Хоть и с некоторыми ограничениями, но сервер может выступать от имени клиента, при этом действует он от своего лица. Сервер может обращаться к ресурсам на том же самом компьютере, что и клиент. Если сервер находится на том же самом компьютере, что и клиент, то он может обращаться к сетевым ресурсам от имени клиента. Если сервер находится на компьютере, отличном от клиента, то он может обращаться только к тем ресурсам, которые находятся на том же самом компьютере, что и сервер. Это значение уровня заимствования установлено для серверных приложений COM+ по умолчанию
Delegate	Сервер полностью выступает от имени клиента, используя все его права вне зависимости от компьютера, на котором расположены необходимые ресурсы. Причем права клиента могут быть предъявлены для доступа к любым машинам в любом количестве. Это самые широкие полномочия, которое можно предоставить

Подобно аутентификации, заимствование прав может быть выполнено только с разрешения клиента. Получение разрешения клиента на использование его прав осуществляется точно так же, как при идентификации: при помощи средств администрирования Component Services и DCOMCNFG или функций API `CoInitializeSecurity()` и `CoSetProxyBlanket()`.

По окончании настройки системы безопасности на уровне приложения она может быть настроена на уровне компонентов, интерфейсов и методов. Для этого достаточно отредактировать соответствующее свойство элемента в древовидной структуре и на вкладке Security. При этом используется диалоговое окно, подобное представленному на рис. 18.3.

Данное диалоговое окно достаточно простое и позволяет задать режим проверки и роли, обладающие доступом к этому элемента.

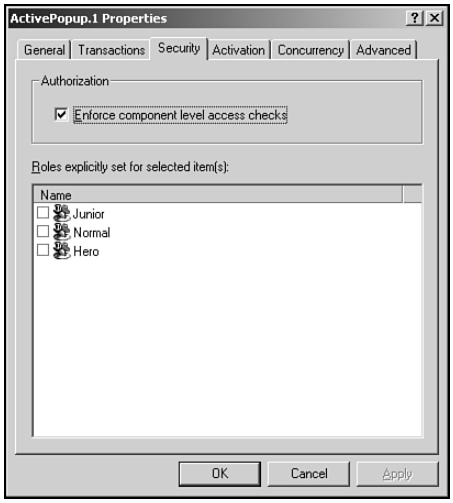

Рис. 18.3. *Настройка системы безопасности компонента COM+*

Многоуровневые приложения

При разработке многоуровневых приложений, в которых реализована система безопасности COM+, необходимо учитывать некоторые моменты, связанные с производительностью. Не нужно забывать о том, что одной из основных задач любой многоуровневой системы является улучшение общей масштабируемости. Одной из оши-

бок, которые часто приводят к противоречию между масштабируемостью и производительностью, является реализация системы безопасности приложения на нескольких уровнях многоуровневой модели. При работе со службами COM+ систему безопасности лучше всего реализовывать на среднем уровне. Например, вместо заимствования прав клиента для доступа к базе данных, значительно эффективнее получить такой доступ при помощи обычного подключения, которое может быть затем распределено между множеством клиентов.

Программная защита

До этого момента речь шла преимущественно о декларативной (то есть управляемой администратором) системе безопасности, однако в приложениях COM+ можно также использовать и программную защиту. В общем случае можно определить, принадлежит ли клиент, вызывающий некоторый метод, к определенной роли. Таким образом на основании роли клиента можно контролировать не только доступ к методам, но и их поведение. Для этого в COM+ используется два подхода. Первый из них заключается в использовании метода `IsCallerInRole()` интерфейса `IObjectContext`. Он определен следующим образом:

```
function IsCallerInRole(const bstrRole: WideString): Bool;
                                                     safecall;
```

В качестве параметра `bstrRole` этой функции передается имя роли, а возвращаемый результат типа `Boolean` указывает на принадлежность текущего клиента указанной роли. Ссылку на контекст текущего объекта можно получить при помощи функции API `GetObjectContext()`, которая определена таким образом:

```
function GetObjectContext: IObjectContext;
```

В следующем фрагменте программного кода перед выполнением некоторой задачи осуществляется проверка клиента на предмет его принадлежности роли `Hero`:

```
var
  Ctx: IObjectContext;
begin
  Ctx := GetObjectContext;
  if (Ctx <> nil) and (Ctx.IsCallerInRole('Hero')) then begin
    // Здесь выполняются действия
  end;
end;
```

Точно также можно использовать метод `IsCallerInRole()` интерфейса `ISecurityCallContext` через ссылку, которая может быть получена при помощи функции API `CoGetCallContext()`. Использование этой версии метода более предпочтительно, так как интерфейс `ISecurityCallContext` предоставляет массу другой информации о системе безопасности (например информацию о клиенте, его аутентификации и уровнях заимствования прав доступа).

Оперативная активизация

Оперативная (JIT – Just-In-Time) активизация относится к встроенным функциям COM+, позволяющим удалять и восстанавливать объекты без ведома клиентского

приложения. При этом сервер потенциально окажется способен взаимодействовать с бо́льшим количеством клиентов, так как сможет освобождать системные ресурсы, временно не используемые одним клиентом, и предавать их другим, которые нуждаются в них в данный момент. Тем не менее, если первому пользователю понадобятся его объекты, то система сможет их восстановить.

Разработчик способен полностью контролировать процесс дезактивации объекта. Кроме того, объект может быть дезактивирован только в том случае, если в данный момент нет информации о его состоянии[1]. Дезактивация объектов осуществляется при помощи методов `SetComplete()` и `SetAbort()` интерфейса `IObjectContext` или метода `SetDeactivateOnReturn()` интерфейса `IContextState`.

Компоненты для работы с очередью

Обычно все разработчики Delphi хорошо осведомлены о преимуществах приложений, в которых используется *сокращенная модель* (briefcase model). С тех пор как в Delphi 3 была реализована технология MIDAS, появилась возможность создавать приложения, которые способны функционировать даже при отключении клиента от сервера. Разработчики Delphi быстро осознали все преимущества сокращенной модели, позволяющей пользователям приложений работать с данными даже при отсутствии соединения с сервером, и начали повсеместно использовать MIDAS и другие подобные технологии. Например, если необходимо создать приложение, позволяющее менеджеру вносить данные о заказчиках в базу на портативном компьютере, а по возвращении в офис синхронизировать их с базой на сервере, то вместо того, чтобы писать длинный и сложный код, достаточно просто разместить в форме набор необходимых компонентов и написать несколько строк кода.

Когда дело касается данных, то все действительно работает очень хорошо. Но как быть с объектами? Для создания приложений, предназначенных для удаленного подключения к объектам, используются такие технологии, как DCOM, MTS/COM+ и CORBA. Они часто применяются при разработке программных средств для различных компаний и их клиентов. Еще чаще технологии удаленного подключения к объектам используются при разработке сложных распределенных приложений. В результате приложения с распределенными компонентами (например приложения для работы с базами данных) также должны уметь функционировать при отключении от сервера.

Компоненты для работы с очередью: сокращенная объектная модель

В основе *компонентов для работы с очередями* (queued components) лежит технология *очереди сообщений Microsoft* (MSMQ — Microsoft Message Queue), которая предоставляет клиентам COM+ возможность асинхронного вызова методов компонентов сервера COM+. По существу это означает, что клиенты могут создавать экземпляры серверных объектов и вызывать их методы независимо от того, доступен сервер или нет. Средства COM+ осуществляют управление этим процессом, размещая вызовы методов в очереди, а непосредственное выполнение этих методов происходит в дальнейшем при восстановлении подключения к серверу. Более того, серверным объектам фактически безразлично, вызываются ли их методы непосредственно или через оче-

[1] *Т.е. объект не используется.* – Прим. ред.

редь COM+. Далее рассмотрим некоторые ключевые моменты использования компонентов, предназначенных для работы с очередью COM+.

На рис. 18.4 схематически представлена внутренняя реализация компонентов для работы с очередью. При обращении клиента к методу такого компонента его вызов перехватывается *регистратором* (recorder), который упаковывает его вместе со всеми параметрами и помещает в очередь. Поскольку клиент успешно передал вызов, ему кажется, что он подключен к серверу, но фактически это не так. Таким образом сервер регистратора представляет собой своего рода прокси для настоящего сервера. Регистратор знает, как вести себя, поскольку он получает информацию о сервере из его библиотеки типов, конфигурационной или регистрационной записи. *Приемник* (listener) считывает и удаляет из очереди сообщение, которое содержит информацию о вызове, а затем передает его на *проигрыватель* (player). И, в заключение, проигрыватель распаковывает информацию обращения (вместе с сопутствующей информацией, например о контексте безопасности клиента) и выполняет метод, обращаясь к серверу.

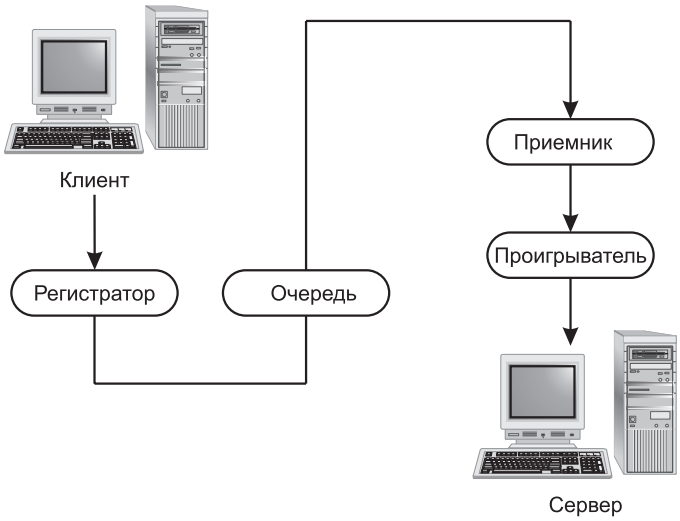

Рис. 18.4. *Архитектура компонента для работы с очередью COM+*

На данном этапе может возникнуть вполне закономерный вопрос: звучит неплохо, но не слишком ли это все сложно? В действительности в представленной архитектуре нет ничего сложного.

Преимущества компонентов для работы с очередью

Перед тем как углубиться в детали реализации, рассмотрим некоторые преимущества компонентов для работы с очередью.

- *Масштабируемость системы.* Система без очереди способна одновременно обрабатывать лишь ограниченное количество серверных объектов. Если все серверные объекты будут заняты обработкой клиентских вызовов, то остальные входящие вызовы будут заблокированы до тех пор, пока какой-либо объект не освободится. Когда в подобных системах одновременно открывается большое

количество транзакций, это приводит к значительному снижению количества параллельно обслуживаемых клиентов. В случае использования очереди вызовов отклик от метода всегда возвращается клиенту сразу же после его помещения в очередь и дальнейшей передачи на сервер. Благодаря такому подходу системы могут обрабатывать большое количество одновременно открытых транзакций.

Масштабируемость также увеличивается и на стороне сервера, так как клиенты не влияют на активность сервера. Серверу с очередью не обязательно быть активным именно в то время, когда клиент осуществляет вызовы методов. Он обязательно должен быть активен только в тот момент, когда вызов обрабатывается регистратором. Сокращение времени простоя сервера приводит к увеличению возможного количества одновременно открытых серверов при том же объеме оперативной памяти.

- *Сокращенная модель.* Как уже упоминалось ранее, очереди COM+ используются в случае удаленного обращения к объектам при отсутствии фактического подключения к серверу, наподобие того как средства MIDAS используются при удаленном обращении к базам данных. При таком подходе клиенты могут работать при отсутствии подключения к сети, а вызовы методов передаются на сервер сразу же после восстановления подключения.
- *Безаварийность.* При разработке приложений, для которых требуется высокая степень надежности (например виртуальный магазин), очень важно учитывать возможность потери связи с серверными объектами. Компоненты для работы с очередями дают идеальную защиту от подобных проблем, так как в случае отключения клиента от сервера вызовы методов будут занесены в очередь, а затем после восстановления подключения будут обработаны.
- *Возможность планирования.* Вместо того чтобы перегружать серверы в часы пиковой активности и оставлять их бездействовать в остальное время, при помощи очередей можно равномерно распределить обработку вызовов в течение дня, снизив тем самым нагрузку на серверы.

Создание сервера

Между разработкой компонента, предназначенного для работы с очередью, и обычного компонента COM/COM+ есть некоторые различия. Они, в основном, связаны с тем, что все методы интерфейсов для работы с очередью должны иметь только входные параметры и не должны возвращать никаких значений. Это объясняется тем, что клиенты не могут ожидать от сервера немедленного возврата каких-либо результатов или значений параметров сразу же после вызова методов. Кроме того, при установке компонентов, предназначенных для работы с очередями, необходимо выполнить некоторые дополнительные действия.

В качестве примера создадим сервер, содержащий один класс COM+ с одним интерфейсом и одним методом. Для облегчения задачи воспользуемся мастером Automation Object Wizard, вызвать который можно в меню File пункт New. Новый объект будет называться `QTest`, а первичному интерфейсу мастер автоматически присвоит имя `IQTest`. В интерфейс `IQTest` добавим один метод, который в редакторе библиотеки типов необходимо определить следующим образом:

```
procedure SendText(Value: WideString;
                   Time: TDateTime) [dispid $00000001]; safecall;
```

Данный метод принимает два параметра, первый из которых — это строка сообщения, а второй — время вызова метода. В рассматриваемой реализации такого метода сообщение просто выводится на экран, а кроме того, производится запись о времени обработки метода сервером в файл журнала `c:\queue.txt`. Содержимое файла реализации этого объекта автоматизации представлено в листинге 18.1.

Листинг 18.1. `TestImpl.pas` — реализация объекта, взаимодействующего с очередью

```
unit TestImpl;

interface

uses
  Windows, ComObj, ActiveX, Srv_TLB, StdVcl;

type
  TQTest = class(TAutoObject, IQTest)
  protected
    procedure SendText(const Value: WideString;
                       Time: TDateTime); safecall;
  end;

implementation

uses ComServ, SysUtils;

procedure TQTest.SendText(const Value: WideString;
                          Time: TDateTime);
const
  SFileName = 'c:\queue.txt';
  SEntryFormat = 'Send time:  %s'#13#10 +
                 'Write time: %s'#13#10 +
                 'Message:    %s'#13#10#13#10;
var
  F: THandle;
  WriteStr: string;
begin
  F := CreateFile(SFileName, GENERIC_WRITE, FILE_SHARE_READ, nil,
                  OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
  if F = INVALID_HANDLE_VALUE then RaiseLastWin32Error;
  try
    FileSeek(F, 0, 2);  // переход к концу файла
    WriteStr := Format(SEntryFormat, [DateTimeToStr(Time),
                       DateTimeToStr(Now), Value]);
    FileWrite(F, WriteStr[1], Length(WriteStr));
  finally
    CloseHandle(F);
  end;
```

```
end;

initialization
  TAutoObjectFactory.Create(ComServer, TQTest, Class_QTest,
                            ciMultiInstance, tmApartment);
end.
```

После создания сервера необходимо установить его в новом приложении COM+ при помощи либо средства администрирования Component Services, либо функций API библиотеки администрирования COM+. Если используется Component Services, то на первом этапе необходимо создать новое пустое приложение. Для этого следует выбрать соответствующий элемент в локальном меню узла COM+ Applications в древовидной структуре и выполнить предлагаемые действия. После создания приложения корректируются его свойства и оно обозначается как приложение для работы с очередями (рис. 18.5). Кроме того, необходимо активизировать режим просмотра очереди, чтобы все сообщения обрабатывались сразу же после их поступления. Диалоговое окно настройки представлено на рис. 18.5.

Для того чтобы установить сервер в приложении COM+, выберите в контекстном меню узла Component древовидной структуры пункты New и Component, что приведет к вызову мастера COM Component Install Wizard, который установит новый компонент с параметрами по умолчанию. При этом для него будет выбрано имя библиотеки DLL созданного ранее сервера COM+. После установки нового объекта в приложении отредактируйте свойства его интерфейса `IQTest` для поддержки очередей (рис. 18.6).

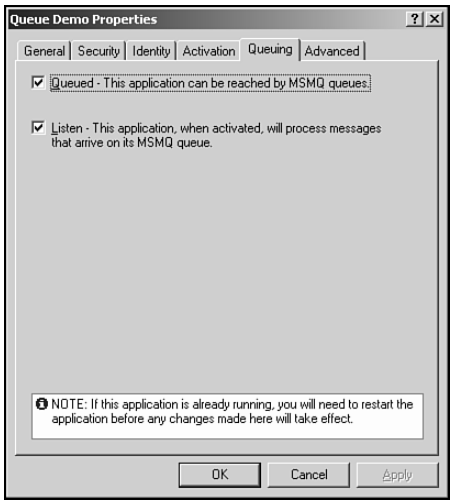

Рис. 18.5. *Настройка приложения COM+, предназначенного для работы с очередью*

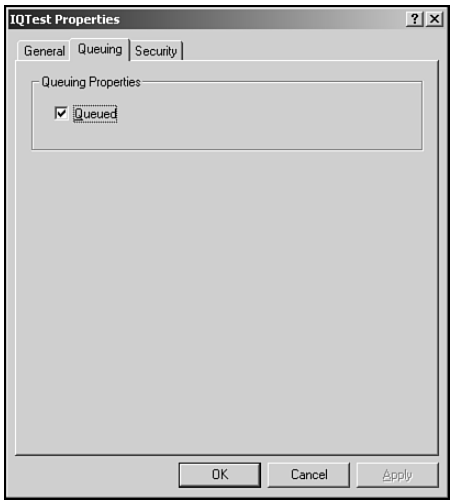

Рис. 18.6. *Установка интерфейса для работы с очередью*

Обратите внимание: согласно требованиям спецификации COM+ очередь должна быть доступна как на уровне приложения COM+, так и на уровне интерфейса.

Создание клиента

Процесс создания клиентского компонента для работы с очередью идентичен процессу создания рассмотренного ранее клиента автоматизации. В данном случае необходимо создать приложение, главная форма которого представлена на рис. 18.7.

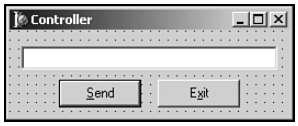

Рис. 18.7. *Клиентское приложение для компонента, предназначенного для работы с очередью*

При щелчке на кнопке Send (Отправить) содержимое поля ввода при помощи метода `SendText()` отправляется на сервер. Программный код модуля, соответствующего этой форме, представлен в листинге 18.2.

Листинг 18.2. `Ctrl.pas` — главный модуль клиента компонента, предназначенного для работы с очередями

```
unit Ctrl;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ColorGrd, ExtCtrls, Srv_TLB, Buttons;

type
  TControlForm = class(TForm)
    BtnExit: TButton;
    Edit: TEdit;
    BtnSend: TButton;
    procedure BtnExitClick(Sender: TObject);
    procedure BtnSendClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    FIntf: IQTest;
  end;

var
  ControlForm: TControlForm;

implementation

{$R *.DFM}

uses ComObj, ActiveX;

// Необходимо импортировать метод CoGetObject, потому что импорт
// в модуле ActiveX некорректен:
function MyCoGetObject(pszName: PWideChar;
                       pBindOptions: PBindOpts; const iid: TIID;
```

```
                        out ppv): HResult; stdcall;
                        external 'ole32.dll' name 'CoGetObject';

procedure TControlForm.BtnExitClick(Sender: TObject);
begin
  Close;
end;

procedure TControlForm.BtnSendClick(Sender: TObject);
begin
  FIntf.SendText(Edit.Text, Now);
  Edit.Clear;
end;

procedure TControlForm.FormCreate(Sender: TObject);
const
  SMoniker: PWideChar = 'queue:/new:{64C576F0-C9A7-420A-9EAB-
↳0BE98264BC9D}';
begin
  // Создать объект, используя моникер, определяющий параметры
  // постановки в очередь
  OleCheck(MyCoGetObject(SMoniker, nil, IQTest, FIntf));
end;

end.
```

Единственное отличие данного модуля от стандартного контроллера автоматизации заключается в создании экземпляра серверного объекта. Например, вместо функции API COM `CoCreateInstance()` этим клиентом используется функция API `CoGetObject()`.

Функция `CoGetObject()` позволяет создавать объекты через так называемый моникер (moniker). В COM+ принят специальный синтаксис строкового моникера, который используется для вызова компонентов из очереди. Строка моникера начинается с `queue:/new:`, после чего следует идентификатор класса или программы серверного объекта. Ниже представлены примеры правильно отформатированных моникеров очереди:

```
queue:/new:Srv.IQTest
queue:/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}
queue:/new:64C576F0-C9A7-420A-9EAB-0BE98264BC9D
```

Ниже перечислены параметры, которые используются в моникерах и могут быть включены в строку для изменения направления или поведения очереди:

- `ComputerName` — значением этого параметра является строка с именем компьютера, содержащего очередь. Он задает имя компьютера в пути к очереди. Если значение данного параметра не указано, то используется имя компьютера, на котором установлено приложение.
- `QueueName` — значением этого параметра является строка с именем очереди на конечном сервере. Он задает имя очереди. Если значение данного параметра не указано, то используется имя очереди, связанной с приложением.

- PathName — путь к очереди должен соответствовать шаблону ComputerName\QueueName. Этот параметр задает полный путь к очереди. Если его значение не определено, то используется путь к очереди, связанной с приложением.
- FormatName — значением этого параметра является название формата очереди (например DIRECT=9CA3600F-7E8F-11D2-88C5-00A0C90AB40E). Он задает название формата очереди.
- AppSpecific — например, AppSpecific=8675309. Значением этого параметра является беззнаковое целое число, которое используется приложением для некоторых специфических задач.
- AuthLevel — MQMSG_AUTH_LEVEL_NONE (0) или MQMSG_AUTH_LEVEL_ALWAYS (1). Этот параметр задает уровень аутентификации сообщений и определяет необходимость запроса сертификата у пользователя, отправившего сообщение.
- Delivery — MQMSG_DELIVERY_EXPRESS (0) или MQMSG_DELIVERY_RECOVERABLE (1). Этот параметр задает характер доставки сообщения и в транзакционных очередях не используется.
- EncryptAlgorithm — CALG_RC2, CALG_RC4 или другие целочисленные значения, распознаваемые средствами COM+ как идентификаторы алгоритмов шифрования. Этот параметр задает тип алгоритма, который используется COM+ для шифрования и дешифрования сообщений.
- HashAlgorithm — CALG_MD2, CALG_MD4, CALG_MD5, CALG_SHA, CALG_SHA1, CALG_MAC, CALG_SSL3_SHAMD5, CALG_HMAC, CALG_TLS1PRF и другие целочисленные значения, распознаваемые средствами COM+ как идентификаторы хеш-функций шифрования.
- Journal — MQMSG_JOURNAL_NONE (0), MQMSG_DEADLETTER (1) или MQMSG_JOURNAL (2). Эти идентификаторы используются при работе с журналом очереди сообщений COM+.
- Label — значением этого параметра может быть любая строка. Он задает метку сообщения длиной в MQ_MAX_MSG_LABEL_LEN символов.
- MaxTimeToReachQueue — INFINITE, LONG_LIVED или какое-либо целочисленное значение, указывающее количество секунд. Этот параметр задает максимальное время в секундах, отводимое сообщению на занесение в очередь.
- MaxTimeToReceive — INFINITE, LONG_LIVED или какое-либо целочисленное значение, указывающее количество секунд. Данный параметр задает максимальное время в секундах, отводимое сообщению до приема конечным приложением.
- Priority — MQ_MIN_PRIORITY (0), Q_MAX_PRIORITY (7), MQ_DEFAULT_PRIORITY (3) или какое-либо целочисленное значение в диапазоне от 0 до 7. Этот параметр задает уровень приоритета сообщения. При этом используются значения, допустимые в MSMQ.
- PrivLevel — MQMSG_PRIV_LEVEL_NONE, NONE, MQMSG_PRIV_LEVEL_BODY, BODY, MQMSG_PRIV_LEVEL_BODY_BASE, BODY_BASE, MQMSG_PRIV_LEVEL_BODY_ENHANCED или BODY_ENHANCED. Этот параметр задает уровень конфиденциальности шифрованных сообщений.

- `Trace` — `MQMSG_TRACE_NONE (0)` или `QMSG_SEND_ROUTE_TO_REPORT_QUEUE (1)`. Этот параметр задает режим трассировки маршрутов для очереди COM+.

Моникер, в котором используются некоторые из перечисленных параметров, может выглядеть следующим образом:

```
queue:Priority=6,ComputerName=foo/new:{64C576F0-C9A7-420A-9EAB-
0BE98264BC9D}
queue:PathName=drevil\myqueue/new:{64C576F0-C9A7-420A-9EAB-
0BE98264BC9D}
```

Запуск сервера

После того как в текстовом поле формы клиентского приложения будет введено несколько строк, файл `c:\queue.txt` на жестком диске создан не будет.

Дело в том, что до передачи сообщений в очередь должно быть запущено серверное приложение. Это можно сделать тремя различными способами:

1. Вручную, при помощи средства Component Services. Для этого достаточно выбрать в контекстном меню узла приложения в древовидной структуре пункт Start.
2. Программно, при помощи функций API библиотеки администрирования COM+.
3. С использованием сценария, подобного сценарию диспетчера задач:

```
dim cat
set cat = CreateObject("COMAdmin.COMAdminCatalog");
cat.StartApplication("YourApplication");
```

Теперь после запуска клиентского приложения на жестком диске будет создан файл `c:\queue.txt`, имеющий примерно следующее содержание:

```
Send time:  7/6/2001 7:15:08 AM
Write time: 7/6/2001 7:15:18 AM
Message:    this is a test

Send time:  7/6/2001 7:15:10 AM
Write time: 7/6/2001 7:15:18 AM
Message:    this is another
```

Объектный пул

В интерфейсе `IObjectControl` есть метод `CanBePooled()`, который раньше работал некорректно, а сервер MTS его просто игнорировал. В COM+ этот метод был исправлен, что позволяет теперь создавать *объектный пул* (object pooling). Пул используется для временного хранения экземпляров отдельных объектов и предоставления доступа к ним нескольким клиентам. Так же, как и в случае с оперативной (JIT) активизацией, целью объектного пула является повышение производительности системы. Однако оперативная активизация используется только в том случае, если на процесс создания и уничтожения объектов не требуется много ресурсов. В противном случае вместо создания и уничтожения объектов лучше постоянно хранить их экземпляры в объектном пуле.

Объект, помещаемый в пул, должен удовлетворять следующим условиям:

- Он не должен учитывать состояния (т.е. данные экземпляра не должны сохраняться между вызовами методов).
- Объект не должен использовать потоков. Это означает, что он не должен быть связан ни с одним потоком и не должен использовать локальное хранилище потока.
- Объект должен допускать агрегацию.
- Ресурсы в транзакции должны распределяться программно, так как диспетчер ресурсов не может выполнять автоматическое распределение на стороне объекта.
- В объекте должен быть реализован интерфейс `IObjectControl`.

События

Того, кто разрабатывал приложения в Delphi, нет необходимости убеждать в важности событий. Без них невозможно определить момент нажатия кнопки или сохранения изменений в записи таблицы. Тем не менее, несмотря на всю важность событий, разработчики COM часто предпочитали их не использовать из-за усложнения приложений. В COM+ была реализована новая модель обработки событий, абсолютно не связанная с *моделью указателей на подключение* (connection points model) от *Byzantine*, которая использовалась в COM.

Процесс взаимодействия между клиентами и серверами COM достаточно прямолинеен: клиенты вызывают методы на серверах, а серверы выполняют соответствующие действия и возвращают клиентам некоторые данные. Такая модель используется, вероятно, в 90% всех приложений COM типа клиент/сервер, однако даже неопытному разработчику COM очевидна ее ограниченность. В частности, на стороне клиента не может происходить быстрое обновление данных, измененных на сервере.

Самым простым способом получения подобной информации является периодический опрос серверов на предмет изменений интересующих клиента данных. Однако такой подход имеет явные недостатки: на выполнение опросов уходит много времени и создается дополнительная нагрузка на сеть, что приводит к снижению общей масштабируемости системы.

Было бы хорошо, воспользоваться какой-нибудь промежуточной системой, через которую клиенты могли бы передавать серверам предопределенные интерфейсы для извлечения информации об изменениях данных. К сожалению, такую систему тяжело реализовать технически. С каждым интерфейсом она будет взаимодействовать посвоему, а на сервере при этом создается специальное приложение для работы с подключениями множества клиентов.

Традиционные средства COM предоставляют более эффективный и структурированный подход, который заключается в использовании событий. При работе с событиями используются указатели на подключения, которые позволяют серверам отслеживать клиентов, запрашивающих информацию об изменениях. При этом для передачи уведомлений серверы вызывают методы клиентов. Указатели на подключения являются примером системы обработки *жестко связанных событий* (TCE — Tightly Coupled Event). В системе TCE клиенты и серверы производят взаимную идентификацию. Кроме того, для использования систем TCE требуется, чтобы клиенты и серверы работали одновременно и в них отсутствовали средства фильтрации событий. У систем с указателями на подключения также есть один характерный недостаток: они

достаточно сложны в реализации и использовании. По этой причине клиенты предпочитают реализовывать интерфейсы событий целиком даже в том случае, если их интересует только один метод.

В спецификации COM+ реализована новая система событий, которая разрешает некоторые старые проблемы и содержит несколько полезных дополнений. Модель событий COM+ известна как система *не жестко связанных событий* (LCE — Loosely Coupled Event). Такое название обусловлено тем, что между серверами (генераторами событий) и клиентами (получателями событий) нет жесткого соединения. Вместо этого, события, создаваемые генераторами, регистрируются в одном каталоге COM+, а события, в которых заинтересованы получатели,— в другом. При генерации нового события средства времени исполнения COM+ сначала просматривают специальную базу данных, чтобы определить клиентов, которым должно быть передано уведомление, а затем передают таким клиентам уведомление о событии. При этом клиенты не обязательно должны быть активными в момент возникновения события, так как они активизируются средствами COM при инициализации события. Кроме того, модуль регистрации событий поддерживает модульность на уровне методов. Это означает, что получатели могут не реализовывать методы для тех событий, которые их не интересуют. Структура системы событий COM+ представлена на рис. 18.8.

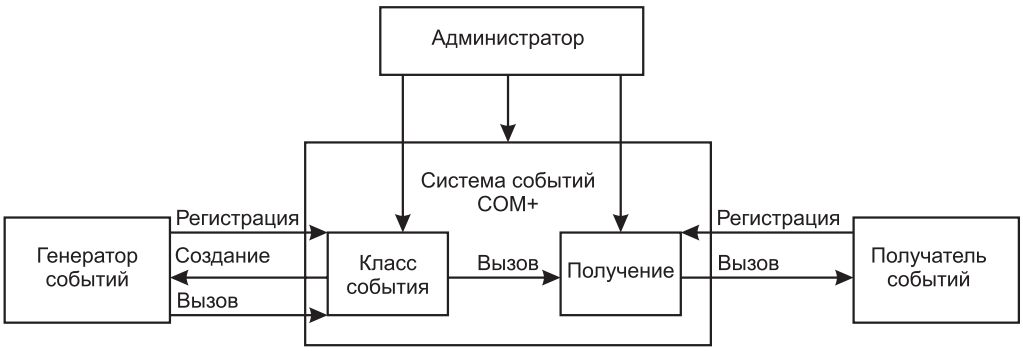

Рис. 18.8. *Архитектура системы событий COM+*

Согласно представленной схеме, процесс начинается в тот момент, когда генератор регистрирует новый класс события. Это может быть сделано при помощи средства администрирования Component Services или метода `ICOMAdminCatalog.InstallEventClass()`. После регистрации объект, реализующий класс события, располагается в памяти и становится доступен средствам времени исполнения COM+. После этого генератор или какой-либо другой объект, способный создать событие, может вызвать функцию API COM `CoCreateInstance()` и, таким образом, обратиться к методам этого объекта для передачи события.

На стороне получателя класс события может быть зарегистрирован для постоянного использования при помощи средства администрирования Component Services или временного при помощи функции администрирования API COM. При постоянной регистрации компонент получателя не нуждается в активизации в момент возникновения события, так как он будет создан автоматически средствами времени исполнения COM+. Временная регистрация класса события применяется для уже ак-

тивных компонентов, которые временно используются для приема уведомлений. При генерации события на стороне сервера средства COM+ выполняют просмотр всех зарегистрированных получателей, поочередно передавая им это новое событие. Порядок просмотра клиентов при этом определить нельзя, но зато можно управлять процессом передачи событий при помощи фильтров. Вопрос фильтрации событий будет подробно рассмотрен несколько позднее.

Фактически процесс создания события COM+ можно разделить на пять этапов:

1. Создание класса события на сервере.
2. Регистрация и настройка сервера класса события.
3. Создание сервера получателя.
4. Регистрация и настройка сервера получателя.
5. Передача событий.

Вскоре данные этапы будут рассмотрены на конкретном примере реализации событий COM+ в Delphi.

Создание сервера класса события

Для того чтобы создать сервер класса события, необходимо прежде всего создать *внутренний сервер COM* (in-process COM server), в который будут добавляться объекты COM. Сервер класса события отличается от обычного сервера COM тем, что он не содержит никаких реализаций. Это — просто сервер, предназначенный для передачи определения класса события.

В Delphi сервер класса события создается при помощи мастера ActiveX Library (для создания сервера COM в виде библиотеки DLL) и мастера Automation Object (для создания класса события и интерфейса). Присвоим новому объекту имя `EventObj`. Для завершения создания сервера мастер откроет *редактор библиотеки типов* (Type Library Editor). Используя его, добавьте в интерфейс `IEventObj` вызов метода `MyEvent` (метод события). Содержимое файла реализации, созданного для данной библиотеки типов, представлено в листинге 18.3.

Листинг 18.3. `PubMain.pas` — главный модуль сервера класса события

```
unit PubMain;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  ComObj, ActiveX, Publisher_TLB, StdVcl;

type
  TEventObj = class(TAutoObject, IEventObj)
  protected
    function MyEvent(const EventParam: WideString): HResult;
                                                     safecall;
    { Защищенные объявления }
  end;

implementation
```

```
uses ComServ;

function TEventObj.MyEvent(const EventParam: WideString): HResult;
begin

end;

initialization
  TAutoObjectFactory.Create(ComServer, TEventObj, Class_EventObj,
                            ciMultiInstance, tmApartment);
end.
```

Вот и все, что необходимо для создания сервера класса события. Обратите внимание на следующее: на данном этапе такой сервер не регистрируется, потому что регистрация выполняется отдельно. Именно об этом пойдет речь при рассмотрении следующего этапа.

Регистрация и настройка сервера класса события

На данном этапе опять будет использоваться средство администрирования Component Services. Это средство часто применяется при разработке приложений COM+. Оно находится в группе Administrative Tools (Средства администрирования) раздела Programs (Программы) меню кнопки Start (Пуск). В первую очередь в средстве Component Services создается новое приложение COM+. Для этого в контекстном меню узла COM+ Applications древовидной структуры нужно выбрать пункты New и Application. В результате будет вызван мастер COM+ Application Install Wizard (рис. 18.9). С помощью этого мастера создайте новое приложение и назовите его `DelphiEventDemo`.

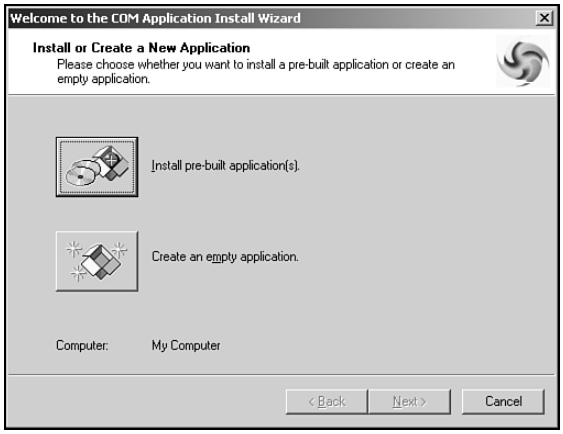

Рис. 18.9. *Создание приложения COM+ при помощи средства Component Services*

После установки приложения COM+ можно установить в нем сервер класса события. Для этого в контекстном меню узла Components древовидной структуры выбери-

те пункты New и Component. В результате будет вызван мастер COM Component Install Wizard (рис. 18.10).

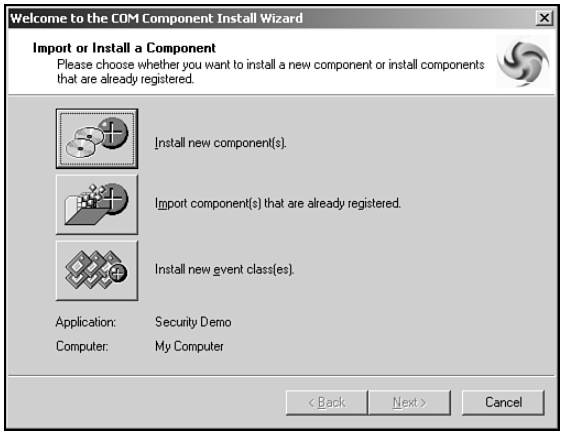

Рис. 18.10. *Создание компонента COM+ при помощи средства Component Services*

В этом мастере установите новый класс события и выберите имя для файла данного класса. Теперь можно приступить к созданию сервера получателя.

Создание сервера получателя

Сервер получателя (subscriber server) — это, по сути, стандартный сервер автоматизации Delphi. Единственным его отличием является необходимость реализации интерфейса события, определенного при создании сервера класса события. Для этого сервер получателя использует библиотеку типов сервера класса события. Кроме того, в раздел реализации такого вспомогательного класса добавляется интерфейс `IEventObj`. Вспомогательный класс `SubObj`, содержащий интерфейсы `ISubObj` и `IEventObj`, представлен на рис. 18.11, а содержимое файла реализации для данной библиотеки типов — в листинге 18.4.

Листинг 18.4. `SubMain.pas` — модуль реализации сервера события

```
unit SubMain;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  ComObj, ActiveX, Subscriber_TLB, Publisher_TLB, StdVcl;

type
  TSubObj = class(TAutoObject, ISubObj, IEventObj)
  protected
    function MyEvent(const EventParam: WideString): HResult;
                                                     safecall;
```

```
    { Защищенные объявления }
  end;

implementation

uses ComServ, Windows;

function TSubObj.MyEvent(const EventParam: WideString): HResult;
begin
  MessageBox(0, PChar(string(EventParam)), 'COM+ Event!', MB_OK);
  Result := S_OK;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TSubObj, Class_SubObj,
                            ciMultiInstance, tmApartment);
end.
```

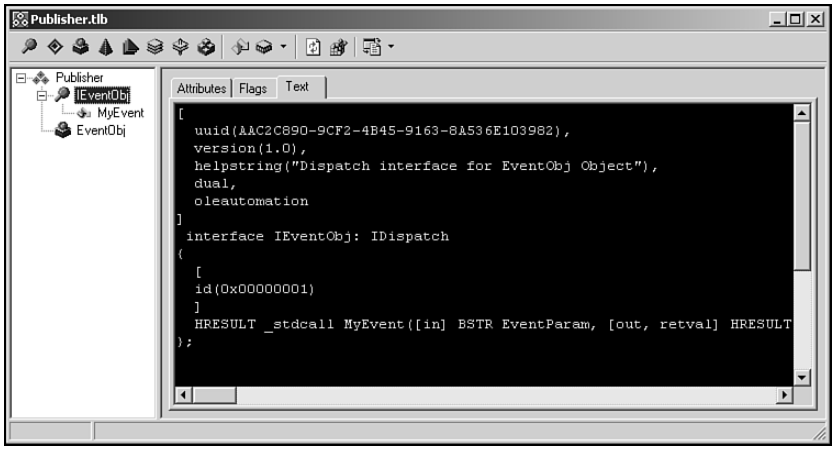

Рис. 18.11. *Интерфейс `IEventObj` в редакторе библиотеки типов*

Как видите, реализация этого события достаточно проста и содержит только вывод окна сообщения с текстовой строкой. Данный сервер также не нужно регистрировать, потому что он является стандартным сервером COM. Остальные вспомогательные действия обсуждаются на следующем этапе.

Регистрация и настройка серверов получателей

Для того чтобы зарегистрировать сервер получателя, необходимо открыть средство администрирования Component Services и выбрать в контекстном меню пункты New и Component, так же, как это делалось для сервера класса события. Отличие состоит лишь в том, что на этот раз в мастере COM Component Install Wizard устанавливает новый компонент и для получателя создается библиотека DLL.

После установки сервера получателя для него можно создать новую *подписку* (subscription). Для этого в контекстном меню узла Subscriptions выберите пункты New

и Subscription. В результате будет вызван мастер New Subscription Wizard, в котором можно установить соответствие между интерфейсами или методами генератора и получателя события. В данном случае для методов получателя выберите интерфейс `IEventObj`, а для класса события — класс `Publisher.EventObj`. В качестве имени этой подписки укажите "Subscription of Doom" и установите незамедлительную активизацию сервера (рис. 18.12).

Полное определение приложения COM+ в окне средства администрирования Component Services представлено на рис. 18.13.

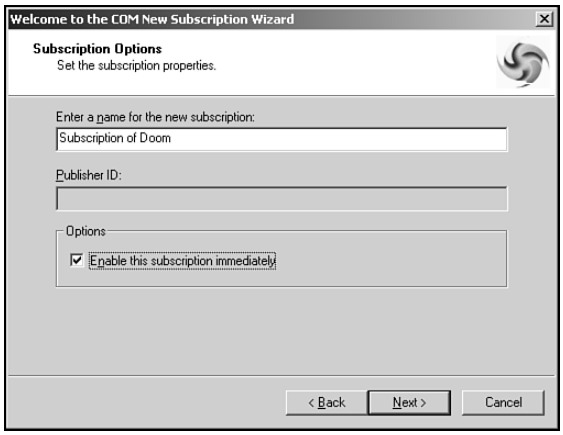

Рис. 18.12. *Мастер подписки из средства Component Services*

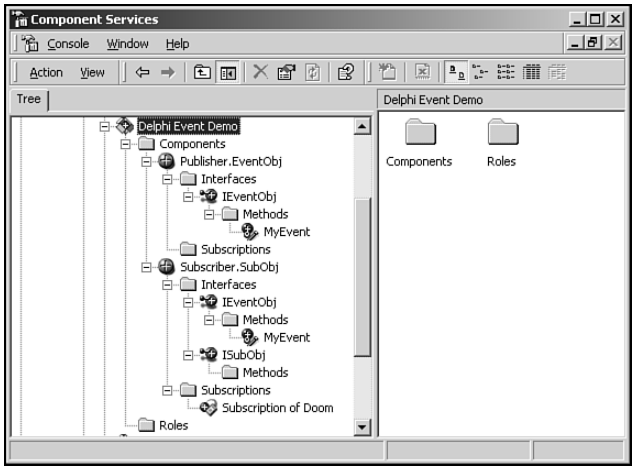

Рис. 18.13. *Приложение, демонстрирующее передачу события, открыто в окне Component Services*

Генерация событий

Процесс установки завершен, и теперь осталось лишь сгенерировать событие. Для этого нужно создать экземпляр класса `EventObj` и вызвать метод `IEventObj.MyEvent`. Проще всего это сделать так, как показано в тестовом приложении, представленном в листинге 18.5.

Листинг 18.5. `TestU.pas` — модуль, передающий не жестко связанное событие

```
unit TestU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Publisher_TLB, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    FEvent: IEventObj;
  end;

var
  Form1: TForm1;

implementation

uses ComObj, ActiveX;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  OleCheck(CoCreateInstance(CLASS_EventObj, nil, CLSCTX_ALL,
           IEventObj, FEvent));
  FEvent.MyEvent('This is a clever string');
end;

end.
```

Результат щелчка на кнопке представлен на рис. 18.14. Обратите внимание: получатель события был создан автоматически средствами COM+, а код обработчика события — выполнен.

Рис. 18.14. *Приложение, демонстрирующее передачу события, в действии*

Также можно отметить тот факт, что на передачу события средствами COM+ затрачено несколько секунд. Это объясняется достаточно большим количеством загружаемых внутренних структур, необходимых для передачи событий COM+. В заключение следует отметить, что события никогда не передаются получателю моментально — такой процесс всегда занимает определенное время.

Более сложные вопросы

Рассмотренный в настоящем разделе материал содержит лишь базовые концепции модели событий COM+, однако в нем не упоминалось еще о нескольких очень важных особенностях. Не был затронут вопрос событий очередей, которые представляют собой синтез событий COM+ и компонентов для работы с очередями. События очередей используются при работе с компонентами, отключенными от сервера. Еще одним вопросом, достойным рассмотрения, являются фильтры событий. Фильтры событий бывают двух типов: фильтры генераторов и параметрические фильтры. Фильтры первого типа позволяют генераторам управлять вызовом методов событий через их класс. Параметрические фильтры дают возможность генераторам перехватывать события в соответствии со значениями их параметров.

Средства времени исполнения

Средства времени исполнения COM+, по существу, аналогичны средствам времени исполнения COM. Они включают в себя все функции API COM (их название начинается с префикса `Co...`), а также вспомогательные функции. Средства времени исполнения отвечают за создание и обработку объектов, маршалинг, посредничество, распределение оперативной памяти и другие низкоуровневые операции, лежащие в основе COM+. Для поддержки многих новомодных служб, о которых кое-кто даже и не слышал, корпорация *Microsoft* добавила в средства времени исполнения COM+ несколько новых элементов, в том числе настраиваемые компоненты, базу данных регистрации, средства работы с контекстом и реализацию новой нейтральной модели потоков.

База данных регистрации (RegDB)

В COM параметры отдельного объекта, в основном, хранятся либо в системном реестре, либо в библиотеке типов, а в COM+ реализована новая концепция, которая заключается в использовании для хранения информации об объектах базы данных регистрации. Библиотеки типов для хранения параметров объектов в COM+ также используются, а работа с системным реестром была оставлена только для совместимости с прежними версиями. К общим параметрам объектов COM+, хранимым в базе данных RegDB, относятся также значение транзакционного уровня и признак поддержки оперативной (JIT) активизации.

Настраиваемые компоненты

Компоненты, параметры которых хранятся в базе данных RegDB, называются *настраиваемыми* (configured). Существуют также и не настраиваемые компоненты. Наилучшим примером не настраиваемого компонента является компонент COM или MTS, который без изменений используется и в среде COM+. Для того чтобы взаимо-

действовать с большинством рассмотренных ранее в настоящей главе служб, компоненты должны быть настраиваемыми.

Контекст

Контекст (context) — это термин, который впервые стал применяться в сервере MTS. Он используется для описания состояния текущих условий работы какого-либо компонента. Данный термин не только перешел в COM+, но и был расширен. В спецификации COM для описания контекста, связанного с каким-либо потоком или процессом, использовался термин *апартамент* (apartment). Этот термин соответствовал контексту времени исполнения данного объекта. В спецификации COM+ термин "апартамент" был заменен на "контекст", который служит для описания состояния объекта на более глубоком уровне (например может выражать состояние транзакции или активизации).

Нейтральные потоки

В спецификации COM+ была впервые реализована новая потоковая модель, известная под названием *потоковый нейтральный апартамент* (TNA — Thread Neutral Apartment). Эта модель была разработана для того, чтобы в приложениях COM+ можно было использовать высокую производительность и масштабируемость многопоточных объектов без усложнения процесса программирования. Раньше программирование задач, связанных с блокированием доступа к совместно используемым данным и ресурсам на сервере, было связано с некоторыми сложностями. Потоковая модель TNA используется в компонентах COM+, не имеющих пользовательского интерфейса. Для компонентов с графическим интерфейсом продолжают использовать обычные потоки, так как дескрипторы окон привязаны к отдельным потокам. Кроме того, в одном процессе может использоваться только один TNA.

Разработка приложений COM+

Теперь, рассмотрев отдельные возможности спецификации COM+, можно приступать к разработке приложения, в котором были бы реализованы такие свойства COM+, как транзакции, контроль за временем существования объектов и совместное использование ресурсов.

Цель — масштабируемость

В наше время основной целью системного проектирования является достижение высокой масштабируемости. В связи с быстрым распространением Internet, intranet, extranet и других видов сетей, объединением корпоративных данных в централизованные хранилища и потребностью в массовом совместном использовании этих данных, стали предъявляться жесткие требования к масштабируемости систем. Современные системы вынуждены одновременно взаимодействовать с гораздо большим количеством пользователей, чем прежде. Это действительно серьезная задача, особенно учитывая такие технические ограничения, как количество возможных одновременных подключений к базе данных, пропускная способность сети, загрузка сервера и т.п. В старые до-

брые времена (в начале 90-х годов) к системам архитектуры клиент/сервер относились как к единственному типу масштабируемых приложений. Но базы данных с увеличением количества триггеров и хранимых процедур становились все сложнее, а клиенты постоянно сталкивались со сложностями реализации своих бизнес-правил. Вскоре стало очевидно, что подобные системы никогда не смогут стать настолько масштабируемыми, чтобы обслужить действительно большое количество пользователей. Поэтому многоуровневая архитектура вскоре заняла лидирующие позиции в сфере построения масштабируемых систем. В этой архитектуре функции приложения и распределенные подключения к базе данных находятся на среднем уровне, что позволяет упростить как саму базу данных, так и клиентскую часть приложения, а кроме того, существенно оптимизировать использование ресурсов.

НА ЗАМЕТКУ

Структура многоуровневой модели построена таким образом, что увеличение пропускной способности приложения приводит к увеличению временных задержек. Другими словами, увеличение масштабируемости приводит к снижению производительности.

Контекст исполнения

Не забывайте о том, что, в отличие от объектов COM, объекты COM+ не работают непосредственно в контексте клиента. К тому же клиенты никогда не получают указатели на интерфейсы непосредственно в объектах. Вместо этого в COM+ между клиентами и объектами используются посредники. При таком подходе посредник, с точки зрения клиента, является точной копией объекта. Учитывая тот факт, что средства COM+ имеют полный контроль над посредниками, они могут управлять также и доступом к методам интерфейсов объектов. Такая возможность может быть использована для управления временем существования и защитой объектов.

Учет состояния объектов

Среди разработчиков, использующих технологию COM+, часто можно услышать дискуссии о различиях между объектами, учитывающими и не учитывающими состояние (stateful и stateless). Несмотря на то что в спецификации COM нет никаких указаний по этому поводу, в большинстве случаев объекты COM учитывают состояние. Это означает, что они постоянно, начиная с момента создания и до момента удаления, содержат какую-то информацию. Проблема, связанная с использованием таких объектов, заключается в их низкой масштабируемости. Указанное объясняется тем, что хранимая информация должна поддерживаться для каждого объекта при обращении к нему каждого клиента. Объекты, не учитывающие состояние, не хранят информацию о своем состоянии в промежутках между вызовами методов. В спецификации COM+ преимущественно используются именно такие объекты, потому что они упрощают оптимизацию приложений. Если какой-либо объект вообще не хранит информации, то теоретически он может быть дезактивирован средствами COM+ в промежутках между обращениями к его методам безо всякого вреда для работы приложения. Такая возможность появилась благодаря тому, что в COM+ клиент работает только с указателями на внутреннего посредника, а не напрямую с самим объектом. И это — не просто теория, а фактическая организация работы с объектами в COM+. Все

экземпляры объектов между вызовами методов уничтожаются для того, чтобы освободить ресурсы, выделенные под объекты. При очередном обращении клиента к объекту посредник COM+ перехватывает этот вызов и автоматически создает новый экземпляр объекта. Такой подход позволяет значительно повысить масштабируемость системы, потому что в каждый момент времени будет существовать относительно немного активных экземпляров класса.

При разработке интерфейсов для работы с объектами, не учитывающими состояние, вероятно, придется уклониться от привычных подходов к проектированию интерфейсов. В качестве примера рассмотрим типичный интерфейс в стиле COM:

```
ICheckbook = interface
['{2CCF0409-EE29-11D2-AF31-0000861EF0BB}']
    procedure SetAccount(AccountNum: WideString); safecall;
    procedure AddActivity(Amount: Integer); safecall;
end;
```

В приложении такой интерфейс можно использовать следующим образом:

```
var
  CB: ICheckbook;
begin
  CB := SomehowGetInstance;
  CB.SetAccount('12345ABCDE');  // Открыть текущий счет
  CB.AddActivity(-100);         // и снять с него 100$
  ...
end;
```

Проблема в данном случае заключается в том, что между вызовами методов объект должен хранить информацию о номере счета, а следовательно, учитывать свое состояние. В COM+ для данного интерфейса используется другой подход, который заключается в передаче *всей* необходимой информации непосредственно в метод `AddActivity()`. В этом случае объекту ненужно учитывать состояния:

```
procedure AddActivity(AccountNum: WideString;
                      Amount: Integer); safecall;
```

Текущее состояние активного объекта еще называют контекстом. Средства COM+ используют контекст каждого активного объекта для получения информации о его специфических параметрах (например о состоянии системы безопасности или о транзакции). Сам объект для получения указателя на интерфейс контекста `IObjectContext` в любой момент может вызвать метод `GetObjectContext()`. Интерфейс `IObjectContext` определен в модуле `Mtx` следующим образом:

```
IObjectContext = interface(IUnknown)
  ['{51372AE0-CAE7-11CF-BE81-00AA00A2FA25}']
  function CreateInstance(const cid, rid: TGUID; out pv): HResult;
                                                          stdcall;
  procedure SetComplete; safecall;
  procedure SetAbort; safecall;
  procedure EnableCommit; safecall;
  procedure DisableCommit; safecall;
  function IsInTransaction: Bool; stdcall;
  function IsSecurityEnabled: Bool; stdcall;
  function IsCallerInRole(const bstrRole: WideString): Bool;
```

```
                                                    safecall;
end;
```

Наиболее важными методами в этом интерфейсе являются `SetComplete()` и `SetAbort()`. При вызове любого из них объект уведомляет средства COM+ о том, что он больше не обрабатывает информацию о состоянии. В ответ на уведомление средства COM+ уничтожат данный объект (клиент об этом, конечно же, ничего знать не будет) и освободят ресурсы для других экземпляров. Если объект принимает участие в транзакции, то методы `SetComplete()` и `SetAbort()` также выполняют соответственно завершение транзакции или отказ от нее.

Контроль за временем существования объектов

В программировании COM существует правило, согласно которому указатели на интерфейсы освобождаются сразу же после того, как в них отпадает необходимость. В традиционной спецификации COM этого правила нужно было придерживаться, чтобы рационально использовать системные ресурсы. Согласно спецификации COM+ объекты, не учитывающие состояния, автоматически уничтожаются после вызова ими метода `SetComplete()` или `SetAbort()`. Удаление экземпляра объекта происходит скрытно от клиента, и клиентская часть приложения не нуждается ни в каких дополнительных модификациях.

Организация приложения COM+

Понятию "приложение" (application) в средствах Component Services соответствует набор компонентов COM+, имеющих общую конфигурацию и параметры. Раньше для определения этого же понятия в терминологии сервера MTS использовалось слово "пакет" (package), однако, к счастью, в COM+ терминология изменилась. Термин "пакет" использовался для обозначения достаточно большого количества понятий, включая пакеты Delphi, пакеты C++Builder, пакеты Oracle и пакеты с подарками на день рождения.

По умолчанию все компоненты COM+ одного и того же пакета запускаются в одном и том же процессе. Это позволяет создавать надежные, отказоустойчивые пакеты. Также важно отметить, что физическое размещение компонентов в пакетах может быть произвольным. Один сервер COM+ может содержать несколько различных объектов COM+, каждый из которых расположен в отдельном пакете.

Для создания приложений используется либо меню Run, пункт Install COM+ Objects IDE Delphi, либо средство Component Services.

Размышления о транзакциях

Одним из ключевых вопросов COM+ являются транзакции. Кто-то может подумать: мой сервер баз данных уже поддерживает транзакции, так зачем они должны поддерживаться еще и моими компонентами? Это — хороший вопрос, заслуживающий хорошего ответа. Средства COM+ позволяют открыть несколько транзакций при одновременном подключении к нескольким базам данных и даже выполнить отдельное

действие из некоторого набора операций, не имеющих непосредственного отношения к базам данных. Для поддержки транзакций объектами COM+ необходимо либо в процессе разработки установить соответствующий флажок для класса объекта в библиотеке типов (эта операция выполняется при помощи мастера Delphi Transactional Object), либо после установки объекта при помощи проводника сервера транзакций (Transaction Server Explorer).

Когда в объектах нужно использовать транзакции? Ответ на этот вопрос очень прост: транзакции используются каждый раз, когда есть процесс, состоящий из нескольких этапов, который необходимо выполнять как единый неделимый оператор. При таком подходе завершение транзакции или отказ от нее будет применяться сразу для всего процесса, а данные никогда не окажутся в некорректном или промежуточном состоянии. Например, при разработке банковской программы необходимо реализовать функцию, возвращающую клиенту чек. В этом случае процесс может состоять из следующих этапов:

- проверка указанной в чеке суммы на счету клиента;
- снятие со счета платы за банковскую операцию по возвращению чека;
- отправка уведомления клиенту.

Для того чтобы выдача чека была произведена корректно, должен быть выполнен каждый из трех этапов. Размещение их в одной транзакции даст гарантию отсутствия ошибок, так как в случае сбоя на одном из этапов произойдет отказ от транзакции, и состояние счета будет возвращено в исходное состояние.

Ресурсы

При работе со спецификацией COM+ постоянно выполняется создание и уничтожение объектов, а также используются транзакции. Поэтому большое значение имеют средства распределения ресурсов (таких как подключения к базе данных) между множеством объектов. В COM+ для этого служат диспетчеры ресурсов и распределители ресурсов. *Диспетчер ресурсов* (resource manager) представляет собой службу, управляющую некоторыми типами долговременных данных (например информацией об остатке на счету или об инвентаре). Корпорация *Microsoft* реализовала диспетчер ресурсов в сервере MS SQL Server. *Распределитель ресурсов* (resource dispenser) управляет недолговременными ресурсами (в частности, подключениями к базе данных). Корпорация *Microsoft* реализовала распределитель ресурсов в средствах ODBC, а *Borland* — в средствах BDE.

Когда в транзакции используются какие-либо ресурсы, то они становятся *частью* этой транзакции. Таким образом, все изменения, внесенные в ресурсы в течение транзакции, будут сохранены (или не сохранены) *одновременно*, при завершении (или отмене) транзакции.

COM+ в Delphi

Теперь пришло время поговорить о реализации технологии COM+ в Delphi. Следует отметить, что средства COM+ поддерживаются только в версии Delphi Client/Server. Компоненты COM+ технически возможно применять и в версиях Standard и

Professional, однако они не позволяют использовать все возможности разработки приложений COM+ в Delphi.

Мастера COM+

Для разработки компонентов COM+ в Delphi используется мастер Transactional Data Module Wizard (мастер модулей транзакционных данных), доступ к которому можно получить во вкладке Multitier диалогового окна New Items, а также мастер Transactional Object Wizard (мастер транзакционных объектов), запускаемый со вкладки ActiveX. Мастер Transactional Data Module Wizard позволяет создавать серверы MIDAS, работающие в среде COM+. Мастер Transactional Object Wizard служит в качестве отправной точки для разработки транзакционных объектов COM+, и именно о нем дальше и пойдет речь. При вызове такого мастера на экране появится диалоговое окно, представленное на рис. 18.15.

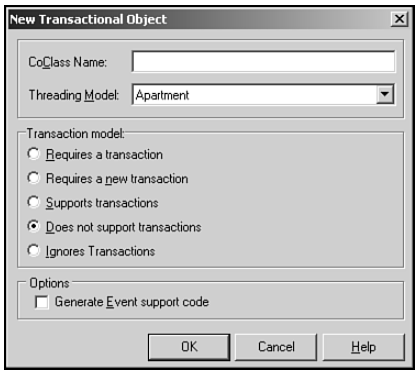

Рис. 18.15. *Мастер COM+ Transactional Object Wizard*

Это диалоговое окно аналогично окну мастера объектов автоматизации, который использовался при разработке приложений Delphi с применением средств COM. По сравнению с мастером объектов автоматизации, в мастере Transactional Object Wizard появилась возможность выбора транзакционной модели, поддерживаемой компонентом COM+. Существуют следующие транзакционные модели:

- Requires a Transaction (транзакция необходима) — компонент всегда создается внутри контекста какой-либо транзакции. При этом он наследует транзакцию своего владельца (если такой существует) или создает новую транзакцию.
- Requires a New Transaction (необходима новая транзакция) — для компонента всегда создается новая транзакция.
- Supports Transactions (транзакция поддерживается) — компонент наследует транзакцию своего владельца. Если владельца не существует, то транзакция вообще не создается.
- Does Not Support Transactions (транзакция не поддерживается) — для компонента транзакция не создается.
- Ignores Transactions (транзакция игнорируются) — компонент работает независимо от контекста транзакции.

Информация о транзакционной модели хранится вместе с классом компонента в библиотеке типов.

После щелчка на кнопке OK мастер создаст пустое определение класса, производного от класса `TMtsAutoObject`. Это определение будет открыто в редакторе Type Library Editor, где в него можно добавить свойства, методы и интерфейсы компонентов COM+. Процесс разработки на данном этапе идентичен процессу разработки объектов автоматизации в Delphi. Несмотря на то что объекты COM+, созданные при помощи мастера Delphi, являются объектами автоматизации (т.е. объектами COM, в которых реализован интерфейс `IDispatch`), это не является техническим требованием COM+. Средства COM также предполагали работу с интерфейсами `IDispatch`, входящими в состав библиотек типов, однако при использовании объектов данного вида в COM+ можно уделить больше внимания функциям компонентов, не отвлекаясь на детали их взаимодействия со средствами COM+. Кроме того, компоненты COM+ должны находиться на внутреннем сервере COM (DLL), так как внешние серверы (EXE) компонентами COM+ не поддерживаются.

Структура COM+

Вышеупомянутый класс `TMtsAutoObject` является базовым для всех объектов COM+, создаваемых при помощи мастера Delphi. Он относительно прост и определен в модуле `MtsObj` следующим образом:

```
type
  TMtsAutoObject = class(TAutoObject, IObjectControl)
  private
    FObjectContext: IObjectContext;
  protected
    { IObjectControl }
    procedure Activate; safecall;
    procedure Deactivate; stdcall;
    function CanBePooled: Bool; stdcall;

    procedure OnActivate; virtual;
    procedure OnDeactivate; virtual;
    property ObjectContext: IObjectContext read FObjectContext;
  public
    procedure SetComplete;
    procedure SetAbort;
    procedure EnableCommit;
    procedure DisableCommit;
    function IsInTransaction: Bool;
    function IsSecurityEnabled: Bool;
    function IsCallerInRole(const Role: WideString): Bool;
  end;
```

Класс `TMtsAutoObject` — это, по существу, класс `TAutoObject`, к которому добавлены возможности по управлению инициализацией и очисткой контекста.

В нем реализован интерфейс `IObjectControl`, отвечающий за инициализацию и очистку компонентов COM+. Он обладает следующими методами:

- `Activate()` — позволяет объекту выполнять инициализацию контекста при активизации. Этот метод вызывается средствами COM+ прежде любых других пользовательских методов компонента.
- `Deactivate()` — позволяет выполнять очистку контекста при деактивации объекта.
- `CanBePooled()` — в MTS не использовался, однако поддерживается в COM+. Описан в настоящей главе ранее.

Класс `TMtsAutoObject` содержит виртуальные методы `OnActivate()` и `OnDeactivate()`, которые происходят от закрытых методов `Activate()` и `Deactivate()`, используемых для активизации и деактивации контекста.

Кроме того, класс `TMtsAutoObject` использует указатель на интерфейс COM+ `IObjectContext` в виде свойства `ObjectContext`. Класс `TMtsAutoObject` позволяет пользователям создавать реализации любых методов интерфейса `IObjectContext`. Например, в реализации метода `TMtsAutoObject.SetComplete()` просто сравнивается значение поля `FObjectContext` со значением `nil`, а затем вызывается метод `FObjectContext.SetComplete()`.

Ниже перечислены все методы интерфейса `IObjectContext` с кратким описанием каждого из них:

- `CreateInstance()` — создает экземпляр еще одного объекта COM+. Этот метод выполняет ту же операцию, что и метод `IClassFactory.CreateInstance()` при использовании спецификации COM.
- `SetComplete()` — извещает средства COM+ о том, что компонент завершил работу и больше не будет поддерживать информацию о внутреннем состоянии. Если компонент является транзакционным, то данный метод указывает также на завершение транзакции. После его вызова средства COM+ могут дезактивировать объект, а затем освободить ресурсы для повышения масштабируемости.
- `SetAbort()` — аналогичен методу `SetComplete()`. Извещает средства COM+ о том, что компонент завершил работу и больше не будет поддерживать информацию о состоянии. Данный метод вызывается в случае возникновения ошибки, поэтому все незавершенные транзакции должны быть прерваны.
- `EnableCommit()` — указывает на то, что компонент находится в состоянии, когда транзакция может быть закрыта вызовом метода `SetComplete()`. Это состояние компонента устанавливается по умолчанию.
- `DisableCommit()` — указывает на то, что компонент находится в некорректном состоянии и для завершения транзакции требуется вызов дополнительных методов.
- `IsInTransaction()` — позволяет компоненту определить, находится ли он внутри контекста транзакции.
- `IsSecurityEnabled()` — позволяет компоненту определить, доступна ли для него система безопасности COM+. Этот метод всегда возвращает значение `True`, кроме тех случаев, когда компонент работает в пространстве клиентского процесса.

- `IsCallerInRole()` — позволяет компоненту определить, принадлежит ли его клиент заданной роли COM+. Данный метод является ключевым в ролевой системе безопасности COM+. Более подробно вопрос ролей будет рассмотрен несколько позднее.

Ядро средств поддержки COM+ определено в модуле `Mtx`. Этот модуль является вариантом файла заголовка `mtx.h` для языка Pascal. Он содержит типы (например `IObjectControl` и `IObjectContext`) и функции, которые используются при работе с API COM+.

Простое приложение Tic-Tac-Toe

Но достаточно теории! Настало время применить полученные знания о COM+ на практике. Средства COM+ поставляются вместе с простым приложением `tic-tac-toe` (крестики-нолики), которое вдохновило автора реализовать эту классическую игру при помощи Delphi. Запустите мастер Transactional Object Wizard и создайте новый объект под названием `GameServer`. При помощи редактора Type Library Editor добавьте в интерфейс `IGameServer`, созданный для нового объекта по умолчанию, три метода: `NewGame()`, `ComputerMove()`, и `PlayerMove()`. Добавьте также два новых перечислимых свойства: `SkillLevels` и `GameResults` — которые будут использоваться вышеуказанными методами. Все эти элементы отображены в окне редактора Type Library Editor, показанном на рис. 18.16.

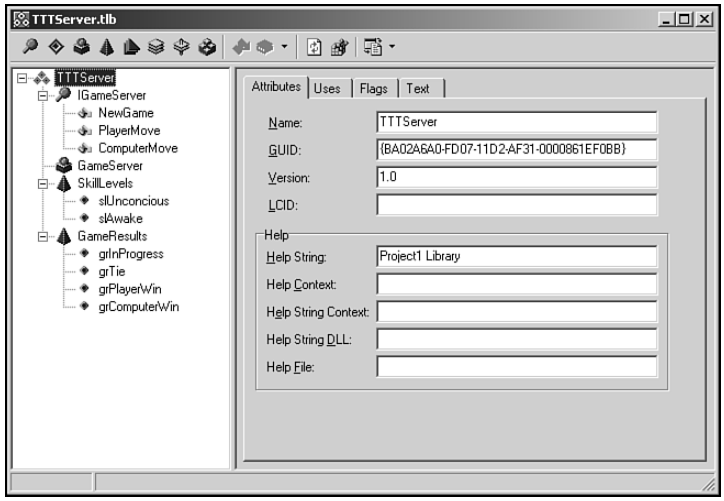

Рис. 18.16. *Сервер `Tic-Tac-Toe` в окне редактора Type Library Editor*

Три новых метода данного интерфейса очень просты и служат для реализации компьютерной версии игры в крестики-нолики. Метод `NewGame` служит для инициализации новой игры со стороны клиента. Метод `ComputerMove` анализирует возможные ходы и выполняет ход со стороны компьютера. Метод `PlayerMove` позволяет сделать ход клиенту. Ранее уже упоминалось о том, что подход к разработке компо-

нентов COM+ отличается от подхода к разработке стандартных компонентов COM. Хорошей иллюстрацией этому является приложение `tic-tac-toe`.

Если бы это был заурядный компонент COM, то для учета состояния игры в методе `NewGame()` понадобилось бы просто инициализировать соответствующую структуру данных. Такой структуре могло бы соответствовать поле экземпляра объекта, к которому могли бы иметь доступ другие методы.

Почему же такой подход нельзя использовать при разработке компонента COM+? Все дело в учете состояния. Как уже упоминалось ранее, для использования всех возможностей COM+ объект не должен хранить информацию о своем состоянии. Но структура компонента ориентирована на обработку данных экземпляра между вызовами методов, что соответствует учету состояния. Поэтому при разработке компонента COM+ лучше всего из метода `NewGame()` возвращать идентификатор игры и использовать его для обработки структур данных при помощи специальных средств распределения ресурсов. Такие средства должны работать вне контекста экземпляра объекта, потому что средства COM+ могут активизировать и дезактивировать экземпляры объекта при каждом вызове метода. Все остальные методы компонента могут принимать идентификатор игры в качестве параметра и извлекать данные из совместно используемого ресурса. При таком подходе состояние не учитывается, так как объект может не быть активным между вызовами метода, а все остальные методы принимают необходимые данные через параметры и средства распределения ресурсов.

Средства доступа к совместно используемым данным в COM+ называются *распределителями ресурсов* (resource dispenser). В частности, распределитель ресурсов Shared Property Manager используется для обработки общих данных компонента. Доступ к этому распределителю ресурсов организован через интерфейс `ISharedPropertyGroupManager`. Распределитель ресурсов Shared Property Manager находится на верхнем уровне иерархии системы запоминающих устройств и поддерживает любое количество групп совместно используемых свойств через интерфейс `ISharedPropertyGroup`. В свою очередь, каждая такая группа может содержать любое количество совместно используемых свойств, доступ к которым организован через интерфейс `ISharedProperty`. Совместно используемые свойства удобны тем, что они используются в спецификации COM+ вне контекста экземпляров любых объектов, а управление доступом к ним осуществляется через блокировки и семафоры распределителя ресурсов Shared Property Manager.

С учетом всего вышесказанного реализация метода `NewGame()` будет иметь следующий вид:

```
procedure TGameServer.NewGame(out GameID: Integer);
var
  SPG: ISharedPropertyGroup;
  SProp: ISharedProperty;
  Exists: WordBool;
  GameData: OleVariant;
begin
  // При проверке безопасности использовать роль пользователя
  CheckCallerSecurity;
  // Получить указатель на группу совместно используемых свойств
  SPG := GetSharedPropertyGroup;
  // Создать или получить совместно используемое
  // свойство NextGameID
```

```
  SProp := SPG.CreateProperty('NextGameID', Exists);
  if Exists then GameID := SProp.Value
  else GameID := 0;
  // Увеличить и сохранить значение свойства NextGameID
  SProp.Value := GameID + 1;
  // Создать массив данных игры
  GameData := VarArrayCreate([1, 3, 1, 3], varByte);
  SProp := SPG.CreateProperty(Format(GameDataStr,
                              [GameID]), Exists);
  SProp.Value := GameData;
  SetComplete;
end;
```

В данном методе сначала проверяется роль пользователя, а затем для получения идентификатора игры применяется совместно используемое свойство. После этого создается вариантный массив для хранения данных игры, а затем эти данные сохраняются в совместно используемом свойстве. В завершение вызывается метод `SetComplete()`, извещающий средства COM+ о возможности деактивации данного экземпляра.

Исходя из этого, можно сформулировать главное правило разработки компонентов COM+: вызывайте метод `SetComplete()` или `SetAbort()` настолько часто, насколько это возможно. В идеале один из таких методов должен вызываться во всех методах компонента, так как в этом случае средства COM+ смогут восстанавливать ресурсы, потребляемые экземпляром компонента, после выхода из каждого метода. Другим словами, в процессе активизации и деактивации объекта не должно потребляться много ресурсов.

Реализация метода `CheckCallerSecurity()` иллюстрирует преимущества ролевой системы безопасности COM+:

```
procedure TGameServer.CheckCallerSecurity;
begin
  // Игра только для членов роли "TTT".
  if IsSecurityEnabled and not IsCallerInRole('TTT') then
    raise Exception.Create('Only TTT role can play tic-tac-toe');
end;
```

При изучении данного фрагмента возникает вопрос: как установить роль `TTT` и определить ее членов? Роли можно определять и программно, однако лучше для этого использовать средство проводника сервера транзакций. После установки компонента можно установить роли при помощи узла Roles, расположенного под узлом каждого пакета в окне Transaction Server Explorer. Важно отметить, что ролевая система безопасности поддерживается только компонентами, которые используются в системах Windows NT. Для компонентов, которые используются в системах Windows 9x/Me, метод `IsCallerInRole()` всегда возвращает значение `True`.

Ниже представлена реализация методов `ComputerMove()` и `PlayerMove()`:

```
procedure TGameServer.ComputerMove(GameID: Integer;
                                   SkillLevel: SkillLevels;
                                   out X, Y: Integer;
                                   out GameRez: GameResults);
var
```

```
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Получить совместно используемое свойство,
  // содержащее данные игры
  SProp := GetSharedPropertyGroup.CreateProperty(Format
↳(GameDataStr, [GameID]), Exists);
  // Получить массив данных игры и блокировки для него,
  // обеспечивающие более эффективный доступ.
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Если игра не окончена, предоставить право хода компьютеру
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then begin
      CalcComputerMove(GameData, SkillLevel, X, Y);
      // Временно сохранить новый массив данных игры
      SProp.Value := PropVal;
      // Проверить окончание игры
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;

procedure TGameServer.PlayerMove(GameID, X, Y: Integer;
                                 out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Получить совместно используемое свойство,
  // содержащее данные игры
  SProp := GetSharedPropertyGroup.CreateProperty(
↳Format(GameDataStr, [GameID]), Exists);
  // Получить массив данных игры и блокировки для него,
  // обеспечивающие более эффективный доступ.
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Проверить окончание игры
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then begin
      // Если клетка не пуста, то передать исключение
      if GameData[X, Y] <> EmptySpot then
        raise Exception.Create('Spot is occupied!');
      // Сделать ход
```

```
      GameData[X, Y] := PlayerSpot;
      // Временно сохранить новый массив данных игры
      SProp.Value := PropVal;
      // Проверить окончание игры
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;
```

Эти методы похожи тем, что оба они принимают данные игры из совместно используемого свойства через параметр `GameID`, обрабатывают эти данные для отображения текущего хода, сохраняют обновленные данные и проверяют окончание игры. Метод `ComputerMove()` также вызывает метод `CalcComputerMove()`, анализирующий состояние данных игры, и выполняет следующий ход. Весь исходный код этого компонента COM+, содержащийся в модуле `ServMain`, представлен в листинге 18.6.

Листинг 18.6. Модуль `ServMain.pas`, содержащий класс `TGameServer`

```
unit ServMain;

interface

uses
  ActiveX, MtsObj, Mtx, ComObj, TTTServer_TLB, StdVcl;

type
  PGameData = ^TGameData;
  TGameData = array[1..3, 1..3] of Byte;

  TGameServer = class(TMtsAutoObject, IGameServer)
  private
    procedure CalcComputerMove(GameData: PGameData;
                               Skill: SkillLevels;
                               var X, Y: Integer);
    function CalcGameStatus(GameData: PGameData): GameResults;
    function GetSharedPropertyGroup: ISharedPropertyGroup;
    procedure CheckCallerSecurity;
  protected
    procedure NewGame(out GameID: Integer); safecall;
    procedure ComputerMove(GameID: Integer;
                           SkillLevel: SkillLevels;
                           out X, Y: Integer;
                           out GameRez: GameResults); safecall;
    procedure PlayerMove(GameID, X, Y: Integer;
                         out GameRez: GameResults); safecall;
  end;

implementation
```

```
uses ComServ, Windows, SysUtils, Variants;

const
  GameDataStr = 'TTTGameData%d';
  EmptySpot = 0;
  PlayerSpot = $1;
  ComputerSpot = $2;

function TGameServer.GetSharedPropertyGroup: ISharedPropertyGroup;
var
  SPGMgr: ISharedPropertyGroupManager;
  LockMode, RelMode: Integer;
  Exists: WordBool;
begin
  if ObjectContext = nil then
    raise Exception.Create('Failed to obtain object context');
  // Создать для объекта группу совместно используемых свойств
  OleCheck(ObjectContext.CreateInstance(CLASS_SharedProp
↳ertyGroupManager, ISharedPropertyGroupManager, SPGMgr));
  LockMode := LockSetGet;
  RelMode := Process;
  Result := SPGMgr.CreatePropertyGroup('DelphiTTT', LockMode,
                                       RelMode, Exists);
  if Result = nil then
    raise Exception.Create('Failed to obtain property group');
end;

procedure TGameServer.NewGame(out GameID: Integer);
var
  SPG: ISharedPropertyGroup;
  SProp: ISharedProperty;
  Exists: WordBool;
  GameData: OleVariant;
begin
  // При проверке безопасности использовать роль пользователя
  CheckCallerSecurity;
  // Получить указатель на группу совместно используемых свойств
  SPG := GetSharedPropertyGroup;
  // Создать или получить совместно используемое
  // свойство NextGameID
  SProp := SPG.CreateProperty('NextGameID', Exists);
  if Exists then GameID := SProp.Value
  else GameID := 0;
  // Увеличить и сохранить значение свойства NextGameID
  SProp.Value := GameID + 1;
  // Создать массив данных игры
  GameData := VarArrayCreate([1, 3, 1, 3], varByte);
  SProp := SPG.CreateProperty(Format(GameDataStr, [GameID]),
                              Exists);
  SProp.Value := GameData;
  SetComplete;
end;
```

```
procedure TGameServer.ComputerMove(GameID: Integer;
                                   SkillLevel: SkillLevels;
                                   out X, Y: Integer;
                                   out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Получить совместно используемое свойство,
  // содержащее данные игры
  SProp := GetSharedPropertyGroup.CreateProperty(
↳Format(GameDataStr, [GameID]), Exists);
  // Получить массив данных игры и блокировки для него,
  // обеспечивающие более эффективный доступ.
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Если игра не окончена, предоставить право хода компьютеру
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then begin
      CalcComputerMove(GameData, SkillLevel, X, Y);
      // Временно сохранить новый массив данных игры
      SProp.Value := PropVal;
      // Проверить окончание игры
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;

procedure TGameServer.PlayerMove(GameID, X, Y: Integer;
                                 out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Получить совместно используемое свойство,
  // содержащее данные игры
  SProp := GetSharedPropertyGroup.CreateProperty(
↳Format(GameDataStr, [GameID]), Exists);
  // Получить массив данных игры и блокировки для него,
  // обеспечивающие более эффективный доступ.
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Проверить окончание игры
    GameRez := CalcGameStatus(GameData);
```

```
    if GameRez = grInProgress then begin
      // Если клетка не пуста, то передать исключение
      if GameData[X, Y] <> EmptySpot then
        raise Exception.Create('Spot is occupied!');
      // Сделать ход
      GameData[X, Y] := PlayerSpot;
      // Временно сохранить новый массив данных игры
      SProp.Value := PropVal;
      // Проверить окончание игры
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;

function TGameServer.CalcGameStatus(GameData:
                                    PGameData): GameResults;
var
  I, J: Integer;
begin
  // Проверить, нет ли победителя
  if GameData[1, 1] <> EmptySpot then begin
    // Проверить верхнюю строку, левый столбец и диагональ из
    // верхнего левого в правый нижний угол
    if ((GameData[1, 1] = GameData[1, 2]) and
        (GameData[1, 1] = GameData[1, 3])) or
       ((GameData[1, 1] = GameData[2, 1]) and
        (GameData[1, 1] = GameData[3, 1])) or
       ((GameData[1, 1] = GameData[2, 2]) and
        (GameData[1, 1] = GameData[3, 3])) then
    begin
      Result := GameData[1, 1] + 1; // Результат - идентификатор
      Exit;                         // клетки + 1
    end;
  end;
  if GameData[3, 3] <> EmptySpot then begin
    // Проверить нижнюю строку и правый столбец
    if ((GameData[3, 3] = GameData[3, 2]) and
        (GameData[3, 3] = GameData[3, 1])) or
       ((GameData[3, 3] = GameData[2, 3]) and
        (GameData[3, 3] = GameData[1, 3])) then
    begin
      Result := GameData[3, 3] + 1; // Результат - идентификатор
      Exit;                         // клетки + 1
    end;
  end;
  if GameData[2, 2] <> EmptySpot then begin
    // Проверить среднюю строку, средний столбец и диагональ из
    // нижнего левого в верхний правый угол
    if ((GameData[2, 2] = GameData[2, 1]) and
        (GameData[2, 2] = GameData[2, 3])) or
```

```
      ((GameData[2, 2] = GameData[1, 2]) and
       (GameData[2, 2] = GameData[3, 2])) or
      ((GameData[2, 2] = GameData[3, 1]) and
       (GameData[2, 2] = GameData[1, 3])) then
    begin
      Result := GameData[2, 2] + 1; // Результат - идентификатор
      Exit;                         // клетки + 1
    end;
  end;
  // Наконец, определяем необходимость продолжения игры
  for I := 1 to 3 do
    for J := 1 to 3 do
      if GameData[I, J] = 0 then begin
        Result := grInProgress;
        Exit;
      end;
  // Если игрок является членом роли TTT, то он
  // автоматически побеждает.
  if IsCallerInRole('TTT') then begin
    Result := grPlayerWin;
    Exit;
  end;
  // В противном случае - ничья
  Result := grTie;
end;

procedure TGameServer.CalcComputerMove(GameData: PGameData;
                                       Skill: SkillLevels;
                                       var X, Y: Integer);
type
  // Используется для поиска возможных ходов по вертикали,
  // горизонтали или диагонали
  TCalcType = (ctRow, ctColumn, ctDiagonal);
  // mtWin = один ход до победы, mtBlock = оппоненту остается один
  // ход до победы, mtOne = я заполнил еще одну клетку на этой
  // линии, mtNew = Я не заполнил ни одной клетки на этой линии.
  TMoveType = (mtWin, mtBlock, mtOne, mtNew);
var
  CurrentMoveType: TMoveType;

  function DoCalcMove(CalcType: TCalcType;
                      Position: Integer): Boolean;
  var
    RowData, I, J, CheckTotal: Integer;
    PosVal, Mask: Byte;
  begin
    Result := False;
    RowData := 0;
    X := 0;
    Y := 0;
    if CalcType = ctRow then begin
      I := Position;
      J := 1;
```

```
end
else if CalcType = ctColumn then begin
  I := 1;
  J := Position;
end
else begin
  I := 1;
  case Position of
    1: J := 1; // просмотр из верхнего левого в нижний
               // правый угол
    2: J := 3; // просмотр из верхнего правого в нижний
               // левый угол
  else
    Exit;   // проверяются только две диагонали
  end;
end;
// Mask содержит биты для игрока или компьютера в зависимости
// от тактики нападения или защиты. Checktotal определяет
// строку для хода.
case CurrentMoveType of
  mtWin: begin
      Mask := PlayerSpot;
      CheckTotal := 4;
    end;
  mtNew: begin
      Mask := PlayerSpot;
      CheckTotal := 0;
    end;
  mtBlock: begin
      Mask := ComputerSpot;
      CheckTotal := 2;
    end;
else
  begin
    Mask := 0;
    CheckTotal := 2;
  end;
end;
// просмотр всех линий в текущем типе CalcType
repeat
  // Получить статус текущей клетки (X, O или пустая)
  PosVal := GameData[I, J];
  // Запомнить последнюю пустую клетку в том случае,
  // если решено заполнить ее
  if PosVal = 0 then begin
    X := I;
    Y := J;
  end else
    // Если клетка уже заполнена, добавить маскирующее
    // значение в RowData
    Inc(RowData, (PosVal and not Mask));
  if (CalcType = ctDiagonal) and (Position = 2) then begin
    Inc(I);
```

```
        Dec(J);
      end else begin
        if CalcType in [ctRow, ctDiagonal] then Inc(J);
        if CalcType in [ctColumn, ctDiagonal] then Inc(I);
      end;
    until (I > 3) or (J > 3);
    // При увеличении RowData необходимо защищаться или наступать,
    // в зависимости от выбранной тактики.
    Result := (X <> 0) and (RowData = CheckTotal);
    if Result then begin
      GameData[X, Y] := ComputerSpot;
      Exit;
    end;
  end;

var
  A, B, C: Integer;
begin
  if Skill = slAwake then begin
    // В первом случае наступать, во втором - защищаться
    for A := Ord(mtWin) to Ord(mtBlock) do begin
      CurrentMoveType := TMoveType(A);
      for B := Ord(ctRow) to Ord(ctDiagonal) do
        for C := 1 to 3 do
          if DoCalcMove(TCalcType(B), C) then Exit;
    end;
    // Попытаться занять центральную клетку
    if GameData[2, 2] = 0 then begin
      GameData[2, 2] := ComputerSpot;
      X := 2;
      Y := 2;
      Exit;
    end;
    // В противном случае занять самую выгодную позицию на линии
    for A := Ord(mtOne) to Ord(mtNew) do begin
      CurrentMoveType := TMoveType(A);
      for B := Ord(ctRow) to Ord(ctDiagonal) do
        for C := 1 to 3 do
          if DoCalcMove(TCalcType(B), C) then Exit;
    end;
    // либо просто найти свободную клетку
    for A := 1 to 3 do
      for B := 1 to 3 do
        if GameData[A, B] = 0 then begin
          GameData[A, B] := ComputerSpot;
          X := A;
          Y := B;
          Exit;
        end;
  end
  else begin
    // выбирать свободную клетку случайным образом
    repeat
```

```
      A := Random(3) + 1;
      B := Random(3) + 1;
    until GameData[A, B] = 0;
    GameData[A, B] := ComputerSpot;
    X := A;
    Y := B;
  end;
end;

procedure TGameServer.CheckCallerSecurity;
begin
  // Игра только для членов роли "TTT".
  if IsSecurityEnabled and not IsCallerInRole('TTT') then
    raise Exception.Create('Only TTT role can play tic-tac-toe');
end;

initialization
  Randomize;  // установить генератор случайных чисел
  TAutoObjectFactory.Create(ComServer, TGameServer,
                            Class_GameServer, ciMultiInstance,
                            tmApartment);
end.
```

Установка сервера

После создания сервера его нужно установить в среде COM+. В Delphi это можно сделать очень просто, выбрав в меню Run пункт Install COM+ Objects. В результате на экране появится диалоговое окно Install COM+ Objects, при помощи которого объекты могут быть установлены в уже существующий или в новый пакет (рис. 18.17).

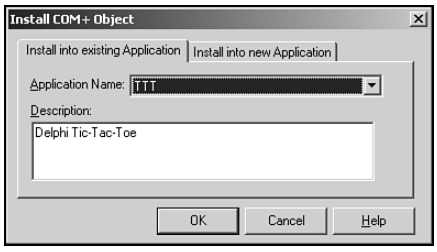

Рис. 18.17. *Установка объекта COM+ при помощи интегрированной среды разработки Delphi*

Выберите компоненты, которые необходимо установить, укажите пакет и щелкните на кнопке OK. Компоненты COM+ можно также установить при помощи проводника сервера транзакций. Процедура установки объектов COM+ значительно отличается от процедуры установки стандартных объектов COM, так как для регистрации сервера COM обычно используется программа командной строки RegSvr32. При помощи проводника сервера транзакций можно также с легкостью установить компоненты COM+ на удаленных компьютерах, что позволяет избежать множества проблем, связанных с настройкой подключений DCOM.

Клиентское приложение

В листинге 18.7 представлен исходный код клиентского приложения, предназначенного для взаимодействия с компонентом COM+ через пользовательский интерфейс.

Листинг 18.7. `UiMain.pas` — главный модуль клиентского приложения

```
unit UiMain;
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Buttons, ExtCtrls, Menus, TTTServer_TLB, ComCtrls;

type
  TRecord = record
    Wins, Loses, Ties: Integer;
  end;

  TFrmMain = class(TForm)
    SbTL: TSpeedButton;
    SbTM: TSpeedButton;
    SbTR: TSpeedButton;
    SbMM: TSpeedButton;
    SbBL: TSpeedButton;
    SbBR: TSpeedButton;
    SbMR: TSpeedButton;
    SbBM: TSpeedButton;
    SbML: TSpeedButton;
    Bevel1: TBevel;
    Bevel2: TBevel;
    Bevel3: TBevel;
    Bevel4: TBevel;
    MainMenu1: TMainMenu;
    FileItem: TMenuItem;
    HelpItem: TMenuItem;
    ExitItem: TMenuItem;
    AboutItem: TMenuItem;
    SkillItem: TMenuItem;
    UnconItem: TMenuItem;
    AwakeItem: TMenuItem;
    NewGameItem: TMenuItem;
    N1: TMenuItem;
    StatusBar: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure ExitItemClick(Sender: TObject);
    procedure SkillItemClick(Sender: TObject);
    procedure AboutItemClick(Sender: TObject);
    procedure SBClick(Sender: TObject);
    procedure NewGameItemClick(Sender: TObject);
  private
    FXImage: TBitmap;
```

```
    FOImage: TBitmap;
    FCurrentSkill: Integer;
    FGameID: Integer;
    FGameServer: IGameServer;
    FRec: TRecord;
    procedure TagToCoord(ATag: Integer; var Coords: TPoint);
    function CoordToCtl(const Coords: TPoint): TSpeedButton;
    procedure DoGameResult(GameRez: GameResults);
  end;

var
  FrmMain: TFrmMain;

implementation

uses UiAbout;

{$R *.DFM}

{$R xo.res}

const
  RecStr = 'Wins: %d, Loses: %d, Ties: %d';

procedure TFrmMain.FormCreate(Sender: TObject);
begin
  // Загрузить изображения "X" и "O" из файла ресурса TBitmaps
  FXImage := TBitmap.Create;
  FXImage.LoadFromResourceName(MainInstance, 'x_img');
  FOImage := TBitmap.Create;
  FOImage.LoadFromResourceName(MainInstance, 'o_img');
  // установить сложность игры по умолчанию
  FCurrentSkill := slAwake;
  // инициализация записи в графическом интерфейсе
  with FRec do
   StatusBar.SimpleText := Format(RecStr, [Wins, Loses, Ties]);
  // Получить экземпляр сервера
  FGameServer := CoGameServer.Create;
  // Начать новую игру
  FGameServer.NewGame(FGameID);
end;

procedure TFrmMain.ExitItemClick(Sender: TObject);
begin
  Close;
end;

procedure TFrmMain.SkillItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do begin
    Checked := True;
    FCurrentSkill := Tag;
  end;
```

```
end;

procedure TFrmMain.AboutItemClick(Sender: TObject);
begin
  // Показать окно "О программе"
  with TFrmAbout.Create(Application) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TFrmMain.TagToCoord(ATag: Integer; var Coords: TPoint);
begin
  case ATag of
    0: Coords := Point(1, 1);
    1: Coords := Point(1, 2);
    2: Coords := Point(1, 3);
    3: Coords := Point(2, 1);
    4: Coords := Point(2, 2);
    5: Coords := Point(2, 3);
    6: Coords := Point(3, 1);
    7: Coords := Point(3, 2);
  else
    Coords := Point(3, 3);
  end;
end;

function TFrmMain.CoordToCtl(const Coords: TPoint): TSpeedButton;
begin
  Result := nil;
  with Coords do
    case X of
      1:
        case Y of
          1: Result := SbTL;
          2: Result := SbTM;
          3: Result := SbTR;
        end;
      2:
        case Y of
          1: Result := SbML;
          2: Result := SbMM;
          3: Result := SbMR;
        end;
      3:
        case Y of
          1: Result := SbBL;
          2: Result := SbBM;
          3: Result := SbBR;
        end;
    end;
```

```
end;

procedure TFrmMain.SBClick(Sender: TObject);
var
  Coords: TPoint;
  GameRez: GameResults;
  SB: TSpeedButton;
begin
  if Sender is TSpeedButton then begin
    SB := TSpeedButton(Sender);
    if SB.Glyph.Empty then begin
      with SB do begin
        TagToCoord(Tag, Coords);
        FGameServer.PlayerMove(FGameID, Coords.X,
                               Coords.Y, GameRez);
        Glyph.Assign(FXImage);
      end;
      if GameRez = grInProgress then begin
        FGameServer.ComputerMove(FGameID, FCurrentSkill, Coords.X,
                                 Coords.Y, GameRez);
        CoordToCtl(Coords).Glyph.Assign(FOImage);
      end;
      DoGameResult(GameRez);
    end;
  end;
end;

procedure TFrmMain.NewGameItemClick(Sender: TObject);
var
  I: Integer;
begin
  FGameServer.NewGame(FGameID);
  for I := 0 to ControlCount - 1 do
    if Controls[I] is TSpeedButton then
      TSpeedButton(Controls[I]).Glyph := nil;
end;

procedure TFrmMain.DoGameResult(GameRez: GameResults);
const
  EndMsg: array[grTie..grComputerWin] of string = (
    'Tie game', 'You win', 'Computer wins');
begin
  if GameRez <> grInProgress then begin
    case GameRez of
      grComputerWin: Inc(FRec.Loses);
      grPlayerWin: Inc(FRec.Wins);
      grTie: Inc(FRec.Ties);
    end;
    with FRec do
     StatusBar.SimpleText := Format(RecStr, [Wins, Loses, Ties]);
    if MessageDlg(Format('%s! Play again?', [EndMsg[GameRez]]),
                  mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      NewGameItemClick(nil);
```

```
  end;
end;

end.
```

На рис. 18.18 показан внешний вид этого приложения в действии. Человек ставит крестики, а компьютер — нолики.

Рис. 18.18. *Игра в крестики-нолики*

Отладка приложений COM+

Компоненты COM+ работают внутри пространства процесса COM+, а не клиента, поэтому на первый взгляд может показаться, что отладка приложения COM+ должна быть намного сложнее отладки обычных приложений. Однако COM+ содержат специальное средство, упрощающее процесс отладки. Загрузите проект сервера, вызовите диалоговое окно Run Parameters (Параметры запуска) (рис. 18.19) и укажите в нем в качестве главного приложения (host application) `mtx.exe`. В качестве параметра программе `mtx.exe` необходимо передать `/p:{`*`package guid`*`}`, где *`package guid`* — это глобальный уникальный идентификатор пакета, указанного в средстве Component Services. Теперь расставьте точки останова и запустите приложение. Вначале ничего не произойдет, так как клиентское приложение еще не запущено. После запуска клиентского приложения произойдет переход в режим отладки.

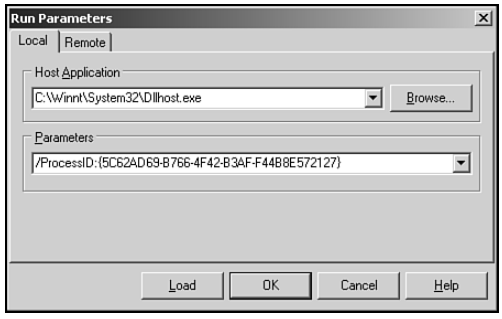

Рис. 18.19. *Диалоговое окно* Run Parameters

Резюме

COM+ — это мощнейшее дополнение к семейству технологий COM. В данную спецификацию добавлены службы контроля за временем существования объектов, поддержки транзакций и системы безопасности. Кроме того, объекты COM+ можно преобразовывать в объекты COM без значительных изменений исходного кода. Таким образом, корпорацией *Microsoft* была разработана усовершенствованная спецификация для разработки распределенных приложений с высокой степенью масштабируемости. В настоящей главе были рассмотрены основные понятия спецификации COM+ и особенности поддержки COM+ в Delphi. На основании представленного примера приложения COM+ были изучены некоторые приемы разработки оптимизированных компонентов COM+. Средства COM+ в сочетании с Delphi предоставляют большие возможности создания масштабируемых многоуровневых приложений. При этом не следует забывать о некоторых различиях между разработкой обычных компонентов COM и компонентов COM+.

ГЛАВА

19

Разработка приложений CORBA

В ЭТОЙ ГЛАВЕ...

Аббревиатура CORBA означает Common Object Request Broker Architecture (*универсальная архитектура брокера объектных запросов*). CORBA — это спецификация, разработанная группой поддержки объектной технологии (Object Management Group, или OMG), которая с помощью разрабатываемых ею стандартов определяет архитектуру решений, принимаемых в качестве основы при реализации объектов, независимо от используемого языка программирования и вычислительной платформы. В отличие от некоторых конкурирующих стандартов (например таких, как технологии COM/DCOM корпорации *Microsoft*), группа OMG не предлагает никаких реализаций определяемых ею стандартов.

Группой OMG четко определено только назначение CORBA и, в определенной степени, подходы, используемые в этой спецификации. В то же время производители программных продуктов могут использовать собственные разработки и методы, которые согласуются с CORBA. Тем не менее у такой свободы есть свои недостатки. Например, группой OMG не определено, как различные реализации CORBA обнаруживают объекты при работе с двумя различными *брокерами объектных запросов* (ORB — Object Request Broker). По этой причине бывает очень непросто согласовать приложения, разработанные различными производителями. Вопрос согласования является одним из ключевых в использовании спецификации CORBA, и работа над ним никогда не прекращается.

Дополнительную информацию о группе OMG можно найти на Web-сайте `www.omg.org`. Он содержит много информации о CORBA, в том числе описание последних спецификаций, учебные курсы, ссылки на Web-сайты разработчиков и т.д.

Следует также отметить, что в Internet доступно множество бесплатных реализаций CORBA. В настоящей главе рассматривается реализация *Borland*, входящая в пакет Delphi 6 Enterprise. В данном случае продукт, в котором реализована спецификация CORBA, называется VisiBroker. Вполне возможно, что этот брокер объектных запросов является самым распространенным в мире. Все динамические библиотеки, необходимые для работы с CORBA, входят в состав Delphi 6. Кроме того, в интегрированной среде разработки внедрены специальные мастера, благодаря которым разработка приложений значительно упрощается.

Характеристики CORBA

Некоторые характеристики CORBA делают эту спецификацию особенно удобной при работе в распределенной среде:

- В CORBA используется объектно-ориентированный подход. Каждый сервер CORBA предоставляет интерфейс, содержащий перечень поддерживаемых методов и типов данных. При этом подробности реализации от пользователя скрыты.
- Прозрачность обнаружения объектов. Истинная сила CORBA заключается в том, что объекты могут быть обнаружены в любом месте. Когда клиентское приложение CORBA запрашивает серверный объект, то расположение сервера заранее не известно. Фактически при использовании CORBA клиентское приложение работает с образом серверного приложения. В то же время все выглядит так, как будто серверный объект работает локально в собственном пространстве процесса. Данный вопрос будет подробно рассмотрен в разделе, посвященном архитектуре CORBA.

- Независимость от языка программирования. Объекты могут быть созданы при помощи различных языков программирования. При этом часто используются языки Java и C++, однако и Delphi также широко применяется, так как данный пакет предоставляет все необходимые средства разработки. Для совместимости с перечисленными языками программирования в объектах CORBA используются соответствующие интерфейсы. Каждый серверный объект должен быть согласован со своим определением интерфейса.

  Из-за различий в языках программирования подробности реализации не могут быть скорректированы клиентами на стороне сервера. В мире CORBA используется именно такая ограниченная объектно-ориентированная схема.

- Поддержка различных платформ и операционных систем. Существуют реализации CORBA для различных платформ и операционных систем. На стороне сервера баз данных часто используется язык Java, а на среднем уровне и стороне клиента — Delphi. Разработчики могут создавать мощные приложения для уже существующих систем. В то же время Delphi позволяет реализовать функции извлечения информации с сервера.

Архитектура CORBA

Архитектура CORBA схематически представлена на рис. 19.1. Общей частью для клиента и сервера является брокер объектных запросов ORB, который организует взаимодействие объектов. При этом используется протокол *Internet Inter-ORB* (IIOP), в основе которого лежит протокол TCP/IP, что гарантирует доставку сообщений и корректную работу во всех системах, где используется TCP/IP. Кроме обработки трафика сообщений брокер ORB также вносит изменения, компенсирующие различия платформ.

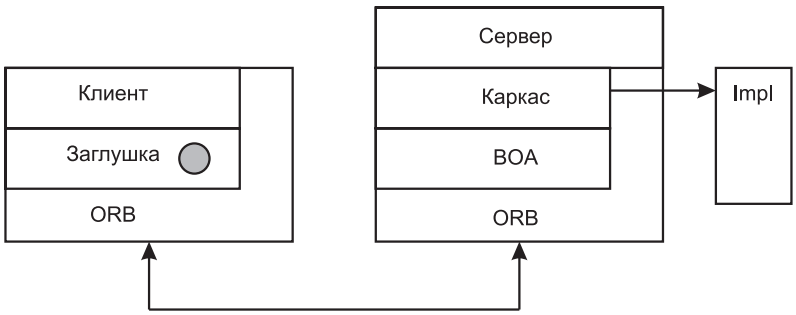

Рис. 19.1. *Архитектура CORBA*

Предположим, что с компьютера на базе процессора *Intel* на рабочую станцию *Sun* было передано число `123`. Без вмешательства в процесс передачи такое число корректно обработано не будет, потому что в процессорах *Intel* и *Sun* используются различные схемы регистров.

Это называют конфликтом “остроконечников и тупоконечников” (Big-Endian/Little-Endian problem). Брокер ORB знает о типе платформы, на которой он запущен, поэтому устанавливает в сообщении CORBA соответствующий флажок. Принимающая сторона

считывает такой флажок и автоматически выполняет корректную обработку данных. Подобный подход гарантирует правильную обработку числа `123` на обеих сторонах.

НА ЗАМЕТКУ

Согласно странице 65 книги *Ру* (Rhu), *Херрона* (Herron) и *Клинкера* (Klinker) *IIOP Complete* (издательства *Addison Wesley*), "термины *остроконечник* и *тупоконечник* — это аналогия с книгой *Джонатана Свифта Путешествия Гулливера*, в которой острова Лилипутия и Блефуску враждовали из-за того, что не могли договориться, с какой стороны разбивать яйца, — с тупой или с острой".

Клиентская сторона состоит из двух дополнительных слоев (см. рис. 19.1). Прямоугольнику клиента соответствует приложение. Более интересным элементом является так называемая *заглушка* (stub). Заглушка — это файл, который автоматически создается специальным компилятором под названием IDL2Pas, входящим в состав Delphi Enterprise. Такой компилятор анализирует файлы, в которых описаны интерфейсы сервера, и создает исходный код на языке Delphi Pascal для работы с CORBA ORB. Документацию по IDL2Pas можно найти в файлах HTML на компакт-диске Delphi 6 в каталоге `Delphi6\Doc\Corba`.

Заглушка содержит один или более классов, "отображающих" (mirror) сервер CORBA. Эти классы содержат несколько открытых интерфейсов и типов данных, аналогичных тем, что предоставлены на сервере. Для взаимодействия с сервером клиент обращается к классам заглушки, которые выполняют роль посредника для доступа к объектам сервера. Символ в блоке заглушки соответствует подключению к серверу. Подключение осуществляется с помощью *связанного* (bind) вызова, поступающего со стороны клиента. В свою очередь, заглушка использует так называемую *объектную ссылку* (object reference) на сервер, которой соответствует символ. После подключения к серверу клиент вызывает один из методов класса заглушки. Заглушка упаковывает запрос и все параметры метода в специальном буфере, а затем передает этот пакет на сервер. Описанный процесс упаковки называется *маршалингом* (marshaling) данных. Далее заглушка через брокер ORB выполняет вызов на сервере метода, соответствующего объектной ссылке. Когда от сервера поступает ответ, класс заглушки принимает сообщение от брокера ORB и передает его клиенту.

Клиент также может напрямую вызывать некоторые специальные функции брокера ORB. Эта логическая связь обозначена на схеме (см. рис. 19.1).

На стороне сервера находится интерфейс ORB, *называемый базовым объектным адаптером* (BOA — Basic Object Adapter). Он отвечает за маршрутизацию сообщений между брокером ORB и интерфейсом каркаса (описываемым далее). В будущем в Delphi предполагается использовать более гибкий настраиваемый *интерфейс переносимого объектного адаптера* (POA — Portable Object Adapter).

Каркас (skeleton) — это класс, который так же, как и заглушка создается компилятором IDL2Pas. Он содержит один или более классов, предоставляющих серверные интерфейсы CORBA. В реализации CORBA на Delphi каркас не содержит каких-либо деталей реализации серверных интерфейсов. Это — просто файл с классами, которые реализуют часть функций сервера. Такой файл называют еще файлом IMPL (сокращение от слова "implementation" — реализация).

Классы в файле IMPL предназначены не только для спецификации CORBA. Одни и те же классы реализации могут использоваться для организации интерфейсов CORBA, COM и других стандартов.

При получении сообщения на стороне сервера брокер ORB передает буфер сообщения интерфейсу BOA, который, в свою очередь, передает этот буфер классу каркаса. Каркас распаковывает данные буфера (*демаршалинг*) и определяет, какие методы из файла IMPL должны быть вызваны. После вызова этих методов каркас упаковывает полученные результаты и значения параметров для передачи клиенту. Буфер ответа опять последовательно передается интерфейсу BOA и ORB, а затем отправляется брокеру ORB на стороне клиента.

OSAgent

Объекты CORBA должны уметь обнаруживать друг друга. Группа OMG реализовала эту задачу при помощи так называемой *службы имен* (naming service), описанной в спецификации CORBA. Служба имен — это программа, запущенная на сервере в сети. Объекты на стороне сервера регистрируются такой службой, благодаря чему они могут быть обнаружены клиентскими приложениями. Для использования службы имен необходимо добавить немного программного кода в приложения — как на стороне клиента, так и на стороне сервера. Расположение приложения службы имен должно быть известно заранее, иначе клиентское приложение не сможет подключиться к серверному объекту.

Установка соединения между клиентами и серверами — это достаточно сложный процесс. В состав VisiBroker входит специальная программа под названием OSAgent, которая самостоятельно осуществляет поиск объектов, что намного проще применения службы имен. VisiBroker не является частью спецификации CORBA и используется только в брокере ORB *Borland*. Так как средства ORB VisiBroker используются внутри реализации CORBA, то для обнаружения объектов и привязки к ним лучше использовать программу OSAgent.

Перед тем как запустить приложение CORBA, созданное с применением средств VisiBroker, необходимо запустить OSAgent. При запуске серверное приложение зарегистрируется самостоятельно, используя этот агент. Клиентское приложение при подключении к серверу сначала обратится к OSAgent, запросит адрес сервера, а затем подключится непосредственно к серверному процессу.

Интерфейсы

Все объекты CORBA описываются своими интерфейсами. Это полностью объектно-ориентированная модель. Серверное приложение предоставляет клиентам доступ только к определенным объявлениям типов, интерфейсам и методам. После того как к интерфейсам открыт доступ, изменять их уже нельзя. Для того чтобы добавить в объект дополнительные свойства, лучше всего на базе старого сервера создать новый, а затем расширить этот новый объект. Таким образом можно предоставлять доступ к новым интерфейсам, обойдя проблему обратной совместимости приложений.

Для описания интерфейсов группа OMG разработала специальный *язык определения интерфейсов* (IDL — Interface Definition Language). Несмотря на то, что IDL напоминает языки C и Java, — это совершенно независимый язык программирования. Все

производители брокеров ORB предоставляют компиляторы IDL для преобразования файлов IDL в программный код на языке определения. Термин "*компилятор IDL*" не совсем корректен, потому что на самом деле, файлы IDL не компилируются в исполняемые файлы. Скорее, это генератор кода, так как результатом его работы является набор файлов с исходным кодом на некотором языке программирования.

Группой OMG были разработаны преобразователи исходного кода для языков C++ и Java. Для C++ в состав брокера ORB входит компилятор IDL2CPP, а для Java — компилятор IDL2Java.

Файлы, созданные компиляторами IDL2, содержат классы заглушек и каркасов, рассмотренных ранее в настоящем разделе. В состав Delphi входит компилятор IDL2Pas, который можно запустить в режиме командной строки или через мастера CORBA интегрированной среды разработки.

Язык определения интерфейсов (IDL)

Язык IDL — это очень большая тема. На компакт-диске Delphi 6 Enterprise в каталоге `Delphi6\Doc\CORBA` есть документ в формате PDF, в котором описывается преобразование исходного кода на языке Object Pascal. В этом документе подробно рассматриваются все типы данных, модули, вопросы наследования и пользовательские типы. В настоящем разделе приведены лишь некоторые наиболее важные аспекты IDL, а более подробную информацию по этой теме можно найти в упомянутом документе.

Файлы IDL должны соответствовать нескольким требованиям. Прежде всего, они должны иметь расширение `.idl` (регистр символов при этом значения не имеет). Другие расширения файлов не допускаются.

Содержимое файла IDL должно соответствовать определенной структуре. Описание интерфейсов чувствительно к регистру символов. Так, в языках C++ и Java, имена `Foo` и `foo` не являются эквивалентными, но в Delphi эти имена будут рассматриваться как одно, что неизбежно приведет к конфликту интерфейса.

Для комментариев в файлах IDL используется синтаксис языка C или C++:

```
// Это комментарий для одной строки.
/* Это пример блочного комментария,
      который может занимать
      несколько строк. */
```

Все ключевые слова языка IDL пишутся в нижнем регистре, иначе они не будут восприняты компилятором IDL2Pas. По возможности следует избегать использования ключевых слов Delphi, так как, согласно спецификации преобразования, всем ключевым словам Delphi должен предшествовать символ подчеркивания (_). Другими словами, зарезервированные слова Delphi лучше не использовать.

С помощью директивы `#include` в файлы IDL можно подключать другие файлы IDL. Это позволяет объединять большие файлы IDL в небольших группах.

Основные типы данных

В языке IDL для описания интерфейсов используется несколько основных типов данных. Они перечислены в табл. 19.1 с указанием соответствующих типов языка Object Pascal.

Таблица 19.1. Основные типы данных языка IDL

Тип языка IDL	*Тип языка Pascal*
boolean	Boolean
Char	Char
wchar	Wchar
octet	Byte
string	AnsiString
wstring	WideString
short	SmallInt
unsigned short	Word
long	Integer
unsigned long	Cardinal
long long	Int64
unsigned long long	Int64
float	Single
double	Double
long double	Extended
fixed	соответствующего типа не существует

В языке IDL нет типа `int`. Вместо него используются целочисленные типы `short`, `long`, `unsigned short` и `unsigned long`. Символьные типы соответствуют типу `ISO Latin-1`, эквивалентному таблице ASCII. Единственным исключением является символ `NUL` (`#0`). Программисты C и C++ просили группу OMG не использовать этот символ, так как он в языке C указывает на конец строки.

Реализация типа `Boolean` зависит от разработчиков. Тип `Boolean` IDL в Delphi соответствует типу с тем же названием, а тип `Any` — типу `Variant`.

Пользовательские типы данных

В IDL можно определять собственные типы. Синтаксис подобен определению структур в языке C. К общим пользовательским типам данных относятся псевдонимы, перечисления, структуры, массивы и последовательности.

Псевдонимы

Псевдонимы используются для присвоения типам данных более удобочитаемых имен. Например, тип года может быть описан следующим образом:

```
typedef short YearType;
```

Перечисления

Перечислениям языка IDL соответствует перечислимый тип Delphi. Перечисление нескольких цветов может выглядеть следующим образом:

```
enum Color(red, white, blue, green, black);
```

Структуры

Структуры IDL аналогичны записям в языке Pascal. Ниже представлен пример структуры, предназначенной для хранения значения времени:

```
struct TimeOfDay {
      short hour;
      short minute;
      short seconds;
};
```

Массивы

Массивы бывают одномерными и многомерными. Они определяются при помощи ключевого слова `typedef`. Вот несколько примеров:

```
typedef Color ColorArray[4];  // одномерный массив типа Color
typedef string StringArray[10][20]; // 10 строк по 20 символов
```

Последовательности

Последовательности в языке IDL используются очень часто. В Delphi им соответствуют массивы переменной длины. Последовательности бывают ограниченными и неограниченными.

```
typedef sequence<Color> Colors;
typedef sequence<long, 1000> NumSeq;
```

Первый параметр в объявлении последовательности обозначает базовый тип массива переменной длины. Второй параметр является не обязательным и задает длину для ограниченной последовательности.

Чаще всего в программировании CORBA последовательности используются для передачи записей баз данных между серверами и клиентами. После получения последовательности клиентским приложением из нее в цикле извлекаются значения всех полей записи. Затем полученная информация используется элементами управления пользовательского интерфейса. При этом средства CORBA могут взаимодействовать со средствами MIDAS.

Параметры методов

Все аргументы методов должны объявляться с одним из трех атрибутов: `in`, `out` или `inout`.

Значение параметра, объявленного как `in`, устанавливается клиентом. В Delphi такому параметру отвечает параметр типа `const`.

Значение параметра, объявленного как `out`, устанавливается сервером. В Delphi такому параметру соответствует параметр типа `var`.

Начальное значение параметра, объявленного как `inout`, устанавливается клиентом. Получив данные, сервер изменяет их значение и возвращает клиенту. Параметру типа `inout` в Delphi соответствует параметр типа `var`.

Модули

Ключевое слово `module` используется для группирования интерфейсов и типов. Имя модуля используется компилятором IDL2Pas для присвоения имени соответствующему модулю Delphi. Интерфейсы и типы, определенные внутри модуля, являются для него локальными. Для обращения к интерфейсу `Bar`, определенному в модуле по имени `Foo`, используется следующий синтаксис: `Foo::Bar`.

В IDL не поддерживаются закрытые (private) или защищенные (protected) типы и методы. Все интерфейсы и методы считаются открытыми (public). Это имеет значение в том случае, когда в файле IDL описаны интерфейсы, предоставляемые сервером внешнему миру. В этом случае некоторые методы или типы желательно скрыть или защитить.

Для того чтобы понять подходы программирования на IDL, лучше всего изучить примеры, написанные другими. В каталоге VisiBroker (по умолчанию `c:\Inprise`) есть каталог `IDL`, содержащий файлы IDL с различными интерфейсами CORBA, типа ORB и различных служб. Эти файлы являются хорошей отправной точкой для изучения, так как в них есть множество примеров объявления и определения интерфейсов, вложенных модулей и ссылок на внешние типы данных.

Наряду с основными понятиями IDL будут рассмотрены несколько примеров разработки приложений CORBA, демонстрирующих возможности распределенного объектного программирования. Оставшаяся часть главы посвящена вопросам разработки серверов и клиентов CORBA.

Модуль Bank

В CORBA есть традиционный пример модуля, аналогичного программе вывода строки "`Hello, world`" в языке C. Он называется `Bank` и содержит простой вызов метода, возвращающего банковский баланс. В нем используются методы `deposit` (внести) и `withdraw` (снять), соответствующие переводу денег на счет и снятию их со счета. Необходимо также реализовать запрет на снятие со счета сумм, превышающих количество наличных денег. Модуль IDL для этого примера представлен в листинге 19.1.

Листинг 19.1. `Bank.idl`

```
module Bank {
    exception WithdrawError {
          float current_balance;
    };

    interface Account {
        void deposit(in float amount);
        void withdraw(in float amount) raises (WithdrawError);
        float balance();
    };
};
```

Исключение объявлено с одним элементом данных. Если клиент попытается снять со счета больше денег, чем это возможно, возникнет исключение, в элементе данных которого будет указан баланс. Клиентское приложение может перехватить такое исключение и отобразить соответствующее сообщение. В данном случае сообщение будет содержать текущий баланс.

Методы `deposit` и `withdraw` эквивалентны процедурам, поэтому они возвращают значение типа `void`. Каждый из этих методов принимает один параметр: количество денег, которые клиент вносит на счет или снимает со счета. Остаток представляет собой число с плавающей запятой, которому в Delphi соответствует тип `single`. Обратите внимание: параметры в методах `deposit` и `withdraw` объявлены как `in`. Это связано тем, что методы передают значения таких параметров от клиента серверу. Метод `balance` — это функция, которая возвращает число с плавающей запятой, содержащее текущий баланс средств на счету.

В состав интегрированной среды разработки Delphi 6 входит множество мастеров, значительно облегчающих разработку клиентской и серверной частей приложений CORBA. Итак, начнем с разработки серверной части. Для вызова мастера выберите в меню File пункты New и Other, а в появившемся диалоговом окне перейдите на вкладку CORBA и дважды щелкните на пиктограмме CORBA server. В результате появится главное окно мастера (рис. 19.2).

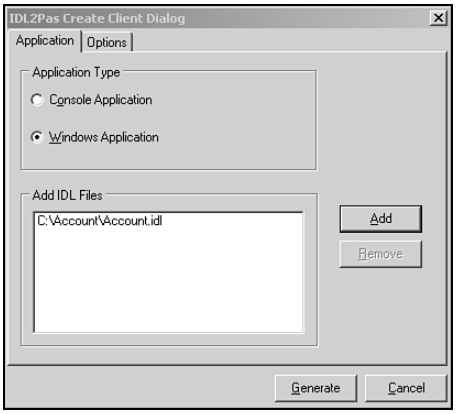

Рис. 19.2. *Мастер CORBA в Delphi 6*

Это окно содержит список всех файлов IDL, которые можно применить для создания исходного кода приложения. Вначале данный список будет пуст. Для того чтобы добавить имена файлов, необходимо щелкнуть на кнопке Add (Добавить) и в стандартном диалоговом окне Open найти каталог с файлом `Bank.idl`, отметить этот файл и щелкнуть на кнопке OK. В результате файл `Bank.idl` будет добавлен в список файлов, обрабатываемых компилятором IDL2Pas. Для данного приложения это единственный файл IDL, поэтому щелкните на кнопке Generate (Создать). В результате будет создано новое приложение сервера.

Компилятор IDL2Pas обработает указанный файл IDL, и мастер создаст соответствующее приложение сервера, состоящее из четырех файлов:

- `Bank_I.pas` — этот файл содержит определения всех интерфейсов и типов.

- `Bank_C.pas` — данный файл содержит все пользовательские типы, исключения и клиентские классы заглушек. Кроме того, все эти пользовательские типы и классы заглушек обладают вспомогательными классами, предназначенными для организации обмена данными с буферами CORBA.
- `Bank_S.pas` — этот файл содержит определение класса каркаса на стороне сервера.
- `Bank_Impl.pas` — данный файл содержит определение общего класса для реализации на стороне сервера. В методы данного класса можно вносить собственный программный код. Такой файл пока что использоваться не будет.

Из этого списка файлов очевидно, что клиентская заглушка, представленная в архитектуре CORBA, находится в файле `Bank_C.pas`, а серверный каркас — в файле `Bank_S.pas`. Реализация серверной части находится в файле `Bank_Impl.pas`.

Определение интерфейса этого приложения представлено в листинге 19.2. В данном случае используется только один интерфейс по имени `Account`. Он содержит три метода, объявленные в файле IDL.

Листинг 19.2. `Bank_I.pas`

```
unit Bank_i;

interface

uses
  CORBA;

type
  Account = interface;

  Account = interface
    ['{99FCA96D-77B2-4A99-7677-E1E0C32F8C67}']
    procedure deposit (const amount : Single);
    procedure withdraw (const amount : Single);
    function  balance : Single;
  end;

implementation

initialization

end.
```

В листинге 19.3 представлено содержимое файла `Bank_C.pas`. В нем объявлено исключение `Overdrawn`. Здесь же определен базовый класс исключений `UserException`.

Листинг 19.3. Файл `Bank_C.pas`

```
unit Bank_c;

interface

uses
  CORBA, Bank_i;

type
  EWithdrawError = class;
  TAccountHelper = class;
  TAccountStub = class;

  EWithdrawError = class(UserException)
  private
    Fcurrent_balance: Single;
  protected
    function  _get_current_balance: Single; virtual;
  public
    property  current_balance: Single read _get_current_balance;
    constructor Create; overload;
    constructor Create(const current_balance: Single); overload;
    procedure Copy(const _Input: InputStream); override;
    procedure WriteExceptionInfo(var _Output: OutputStream);
                                                 override;
  end;

  TAccountHelper = class
    class procedure Insert (var _A: CORBA.Any; const _Value:
                            Bank_i.Account);
    class function  Extract(var _A: CORBA.Any): Bank_i.Account;
    class function  TypeCode: CORBA.TypeCode;
    class function  RepositoryId: string;
    class function  Read (const _Input: CORBA.InputStream):
                                          Bank_i.Account;
    class procedure Write(const _Output: CORBA.OutputStream;
                          const _Value: Bank_i.Account);
    class function  Narrow(const _Obj: CORBA.CORBAObject;
                           _IsA: Boolean = False): Bank_i.Account;
    class function  Bind(const _InstanceName: string = '';
                         _HostName: string = ''): Bank_i.Account;
                                                   overload;
    class function  Bind(_Options: BindOptions;
                         const _InstanceName: string = '';
                         _HostName: string = ''): Bank_i.Account;
                                                   overload;
  end;

  TAccountStub = class(CORBA.TCORBAObject, Bank_i.Account)
  public
    procedure deposit  ( const amount : Single); virtual;
    procedure withdraw ( const amount : Single); virtual;
    function  balance: Single; virtual;
  end;
```

```
implementation

var

  WithdrawErrorDesc: PExceptionDescription;

function EWithdrawError._get_current_balance: Single;
begin
  Result := Fcurrent_balance;
end;

constructor EWithdrawError.Create;
begin
  inherited Create;
end;

constructor EWithdrawError.Create(const current_balance: Single);
begin
  inherited Create;
  Fcurrent_balance := current_balance;
end;

procedure EWithdrawError.Copy(const _Input: InputStream);
begin
  _Input.ReadFloat(Fcurrent_balance);
end;

procedure EWithdrawError.WriteExceptionInfo(var _Output:
                                            OutputStream);
begin
  _Output.WriteString('IDL:Bank/WithdrawError:1.0');
  _Output.WriteFloat(Fcurrent_balance);
end;

function  WithdrawError_Factory: PExceptionProxy; cdecl;
begin
  with Bank_c.EWithdrawError.Create() do Result := Proxy;
end;

class procedure TAccountHelper.Insert(var _A: CORBA.Any;
                                      const _Value: Bank_i.Account);
begin
  _A := Orb.MakeObjectRef( TAccountHelper.TypeCode,
                           _Value as CORBA.CORBAObject);
end;

class function TAccountHelper.Extract(var _A:
                                        CORBA.Any): Bank_i.Account;
var
  _obj: Corba.CorbaObject;
begin
  _obj := Orb.GetObjectRef(_A);
```

```
  Result := TAccountHelper.Narrow(_obj, True);
end;

class function TAccountHelper.TypeCode: CORBA.TypeCode;
begin
  Result := ORB.CreateInterfaceTC(RepositoryId, 'Account');
end;

class function TAccountHelper.RepositoryId: string;
begin
  Result := 'IDL:Bank/Account:1.0';
end;

class function TAccountHelper.Read(const _Input:
                              CORBA.InputStream): Bank_i.Account;
var
  _Obj: CORBA.CORBAObject;
begin
  _Input.ReadObject(_Obj);
  Result := Narrow(_Obj, True)
end;

class procedure TAccountHelper.Write(const _Output:
             CORBA.OutputStream; const _Value: Bank_i.Account);
begin
  _Output.WriteObject(_Value as CORBA.CORBAObject);
end;

class function TAccountHelper.Narrow(const _Obj:
              CORBA.CORBAObject; _IsA: Boolean): Bank_i.Account;
begin
  Result := nil;
  if (_Obj = nil) or
     (_Obj.QueryInterface(Bank_i.Account, Result) = 0) then exit;
  if _IsA and _Obj._IsA(RepositoryId) then
    Result := TAccountStub.Create(_Obj);
end;

class function TAccountHelper.Bind(const _InstanceName:
            string = ''; _HostName: string = ''): Bank_i.Account;
begin
  Result := Narrow(ORB.bind(RepositoryId, _InstanceName,
                            _HostName), True);
end;

class function TAccountHelper.Bind(_Options: BindOptions;
                           const _InstanceName: string = '';
                           HostName: string = ''): Bank_i.Account;
begin
  Result := Narrow(ORB.bind(RepositoryId, _Options, _InstanceName,
                  _HostName), True);
end;
```

```
procedure TAccountStub.deposit ( const amount: Single);
var
  _Output: CORBA.OutputStream;
  _Input:  CORBA.InputStream;
begin
  inherited _CreateRequest('deposit', True, _Output);
  _Output.WriteFloat(amount);
  inherited _Invoke(_Output, _Input);
end;

procedure TAccountStub.withdraw ( const amount : Single);
var
  _Output: CORBA.OutputStream;
  _Input:  CORBA.InputStream;
begin
  inherited _CreateRequest('withdraw', True, _Output);
  _Output.WriteFloat(amount);
  inherited _Invoke(_Output, _Input);
end;

function  TAccountStub.balance: Single;
var
  _Output: CORBA.OutputStream;
  _Input:  CORBA.InputStream;
begin
  inherited _CreateRequest('balance', True, _Output);
  inherited _Invoke(_Output, _Input);
  _Input.ReadFloat(Result);
end;

initialization
Bank_c.WithdrawErrorDesc := RegisterUserException('WithdrawError',
                                  'IDL:Bank/WithdrawError:1.0',
                                  @Bank_c.WithdrawError_Factory);

finalization
UnRegisterUserException(Bank_c.WithdrawErrorDesc);

end.
```

В листинге 19.4 представлено определение класса реализации `Account`. Данный класс привязан к CORBA, поэтому может быть использован и в других приложениях или интерфейсах. Его методы объявлены в файле `Bank.idl`. Для реализации полнофункционального сервера в методы класса `TAccount` был внесен дополнительный программный код.

Листинг 19.4. Класс реализации для сервера `Bank`

```
unit Bank_impl;
```

```
interface

uses
  SysUtils, CORBA, Bank_i, Bank_c;

type
  TAccount = class;

  TAccount = class(TInterfacedObject, Bank_i.Account)
  protected
    _balance: Single;
  public
    constructor Create;
    procedure deposit  (const amount: Single);
    procedure withdraw (const amount: Single);
    function  balance: Single;
  end;

implementation

constructor TAccount.Create;
begin
  inherited;
  _balance := random(10000);
end;

procedure TAccount.deposit(const amount: Single);
begin
  if amount > 0 then
    _balance := _balance + amount;
end;

procedure TAccount.withdraw(const amount: Single);
begin
  if amount < _balance then
    _balance := _balance - amount
  else
    raise EWithdrawError.Create(_balance);
end;

function  TAccount.balance: Single;
begin
  result := _balance;
end;

initialization
  randomize;

end.
```

Класс `TAccount` является производным от класса `TInterfacedObject`, а значит, подсчет ссылок он выполняет автоматически. Реализация интерфейса `Account` содержится в файле `Bank_I.pas`. В методе `deposit` выполняется простая проверка, исключающая передачу отрицательного числа. В методе `withdraw` выполняется про-

верка суммы денег, снимаемой клиентом со счета. Если эта сумма превышает размер остатка на счету, то будет передано исключение с указанием текущего остатка. Клиентская часть приложения может обработать это исключение и отобразить пользователю соответствующую информацию. Метод `balance` возвращает текущий баланс.

В листинге 19.5 представлен класс заглушки, который используется клиентским приложением в качестве посредника. Подобно каркасу сервера, класс заглушки содержит три метода, определенные для интерфейса `Account` в файле IDL.

Листинг 19.5. Класс заглушки на стороне клиента

```
TAccountStub = class(CORBA.TCORBAObject, Bank_i.Account)
  public
    procedure deposit  ( const amount: Single); virtual;
    procedure withdraw ( const amount: Single); virtual;
    function  balance: Single; virtual;
  end;
```

В листинге 19.6 метод `deposit()` представлен полностью. Два буфера потоков CORBA объявлены как локальные переменные. Метод `CreateRequest()` обращается к брокеру ORB, который определяет корректный выходной буфер и заносит в него информацию. Заглушка передает имя метода, который будет вызываться на стороне сервера, а также определяет, требуется ли ожидание завершения выполнения задачи на сервере. В зависимости от этого вызов будет односторонним или двусторонним.

Листинг 19.6. Метод `deposit` класса заглушки

```
procedure TAccountStub.deposit(const amount: Single);
var
  _Output: CORBA.OutputStream;
  _Input:  CORBA.InputStream;
begin
  inherited _CreateRequest('deposit', True, Output);
  _Output.WriteFloat(amount);
  inherited _Invoke(Output, Input);
end;
```

Теперь данные, предназначенные для передачи на сервер, необходимо занести в выходной буфер. В данном случае в буфер заносится сумма, вносимая на счет. В конце вызывается метод `Invoke`. Это — еще одно обращение к брокеру ORB, после которого запрос и содержимое выходного буфера передается на сервер. Выполнение приложения на стороне клиента продолжается только после того, как сервер завершит обработку запроса. При этом в случае вызова метода, представляющего собой функцию (например метод `balance`), возвращаемый результат будет занесен во входной буфер. Программный код для считывания значений из входного буфера создается компилятором IDL2Pas. В данном примере вызываемый метод представляет собой процедуру, поэтому никакого результата не возвращается.

Весь код заглушки создается компилятором IDL2Pas автоматически, и вносить в него какие-либо изменения самостоятельно не требуется. Однако важно понимать, что же происходит в этом программном коде.

Завершающая часть кода приложения имеет отношение к графическому интерфейсу клиентской части. Интерфейс клиента состоит из трех кнопок, двух полей ввода текста и одной метки (рис. 19.3). Все переменные интерфейса CORBA объявлены как типы интерфейса. В данном случае интерфейс `Account` объявлен как тип `Account`. При таком подходе переменные инициализируются на основании определенного в файле `Bank_i.pas` типа, который содержит три метода (определены в файле `Bank.idl`). Еще одно преимущество использования переменных типа интерфейса заключается в автоматическом подсчете ссылок, который должен выполняться для всех объектов CORBA. Программный код для этого автоматически создается компилятором IDL2Pas.

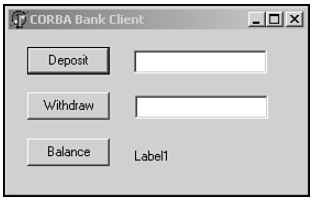

Рис. 19.3. *Клиентское приложение CORBA*

Наиболее интересной частью приложения является обработчик события `btnWithdraw.OnClick`. Исходный код клиентской части представлен в листинге 19.7. В методе `Withdraw()` сопоставляется сумма, снимаемая клиентом, и остаток на счету. Если снимаемая сумма превышает остаток, то возникает исключение. Обратите внимание: передача исключений в CORBA идентична передаче исключений в Delphi. Исключение Delphi преобразуется в исключение CORBA автоматически.

Листинг 19.7. Исходный код клиентской части приложения

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Corba, Bank_c, Bank_i, StdCtrls;

type
  TForm1 = class(TForm)
    btnDeposit: TButton;
    btnWithdraw: TButton;
    btnBalance: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    procedure btnDepositClick(Sender: TObject);
    procedure btnWithdrawClick(Sender: TObject);
    procedure btnBalanceClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
```

```
  private
  { Закрытые объявления }
  protected
    Acct: Account;
    procedure InitCorba;
  { Защищенные объявления }
  public
  { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.InitCorba;
begin
  CorbaInitialize;
  // Привязка к серверу Corba
  Acct := TAccountHelper.bind;
end;

procedure TForm1.btnDepositClick(Sender: TObject);
begin
  Acct.deposit(StrToFloat(Edit1.text));
end;

procedure TForm1.btnWithdrawClick(Sender: TObject);
begin
  try
    Acct.withdraw(StrToFloat(Edit2.Text));
  except
    on e: EWithdrawError do
      ShowMessage('Withdraw Error. The balance = ' +
                  FormatFloat('$##,##0.00', E.current_balance));
  end;
end;

procedure TForm1.btnBalanceClick(Sender: TObject);
begin
  label1.caption := FormatFloat('Balance = $##,##0.00',
                                acct.balance);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  InitCorba;
end;

end.
```

После компиляции клиентской и серверной частей приложения необходимо запустить программу OSAgent. В системах Windows NT, OSAgent VisiBroker может быть

установлен как служба. В других операционных системах он запускается вручную. Для ручного запуска OSAgent на любой платформе MS Windows выберите в меню кнопки Start пункт Run и наберите команду **`OSAgent -C`**, которая запустит OSAgent в консольном режиме. Кроме того, пиктограмма данного агента появится на панели задач.

Затем запускается серверное приложение, и только после этого — клиентское. Графический интерфейс клиентского приложения представлен на рис. 19.3. Он состоит из трех кнопок, двух полей ввода текста и метки, отображающей баланс. Чтобы получить с сервера исходное значение баланса, щелкните на кнопке Balance. Затем внесите на счет определенную сумму и щелкните на кнопке Balance еще раз, чтобы обновить значение на стороне клиента. При вызове методов `deposit` и `withdraw` значение баланса также автоматически обновится на стороне клиента. Теперь попытайтесь снять со счета сумму, превышающую остаток. В результате на экране появится сообщение об ошибке.

Сложные типы данных

Следующий пример не имеет большого практического значения, но иллюстрирует применение ряда сложных типов данных в файлах IDL CORBA. В листинге 19.8 представлено содержимое файла IDL для *дополнительных типов данных* (ADT — Advanced Data Types).

Листинг 19.8. `ADT.idl`

```
// Файл IDL для ADT
//
// Демонстрация различных структур данных IDL
//
// использование псевдонима для строковых типов

typedef string Identifier;

enum EnumType
{
  first,
  second,
  third
};

struct StructType
{
  short s;
  long l;
  Identifier i;
};

const unsigned long ArraySize = 3;

typedef StructType StructArray[ArraySize];

typedef sequence<StructType> StructSequence;
```

```
interface ADT
{

  void Test1(in Identifier st,
             in EnumType myEnum,
             inout StructType myStruct);

  void Test2(out StructType myStruct,
             in StructArray myStructArray,
             out StructSequence myStructSeq);

};
```

Первый тип данных демонстрирует использование псевдонима для преобразования строковых типов. Все строки в данном примере имеют тип `Identifier`. Тип `EnumType` содержит три значения: `first`, `second` и `third`.

Тип `StructType` подобен записи (record) языка Pascal. Эта структура данных состоит из трех элементов типа `short`, `long` и `string` (преобразован в псевдоним `Identifier`). Размер массива хранится в константе `ArraySize`.

В следующих двух элементах файла IDL объявляются типы, основанные на предыдущих определениях. Массив `StructArray` может состоять не более чем из трех элементов (индексация начинается с нуля). Последовательность — это динамический массив. В последней конструкции `typedef` объявляется последовательность элементов типа `StructType`.

И, наконец, в интерфейсе ADT определены два метода: `Test1` и `Test2`. Параметры этих методов подобраны таким образом, чтобы продемонстрировать различные направления перемещения данных. Параметры `In` создаются и инициализируются на стороне клиента. Параметры `Out` создаются и инициализируются на стороне сервера. Параметры `InOut` создаются и инициализируются на стороне клиента, модифицируются на стороне сервера и возвращаются клиенту с новыми значениями элементов данных.

В листинге 19.9 представлено содержимое файла интерфейса `ADT_I.pas`. Обратите внимание, что конструкции `typedef` определены именно в этом файле. Кроме того, интерфейс создается для типа `StructType`. Всем сложным типам соответствуют объекты языка Object Pascal с методами `get` и `set` и одним вспомогательным классом, реализующим упаковку (маршалинг) данных в буфер CORBA.

Листинг 19.9. Файл `ADT_I.pas`

```
unit adt_i;

interface

uses
  CORBA;

type
```

```
  EnumType = (first, second, third);

const
  { (Не изменяйте значения, присвоенные этим константам.) }

  ArraySize: Cardinal = 3;

type
  StructType = interface;
  ADT = interface;

  Identifier = AnsiString;

  StructArray = array[0..2] of adt_i.StructType;

  StructSequence = array of adt_i.StructType;

  StructType = interface
    ['{B4A1845D-4DB0-9B2E-A2E3-001F2D6B8C81}']
    function  _get_s: SmallInt;
    procedure _set_s (const s: SmallInt);
    function  _get_l: Integer;
    procedure _set_l (const l: Integer);
    function  _get_i: adt_i.Identifier;
    procedure _set_i (const i: adt_i.Identifier);
    property  s: SmallInt read _get_s write _set_s;
    property  l: Integer read _get_l write _set_l;
    property  i: adt_i.Identifier read _get_i write _set_i;
  end;

  ADT = interface
    ['{203B9E07-735F-2980-CB02-353A7C6A5B68}']
    procedure Test1 (conststr: adt_i.Identifier;
                     const myEnum: adt_i.EnumType;
                     var   myStruct: adt_i.StructType);
    procedure Test2 (out   myStruct: adt_i.StructType;
                     const myStructArray: adt_i.StructArray;
                     out   myStructSeq: adt_i.StructSequence);
  end;

implementation

initialization

end.
```

В листинге 19.10 представлена реализация серверной части. При считывании значений параметров метода в файле IDL определяется направление передачи данных. Параметру `out` соответствует передача данных клиенту, а параметру `in` — передача данных на сервер.

Для всех параметров `out` на стороне сервера до возврата результата клиенту должны быть созданы и инициализированы соответствующие структуры данных. Все

параметры, определенные как const или var, имеют связанные с ними структуры данных.

Листинг 19.10. Файл реализации серверной части для ADT

```
unit adt_impl;

interface

uses
  SysUtils,
  CORBA,
  adt_i,
  adt_c;

type
  TADT = class;

  TADT = class(TInterfacedObject, adt_i.ADT)
  protected
    {**********************************}
    {*** Пользовательские переменные ***}
    {**********************************}
  public
    constructor Create;
    procedure Test1 ( const st: adt_i.Identifier;
                      const myEnum: adt_i.EnumType;
                      var   myStruct: adt_i.StructType);
    procedure Test2 ( out   myStruct: adt_i.StructType;
                      const myStructArray: adt_i.StructArray;
                      out   myStructSeq: adt_i.StructSequence);
  end;

implementation

uses ServerMain;

constructor TADT.Create;
begin
  inherited;
end;

procedure TADT.Test1 ( const st: adt_i.Identifier;
                       const myEnum: adt_i.EnumType;
                       var   myStruct: adt_i.StructType);
begin
  Form1.Memo1.Lines.Add('String from Client: ' + st);

  case myEnum of
    first:  Form1.Memo1.Lines.Add('Enum value is "first"');
    second: Form1.Memo1.Lines.Add('Enum value is "second"');
    third:  Form1.Memo1.Lines.Add('Enum value is "third"');
```

```
  end;

  Form1.Memo1.Lines.Add(Format('myStruct.s = %d', [myStruct.s]));
  Form1.Memo1.Lines.Add(Format('myStruct.l = %d', [myStruct.l]));
  Form1.Memo1.Lines.Add(Format('myStruct.i = %s', [myStruct.i]));

  myStruct.s := 10;
  myStruct.l := 1000;
  myStruct.i := 'This is the return string from the Server';
end;

procedure TADT.Test2 ( out   myStruct:      adt_i.StructType;
                       const myStructArray: adt_i.StructArray;
                       out   myStructSeq:   adt_i.StructSequence);
var
  k: integer;
  tempSeq: StructSequence;
begin
  myStruct := TStructType.Create(20, 2000,
                      'Hello from the server structType Test 2');

  for k := 0 to ArraySize - 1 do
    With Form1.Memo1.Lines do begin
      Add(Format('myStructArray[%d].s = %d',
                                [k, myStructArray[k].s]));
      Add(Format('myStructArray[%d].l = %d',
                                [k, myStructArray[k].l]));
      Add(Format('myStructArray[%d].i = %s',
                                [k, myStructArray[k].i]));
    end;

  SetLength(tempSeq, 2);
  for k := 0 to 1 do
    tempSeq[k] := TStructType.Create(k + 100, k + 1000,
                                     Format('k = %d', [k]));
  myStructSeq := tempSeq;
end;

initialization

end.
```

Пользовательский интерфейс клиентского приложения состоит из двух кнопок и поля memo. Каждой кнопке соответствует один из методов проверки ADT. Результаты вызова этих методов отображаются в поле memo. Исходный код клиентской части представлен в листинге 19.11.

Листинг 19.11. Клиентская часть ADT

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Corba, adt_c, adt_i, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
  { Закрытые объявления }
  protected
    myADT: ADT;
    procedure InitCorba;
  { Защищенные объявления }
  public
  { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.InitCorba;
begin
  CorbaInitialize;
  myADT := TADTHelper.bind;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  initCorba;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  temp: StructType;
begin
  temp := TStructType.Create(50, 500,
                     'This is the client struct in Test 1');
  myADT.Test1('Hello from the Test1 Client', first, temp);

  with Memo1.Lines do begin
```

```
    Add('Response from server inout struc var:');
    Add(Format('myStruct.s = %d', [temp.s]));
    Add(Format('myStruct.l = %d', [temp.l]));
    Add(Format('myStruct.i = %s', [temp.i]));
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  I: Integer;
  temp: StructType;
  tempSeq: StructSequence;
  tempArray: StructArray;
begin
   temp := TStructType.Create(0,0,'test');
   SetLength(tempSeq, 2);

   for I := 0 to ArraySize -1 do
     tempArray[I] := TStructType.Create(200 + I, 2000 + I,
                                Format('Stuct %d in Array', [I]));

   myADT.Test2(temp, tempArray, tempSeq);

   with Memo1.Lines do begin
     Add(Format('struct.s = %d', [temp.s]) );
     Add(Format('struct.l = %d', [temp.l]) );
     Add(Format('struct.i = %s', [temp.i]) );
   end;

   for I := 0 to 1 do with Memo1.Lines do begin
     Add( Format('tempSeq[%d].s = %d', [I, tempSeq[I].s]) );
     Add( Format('tempSeq[%d].l = %d', [I, tempSeq[I].l]) );
     Add( Format('tempSeq[%d].i = %s', [I, tempSeq[I].i]) );
   end;
end;

end.
```

Для запуска этого приложения необходимо откомпилировать его исходный код и убедиться в том, что запущен OSAgent. При щелчке на любой из кнопок происходит обмен данными между клиентом и сервером. Данные, принимаемые на любой из сторон, отображаются в поле memo.

Delphi, CORBA и Enterprise Java Beans (EJB)

В настоящем разделе описываются методы подключения приложений Delphi CORBA к средствам EJB, устанавливаемым на *сервер приложений Borland* (Borland Application Server). Для разработки и установки EJB в данном примере используется

пакет Borland JBuilder 5 и сервер приложений Borland версии 4.51. Бесплатные демонстрационные версии этих продуктов можно получить на Web-сайте *Borland* по адресу `www.borland.com`.

Интенсивный курс EJB для программистов Delphi

Несколько лет тому назад компания *Sun Microsystems* выпустила платформу J2EE. Это расширило Java до уровня корпоративной распределенной объектной среды. Спецификация J2EE достаточно сложная, однако с точки зрения разработчика приложений она может быть разбита на несколько простых концепций.

Одним из ключевых элементов платформы J2EE является компактный, независимый от платформы, масштабируемый объект — *Enterprise Java Bean* (EJB), разработанный для выполнения одной специфической задачи. Идея заключается в том, что все функции корпоративного приложения выполняются при помощи множества различных специализированных объектов EJB. Таким образом, приложение обращается к какому-либо объекту EJB только в том случае, если необходимо использовать его функции.

EJB — специализированный компонент

В терминах Delphi EJB представляет собой компонент. Примером EJB является компонент, который подключается к базе данных и извлекает ее записи для последующей передачи другим элементам приложения. В других случаях объекты EJB могут выполнять вычисления на основании получаемой ими информации (например расчет налога при продаже товара).

Контейнеры для хранения объектов EJB

В Delphi все компоненты хранятся в пакетах и устанавливаются в интегрированной среде разработки. Выбор компонентов в интегрированной среде разработки выполняется при помощи палитры компонентов. Исходный код компонента в приложении создается при его размещении в форме. При удалении компонента из формы удаляется и его определение в исходном коде.

Подобный подход применяется к объектам EJB, и не только в интегрированной среде разработки Delphi. В спецификации J2EE описан элемент под названием *контейнер EJB* (EJB container). Все объекты EJB хранятся в контейнерах, наподобие того как компоненты хранятся в пакетах Delphi. Контейнер управляет процессом создания и уничтожения объектов EJB.

Использование EJB предопределенных API

В сервер приложений *Borland* использует внедренный контейнер EJB. Аналогично интегрированной среде разработки Delphi и ее компонентам, контейнер EJB использует предопределенный набор интерфейсов API, задающих поведение объектов EJB. Разработчик EJB добавляет методы, реализующие специфическую функцию, но без предопределенных API контейнер не сможет корректно управлять заключенными в нем объектами.

Кроме создания и удаления объектов EJB, функции интерфейса API отвечают и за маршрутизацию сообщений, и за организацию обратных вызовов. Контейнер выполняет также и много других операций, описанных в спецификации, но в данной книге они рассматриваться не будут.

Интерфейсы Home и Remote

В состав набора предопределенных API для всех EJB должны входить два интерфейса. Первый называется `Home`, а второй — `Remote`. Интерфейсу `Home` соответствует метод инициализации, который вызывается приложением для получения экземпляра EJB. Кроме того, интерфейс `Home` отвечает за создание экземпляров интерфейса `Remote` и их последующую передачу вызывающему приложению.

Интерфейс `Remote` содержит все методы, которые могут использоваться в вызывающей программе. Они аналогичны интерфейсам, объявленным в файле IDL. Итак, для получения экземпляра интерфейса `Remote` клиентское приложение вызывает интерфейс `Home` объекта EJB. После этого клиент может вызывать любые предоставленные методы интерфейса `Remote`.

Типы объектов EJB

Все объекты EJB можно разделить на две категории:

- Session beans (элементы уровня сеанса)
- Entity beans (собственно объекты)

Session bean, обычно, являются объектами EJB без состояния (stateless). Имеется в виду, что между вызовами объект не хранит никакой информации о вызывающем приложении. Он не отслеживает положение клиента в последовательности вызовов и потому не имеет эквивалента в машине состояния (state machine). Вполне возможно создать session bean, учитывающий состояние (stateful), но все необходимые для этого функции придется разрабатывать самостоятельно.

Обычно entity bean используют в качестве оболочки для записей баз данных. Объекты этого типа учитывают состояние, так как обрабатываемые ими данные (записи) должны сохраняться между вызовами.

Между session и entity bean есть еще одно существенное различие. При подключении к session bean для вызывающего клиента создается отдельный экземпляр такого объекта. При обращении с этому же session bean другого клиента создается еще один экземпляр. Таким образом, каждый клиент использует собственный экземпляр session bean.

При вызове клиентом entity bean создается всего лишь один его экземпляр, и при обращении к нему другого клиента они будут использовать его совместно. Управление всеми entity bean осуществляется контейнером в пуле соединения.

Настройка JBuilder 5 для разработки EJB

Проще всего создать EJB при помощи пакета *Borland* JBuilder 5. Для подключения к Delphi EJB должен быть размещен на сервере приложений *Borland* версии 4.51 или выше. Демонстрационные версии JBuilder 5 и сервера приложений AppServer можно

получить на Web-сайте *Borland*. Сначала обычно устанавливают сервер AppServer, а затем — JBuilder.

Перед запуском JBuilder создайте каталог проектов, в котором будут сохраняться все приложения JBuilder. Обычно для этого каталога используют имя наподобие `c:\MyProjects`.

После запуска JBuilder 5 мастера создания объектов EJB могут оказаться недоступны. В этом случае необходимо настроить пакет JBuilder таким образом, чтобы он был связан с сервером AppServer.

Для настройки поддержки EJB в пакете JBuilder 5 выполните следующие действия:

1. Запустите JBuilder 5 и выберите в меню Tools пункт Enterprise Setup. На экране появится диалоговое окно с параметрами конфигурации CORBA.
2. Во вкладке CORBA выберите пункт VisiBroker. Пакет JBuilder поставляется с брокером VisiBroker для Java.
3. Щелкните на кнопке Edit (Правка). В появившемся на экране диалоговом окне Edit Configuration введите путь к брокеру ORB. Это — путь размещения программы IDL2Java (обычно: `c:\Borland\AppServer\bin`).
4. Перейдите во вкладку Application (Приложение), на которой указывается тип используемого сервера приложений. Выберите тип BAS 4.5 и укажите корректный каталог (`c:\Borland\AppServer`).
5. Выберите в меню Projects (Проекты) пункт Default Project Proprieties (Свойства проекта по умолчанию) и перейдите во вкладку Servers (Серверы). Удостоверьтесь, что выбран именно сервер Borland Application Server. Если это не так, то щелкните на кнопке с многоточием и выберите его.

Итак, настройка JBuilder завершена. Теперь можно приступить к созданию первого EJB.

Разработка простого EJB "Hello, world"

Позаимствуем для первого EJB классическое приложение языка C, которое выводит на экран строку "`Hello, world`". В данном примере будет рассмотрен только процесс создания EJB. Более сложные объекты EJB разрабатываются по такой же схеме, но при этом в них реализуются дополнительные методы интерфейса `Remote`.

Запустите сначала сервер приложений *Borland*, а затем — JBuilder. В настоящей разделе будет рассмотрен процесс разработки EJB и его установки на сервере приложений. Затем с помощью Delphi будет разработана клиентская часть для подключения к созданному объекту. При разработке реальных приложений чаще всего применяется именно такой подход, хотя опытные программисты Java, используя пакет JBuilder, могут разрабатывать и проверять EJB исключительно в нем.

Чтобы создать EJB "`Hello, world`", необходимо выполнить следующее:

1. Закройте в JBuilder все активные проекты, выберите в меню File пункт New Project. Присвойте проекту имя "`HelloWorld`". Это же имя будет использовано для файла проекта при его сохранении на диске.

2. Теперь в проект необходимо добавить группу EJB. Выберите в меню File пункты New и Enterprise, а в появившемся диалоговом окне — пиктограмму Empty EJB Group (Пустая группа EJB). Присвойте этой группе название **`HelloGroup`**. Обратите внимание на поле ввода текста, в котором указывается имя файла с расширением `.jar`. Данный файл предназначен для хранения откомпилированного приложения и по своей сути аналогичен архивному файлу с расширением `.zip`. Все файлы с двоичным кодом Java архивируются в файле JAR. Это — единственный файл, который необходим для установки приложения. Присвойте ему имя `HelloWorld.jar`.
3. Теперь добавьте в проект новый EJB. Для этого выберите в меню File пункты New и Enterprise, а в появившемся диалоговом окне выберите пиктограмму Enterprise Java Bean и укажите имя **`HelloBean`**. JBuilder 5 автоматически создаст новый bean и все необходимые интерфейсы.
4. Отметьте в окне проекта файл `HelloBean.java` и перейдите на вкладку Source (Исходный код). Приведите исходный код объекта в соответствие с тем, который представлен в листинге 19.12.

Листинг 19.12. Исходный код JavaBean

```
package helloworld;

import java.rmi.*;
import javax.ejb.*;
import java.lang.String;

public class HelloBean implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() throws RemoteException {
  }
  public void ejbActivate() throws RemoteException {
  }
  public void ejbPassivate() throws RemoteException {
  }
  public void setSessionContext(SessionContext sessionContext)
throws RemoteException {
    this.sessionContext = sessionContext;
  }
  public String sayHello() {
     return "Hello, world";
  }
}
```

5. Единственное изменение должно быть внесено в последнем методе `sayHello()`. Он возвращает строку, поэтому в начале исходного кода необходимо подключить пакет `java.lang.String`.

6. Теперь с помощью интерфейса Remote необходимо предоставить доступ к методу sayHello(). Для этого перейдите во вкладку Bean, расположенную внизу окна исходного кода. Теперь перейдите на вкладку Methods, расположенную в нижней части вкладки Bean. Метод sayHello() отображен в списке со сброшенным флажком. Установите флажок. В результате метод sayHello() будет связан с интерфейсом Remote (а следовательно, доступен извне). Это можно проверить, дважды щелкнув мышью на файле Hello.java в окне проекта. В результате на экране появится исходный код интерфейса Remote. Обратите внимание на то, что теперь в нем указан метод sayHello().
7. Сохраните и откомпилируйте проект. Ошибок при этом возникнуть не должно.

Разработка в JBuilder демонстрационного клиентского приложения

JBuilder позволяет разработать клиентское приложение Java для проверки нового EJB. Для этого:

1. Выберите в меню File пункты New и Enterprise, а затем пиктограмму EJB Test Client. Присвойте новому приложению имя HelloTestClient1.java.
2. JBuilder автоматически создаст файл с исходным кодом. Перейдите в начало метода Main и внесите изменения, представленные в листинге 19.13.

Листинг 19.13. Клиентское приложение Java для проверки нового EJB

```
/** Метод Main */

  public static void main(String[] args) {
    HelloTestClient1 client = new HelloTestClient1();
    client.create();
    client.sayHello();

    // Клиентский объект используется для вызова оболочки
    // интерфейса Home, позволяющей создать ссылку на объект
    // интерфейса Remote. Если возвращаемое значение соответствует
    // типу интерфейса Remote, то оно может использоваться для
    // доступа к его методам. Кроме того, для обращения к
    // оболочкам интерфейса Remote можно использовать только
    // клиентский объект.
  }
```

В метод Main были добавлены вызовы методов create() и sayHello().

Разработка клиента и проверка EJB

Для разработки клиентской части выполните следующие действия:

1. Выделите в окне проекта элемент HelloGroup, щелкните на нем правой кнопкой мыши и в контекстном меню выберите пункт Run (Выполнить). В результате EJB

будет запущен, о чем будут свидетельствовать сообщения в соответствующем окне JBuilder. Это займет 20 – 30 секунд, в зависимости от частоты процессора и объема оперативной памяти. Не забывайте, что Java требуется много ресурсов.

2. Выделите клиентское приложение и щелкните на нем правой кнопкой мыши. Выберите из контекстного меню пункт Run. В результате приложение будет запущено и на экране появится окно сообщения со строкой "`Hello, world`". Это означает, что EJB работает корректно.

3. Теперь можно дезактивировать группу EJB, щелкнув на красной кнопке Stop, расположенной в нижней части окна сообщений.

Установка EJB на сервер приложений

Чтобы установить EJB на сервере приложений, необходимо выполнить следующие действия:

1. Выберите в меню Tools пункт EJB Deployment (Установка EJB) и выполните действия, предложенные мастером установки EJB.

2. После того как EJB будет установлен на сервер приложений, он будет запускаться автоматически. Щелкните в мастере несколько раз на кнопке Next (Далее), пока не достигнете этапа 4.

3. На четвертом этапе мастера необходимо выбрать контейнер EJB. Убедитесь, что сервер приложений *Borland* запущен, и щелкните на кнопке Add EJB Container (Добавить контейнер EJB). В появившемся диалоговом окне выберите контейнер AppServer и щелкните на кнопке OK. После этого выполните все остальные этапы мастера.

Создание файла SIDL

Компанией *Borland* была разработана собственная методика сопряжения EJB с интерфейсом сервера приложений. Она получила название *упрощенный язык определения интерфейсов* (SIDL – Simplified Interface Definition Language или Simplified IDL). Подобное сопряжение гарантирует, что устаревшие приложения CORBA получат возможность обращаться к EJB, используя стандарт CORBA 2.1 или выше. Этот инструмент, называемый компилятором SIDL, поставляется в комплекте с сервером приложений и позволяет, проанализировав интерфейс `Remote`, создавать соответствующие файлы IDL.

Кроме того, *Borland* предоставляет бесплатное дополнение к пакету JBuilder, которое при помощи компилятора SIDL конвертирует интерфейсы EJB в IDL. Это дополнение находится на прилагаемом компакт-диске и представляет собой файл `ot-SIDL.jar`, расположенный в том же каталоге, что и примеры исходного кода настоящей главы. Скопируйте этот файл в каталог `c:\JBuilder5\lib\ext` (или другой эквивалентный каталог, в котором установлен JBuilder). После этого для активизации дополнения достаточно перезапустить JBuilder.

После запуска JBuilder выберите в меню Tools пункт IDE Options (Параметры среды) и в появившемся диалоговом окне перейдите на вкладку SIDL. Укажите каталог для выходных данных. В данном случае необходимо указать каталог, в котором сохраняются файлы проекта `HelloWorld` (например, **`c:\MyProjects\HelloWorld`**).

Кроме того, новое средство добавило к интерфейсу EJB `Remote` контекстное меню.

Отметьте в окне проекта файл `HelloHome.java` и щелкните на нем правой кнопкой мыши. Теперь в контекстном меню есть пункт Generate Simplified IDL (Создать SIDL). После выбора этого пункта будет запущен компилятор SIDL. Результат его работы будет сохранен в каталоге `classes`.

Разработка в Delphi клиента EJB

Теперь, когда разработка EJB завершена, можно использовать файл IDL, который был создан компилятором SIDL, при разработке на Delphi клиента CORBA, способного взаимодействовать с EJB. В подкаталоге `classes` текущего проекта есть файл `HelloHome.idl`. Кроме этого файла при разработке приложения понадобится копия файла `sidl.idl`, расположенного в каталоге `Delphi6\Demos\Corba\Idl2pas\EJB\EuroConverter`. Создайте новый каталог, скопируйте в него два указанных файла и выполните следующие действия:

1. Запустите Delphi и выберите в меню File пункты New, Other, CORBA, а затем — пиктограмму CORBA Client Application (Клиентское приложение CORBA).
2. Добавьте в список файлов, предназначенных для обработки, файл `HelloHome.idl`. Файл `sidl.idl` в этот список добавлять не нужно, поскольку он будет автоматически подключен в файле `HelloHome.idl` при помощи директивы `include`.
3. Мастер создаст новое клиентское приложение CORBA. Сохраните проект и назовите его **`HelloClient`**. В результате работы мастера будет создано множество файлов. Кроме модуля `Unit1.pas` понадобятся только два файла: `HelloHome_HelloWorld_i.pas` и `HelloHome_HelloWorld_c.pas`. Все остальные файлы можно закрыть.
4. Разместите в главной форме приложения кнопку и метку. Теперь форма должна выглядеть примерно так, как показано на рис. 19.4.

Рис. 19.4. *Клиент EJB в Delphi*

5. В методе формы `OnCreate()` укажите вызов метода **`initCorba`**.
6. В метод `initCorba()` внесите изменения в соответствии с листингом 19.14. В данном случае необходимо добавить две переменные в определение класса. Первая из них соответствует интерфейсу `Home`, а вторая — интерфейсу `Remote`. Интерфейс `Home` используется для создания экземпляров интерфейса `Remote`. После создания экземпляра интерфейса `Remote` можно вызывать методы для обращения к EJB. Итак, в метод `initCorba()` необходимо внести программный код, в котором выполняется привязка к интерфейсу `Home` и создается объект интерфейса `Remote`.
7. Создайте для кнопки обработчик события `OnClick` и внесите в него программный код в соответствии с листингом 19.14.

8. Откомпилируйте клиентское приложение. В случае возникновения ошибок обратитесь к примечанию, представленному ниже.

Листинг 19.14. Главный файл приложения-клиента EJB

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Corba, HelloHome_c, HelloHome_helloworld_c,
  HelloHome_helloworld_i, HelloHome_i, HelloHome_sidl_javax_ejb_c,
  HelloHome_sidl_javax_ejb_i, HelloHome_sidl_java_lang_c,
  HelloHome_sidl_java_lang_i, HelloHome_sidl_java_math_c,
  HelloHome_sidl_java_math_i, HelloHome_sidl_java_sql_c,
  HelloHome_sidl_java_sql_i, HelloHome_sidl_java_util_c,
  HelloHome_sidl_java_util_i, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
  { Закрытые объявления }
  protected
    myHome: HelloHome;
    myRemote: Hello;
    procedure InitCorba;
  { Защищенные объявления }
  public
  { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.InitCorba;
begin
  CorbaInitialize;

  myHome := THelloHomeHelper.Bind;
  myRemote := myHome._create;
end;

procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
  initCorba;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := myRemote.sayHello;
end;

end.
```

НА ЗАМЕТКУ

В данном случае при работе с Delphi 6 могут возникнуть две ошибки. Обе они устранены в пакете обновления Service Pack 1. Первая ошибка проявляется в сообщении компилятора о том, что в один из модулей не может автоматически подключить другой. В этом случае просто укажите соответствующее имя модуля в разделе `uses`.

Вторая ошибка проявляется во время выполнения. Она связана с методом `create()` интерфейса `Home`. Вначале слово *Create* было внесено в список зарезервированных слов компилятора IDL2Pas. В результате, когда компилятор встречает это слово, он помещает перед ним символ подчеркивания (_). Когда объект EJB обращается к методу `_create()`, это приводит к возникновению исключения, потому что метода с таким именем нет.

Для устранения данной ошибки необходимо найти реализацию метода `THelloHomeStub._create()` в файле `HelloHome_helloworld_c.pas` (соответствующий фрагмент кода представлен ниже). Первый параметр в методе `_CreateRequest()` указывает серверу CORBA на вызываемый метод. Если в первом параметре указан метод `_create`, то измените это значение на `create`:

```
function THelloHomeStub._create: HelloHome_helloworld_i.Hello;
var
  _Output: CORBA.OutputStream;
  _Input:  CORBA.InputStream;
begin
  inherited _CreateRequest('create', True, _Output);
  inherited _Invoke(_Output, _Input);
  Result := HelloHome_helloworld_c.THelloHelper.Read(_Input);
end;
```

Обе эти ошибки устранены в первом пакете обновления для Delphi 6.

Запуск приложения

Для запуска приложения необходимо выполнить следующие действия:

1. Запустите OSAgent.
2. Убедитесь в том, что сервер приложений *Borland* (AppServer) также запущен.

3. Запустите приложение `HelloClient`. Щелкните на кнопке. В результате текст метки изменится на "`Hello, world`".

Более сложные EJB можно разработать точно также, на основании менее сложных. В Java, для расширения возможностей EJB, достаточно всего лишь добавить бо́льшее количество методов и интерфейсов. По существу, клиентский процесс останется тем же самым. В Delphi клиент CORBA будет собран компилятором SIDL из файлов IDL, обработанных мастером CORBA интегрированной среды разработки Delphi.

CORBA и Web-службы

Расширить приложение CORBA для использования в архитектуре Web-служб достаточно просто. Согласно спецификации SOAP ссылки на объекты между приложениями передаваться не могут, поэтому для изоляции клиентов SOAP от деталей реализации приложений CORBA необходимо внести небольшие изменения на среднем уровне.

В следующем примере, созданный в предыдущем разделе EJB, используется клиентами SOAP. Структура такого приложения представлена на рис. 19.5.

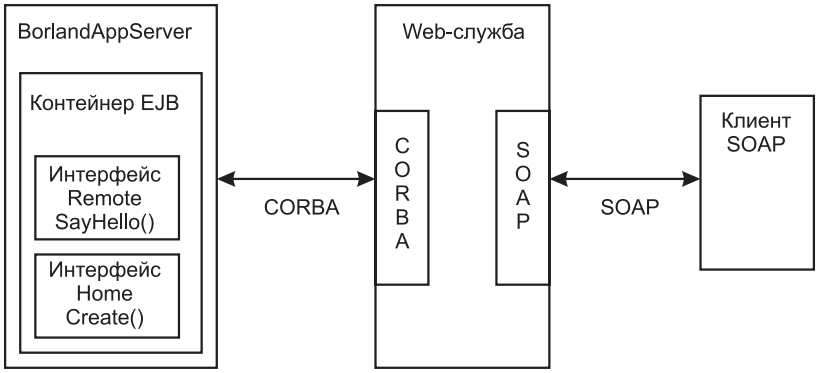

Рис. 19.5. *Пример архитектуры CORBA и Web-служб*

EJB устанавливается на сервере `BorlandAppServer`. Он предоставляет интерфейс CORBA для всех клиентов CORBA. В приложении Web-служб будут одновременно использоваться и сервер SOAP, и клиент CORBA. Участок приложения, который отвечает за взаимодействие с сервером SOAP, будет применять вызовы, основанные на клиентском интерфейсе CORBA.

При таком подходе будут использованы возможности EJB для возвращения результатов тем клиентам, которые могут обращаться только к интерфейсам SOAP. Это означает, что приложение, применяющее спецификацию SOAP, может получать доступ к приложениям, использующим EJB. Благодаря этому возможности клиентских приложений значительно расширяются. Кроме того, такой подход позволяет избежать установки на стороне клиента брокера ORB CORBA.

В данном примере EJB будет использоваться точно тот же, как и в предыдущем разделе. Никаких изменений в его интерфейс и функции вноситься не будет. Эта часть приложения закончена.

Создание Web-службы

Для создания Web-службы будут использоваться файлы IDL из предыдущего раздела. Клиентские файлы можно создать при помощи компилятора DL2Pas, запущенного из командной строки. Создайте каталог для хранения файлов проекта и скопируйте в него файлы `SIDL.idl` и `HelloHome.idl`. Затем откройте окно командной строки и наберите следующее:

```
IDL2Pas HelloHome.idl
```

Компилятор IDL2Pas создаст файлы приложения. В данном случае на стороне клиента CORBA будут использованы два файла: `HelloHome_I.pas` и `HelloHome_C.pas`.

Для построения данного проекта необходимо предварительно установить мастер Invokamatic, который позволяет создавать приложения SOAP всего за пару минут. При регистрации Delphi 6 предоставляется доступ к официальному Web-сайту зарегистрированных пользователей Delphi 6. На этом Web-сайте можно получить дополнительные средства eXtreme Toys, в состав которых входит и мастер Invokamatic. Получите этот мастер и установите его в интегрированной среде разработки.

Теперь создайте приложение Web-службы, выполнив следующие действия:

1. Закройте в Delphi все активные проекты, выберите в меню File пункты New, Other, Web Services и пиктограмму Soap Server Application.
2. В качестве типа приложения Web-сервера выберите Web App Debugger Executable и присвойте ему имя `coHelloWorld` (поле ввода coClass). Каждый раз при повторении этих действий либо используйте уникальные имена, либо по окончании работы с приложением удаляйте его регистрационные данные.
3. Мастер создаст заготовку приложения. Сохраните ее в каталоге проекта, где уже находятся файлы IDL. Присвойте файлу модуля имя `ServerMod.pas`, файлу главной формы — имя `ServerMain.pas`, а файлу приложения — `Server.dpr`.
4. Теперь выберите в меню File пункты New, Other, Web Services и выберите пиктограмму Invokamatic Wizard.
5. На экране появится диалоговое окно с запросом на ввод имени. Введите **`HelloWorldSoap`**. Это имя будет использовано для автоматического именования интерфейсов и файлов. В раскрывающемся списке Invokable Class (Вызываемый класс) выберите значение TInvokable Class. После щелчка на кнопке OK в проект будут добавлены два новых модуля. Первый — это модуль интерфейса, а второй — модуль реализации.
6. Перейдите в модуль интерфейса и добавьте в интерфейс `IHelloWorldSoapIntf` следующий метод:

   ```
   function sayHello: string; stdcall;
   ```

   Дескриптор `stdcall` необходим для обеспечения корректного вызова (в соответствии с соглашением о вызовах).
7. Теперь скопируйте объявление этого метода в модуль реализации класса `THelloWorldSoapIntf` раздела `public`.
8. Поместите курсор в любом месте строки метода и для автоматического завершения реализации класса нажмите комбинацию клавиш <Ctrl+Shift+C>. Чтобы

проверить корректность работы клиента, укажите в качестве результата, возвращаемого методом, следующую строку:

```
result := 'Hello, world';
```

9. Сохраните, откомпилируйте и запустите программу. В момент запуска она зарегистрирует интерфейс метода. Убедитесь, что отладчик Web App Debugger запущен (запускается в меню Delphi Tools). Теперь можно запустить сервер и проверить доступные интерфейсы, щелкнув мышью на поле URL в отладчике Web App Debugger.

Создание клиентского приложения SOAP

Чтобы создать клиентское приложение SOAP, выполните следующие действия:

1. Закройте активный проект, выберите в меню File пункты New и Application.
2. Разместите на форме одну метку и одно поле ввода текста. Добавьте в раздел `uses` модуля формы файлы интерфейсов `SoapHTTPClient` и `HelloWorldSoapIntf`.
3. Объявите переменную `mySoap` типа `IHelloWorldSoap`.
4. В обработчик события `OnClick` кнопки введите код согласно листингу 19.15.
5. Сохраните и откомпилируйте программу.
6. Запустите приложение и щелкните на кнопке Say Hello. После небольшой задержки, связанной с загрузкой серверного приложения, текст метки изменится на "`Hello, world`".

Это — предварительный тест, демонстрирующий взаимодействие клиента и сервера SOAP. Теперь для завершения работы над приложением можно добавить в серверный проект клиентскую часть CORBA.

Листинг 19.15. Класс главной формы клиента SOAP

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, SoapHTTPClient,
  HelloWorldSoapIntf;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Закрытые объявления }
    mySoap: IHelloWorldSoap;
  public
```

```
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
  x: THTTPRio;
begin
  x := THTTPRio.Create(nil);
  x.URL := 'http://localhost:1024/Server.exe/SOAP/';
  mySoap := x as IHelloWorldSoap;
  Label1.Caption := mySoap.sayHello;
end;

end.
```

Добавление в проект Web-службы клиентского кода CORBA

Чтобы добавить клиентские файлы CORBA в проект Web-сервера, выполните следующие действия:

1. Скопируйте файлы `*_i.pas` и `*_c.pas` из каталога клиентского приложения EJB, разработанного в предыдущем разделе. Файл интерфейса представлен в листинге 19.16.

Листинг 19.16. Файл интерфейса SOAP

```
{Модуль объявления общедоступных интерфейсов для IHelloWorldSoap}

unit HelloWorldSoapIntf;

interface

uses
  Types, XSBuiltIns;

type
  IHelloWorldSoap = interface(IInvokable)
    ['{CA738F7B-B111-4F12-BEBD-C2ADDD80C3E2}']
    // Здесь при помощи обычного кода на языке Object Pascal
    // объявляются вызываемые методы. Не забудьте соглашение о
    // вызовах! (используйте stdcall). Например:
    // function Add(const First, Second: double): double; stdcall;
    // function Subtract(const First, Second: double): double;
```

```
    //                                                    stdcall;
    // function Multiply(const First, Second: double): double;
    //                                                    stdcall;
    // function Divide(const First, Second: double): double;
    //                                                    stdcall;
    function sayHello: String; stdcall;
  end;

implementation

uses
  InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(IHelloWorldSoap),
                                '', '');

end.
```

2. Добавьте две открытые переменные в раздел `public` класса `Form1`, как это показано в листинге 19.17. В связи с тем, что эти переменные открытые, они будут доступны всем остальным модулям приложения.
3. Добавьте в модуль главной формы метод `OnCreate()`. Его реализация должна выглядеть согласно коду листинга 19.17.
4. Наконец, внесите изменения в метод `sayHello()` в файле `HelloWorld-SoapImpl.pas` согласно листингу 19.18.
5. Сохраните проект и откомпилируйте сервер.
6. Перед запуском клиента убедитесь в том, что запущены OSAgent и *Borland* AppServer. В приложении после щелчка на кнопке Say Hello появится надпись "`Hello, world`".

Этот достаточно простой пример демонстрирует процесс применения EJB в клиентских приложениях SOAP, что открывает новые возможности использования приложений на платформе J2EE, а также других типов приложений Delphi.

Листинг 19.17. Класс главной формы сервера SOAP

```
unit ServerMain;

interface

uses
  SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Corba,
  HelloHome_helloworld_i, HelloHome_helloworld_c;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Закрытые объявления }
  public
```

```
    { Открытые объявления }
    myHome:   HelloHome;
    myRemote: Hello;
  end;

var
  Form1: TForm1;

implementation

uses ComApp;

{$R *.DFM}

const
CLASS_ComWebApp: TGUID = '{63859D3A-005F-43BB-8E64-85A466D9C364}';

procedure TForm1.FormCreate(Sender: TObject);
begin
  CorbaInitialize;
  myHome   := THelloHomeHelper.bind;
  myRemote := myHome._create;
end;

initialization
  TWebAppAutoObjectFactory.Create(Class_ComWebApp,
                            'coHelloWorld', 'coHelloWorld Object');

end.
```

Листинг 19.18. Класс реализации сервера SOAP

```
{ Модуль объявления общедоступных интерфейсов для IHelloWorldSoap,
реализующий IHelloWorldSoap }

unit HelloWorldSoapImpl;

interface

uses
  HelloWorldSoapIntf, InvokeRegistry, ServerMain;

type
  THelloWorldSoap = class(TInvokableClass, IHelloWorldSoap)
    // Удостоверьтесь, что в интерфейсе IHelloWorldSoap
    // реализованы все вызываемые методы. Затем сохраните этот
    // файл и, нажав комбинацию клавиш <Ctrl+Space>, вызовите
    // CodeInsight(tm) для заполнения этого раздела реализации.
    // Отметьте все объявления интерфейсов для IHelloWorldSoap и
    // нажмите клавишу <Enter>. После вставки объявлений вызовите
    // ClassCompletion(tm), нажав комбинацию клавиш
```

```
    // <Ctrl+Shift+C>, и создайте заглушки реализации.
    function sayHello : String; stdcall;
  end;

implementation

{ THelloWorldSoap }

function THelloWorldSoap.sayHello: String;
begin
//  result := 'Hello, world';  // проверка клиента SOAP
  result := ServerMain.Form1.myRemote.sayHello;
end;

initialization
  InvRegistry.RegisterInvokableClass(THelloWorldSoap);

end.
```

Резюме

В настоящей главе содержится введение в разработку приложений CORBA на Delphi. Вначале рассматривались основы архитектуры CORBA, а также была продемонстрирована разработка достаточно простого приложения Bank. Затем рассматривались более сложные структуры данных.

Затем обсуждались вопросы разработки приложений масштаба предприятия и изучался процесс создания EJB средствами JBuilder 5, установки его на сервере приложений *Borland* и подключения к нему клиента CORBA, разработанного в Delphi.

И, наконец, этот EJB был расширен за счет объединения средств CORBA и Web-служб при помощи протокола SOAP. Такой подход позволяет клиентам, использующим протокол SOAP, обращаться к любому EJB через Web-службы. Причем клиенту вовсе не обязательно заботиться об установке CORBA на сервере. Это открывает новые возможности для модернизации устаревших корпоративных приложений.

ГЛАВА

20

Приложения BizSnap: разработка Web-служб SOAP

В ЭТОЙ ГЛАВЕ...

Зачастую решающим фактором успешной реализации приложения электронной коммерции является *быстрота* его разработки. Заказчик не любит ждать. К счастью, *Borland* позаботилась и об этом: в состав Delphi 6 входит новое средство ускоренной разработки по имени *BizSnap*. BizSnap — это технология, которая, используя протокол SOAP, интегрирует Web-службы и XML в Delphi 6.

Что такое Web-службы?

Borland описывает Web-службы следующим образом:

- Используя в качестве платформы Internet и инфраструктуру Web, Web-службы позволяют установить взаимодействие между приложениями, бизнес-процессами, заказчиками и поставщиками по всему миру, используя при этом стандартные языки и машинно-независимые протоколы Internet.

Обычно распределенные приложения состоят из серверов и клиентов, где серверы обеспечивают клиентам необходимые функциональные возможности. Любое распределенное приложение способно обладать множеством серверов, которые, в свою очередь, могут быть клиентами друг друга. Web-службы представляют собой новый тип серверного компонента, предназначенного для приложений с распределенной архитектурой. Web-службы — это приложения, которые для реализации своих функциональных возможностей используют обычные протоколы Internet.

Поскольку для коммуникации Web-службы используют открытые стандарты, они априори предполагают возможность взаимодействия различных платформ. Например, с точки зрения клиентского приложения, Web-служба, установленная на машине с операционной системой Solaris от *Sun*, будет выглядеть (по всем параметрам) точно так же, как и служба, установленная на машине Windows NT. До появления Web-служб обеспечение подобного взаимодействия было чрезвычайно дорогой и трудоемкой задачей, требующей высокой квалификации исполнителей.

Открытый характер, возможность использовать любые сетевые аппаратные средства и программное обеспечение делает Web-службы мощнейшим и весьма привлекательным инструментом для разработки как локальных сетевых приложений, так и коммерческих бизнес-приложений.

Протокол SOAP

SOAP — это сокращение от *Simple Object Access Protocol* (простой протокол доступа к объектам). SOAP представляет собой упрощенный протокол, используемый для обмена данными в распределенной среде. Он похож на CORBA или DCOM, но обладает меньшим количеством функциональных возможностей, а следовательно, и меньшими непроизводительными затратами. Для обмена информацией, при взаимодействии, SOAP использует документы XML, HTTP или HTTPS. Спецификация SOAP доступна в Internet по адресу `http://www.w3.org/TR/SOAP/`.

Чтобы предоставить пользователю информацию о себе, Web-службы используют специальную разновидность XML, называемую WSDL. WSDL — это сокращение от *Web Services Description Language* (язык описания Web-служб). С помощью WSDL клиентские приложения могут выяснить, на что способна Web-служба, каковы ее функции, где она расположена и как ее вызвать.

Но самым замечательным в BizSnap является то, что вовсе не обязательно изучать все подробности SOAP, XML или WSDL, чтобы создать приложение Web-службы.

В настоящей главе показано, насколько легко и просто создать Web-службу. Затем будет продемонстрировано, как обратиться к этой службе из клиентского приложения.

Разработка Web-служб

Для демонстрации разработки Web-службы используем классический пример конвертора градусов по Фаренгейту в градусы Цельсия. Оформим его как Web-службу.

Web-служба, написанная в Delphi, состоит из трех основных элементов. Первый — это Web-модуль, содержащий несколько компонентов SOAP (рассмотрим их несколько позже). Такой модуль создается автоматически, с помощью мастера SOAP Server Wizard. Два остальных элемента придется создавать самостоятельно. Второй элемент — это реализация класса. Фактически он представляет собой код действий, собственно выполняемых Web-службой. Третьим элементом является интерфейс к данному классу. Интерфейс предоставляет доступ только к тем членам класса, которые должны быть доступны из внешнего мира через эту Web-службу.

Мастер Web-служб расположен во вкладке WebServices хранилища объектов Delphi. Вкладка содержит три элемента. Рассмотрим сначала мастер Soap Server Application. При выборе этого мастера на экране появится диалоговое окно New Soap Server Application (Новое приложение сервера Soap) (рис. 20.1). Тем, кто уже разрабатывал Web-серверы, это диалоговое окно покажется знакомым. Фактически Web-службы представляют собой Web-серверы, обрабатывающие специфические запросы SOAP.

Рис. 20.1. *Диалоговое окно New Soap Server Application*

В данном примере выберем CGI Stand-alone Executable (Автономная исполняемая программа CGI). Щелкните на кнопке OK и мастер создаст новый класс `TWebModule`, как показано на рис. 20.2.

Рассмотрим класс TWebModule

Класс `TWebModule` содержит три компонента:

- `THTTPSoapDispatcher` принимает сообщения SOAP и передает их вызываемому компоненту (invoker), указанному в его свойстве `Dispatcher`.
- `THTTPSoapPascalInvoker` принимает сообщение, переданное компонентом `THTTPSoapDispatcher`. Именно он указывается в свойстве `Dispatcher`. Получив сообщение SOAP, этот компонент интерпретирует его, а затем обращается к *вызываемому* (invokable) интерфейсу, связанному с данным сообщением.

- `TWSDLHTMLPublish` используется для публикации списка документов WSDL, содержащих информацию об интерфейсах, доступных для вызова, что позволяет клиенту, отличному от Delphi, выявлять и использовать методы, доступные через эту Web-службу.

На данном этапе разработки Web-модуль не задействован. Но вызываемый интерфейс необходимо определить и реализовать.

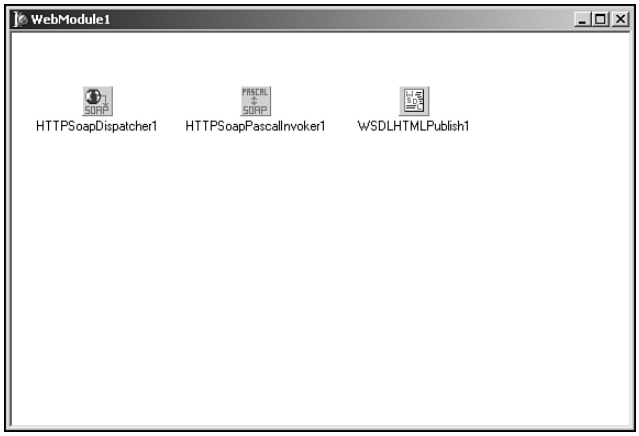

Рис. 20.2. *Созданный мастером Web-модуль*

Определение вызываемого интерфейса

Создайте новый модуль и разместите в нем определение интерфейса. Листинг 20.1 демонстрирует исходный код модуля, созданного для демонстрационного приложения, содержащегося на прилагаемом CD. Этот модуль находится в файле `TempConverterIntf.pas`.

Листинг 20.1. `TempConverter.pas` — определение вызываемого интерфейса

```
unit TempConverterIntf;

interface

type
  ITempConverter = Interface(IInvokable)
    ['{6D239CB5-6E74-445B-B101-F76F5C0F6E42}']
    function FahrenheitToCelsius(AFValue: double): double;
                                                   stdcall;
    function CelsiusToFahrenheit(ACValue: double): double;
                                                   stdcall;
    function Purpose: String; stdcall;
  end;

implementation
```

```
uses
  InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(ITempConverter));

end.
```

Этот небольшой модуль содержит только интерфейс, который определяет методы, подлежащие публикации в качестве функций создаваемой Web-службы. Обратите внимание: данный интерфейс является потомком интерфейса `IInvokable`. Базовый интерфейс `IInvokable` очень прост. Он компилируется с директивой `{M+}`, гарантирующей, что все его потомки будут содержать информацию RTTI. Это позволит Web-службам и клиентам правильно понимать код и символьную информацию, которой они обмениваются друг с другом.

В данном примере определены два метода применяемые для пересчета температуры, а также метод `Purpose()`, который возвращает строку. Обратите внимание, этому интерфейсу был присвоен GUID (глобальный уникальный идентификатор). Чтобы в редакторе кода создать подобную уникальную последовательность, достаточно нажать комбинацию клавиш <Ctrl+Shift+G>.

СОВЕТ

Каждый метод вызываемого интерфейса определен как соответствующий соглашению о вызовах `stdcall`. Это необходимо, в противном случае вызываемый интерфейс работать не будет.

И, наконец, последними из упомянутых элементов являются пользователь модуля `InvokeRegistry` и вызов метода `InvRegistry.RegisterInterface()`. При передаче сообщения SOAP компонент `THTTPSoapPascalInvoker` должен быть способен обнаружить вызываемый интерфейс. Обращение к методу `RegisterInterface()` зарегистрирует интерфейс в *реестре вызовов* (invocation registry). При рассмотрении кода клиента (несколько позже) можно будет заметить, что вызов метода `RegisterInterface()` осуществляется и на клиенте. Регистрация сервера обязательна, поскольку для обращения к интерфейсу необходимо найти его реализацию. На клиенте этот метод используется для того, чтобы позволить компонентам обнаружить информацию о вызываемом интерфейсе и о том, как его вызвать. Расположение вызова функции `RegisterInterface()` в блоке инициализации гарантирует обращение к нему при запуске службы.

Реализация вызываемого интерфейса

Реализация вызываемого интерфейса ничем не отличается от реализации любого другого интерфейса. В листинге 20.2 приведен исходный код интерфейса пересчета температуры.

Листинг 20.2. `TempConverterImpl.pas` — реализация вызываемого интерфейса

```
unit TempConverterImpl;

interface

uses
  InvokeRegistry, TempConverterIntf;

type

  TTempConverter = class(TInvokableClass, ITempConverter)
  public
    function FahrenheitToCelsius(AFValue: double): double;
                                                   stdcall;
    function CelsiusToFahrenheit(ACValue: double): double;
                                                   stdcall;
    function Purpose: String; stdcall;
  end;

implementation

{ TTempConverter }

function TTempConverter.CelsiusToFahrenheit(ACValue: double):
                                                     double;
begin
// Tf = (9/5)*Tc+32
  Result := (9/5)*ACValue+32;
end;

function TTempConverter.FahrenheitToCelsius(AFValue: double):
                                                     double;
begin
// Tc = (5/9)*(Tf-32)

  Result := (5/9)*(AFValue-32);
end;

function TTempConverter.Purpose: String;
begin
  Result := 'Temperature converstions';
end;

initialization
  InvRegistry.RegisterInvokableClass(TTempConverter);
end.
```

Во-первых, обратите внимание на то, что реализация интерфейса является потомком класса `TInvokableClass`. На это существуют две причины. Обе они описаны в интерактивной справочной системе Delphi 6:

- Реестр вызовов (`InvRegistry`) способен создать экземпляры класса `TInvokableClass` (поскольку он имеет виртуальный конструктор), а следовательно, и его потомки тоже. Это позволяет реестру обеспечивать вызываемый интерфейс в приложении Web-службы экземпляром вызываемого класса, который способен обрабатывать входящий запрос.
- Класс `TInvokableClass` представляет собой объект интерфейса, который способен самостоятельно удалить себя из памяти, когда счетчик ссылок на него станет равен нулю. Вызываемые компоненты неспособны определить момент, когда необходимо удалять из памяти экземпляры классов реализации вызываемых ими интерфейсов. Но поскольку на это способен класс `TInvokableClass`, то можно не утруждать себя разработкой собственного механизма контроля срока существования этого объекта.

Кроме того, можно заметить, что класс `TTempConverter` реализует интерфейс `ITempConverter`. Реализация методов пересчета температуры очевидна и объяснений не требует.

В разделе инициализации обращение к методу `RegisterInvokableClass()` регистрирует класс `TTempConverter` в реестре вызовов. Это необходимо осуществить только на сервере, чтобы Web-служба была способна вызвать соответствующую реализацию интерфейса.

Вот и все, что нужно было сделать для создания простой Web-службы. На этом этапе можно откомпилировать Web-службу и поместить полученный в результате файл в исполняемый каталог Web-сервера (IIS или Apache). Как правило, исполняемый каталог (каталог, обладающим правом на исполнение кода) называется `\Scripts` или `\cgi-bin`.

Проверка Web-службы

На компьютере авторов книги для просмотра документа WSDL, созданного разработанной Web-службой, использовался URL `http://127.0.0.1/cgi-bin/TempConvWS.exe/wsdl/ITempConverter`. Сама служба размещалась на сервере Apache. Чтобы получить список всех интерфейсов созданной в Delphi Web-службы, URL должен заканчиваться папкой `wsdl`. Для просмотра конкретного документа WSDL этой службы добавьте имя необходимого интерфейса (например `ITempConverter`). Возникающий в результате документ WSDL представлен в листинге 20.3.

Листинг 20.3. Результирующий документ WSDL, созданный Web-службой

```
<?xml version="1.0" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   name="ITempConverterservice"
   targetNamespace="http://www.borland.com/soapServices/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
- <message name="FahrenheitToCelsiusRequest">
    <part name="AFValue" type="xs:double" />
  </message>
- <message name="FahrenheitToCelsiusResponse">
    <part name="return" type="xs:double" />
```

```
  </message>
- <message name="CelsiusToFahrenheitRequest">
    <part name="ACValue" type="xs:double" />
  </message>
- <message name="CelsiusToFahrenheitResponse">
    <part name="return" type="xs:double" />
  </message>
  <message name="PurposeRequest" />
- <message name="PurposeResponse">
    <part name="return" type="xs:string" />
  </message>
- <portType name="ITempConverter">
-   <operation name="FahrenheitToCelsius">
      <input message="FahrenheitToCelsiusRequest" />
      <output message="FahrenheitToCelsiusResponse" />
    </operation>
-   <operation name="CelsiusToFahrenheit">
      <input message="CelsiusToFahrenheitRequest" />
      <output message="CelsiusToFahrenheitResponse" />
    </operation>
-   <operation name="Purpose">
      <input message="PurposeRequest" />
      <output message="PurposeResponse" />
    </operation>
  </portType>
- <binding name="ITempConverterbinding" type="ITempConverter">
    <soap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http" />
-   <operation name="FahrenheitToCelsius">
      <soap:operation soapAction="urn:TempConverterIntf-
↳ITempConverter#FahrenheitToCelsius" />
-     <input>
        <soap:body use="encoded"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:TempConverterIntf-ITempConverter" />
      </input>
-     <output>
        <soap:body use="encoded"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:TempConverterIntf-ITempConverter" />
      </output>
    </operation>
-   <operation name="CelsiusToFahrenheit">
      <soap:operation
       soapAction="urn:TempConverterIntf-
↳ITempConverter#CelsiusToFahrenheit" />
-     <input>
        <soap:body use="encoded"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:TempConverterIntf-ITempConverter" />
      </input>
-     <output>
        <soap:body use="encoded"
```

```
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:TempConverterIntf-ITempConverter" />
      </output>
    </operation>
-   <operation name="Purpose">
      <soap:operation soapAction="urn:TempConverterIntf-
↳ITempConverter#Purpose" />
-     <input>
        <soap:body use="encoded"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:TempConverterIntf-ITempConverter" />
      </input>
-     <output>
        <soap:body use="encoded"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:TempConverterIntf-ITempConverter" />
      </output>
    </operation>
  </binding>
- <service name="ITempConverterservice">
-   <port name="ITempConverterPort"
     binding="ITempConverterbinding">
      <soap:address
       location="http://127.0.0.1/cgi-bin/TempConvWS.exe
↳/soap/ITempConverter" />
    </port>
  </service>
</definitions>
```

Теперь покажем, насколько просто организовать обращение к Web-службе.

Обращение клиента к Web-службе

Для обращения к Web-службе необходимо знать URL ее документа WSDL. В данном случае это тот же самый URL, который использовался ранее.

Чтобы продемонстрировать указанное, было создано простое приложение с единственной главной формой (рис. 20.3).

Это приложение на самом деле очень простое: пользователь вводит в поле температуру, щелкает на необходимой кнопке и пересчитанное значение отображается в поле Result. Исходный код данного приложения приведен в листинге 20.4.

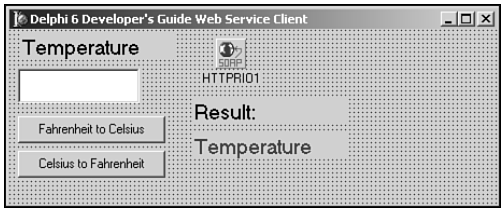

Рис. 20.3. *Главная форма приложения-клиента Web-службы*

Листинг 20.4. Клиент Web-службы

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Rio, SoapHTTPClient;

type
  TMainForm = class(TForm)
    btnFah2Cel: TButton;
    btnCel2Fah: TButton;
    edtArguement: TEdit;
    lblTemperature: TLabel;
    lblResultValue: TLabel;
    lblResult: TLabel;
    HTTPRIO1: THTTPRIO;
    procedure btnFah2CelClick(Sender: TObject);
    procedure btnCel2FahClick(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

uses TempConvImport;

{$R *.dfm}

procedure TMainForm.btnFah2CelClick(Sender: TObject);
var
  TempConverter: ITempConverter;
  FloatVal: Double;
begin
  TempConverter := HTTPRIO1 as ITempConverter;
  FloatVal := TempConverter.FahrenheitToCelsius(
↳StrToFloat(edtArguement.Text));
  lblResultValue.Caption := FloatToStr(FloatVal);
end;

procedure TMainForm.btnCel2FahClick(Sender: TObject);
var
  TempConverter: ITempConverter;
  FloatVal: Double;
begin
  TempConverter := HTTPRIO1 as ITempConverter;
  FloatVal := TempConverter.CelsiusToFahrenheit(StrToFloat(
↳edtArguement.Text));
```

```
  lblResultValue.Caption := FloatToStr(FloatVal);
end;

end.
```

В главной форме размещен компонент THTTPRIO, представляющий собой удаленный вызываемый объект, выступающий в роли локального посредника для Web-службы, которая, весьма вероятно, находится где-нибудь на удаленной машине. Два обработчики события TButton содержат код, осуществляющий вызов удаленных объектов Web-службы. Обратите внимание, что компонент THTTPRIO необходимо привести к типу ITempConverter, это позволит осуществить обращение к его вызываемым методам.

Перед запуском данного кода на выполнение компонент THTTPRIO следует подготовить, что выполняется в несколько этапов.

Создание модуля импорта для удаленного вызываемого объекта

Прежде чем компонент THTTPRIO можно будет использовать, необходимо создать модуль импорта для вызываемого объекта. К счастью, *Borland* существенно упростила этот процесс, создав специальный мастер. Такой мастер доступен на вкладке WebServices хранилища объектов. Вначале его диалоговое окно будет выглядеть так, как показано на рис. 20.4.

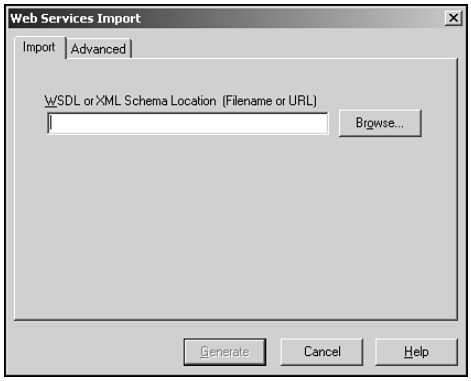

Рис. 20.4. *Мастер Web Services Import*

Чтобы импортировать Web-службу в клиентское приложение, необходимо поместить путь к документу WSDL (уже упомянутый URL) в поле WSDL or XML Schema Location (Filename or URL), а затем щелкнуть на кнопке Generate и модуль импорта будет создан. Модуль импорта для разрабатываемой Web-службы представлен в листинге 20.5. Он выглядит почти точно так же, как и рассмотренный ранее модуль определения интерфейса.

Листинг 20.5. Модуль импорта Web-службы

```
Unit TempConvImport;

interface

uses
  Types, XSBuiltIns;

type

  ITempConverter = interface(IInvokable)
    ['{684379FC-7D4B-4037-8784-B58C63A0280D}']
    function FahrenheitToCelsius(const AFValue: Double): Double;
                                                         stdcall;
    function CelsiusToFahrenheit(const ACValue: Double): Double;
                                                         stdcall;
    function Purpose: WideString; stdcall;
  end;

implementation

uses
  InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(ITempConverter),
                      'urn:TempConverterIntf-ITempConverter', '');

end.
```

Как только все будет закончено, можно вернуться к главной форме клиентского приложения и применить только что созданный модуль импорта. Это позволит главной форме воспользоваться новым интерфейсом.

Использование компонента THTTPRIO

Для компонента `THTTPRIO` должны быть заданы три свойства. Сначала еще раз укажите в свойстве `WSDLLocation` путь к документу WSDL. Как только этот параметр будет установлен, можно будет раскрыть список свойства `Service` и выбрать единственный доступный параметр. Затем проделайте то же самое со свойством `Port`. На данном этапе клиентское приложение можно запустить.

Запуск Web-службы

Теперь, когда все элементы приложения готовы, создайте обработчики события `OnClick` для кнопок. Они должны выглядеть так, как в листинге 20.6.

Листинг 20.6. Обработчики события `OnClick`

```
procedure TMainForm.btnFah2CelClick(Sender: TObject);
var
  TempConverter: ITempConverter;
  FloatVal: Double;
begin
  TempConverter := HTTPRIO1 as ITempConverter;
  FloatVal := TempConverter.FahrenheitToCelsius(StrToFloat(
↳edtArguement.Text));
  lblResultValue.Caption := FloatToStr(FloatVal);
end;

procedure TMainForm.btnCel2FahClick(Sender: TObject);
var
  TempConverter: ITempConverter;
  FloatVal: Double;
begin
  TempConverter := HTTPRIO1 as ITempConverter;
  FloatVal := TempConverter.CelsiusToFahrenheit(StrToFloat(
↳edtArguement.Text));
  lblResultValue.Caption := FloatToStr(FloatVal);
end;
```

При вводе этого кода обратите внимание, что интерактивная подсказка Delphi (CodeInsight) доступна и для Web-службы. Это связано с тем, что Delphi адаптировал Web-службу в состав приложения как базовый объект. В Delphi импликация широко распространена: любая Web-служба, привнесенная в приложение Delphi, независимо от того, устанавливается ли она на Solaris, Windows, Linux или мейнфрейме, и независимо от языка, на каком она написана, будет взаимодействовать с ним. Благодаря настолько плотной интеграции, кроме интерактивной подсказки, приложение, написанное для работы с Web-службой, получает также механизм проверки типов и все возможности для отладки.

Резюме

Web-службы — это новый мощнейший инструмент, использующий открытые стандарты и существующую инфраструктуру для разработки корпоративных приложений независимых ни от платформ, ни от операционных систем.

В настоящей главе было рассказано, как создать простую Web-службу и клиент для нее. Были рассмотрены действия, которые необходимо предпринять для установки этой службы на сервер и правильного подключения компонента `THTTPRIO` клиента. Указанного вполне достаточно, чтобы приступить к разработке Web-служб повышенной сложности. Более подробная информация по этой теме, а также примеры Web-служб находятся на сайте сообщества *Borland*, который мы настоятельно рекомендуем посетить. Кроме всего прочего, там находится статья Дэниела Полищука "*Managing Sessions with Delphi 6 Web Services*" (идентификатор: 27575), посвященная этой теме. Она доступна по адресу `http://community.borland.com/article/0,1410,27575,00.html`.

ГЛАВА

21

Разработка приложений DataSnap

В ЭТОЙ ГЛАВЕ...

О *многоуровневых приложениях* (multitier application) сегодня говорят так же много, как и о любой другой области программирования. На это существуют свои причины. Многоуровневые приложения имеют массу преимуществ перед традиционными приложениями клиент/сервер. Технология DataSnap компании *Borland* предоставляет в распоряжение разработчика один из способов создания многоуровневых приложений, основанный на приемах и опыте, полученных при изучении Delphi. В настоящей главе можно ознакомиться с общей информацией о разработке многоуровневых приложений, а также увидеть, как можно использовать все эти принципы для создания полнофункциональных приложений DataSnap.

Механизм построения многоуровневого приложения

Поскольку в этой главе речь пойдет о многоуровневых приложениях, вначале будет полезно разобраться в том, что же на самом деле представляет собой уровень. *Уровень* (tier) в данном случае является слоем приложения, обеспечивающим некоторый специфический набор функций. В приложениях баз данных можно выделить три основных уровня.

- *Уровень данных* (data). Уровень данных отвечает за хранение данных. Как правило, на этом уровне используется такие *системы управления реляционными базами данных* (RDBMS — Relational Database Management System), как Microsoft SQL Server, Oracle или InterBase.
- *Бизнес-уровень* (business). Этот уровень предназначен для получения данных с уровня данных в формате, соответствующем приложению, и выполнения окончательной проверки данных. Данный уровень также известен как *уровень бизнес-правил* (enforcing business rules) Именно на этом уровне находятся сервера приложений.
- *Уровень представления* (presentation). Этот уровень известен также как уровень графического интерфейса пользователя (GUI tier) — именно на данном уровне осуществляется отображение данных в формате, отвечающем клиентскому приложению. Уровень представления всегда взаимодействует с уровнем бизнес-правил и никогда не связывается напрямую с уровнем данных.

В традиционных приложениях клиент/сервер используется архитектура, подобная представленной на рис. 21.1. Обратите внимание: библиотеки доступа к данным клиента должны располагаться на каждой машине клиента. Исторически сложилось так, что этот момент всегда являлся камнем преткновения при установке приложений клиент/сервер — из-за несовместимости версий используемых библиотек DLL. Кроме того, поскольку большинство бизнес-правил реализовано в клиентской части приложения, при их обновлении приходится изменять программный продукт на каждой клиентской машине.

В многоуровневых приложениях используется архитектура, показанная на рис. 21.2. Применение такой архитектуры сулит множество преимуществ по сравнению с аналогичным приложением на базе архитектуры клиент/сервер.

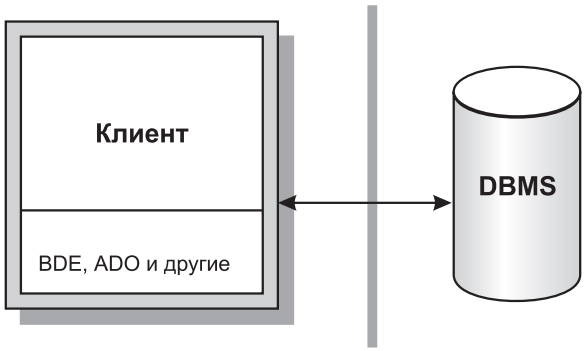

Рис. 21.1. *Традиционная архитектура клиент/сервер*

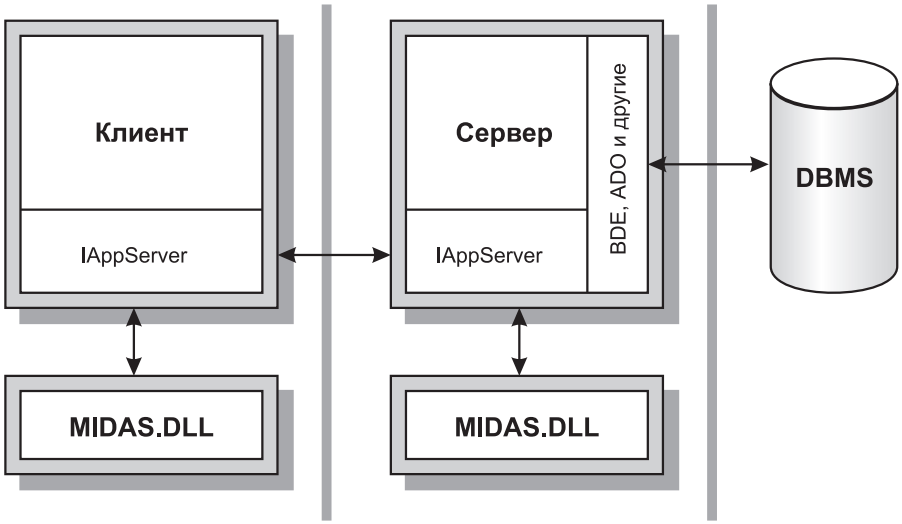

Рис. 21.2. *Многоуровневая архитектура*

Преимущества многоуровневой архитектуры

В следующих разделах будут рассмотрены основные преимущества многоуровневой архитектуры.

Централизованная бизнес-логика

В большинстве приложений клиент/сервер каждому клиентскому приложению для решения поставленной задачи необходимо опираться на собственные бизнес-правила. Это приводит не только к увеличению размера исполняемой части, но и вынуждает раз-

работчика программного обеспечения осуществлять строгий контроль за совместимостью версий. Если пользователь A использует более раннюю версию приложения, чем пользователь B, то бизнес-правила могут выполняться несогласованно, вследствие чего в данных возникают логические ошибки. Реализация бизнес-правил на уровне серверного приложения требует создания и поддержки лишь одной его копии, поскольку любой пользователь, работающий с серверным приложением, будет использовать одну и ту же копию бизнес-правил. В приложениях клиент/сервер некоторые проблемы могут быть решены средствами RDBMS, но не все они обеспечивают одинаковые возможности. Кроме того, применение хранимых процедур делает приложения менее переносимыми. При использовании многоуровневого подхода бизнес-правила не зависят от RDBMS, что упрощает решение задачи обеспечения независимости баз данных.

Архитектура "тонкого" клиента

Помимо упомянутых бизнес-правил, типичное приложение клиент/сервер реализует большинство функций уровня доступа к данным. Это приводит к увеличению размера клиентской исполняемой части приложения, получившей название *"толстого" клиента* (fat client). В частности, приложения баз данных Delphi, предоставляющие доступ к SQL-серверу, на клиентской машине нуждаются в наличии BDE, драйверов SQL Links и/или ODBC, а также клиентских библиотек, необходимых для взаимодействия с сервером. После установки все эти файлы нужно соответствующим образом настроить, что существенно усложняет процесс установки приложения. При использовании технологии DataSnap доступ к данным контролируется серверной частью приложения, тогда как клиентской частью обеспечивается лишь представление данных. Это означает, что на клиентской машине достаточно установить лишь клиентское приложение и одну библиотеку DLL, обеспечивающую взаимодействие клиента с сервером. В этом и заключается особенность архитектуры *"тонкого" клиента* (thin-client).

Автоматическое согласование ошибок

Delphi обладает встроенным механизмом, облегчающим обработку ошибок доступа к данным. В многоуровневых приложениях согласование ошибок необходимо по тем же причинам, что и при использовании кэширования обновлений. Данные копируются на машину клиента, где и осуществляется их изменение. Несколько клиентов могут одновременно работать с одной и той же записью. Механизм согласования ошибок доступа к данным помогает пользователю определить, что следует сделать с записями, измененными со времени их последней загрузки с сервера. Согласно общей идеологии Delphi, если предоставляемых стандартных возможностей недостаточно, то их можно расширить или создать свои собственные.

Модель "портфеля"

Модель "портфеля" (briefcase model) основана на концепции обычного портфеля. В него можно поместить важные документы, переносить их с места на место и извлекать при необходимости. Delphi предоставляет возможность упаковывать все данные и сохранить их на диске портативного компьютера, что позволит работать с ними, находясь в пути, без реального соединения с серверной частью приложения или сервером базы данных.

Отказоустойчивость

Если компьютер, выполняющий функции сервера, станет в какой-то момент времени недоступным, то было бы неплохо иметь возможность динамически переключиться на резервный сервер без повторной компиляции клиентских и серверных приложений. Delphi обеспечивает и такую возможность.

Балансировка загрузки

При установке клиентского приложения на все возрастающем количестве пользовательских машин неизбежно возникновение проблемы превышения предельно допустимой нагрузки на сервер. Существует два способа балансировки сетевого трафика: статический и динамический. При статической балансировке нагрузки просто подключается еще одна серверная машина и половина пользователей переключается на нее, тогда как другая половина по-прежнему будет работать с исходным сервером. Однако при этом трудно обеспечить действительно равномерное распределение нагрузки между серверами, поскольку интенсивность работы с серверами разных групп пользователей различна. При использовании динамической балансировки нагрузки каждому клиентскому приложению можно указать конкретный сервер, к которому оно должно обращаться. Существует много различных алгоритмов динамической балансировки нагрузки (например случайный, последовательный, с минимизацией пользователей сети и с минимизацией сетевого трафика). Начиная с версии 4, Delphi обладает компонентом, позволяющим реализовать последовательную балансировку нагрузки серверов.

Типичная архитектура приложения DataSnap

На рис. 21.3 показано, как выглядит типичное приложение DataSnap после его создания. Центральным элементом является *модуль данных* (DM — Data Module). *Модуль удаленных данных* (RDM — Remote Data Module) — это потомок классического модуля данных, появившегося еще в Delphi версии 2. Подобный модуль данных представлял собой специальный контейнер (форму), в котором можно было размещать только невизуальные компоненты. Удаленный модуль данных в этом смысле ничем не отличается от своего предшественника. Кроме того, модуль RDM является объектом COM, или, если сказать более точно, объектом автоматизации. Службы, экспортируемые модулем RDM, будут доступны на всех клиентских машинах.

Рассмотрим некоторые параметры, доступные при создании модуля RDM. На рис. 21.4 показано диалоговое окно, возникающее при выборе в меню File пунктов New и Remote Data Module.

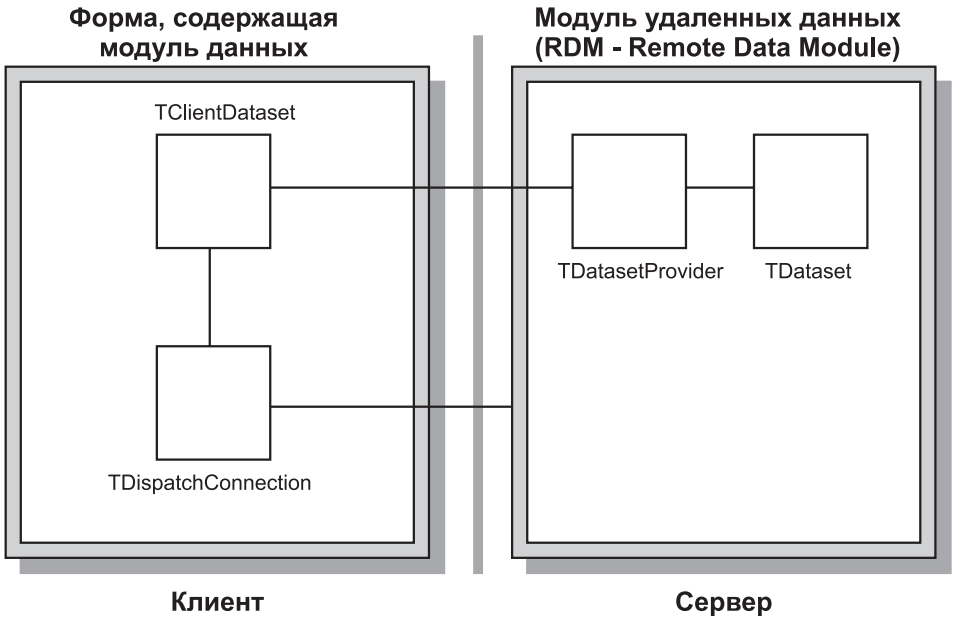

Рис. 21.3. *Типичное приложение DataSnap*

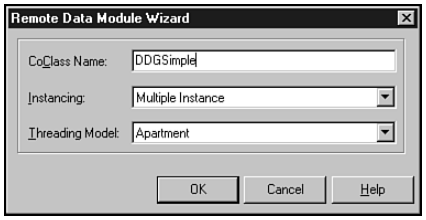

Рис. 21.4. *Диалоговое окно мастера создания нового модуля удаленных данных*

Сервер

Теперь, рассмотрев схему типичного приложения DataSnap в целом, познакомимся с процессом его создания в Delphi. Сначала изучим некоторые параметры, доступные при настройке сервера.

Способы создания экземпляров

Способ создания экземпляров определяет, сколько копий процесса сервера будет запущено. На рис. 21.5 показано, как выбранный параметр управляет поведением сервера.

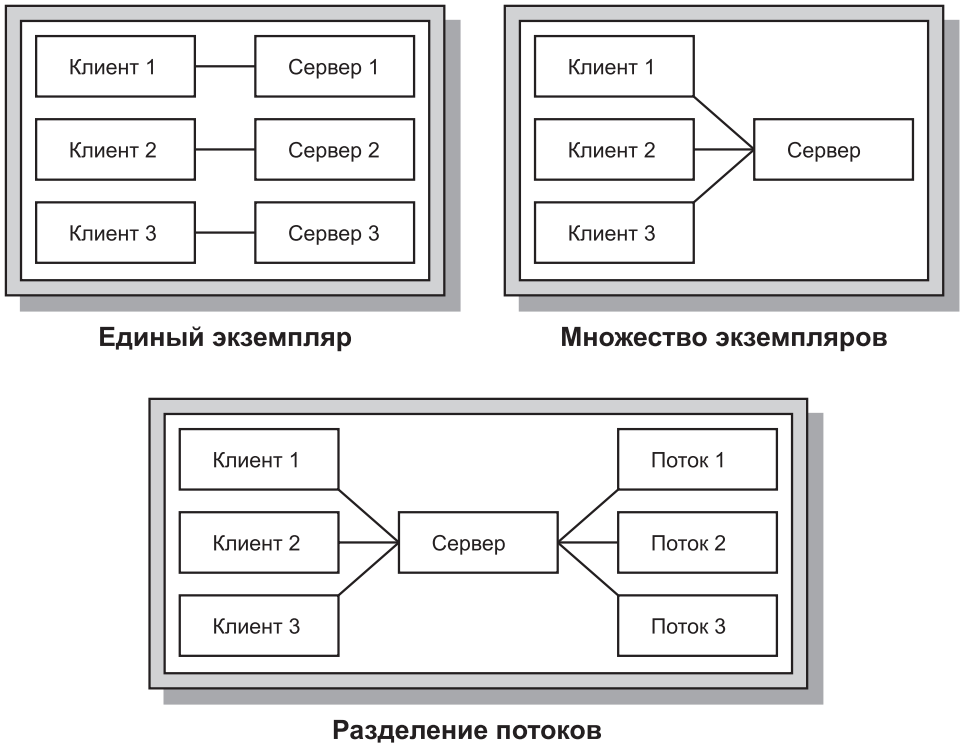

Рис. 21.5. *Поведение сервера в зависимости от способа создания экземпляров*

Для сервера COM доступны следующие способы создания экземпляров модулей удаленных данных.

- `ciSingleInstance`. Каждый клиент, получая доступ к серверу COM, использует отдельный экземпляр сервера. Это означает, что каждый клиент потребляет ресурсы отдельно загружаемого экземпляра сервера. Такой подход обеспечивает параллельный доступ клиентов. Если решено выбрать этот параметр, то не забывайте о существующих ограничениях ядра BDE, которые могут сильно снизить привлекательность данного варианта. Так, ядро BDE 5.01 позволяет создавать не более 48 процессов на одной машине. Поскольку каждый клиент порождает новый процесс сервера, то с одним сервером одновременно установить соединение могут не более 48 клиентов.
- `ciMultiInstance`. Каждый клиент, осуществляющий доступ к серверу COM, использует один и тот же экземпляр сервера. Обычно это означает, что клиент вынужден ожидать, пока предыдущий клиент не освободит сервер COM. Более подробная информация о влиянии значений параметра Threading Choices на поведение сервера приведена в следующем разделе. Это решение эквивалентно последовательному доступу клиентов к серверу. Все клиенты используют одно и то же соединение с базой данных, поэтому свойство `TDataBase.HandleShared` должно содержать значение `True`.

- `ciInternal`. Сервер COM не может быть создан из внешних приложений. Этот подход оказывается полезным, если необходимо управлять доступом к объекту COM с помощью промежуточного *уровня прокси* (proxy layer).

Обратите внимание, что настройка объекта DCOM непосредственно влияет на режим создания экземпляров. Более подробная информация по этой теме приведена далее в настоящей главе.

Выбор модели потоков

Поддержка потоков в Delphi 5 претерпела коренные изменения. В версии 4 выбор модели потоков для сервера EXE не имел особого смысла. В системном реестре просто устанавливался флажок, сообщающий подпрограммам COM, что функции данной библиотеки DLL могут выполняться с использованием указанной модели потоков. В Delphi 5 и 6 выбор модели потоков применим и для серверов EXE, что позволяет подпрограммам COM распределять соединения по отдельным потокам без необходимости применения дополнительного кода. Для модулей RDM допустимы следующие модели потоков.

- *Одиночная* (single). Выбор этой модели означает, что сервер одновременно может обрабатывать лишь один запрос. При использовании одиночной модели нет необходимости заботиться об управлении потоками, поскольку действия сервера ограничены одним потоком, а необходимую синхронизацию сообщений осуществляют подпрограммы COM. Однако выбор такой модели окажется неудачным, если планируется создать многопользовательскую систему, поскольку клиент B, прежде чем начать свою работу, будет вынужден ожидать, пока не завершится обработка запроса клиента A. Этот подход чаще всего оказывается абсолютно неприемлемым, поскольку нет никаких гарантий, что клиент A не занят в данный момент формированием ежедневного отчета или не выполняет какие-либо другие операции, требующие значительных затрат времени.
- *Раздельная* (apartment). Выбор этой модели потоков дает самые высокие результаты в сочетании с присвоением параметру способа создания экземпляров значения `ciMultiInstance`. В этом случае, благодаря параметру `ciMultiInstance`, все клиенты будут совместно использовать один процесс сервера, однако при этом работа одного клиента с сервером не будет блокировать работу другого клиента, поскольку каждый из них будет работать с собственным потоком сервера. При выборе данной модели потоков гарантируется, что данные экземпляра модуля RDM будут использоваться в безопасном режиме. Однако при этом необходимо защитить доступ к глобальным переменным, для чего можно воспользоваться любым из существующих способов синхронизации потоков — например методом `PostMessage()`, критическими секциями, мьютексами, семафорами или классом-оболочкой Delphi `TMultiReadExclusiveWriteSynchronizer`. Именно эту модель потоков предпочтительнее использовать при работе с наборами данных BDE. Обратите внимание, что при использовании такой модели потоков наряду с наборами данных BDE потребуется поместить в модуль RDM компонент `TSession` и установить значение его свойства `AutoSessionName` равным `True`. В этом случае BDE сможет адаптироваться к требованиям используемой модели потоков.

- *Свободная* (Free). Эта модель обеспечивает еще бо́льшую гибкость, позволяя передавать серверу несколько запросов клиентов. Но при использовании такой модели необходимо обеспечить защиту от конфликтов потоков всех данных — как данных экземпляра, так и глобальных переменных. Эту модель предпочтительнее использовать вместе с объектами ADO.
- *Обе модели* (Both). Эта модель так же эффективна, как и свободная модель, но за одним исключением: при ее использовании обратные вызовы преобразуются в последовательную форму автоматически.

Выбор способа доступа к данным

В Delphi 6 Enterprise реализовано несколько различных способов доступа к данным. В частности, по-прежнему поддерживается BDE, благодаря чему можно использовать компоненты, производные от класса `TDBDataSet`, такие как `TTable`, `TQuery` и `TStoredProc`. Однако dbExpress обеспечивает более гибкую архитектуру для доступа к данным. Кроме того, теперь можно воспользоваться средствами технологии ADO и напрямую связываться с сервером InterBase при помощи компонентов класса `TDataSet`.

Публикация служб

RDM отвечает за определение перечня служб, доступных клиенту. Если компонент `TQuery` модуля удаленных данных необходимо сделать доступным клиенту, то наряду с ним в RDM необходимо поместить и компонент `TDataSetProvider`. Затем компонент `TDataSetProvider` связывается с компонентом `TQuery` через свойство `TDataSetProvider.DataSet`. Позже, когда клиенту потребуется использовать данные компонента `TQuery`, в этом ему поможет вновь созданный компонент `TDataSetProvider`. Чтобы указать, какие из провайдеров сервера должен видеть клиент, установите значение их свойства `TDataSetProvider.Exported` равными `True` или `False`.

В то же время, если клиенту не нужен весь набор данных сервера, а требуется лишь осуществить вызов метода, то можно обеспечить и такую возможность. Переместив фокус ввода на модуль удаленных данных, выберите в меню Edit пункт Add to Interface и введите в раскрывшемся диалоговом окне прототип стандартного метода. После обновления библиотеки типов можно будет создать в коде реализацию этого метода.

Клиент

Завершив построение сервера, можно приступить к созданию клиента, который будет использовать службы, предоставляемые сервером. Рассмотрим параметры, доступные при построение клиента DataSnap.

Выбор соединения

Иерархия классов Delphi для соединения клиента с сервером начинается с класса `TDispatchConnection`. Этот базовый объект является родительским для всех типов соединений, которые будут рассмотрены ниже. В случае, если тип соединения не играет существенной роли, то речь будет идти просто об объекте класса `TDispatchConnection`..

Класс `TDCOMConnection` обеспечивает базовую защиту и аутентификацию, используя стандартную реализацию соответствующих служб в Windows. Этот тип соединения

особенно полезен при использовании приложений в intranet и extranet (когда его применяют пользователи в пределах одного домена). В рамках технологии DCOM можно использовать раннее связывание, а также обратные вызовы и компоненты `ConnectionPoints` (обратные вызовы можно применять и при использовании сокетов, однако в этом случае придется ограничиться поздним связыванием). К недостаткам использования такого типа соединения можно отнести следующие.

- Во многих случаях усложняется настройка.
- Этот тип соединения плохо согласуется с концепцией брандмауэров.
- Для компьютеров под управлением Windows 95 требуется установка поддержки DCOM95.

Гораздо проще настроить соединение, предоставляемое компонентом `TSocketConnection`. Кроме того, оно использует лишь один порт для всего трафика DataSnap, поэтому настроить брандмауэр администратору будут существенно проще, чем при работе с DCOM. Для поддержки такой схемы работы потребуется запустить ScktSrvr (находящуюся в каталоге `<Delphi>\BIN`), т.е. для работы соединения на сервере потребуется лишь один дополнительный файл. В Delphi 4 также требовалось наличие пакета WinSock2, что для пользователей Windows 9x означает установку дополнительного программного обеспечения. Однако, если приложение работает в Delphi 6, причем обратные вызовы не используются, то можно установить значение свойства `TSocketConnection.SupportCallbacks` равным `False`. В этом случае на клиентских машинах достаточно использовать пакет WinSock1.

Начиная с версии Delphi 4 стало возможным использование компонента `TCORBAConnection`. Это эквивалент DCOM в рамках открытого стандарта, включающего в себя множество функциональных возможностей для осуществления в создаваемом приложении автоматического поиска, обеспечения отказоустойчивой архитектуры и автоматической балансировки загрузки. Применение CORBA окажется неизбежным при создании независимых ни от платформы, ни от языка приложений DataSnap, использующих соединения.

Впоследствии появился компонент `TWebConnection`. Этот компонент соединения позволяет поддерживать взаимодействие составных частей приложения на основе протокола HTTP или HTTPS. На использование такого типа соединения накладываются следующие ограничения.

- Не поддерживаются обратные вызовы любого типа.
- На машине клиента должна быть установлена библиотека `WININET.DLL`.
- На серверной машине должен быть установлен сервер Internet. Либо служба MS Internet Information Server (IIS) версии 4.0, либо сервер Netscape версии не ниже 3.6.

Но эти ограничения являются оправданными, если приложение работает через Internet или должно проходить брандмауэр, которым вы не можете управлять.

В Delphi 6 появился новый тип соединения: `TSOAPConnection`. Его поведение аналогично `TWebConnection`, но применяется для соединения с Web-службой DataSnap. В отличие от применения других компонентов соединения DataSnap, здесь нельзя использовать свойство `TSoapConnection.AppServer` для вызова тех методов интерфейса сервера приложения, которые не являются методами `IAppServer`. Тут, вместо под-

ключения с помощью интерфейса приложения к модулю данных SOAP, используется отдельный объект THTTPRIO.

Помните, что использование всех этих транспортных компонентов подразумевает корректную установку в системе протокола TCP/IP. Существует лишь одно исключение, когда посредством DCOM связываются два компьютера под управлением Windows NT. В этом случае для определения типа используемого протокола DCOM нужно запустить утилиту DCOMCNFG, а затем переместить требуемый протокол в верхнюю часть списка, расположенного во вкладке Default Protocols. DCOM для Windows 9x поддерживает лишь протокол TCP/IP.

Подключение компонентов

Из рис. 21.3 видно, что составные части приложения DataSnap взаимодействуют через границы уровней. В настоящем разделе рассматриваются основные свойства и компоненты, благодаря которым клиент может взаимодействовать с сервером.

Для обеспечения взаимодействия клиента с сервером необходимо использовать один из потомков класса TDispatchConnection, рассмотренных в предыдущем разделе. У каждого компонента имеются свойства, присущие только данному типу соединения, однако все они позволяют определить, где найти серверное приложение. Класс TDispatchConnection представляет собой аналог компонента TDatabase при использовании в приложениях клиент/сервер, поскольку он определяет канал связи с внешней системой и работает как трубопровод для всех остальных компонентов при их взаимодействии с элементами этой системы.

Установив соединение с сервером, необходимо выбрать способ использования служб, предоставляемых сервером. Для этого в клиентскую часть приложения следует поместить компонент TClientDataSet и связать его с компонентом TDispatchConnection. После этого в списке свойства ProviderNames можно будет просматривать перечень всех экспортируемых сервером провайдеров. Таким образом, в приложениях DataSnap компонент TClientDataSet функционально аналогичен компоненту TTable в приложениях клиент/сервер.

С помощью свойства TDispatchConnection.AppServer можно также вызывать пользовательские методы, существующие на сервере. Например, в следующей строке кода на сервере вызывается функция Login, которой передаются два строковых параметра, а возвращаемое значение имеет тип Boolean:

```
LoginSucceeded := DCOMConnection1.AppServer.Login(UserName,
                                                  Password);
```

Использование DataSnap для создания приложений

Теперь, ознакомившись со множеством параметров, доступных при разработке приложений с использованием технологии DataSnap, попробуем применить их на практике для разработки реального приложения.

Установка сервера

Сначала познакомимся с механизмом построения серверного приложения, а затем обсудим, как создается клиентская часть приложения.

Модуль удаленных данных (RDM)

Основу серверного приложения составляет модуль удаленных данных. Чтобы создать модуль RDM для нового приложения, выберите пиктограмму Remote Data Module во вкладке Multitier диалогового окна New Items (меню File пункт New). Появится диалоговое окно, в котором можно выполнить начальную настройку некоторых параметров модуля удаленных данных.

Наиболее важным параметром является имя модуля RDM, поскольку идентификатор `ProgID` для данного приложения сервера будет построен с использованием имени проекта и имени модуля удаленных данных. Например, если проект (файл `.DPR`) имеет имя `AppServer`, а удаленный модуль данных — `MyRDM`, то идентификатор `ProgID` будет иметь вид `AppServer.MyRDM`. Убедитесь, что для сервера корректно выбран способ создания экземпляров и модель взаимодействия потоков.

Поскольку компоненты `TSocketConnection` и `TWebConnection` обходят стандартный процесс аутентификации Windows, необходимо обязательно удостовериться в том, что на сервере запускаются именно те объекты, которые были определены. Для этого можно поместить в системный реестр определенные значения. Тем самым приложению DataSnap будут переданы сведения о том, какие именно объекты предполагается запускать. К счастью, все, что необходимо для этого сделать, — так это переопределить метод класса `UpdateRegistry`. При создании модуля удаленных данных Delphi автоматически создает реализацию данного метода, показанную в листинге 21.1.

Листинг 21.1. Метод `UpdateRegistry` для класса модуля удаленных данных

```
class procedure TDDGSimple.UpdateRegistry(Register: Boolean;
                                  const ClassID, ProgID: string);
begin
  if Register then begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end else begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
```

Этот метод вызывается каждый раз при регистрации или отмене регистрации сервера. Кроме записей в системном реестре, соответствующих объектам COM и создаваемых при вызове унаследованного метода `UpdateRegistry`, можно вызывать также методы `Enable`*`XXX`*`Transport` и `Disable`*`XXX`*`Transport`. Причем объект будет помечен как защищенный.

НА ЗАМЕТКУ

Компонент `TSocketConnection` отображает в свойстве `ServerName` лишь зарегистрированные защищенные объекты. Если обеспечивать безопасность не нужно, сбросьте в меню Connections утилиты `SCKTSRVR` флажок Registered Objects Only.

Провайдеры

Поскольку основной задачей сервера приложений является предоставление данных клиенту, необходимо обеспечить возможность передачи данных с сервера в формате, понятном клиенту. К счастью, реализация технологии DataSnap в Delphi обладает компонентом `TDataSetProvider`, который существенно упрощает решение этой задачи.

Сначала поместите в удаленный модуль данных компонент `TQuery`. Если в приложении используется реляционная база данных, то потребуется также и компонент `TDatabase`. Теперь нужно связать компоненты `TQuery` и `TDatabase`, а затем определить в свойстве `SQL` простой запрос — например `select * from customer`. И, наконец, поместите в модуль удаленных данных компонент `TDataSetProvider` и свяжите его с компонентом `TQuery` через свойство `DataSet`. Свойство `Exported` объекта `DataSetProvider` определяет, будет ли этот провайдер видимым для клиентов. С помощью этого свойства можно легко управлять видимостью провайдеров и во время выполнения приложения.

НА ЗАМЕТКУ

Хотя в данном разделе основное внимание уделяется компоненту `TDBDataSet`, ориентированному на работу с BDE, этими же принципами можно руководствоваться при организации доступа к данным с помощью любого другого компонента, производного от компонента `TDataSet`. В эту категорию входят также компоненты dbExpress, ADO и InterBase Express, рассмотрение которых выходит за рамки настоящей книги.

Регистрация сервера

После создания приложения сервера его необходимо зарегистрировать как объект COM, что сделает его доступным клиентским приложениям. Способы регистрации, обсуждавшиеся в главе 15, “Разработка приложений COM”, вполне применимы и в случае серверов DataSnap. Достаточно запустить приложение сервера — и требуемый параметр будет добавлен в системный реестр. Но, прежде чем регистрировать сервер, необходимо сохранить проект. Это гарантирует, что, начиная с данного момента, будет использоваться корректный идентификатор `ProgID`.

Если приложение необходимо запустить лишь для регистрации, а не для реальной работы, то при запуске задайте в командной строке параметр `/regserver`. При этом будет выполнен процесс регистрации, после чего приложение немедленно завершится. Чтобы удалить параметры системного реестра, связанные с данным приложением, укажите в командной строке параметр `/unregserver`.

Создание клиента

Теперь, когда существует работоспособное серверное приложение, рассмотрим основные задачи, выполняемые при создании клиента. Обсудим, как возвращать данные, как их редактировать, как обновлять базу данных с учетом изменений, выполненных клиентом, и, наконец, как обработать ошибки, возникающие в процессе обновления.

Возвращение данных

В процессе работы приложения баз данных необходимо постоянно передавать данные от сервера клиенту для их редактирования. Помещение данных в локальный кэш снижает нагрузку на сеть и минимизирует время выполнения транзакций. В предыдущих версиях Delphi для решения этой задачи приходилось использовать кэширование обновлений. В целом аналогичные решения применяются и в приложениях DataSnap.

Клиент взаимодействует с сервером через компонент `TDispatchConnection`. Эту задачу можно легко решить, поместив в компонент `TDispatchConnection` имя компьютера, на котором расположен сервер. При использовании компонента `TDCOMConnection` можно задать полностью квалифицированное имя домена (например `nt.dmiser.com`), IP-адрес компьютера (к примеру `192.168.0.2`) или его имя в NetBIOS (например `nt`). В протоколе DCOM существует ошибка, поэтому имя `localhost` использовать не рекомендуется. При использовании компонента `TSocketConnection` в свойстве `Address` задаются IP-адреса, а в свойстве `Host` — полностью квалифицированное имя домена. О параметрах компонента `TWebConnection` поговорим немного позже.

Задав местоположение приложения-сервера, необходимо предоставить компоненту `TDispatchConnection` способ его идентификации. Это можно сделать с помощью свойства `ServerName`. При установке значения свойства `ServerName` автоматически заполняется свойство `ServerGUID`, которое является очень важным. В самом общем случае, если клиентское приложение подлежит распространению, следует удалить значение из свойства `ServerName`, оставив значение лишь в свойстве `ServerGUID`.

НА ЗАМЕТКУ

При использовании компонента `TDCOMConnection` в свойстве `ServerName` будет отображаться список серверов, зарегистрированных на *данном* компьютере. Однако компонент `TSocketConnection` является более "интеллектуальным" и отображает список серверов, зарегистрированных на удаленной машине.

Наконец, чтобы реально установить соединение с приложением сервера, установите значение свойства `TDispatchConnection.Connected` равным `True`.

Теперь, когда клиент соединен с сервером, необходимо определить способ использования провайдера сервера. Для этого воспользуйтесь компонентом `TClientDataSet`. Данный компонент применяется для связи с провайдером (а значит, и с компонентом `TQuery`, который тоже связан с провайдером) на сервере.

Прежде всего необходимо связать компонент `TClientDataSet` с компонентом `TDispatchConnection`, установив соответствующее значение свойства `Remote-`

Server компонента TClientDataSet. В результате в свойстве ProviderName будет содержаться список доступных провайдеров выбранного сервера.

С этого момента все готово, чтобы открыть *набор данных клиента* ClientDataSet (CDS – Client Data Set).

Поскольку компонент TClientDataSet является потомком виртуального класса TDataSet, для него можно применять большинство из приемов, рассмотренных при обсуждении использования компонента TDBDataSet в приложениях клиент/сервер. Например, установка значения свойства Active равным True приведет к открытию компонента TClientDataSet и отображению данных. Единственным отличием от установки свойства TTable.Active является то, что компонент TClientDataSet получает необходимые данные от приложения сервера.

Редактирование данных в клиентском приложении

Все записи, переданные сервером компоненту TClientDataSet, хранятся в его свойстве Data. Это свойство содержит представление пакета данных DataSnap с формате Variant. Компоненту TClientDataSet известно, как такой пакет данных преобразовать в более удобную форму. Использование типа Variant обусловлено тем, что для подсистемы COM доступно ограниченное количество типов данных.

При выполнении операций с записями набора данных клиента копии вставленных, модифицированных или удаленных записей помещаются в свойство Delta. Этим обеспечивается высокая эффективность приложений DataSnap при передаче обновлений обратно в серверную часть приложения и, в конечном счете, в базу данных.

Для свойства Delta используется чрезвычайно эффективный формат. В этом свойстве каждой операции вставки или удаления соответствует одна запись, а для каждого обновления – две записи. Обновляемые записи хранятся тоже достаточно эффективно. Исходная запись содержится в первой записи, тогда как соответствующая ей модифицируемая запись хранится во второй записи обновления. Однако в модифицированной записи содержатся лишь измененные поля.

Интересно то, что свойство Delta совместимо со свойством Data. Другими словами, хранящееся в нем значение можно напрямую присвоить свойству Data другого компонента ClientDataSet. Благодаря этому текущее содержимое свойства Delta можно анализировать в любой момент.

Для редактирования данных можно использовать различные методы компонента TClientDataSet. В дальнейшем мы будем называть их *методами управления изменениями* (change control method). Они позволяют различными способами модифицировать данные, внесенные в набор данных клиента.

НА ЗАМЕТКУ

Компонент TClientDataSet оказался гораздо более полезным, чем предполагалось изначально. Он обеспечивает прекрасный способ хранения таблиц в оперативной памяти (in-memory), что напрямую никак не связано с технологией DataSnap. Кроме того, поскольку компонент TClientDataSet может использоваться для передачи данных через свойство Data и другие свойства Delphi, он оказывается полезным и при реализации различных шаблонов объектно-ориентированного программирования. Обсуждение этих приемов выходит за рамки излагаемого в настоящей главе материала. Более подробная информация по данной теме находится на Web-странице по адресу: http://xapware.com или http://xapware.com/ddg.

Отмена внесенных изменений

Большинство пользователей знакомы с текстовыми процессорами, в которых поддерживается команда отмены изменений (Undo). Воспользовавшись ею, можно отменить последнее внесенное изменение и вернуться в исходное состояние. То же самое можно осуществить и с помощью вызова метода `cds.Customer.UndoLastChange()` компонента `TClientDataSet`. Используемый при этом стек имеет неограниченную длину, так что при необходимости есть возможность вернуться к самому началу сеанса редактирования. Параметр, передаваемый этому методу, определяет, нужно ли позиционировать курсор на соответствующую запись.

Если необходимо отменить все обновления одновременно, то последовательный вызов метода `UndoLastChange()` является не самым лучшим решением. Чтобы отменить все изменения, вносимые на протяжении одного сеанса редактирования, следует просто вызвать метод `cdsCustomer.CancelUpdates()`.

Возврат к исходной версии

Для возвращения определенной записи в исходное состояние (т.е. в такое, в котором она находилась в момент извлечения из базы данных) можно воспользоваться другим методом. Чтобы сделать это, переместите курсор в наборе данных клиента на запись, которую нужно восстановить, и вызовите метод `cdsCustomer.RevertRecord()`.

Транзакции клиента: свойство SavePoint

И, наконец, свойство `SavePoint` позволяет клиенту использовать транзакции. Данное свойство идеально подходит для разработки сценариев типа “что если”. Присвоив значение свойства `SavePoint` некоторой переменной, можно сохранить “моментальный снимок” данных. После этого пользователь может продолжать редактирование. Если в какой-либо момент пользователь решит, что в снимке содержатся именно те данные, которые ему нужны, то можно вернуть значения сохраненные ранее в свойстве `SavePoint`. При этом набор данных клиента вернется в то состояние, в котором он находился до сохранения. Обратите внимание: для реализации сложного сценария может понадобиться несколько уровней `SavePoint`.

СОВЕТ

Позволим себе одно предостережение относительно свойства `SavePoint`: при вызове метода `UndoLastChange()` значение этого свойства можно “испортить”. Например, предположим, что пользователь выполнил редактирование двух записей и сохранил значение свойства `SavePoint`. Затем он внес изменения в следующую запись, после чего для отмены внесенных изменений дважды воспользовался методом `UndoLastChange()`. Поскольку компонент `TClientDataSet` перешел в состояние, предшествующее сохраненному в свойстве `SavePoint`, то значение этого свойства стало неопределенным.

Согласование данных

По завершении внесения изменений в локальную копию данных, содержащуюся в компоненте `TClientDataSet`, эти изменения необходимо перенести в базу данных. Указанное можно осуществить с помощью метода `cdsCustomer.ApplyUpdates()`. При вызове этого метода приложению сервера будет передано свойство `Delta`, после

чего поступившие изменения будут внесены в базу данных в соответствии с механизмом согласования, установленным для обрабатываемого набора данных. Все обновления выполняются в рамках контекста одной транзакции. Кратко остановимся на обработке ошибок в ходе этого процесса.

Параметр, передаваемый методу `ApplyUpdates()`, определяет количество ошибок, появившихся в процессе обновления, после возникновения которых считается, что обновление завершилось неудачно. В этом случае все внесенные изменения последовательно отменяются. В данном контексте под *ошибками* (errors) подразумевается неудачный поиск по ключу, нарушение целостности ссылок или любые другие ошибки базы данных. Если для этого параметра установлено нулевое значение, то тем самым задается недопустимость любых ошибок. Таким образом, при возникновении какой-либо ошибки, все внесенные изменения в базе данных зафиксированы не будут. Это значение используется чаще всего, поскольку оно наиболее точно соответствует основным принципам использования баз данных.

Однако при желании можно задать и ненулевое число допустимых ошибок. Тогда в базу данных будут перенесены только все успешные записи. Предельным расширением этой концепции является передача в качестве параметра метода `ApplyUpdates()` значения `-1`. В этом случае приложением DataSnap в базе данных будет сохранена каждая отдельная запись, которую можно сохранить, без учета количества произошедших ошибок. Другими словами, при использовании этого значения транзакция всегда будет завершена успешно.

Если необходимо получить полный контроль над процессом обновления, в том числе возможность изменения операторов SQL, используемых для вставки, обновления или удаления, то можно воспользоваться обработчиком события `TDataSetProvider.BeforeUpdateRecord()`. Например, при удалении записи может потребоваться не удалять ее из базы данных физически. Вместо этого есть возможность установить флажок, сообщающий приложениям, что данная запись недоступна. Позднее можно пересмотреть такие записи и выполнить операцию физического удаления. В следующем фрагменте кода демонстрируется, как это можно осуществить:

```
procedure TDataModule1.Provider1BeforeUpdateRecord(Sender:
          TObject; SourceDS: TDataset; DeltaDS: TClientDataset;
          UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  if UpdateKind=ukDelete then begin
    Query1.SQL.Text:='update CUSTOMER set STATUS="DEL" where
↳ ID=:ID';
    Query1.Params[0].Value:=DeltaDS.FieldByName('ID').OldValue;
    Query1.ExecSQL;
    Applied:=true;
  end;
end;
```

В процессе управления потоком и содержимым процесса обновления можно создать любое количество запросов, опираясь на различные факторы, например на значение параметра `UpdateKind` и значения полей в наборе `DataSet`. При просмотре или модификации записей, содержащихся в параметре `DeltaDS`, удостоверьтесь, что используются свойства `OldValue` и `NewValue` соответствующего объекта `TField`. При использовании свойства `TField.Value` или `TField.As`*`XXX`* можно получить непредсказуемый результат.

Кроме того, при передаче обновлений в базу данных можно использовать бизнес-правила или вообще отменить передачу. Любое исключение, возникшее на этом этапе, будет передано механизму обработки ошибок DataSnap, который рассматривается ниже.

По завершении транзакции можно проанализировать возникшие ошибки. При возникновении ошибки приостанавливается работа и сервера, и клиента, позволяя принять меры к устранению ошибки, ее регистрации и выполнению любых других действий, необходимых в данном случае.

Первая остановка при ошибке осуществляется при обработке события `DataSetProvider.OnUpdateError`. Этот обработчик прекрасно подходит для обработки ожидаемых ошибок или для их устранения без вмешательства клиента.

Конечным получателем ошибок является клиентское приложение, в котором пользователю можно позволить самому принять решение о том, как поступить с ошибочной записью. Для этого необходимо определить обработчик события `TClientDataSet.OnReconcileError`.

Такой подход оказывается исключительно полезным, поскольку DataSnap основан на оптимистической стратегии блокировки записей. Данная стратегия позволяет нескольким пользователям одновременно работать с одной и той же записью. В принципе, это может быть причиной конфликтов, возникающих в том случае, если приложение DataSnap предпринимает попытку зафиксировать текущее состояние данных в базе, но обнаруживает, что со времени извлечения некоторая запись оказалась модифицированной.

Использование стандартного диалогового окна согласования ошибок

К счастью, компания *Borland* предоставила в распоряжение разработчика стандартное диалоговое окно, которое можно использовать для отображения ошибок. Это окно показано на рис. 21.6. Исходный код такого модуля также доступен, поэтому при необходимости его можно модифицировать. Для использования данного диалогового окна выберите в меню File пункт New, а затем — пиктограмму Reconcile Error Dialog во вкладке Dialogs. Не забудьте удалить этот модуль из списка Autocreate Forms, в противном случае при компиляции возникнут ошибки.

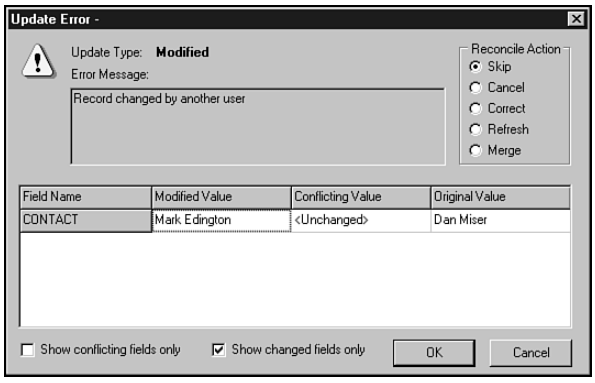

Рис. 21.6. *Диалоговое окно* Reconcile Error *в действии*

Почти все действия этого модуля реализованы в функции HandleReconcileError(), связанной с событием OnReconcileError. На практике обработчик события OnReconcileError обычно вызывает функцию HandleReconcileError. Благодаря этому конечный пользователь на клиентской машине получает возможность взаимодействовать с процессом согласования ошибок на сервере и определять способ обработки этих ошибок. Обработчик события OnReconcileError может быть реализован следующим образом:

```
procedure TMyForm.CDSReconcileError(Dataset: TCustomClientDataset;
                      E: EReconcileError; UpdateKind: TUpdateKind;
                      var Action: TReconcileAction);
begin
  Action:=HandleReconcileError(Dataset, UpdateKind, E);
end;
```

Значение параметра Action определяет действия, выполняемые приложением DataSnap над текущей записью. Другие факторы, определяющие последовательность действий в этом методе, будут рассмотрены несколько позднее. Далее приведен перечень возможных действий.

- raSkip. Не обновлять текущую запись базы данных. Оставить измененную запись в буфере клиента.
- raMerge. Объединить поля текущей записи с записью базы данных. Эта запись не будет применяться к уже вставленным записям.
- raCorrect. Обновить запись базы данных с использованием заданных значений. При выборе этого действия в диалоговом окне Reconcile Error значения можно редактировать в сетке (grid). Этот метод нельзя применять, если запись базы данных редактировалась другим пользователем.
- raCancel. Не обновлять запись базы данных и удалить ее из буфера клиента.
- raRefresh. Обновить запись в буфере клиента с использованием данных текущей записи базы.
- raAbort. Полностью прервать операцию обновления.

Данные параметры имеют смысл (а значит, и отображаются) не во всех случаях. Чтобы действия raMerge и raRefresh были доступны, запись должна была идентифицирована приложением с помощью первичного ключа базы данных. Для этого необходимо установить свойство TField.ProviderFlags.pfInKey компонента TDataset в состояние True для всех полей, указанных в первичном ключе.

Дополнительные параметры, повышающие надежность приложения

Овладев основами создания приложений DataSnap, читатель неизбежно задается вопросом: “Что дальше?”. В настоящем разделе будут более подробно рассмотрены различные аспекты использования технологии DataSnap, а также новые средства управления работой приложения.

Методы оптимизации клиентской части приложения

Модель организации доступа к данным, применяемая в приложениях DataSnap, довольно изящна. Но, поскольку все записи компонента TClientDataSet хранятся в оперативной памяти, необходимо проявлять осторожность при определении результирующих наборов данных, возвращаемых компоненту TClientDataSet. Следует обязательно удостовериться в том, что приложение сервера разработано правильно и возвращает лишь требуемые записи. Для ограничения количества записей, возвращаемых клиенту в каждый момент времени, можно использовать прием, описанный ниже.

Ограничение размеров пакета данных

При открытии компонента TClientDataSet сервер извлекает из источника данных количество записей, указанное в свойстве TClientDataSet.PacketRecords. Однако приложение DataSnap принимает столько записей, сколько ему необходимо для заполнения данными всех визуальных элементов управления. Например, если компонент TDBGrid в форме отображает одновременно до 10 записей, а значение свойства PacketRecords установлено равным 5, то в начальной выборке данных будет содержаться 10 записей, а во всех последующих пакетах данных — по 5 записей. Если значение этого свойства равно -1 (по умолчанию), то передаются все записи. Если свойство PacketRecords принимает положительное значение, то оно определяет состояние приложения. Указанное обусловлено тем, что приложение сервера должно отслеживать перемещение курсора каждого клиента, чтобы сервер в ответ на запрос клиента мог вернуть соответствующий пакет записей. Но состояние клиента можно отслеживать самостоятельно, передавая серверу позицию его последней записи. Именно это и осуществляется в следующем фрагменте кода:

```
Server RDM:
procedure TStateless.DataSetProvider1BeforeGetRecords(Sender:
                            TObject; var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do begin
    DataSet.Open;
    if VarIsEmpty(OwnerData) then
      DataSet.First
    else begin
      while not DataSet.Eof do begin
        if DataSet.FieldByName('au_id').Value = OwnerData then
          break;
      end;
    end;
  end;
end;

procedure TStateless.DataSetProvider1AfterGetRecords(Sender:
                            TObject; var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do begin
    OwnerData := Dataset.FieldValues['au_id'];
```

```
    DataSet.Close;
  end;
end;

Client:
procedure TForm1.ClientDataSet1BeforeGetRecords(Sender: TObject;
                                        var OwnerData: OleVariant);
begin
  // KeyValue - закрытая переменная типа OleVariant
  if not (Sender as TClientDataSet).Active then
    KeyValue := Unassigned;
  OwnerData := KeyValue;
end;

procedure TForm1.ClientDataSet1AfterGetRecords(Sender: TObject;
                                      var OwnerData: OleVariant);
begin
    KeyValue := OwnerData;
end;
```

При использовании автоматической выборки метод `TCLientDataSet.Last()` извлекает остальные записи результирующего набора данных. Для получения этого же результата можно нажать в компоненте `TDBGrid` комбинацию клавиш <Ctrl+End>. Во избежание проблем установите значение свойства `TCLientDataSet.FetchOnDemand` равным `False`. Это свойство определяет, будет ли пакет данных извлекаться автоматически, если пользователь прочел все существующие записи в клиентском приложении. Если же такую возможность необходимо реализовать в коде, то воспользуйтесь методом `GetNextPacket()`, возвращающим следующий пакет данных.

НА ЗАМЕТКУ

Обратите внимание: приведенный выше код перебирает набор данных до тех пор, пока не находит соответствующую запись. Это сделано для того, чтобы односторонние наборы данных, типа dbExpress, могли использовать этот же самый код без модификации. Конечно, существует множество способов поиска соответствующий записи, например с помощью оператора SQL или параметрического запроса, но данный пример сосредоточен на механизме передачи ключа между клиентом и сервером.

Использование модели "портфеля"

Другим видом оптимизации, позволяющим уменьшить сетевой трафик, является использование модели "портфеля". Для этого присвойте имя файла свойству `TCliDataSet.Filename`. Если заданный файл уже существует на данном компьютере, то компонент `TCLientDataSet` откроет локальную копию файла вместо того, чтобы считывать данные из приложения сервера. Это чрезвычайно удобно для таких редко изменяемых элементов, как таблицы подстановок.

СОВЕТ

Если в свойстве `TClientDataSet.Filename` задано имя файла с расширением `.XML`, то пакет данных будет сохранен в формате XML, что позволяет использовать для его обработки любые инструменты XML, доступные при работе с файлом портфеля.

Передача на сервер динамических запросов SQL

Иногда со стороны клиента приходится модифицировать основные свойства компонента `TDataSet` (например свойство `SQL` компонента `TQuery`). Как следует из принципов создания многоуровневых приложений, это может оказаться достаточно эффективным и элегантным решением. Delphi значительно упрощает решение данной задачи.

Для формирования специальных запросов необходимо выполнить два действия. Во-первых, поместите необходимый оператор запроса SQL в свойство `TClientDataSet.CommandText`. Во-вторых, установите значение свойства `DataSetProvider. Options` равным `poAllowCommandText`. При открытии компонента `TClientDataSet` или вызове метода `TClientDataSet.Execute()` запрос из свойства `CommandText` будет передан серверу. Этот же прием можно использовать для изменения таблицы или имени хранимой процедуры на сервере.

Методы сервера приложений

DataSnap обладает многочисленными событиями, позволяющими настроить поведение приложения. Практически для любого метода интерфейса `IAppServer` существуют события типа `Before`*`XXX`* и `After`*`XXX`*. Эти события могут оказаться особенно полезными, если необходимо придать приложению сервера полную независимость от состояния процессов пользователей.

Разрешение конфликтов записи

В предыдущих разделах при обсуждении механизма согласования кратко упоминалось о том, что если два пользователя работают с одной и той же записью, то при попытке второго пользователя передать запись на сервер может возникнуть ошибка. К счастью, Delphi позволяет осуществлять полный контроль над этим процессом и избежать подобных конфликтов.

Свойство `TDataSetProvider.UpdateMode` применяется для формирования оператора SQL, который будет использоваться для проверки того, изменилась ли запись с момента ее последнего извлечения из базы данных. Вернемся к сценарию, в котором два пользователя редактируют одну и ту же запись. Ниже приведены возможные значения свойства `TDataSetProvider.UpdateMode`, определяющие возможности работы с записью каждого пользователя.

- `upWhereAll`. Этот параметр является самым строгим, но он обеспечивает наиболее высокую степень уверенности в том, что запись не изменялась со времени ее первоначального извлечения из базы данных. Если два пользователя редактируют одну и ту же запись, то первый пользователь может эту запись обновлять, тогда как второй пользователь получит сообщение об ошибке `Another user changed the record` (Запись изменена другим пользователем). Если в дальнейшем понадобится уточнить, какие именно поля следует проверять, то удалите значение `pfInWhere` из соответствующих свойств `TField.ProviderFlags`.
- `upWhereChanged`. Если установлен этот параметр, то два пользователя смогут одновременно редактировать одну и ту же запись. Если оба пользователя редактируют различные поля одной и той же записи, то это не приведет к конфликту. Например, если пользователь A модифицирует поле `Address` и обнов-

ляет запись, то пользователь B, в свою очередь, может отредактировать поле BirthDate, а затем успешно обновить эту же запись.

- upWhereKeyOnly. Данный параметр означает наименее строгую проверку. Запись базы данных может изменить каждый из пользователей. При этом существующая запись всегда замещается новой. Таким образом, при использовании этого параметра реализуется принцип "выигрывает последний".

Прочие параметры сервера

Для того чтобы определить, как приложение DataSnap будет управлять пакетами данных, можно использовать и другие допустимые значения свойства TDataSetProvider.Options. Например, при добавлении значения poReadOnly набор данных для клиента будет доступен только для чтения. Если указано значение poDisableInserts, poDisableDeletes или poDisableEdits, то клиент не сможет выполнить операцию вставки, удаления или редактирования соответственно, а в случае попытки выполнить запрещенную операцию будут активизированы соответствующие обработчики событий OnEditError или OnDeleteError.

При использовании вложенных наборов данных из главной записи можно выполнять каскадные обновления или удаления в подчиненных записях, если добавить в свойство DataSetProvider.Options значения poCascadeUpdates или poCascadeDeletes. При использовании этого свойства необходимо, чтобы база данных поддерживала каскадную проверку целостности ссылок.

Недостатком предыдущих версий технологии DataSnap (она называлась MIDAS) была невозможность простого перенесения изменений, сделанных на сервере, в компонент TCLientDataSet клиентского приложения. Можно было лишь сортировать записи с помощью метода RefreshRecord (или в некоторых случаях повторно заполнить весь набор данных).

При добавлении в свойство DataSetProvider.Options значения poPropogateChanges все изменения данных, внесенные на сервере (например в обработчике события DataSetProvider.BeforeUpdateRecord, предназначенного для реализации бизнес-правил), будут автоматически переданы обратно компоненту TClientDataSet. Более того, если в свойство DataSetProvider.Options добавить значение poAutoRefresh, то компоненту TClientDataSet автоматически передается значение свойства AutoIncrement, а также значения, используемые по умолчанию.

СОВЕТ

Режим poAutoRefresh в исходной версии Delphi 5 и 6 не работает. В более поздних версиях эта ошибка будет устранена. Чтобы обойти эту проблему, необходимо либо воспользоваться методом Refresh() компонента TClientDataSet, либо самостоятельно управлять всем процессом обновления данных.

До сих пор при обсуждении процесса предотвращения конфликтов речь шла о выполнении согласования стандартными средствами SQL, а события компонента TDataSet в процессе согласования не использовались. Для поддержки использования этих событий в процессе разрешения конфликтов было создано свойство TDataSetProvider.ResolveToDataSet. Например, если значение этого свойства установлено равным True, то можно использовать большинство событий компонента TDataSet. Следите за тем, чтобы все используемые события вызывались лишь во время пе-

редачи обновлений обратно на сервер. Другими словами, если на сервере определено событие TQuery.BeforeInsert, то оно должно возникать только при вызове метода TClientDataSet.ApplyUpdates. События сервера нельзя объединить с соответствующими событиями компонента TClientDataSet.

Поддержка связи "главная–подчиненная"

Знакомство с приложениями баз данных оказалось бы неполным без упоминания о связях между таблицами типа "главная–подчиненная" (master/detail). При использовании технологии DataSnap эту связь можно реализовать двумя способами.

Вложенные наборы данных

Вложенные наборы данных (nested dataset) позволяют помещать подчиненные наборы данных непосредственно в главную таблицу. Помимо того, что это позволяет выполнять обновление главной и подчиненных записей за одну транзакцию, стало возможным хранить все главные и подчиненные записи в одном файле портфеля. Можно также использовать все преимущества компонента DBGrid, который теперь позволяет отображать подчиненные наборы данных в отдельных окнах. При использовании вложенных наборов данных не забывайте о следующем: при выборке главной записи извлекаются и все связанные с ней подчиненные записи, которые затем передаются клиенту. При использовании нескольких уровней вложенности подчиненных наборов данных это может послужить причиной существенного снижения эффективности. Например, при извлечении одной главной записи, имеющей 10 подчиненных, каждая из которых связана, в свою очередь, с тремя подчиненными записями следующего уровня, из базы данных будет получена 41 запись. При использовании связи со стороны клиента сначала будут получены только 14 записей, а остальные вложенные записи будут извлекаться по мере перемещения по подчиненному клиентскому набору данных (TClientDataSet).

Для создания вложенных наборов данных в приложении сервера необходимо определить отношение "главная–подчиненная". Это осуществляется точно так же, как и в приложениях клиент/сервер, а именно — с помощью оператора SQL для объекта подчиненного запроса TQuery, который включал бы *параметр связи* (link parameter). Например:

```
"select * orders where custno=:custno"
```

Затем присвойте свойству TQuery.Datasource подчиненного запроса TQuery значение, указывающее на компонент TDatasource, связанный с компонентом TDataSet главного набора данных. После установки связи останется лишь экспортировать компонент TDataSetProvider, связанный с главным набором данных. Приложение DataSnap достаточно интеллектуально, чтобы понять, что у главного набора данных имеются подчиненные наборы, связанные с ним и передаваемые клиенту как экземпляры класса TDataSetField.

В клиентской части приложения присвойте имя главного провайдера свойству TClientDataSet.ProviderName. Затем добавьте в компонент TClientDataSet постоянные поля. Обратите внимание на последнее поле в окне редактора полей. Его имя совпадает с именем подчиненного набора данных на сервере и имеет тип TDataSetField. На данный момент уже существует достаточно информации, чтобы использовать вложенный набор данных в коде. Упростить задачу можно, добавив компонент TClientDataSet для

подчиненного набора данных и присвоив его свойству DataSetField имя соответствующего компонента TDataSetField главного набора данных. При этом важно помнить, что в подчиненном наборе данных не нужно устанавливать никаких других свойств, таких, например, как RemoteServer, ProviderName, MasterSource, MasterFields или PacketRecords. Следует лишь установить значение свойства DataSetField. Теперь с подчиненным набором данных можно связать необходимые элементы управления.

После завершения работы со вложенными наборами данных внесенные изменения необходимо зафиксировать в базе. Это можно осуществить с помощью вызова метода ApplyUpdates главного компонента TCLientDataSet. В результате приложение DataSnap одной транзакцией, содержащей и подчиненные наборы данных, внесет все изменения в главный набор данных, находящийся на сервере.

Соответствующий пример можно найти на прилагаемом компакт-диске — в подкаталоге \NestCDS каталога, соответствующего данной главе.

Связь в клиентской части приложения

Напомним о некоторых предостережениях, которые упоминались при рассмотрении вложенных наборов данных. Альтернативой использованию вложенных наборов данных является создание связи "главная–подчиненная" в клиентской части приложения. В этом случае на сервере необходимо создать компоненты TDataSet и TDataSetProvider для главного и подчиненного наборов данных.

В клиентской части свяжите с наборами данных два компонента TClientDataSet, экспортируемых на сервер. Затем создайте связь "главная–подчиненная", присвоив свойству TClientDataSet.MasterSource подчиненного набора данных источник данных TDataSource, связанный с главным набором данных.

При установке свойства MasterSource компонента TClientDataSet свойству PacketRecords присваивается нулевое значение. Это означает, что в данном случае приложение DataSnap должно получить с сервера только метаданные. Но если свойство PacketRecords имеет нулевое значение в контексте связи "главная–подчиненная", то смысл несколько изменяется. В таком случае приложение DataSnap для каждой главной записи будет извлекать весь набор связанных данных подчиненных записей. Одним словом, оставьте в свойстве PacketRecords значение, используемое по умолчанию.

Для передачи данных, связанных по принципу "главная–подчиненная", в базу данных за одну транзакцию разработчику необходимо написать свою собственную реализацию метода ApplyUpdates. Данная задача далеко не так проста, как большинство задач, решаемых в Delphi. Но в этом случае разработчик получает полный контроль над процессом обновления.

Применение обновлений к единственной таблице обычно осуществляется с помощью метода TCLientDataSet.ApplyUpdates. Этот метод передает измененные записи от компонента ClientDataSet его провайдеру на среднем уровне, который, в свою очередь, записывает изменения в базу данных. Причем все необходимые действия осуществляются в рамках одной транзакции и могут быть выполнены без вмешательства программиста. Чтобы то же самое осуществить при использовании связи "главная–подчиненная", следует понимать, какие действия выполняются Delphi при вызове метода TClientDataSet.ApplyUpdates.

Все изменения, внесенные в компонент TClientDataSet, хранятся в свойстве Delta. В этом свойстве содержится вся информация, которая в конечном счете будет записана в базу данных. В следующем фрагменте кода иллюстрируется процесс обновления, при ко-

тором содержимое свойств Delta передается обратно в базу данных. В листингах 21.2 и 21.3 приведены фрагменты кода клиента и сервера для выполнения обновлений при использовании связи "главная–подчиненная".

Листинг 21.2. Клиентская часть для выполнения обновления

```
procedure TClientDM.ApplyUpdates;
var
  MasterVar, DetailVar: OleVariant;
begin
  Master.CheckBrowseMode;
  Detail_Proj.CheckBrowseMode;
  if Master.ChangeCount > 0 then MasterVar := Master.Delta
  else MasterVar := NULL;
  if Detail.ChangeCount > 0 then DetailVar := Detail.Delta
  else DetailVar := NULL;
  RemoteServer.AppServer.ApplyUpdates(DetailVar, MasterVar);
  { Согласование ошибок в пакетах данных. Поскольку предполагается
    отсутствие ошибок, то они могут содержаться лишь в одном
    пакете. Обновление данных производится лишь тогда, когда ни в
    одном из пакетов нет ошибок. }
  if not VarIsNull(DetailVar) then Detail.Reconcile(DetailVar)
  else
  if not VarIsNull(MasterVar) then Master.Reconcile(MasterVar)
  else begin
    Detail.Reconcile(DetailVar);
    Master.Reconcile(MasterVar);
    Detail.Refresh;
    Master.Refresh;
  end;
end;
```

Листинг 21.3. Серверная часть для выполнения обновления

```
procedure TServerRDM.ApplyUpdates(var DetailVar,
                                  MasterVar: OleVariant);
var
  ErrCount: Integer;
begin
  Database.StartTransaction;
  try
    if not VarIsNull(MasterVar) then begin
      MasterVar := cdsMaster.Provider.ApplyUpdates(MasterVar, 0,
                                                   ErrCount);
      if ErrCount > 0 then
        SysUtils.Abort;    // Отмена внесенных изменений
    end;
    if not VarIsNull(DetailVar) then begin
      DetailVar := cdsDetail.Provider.ApplyUpdates(DetailVar, 0,
                                                   ErrCount);
```

```
      if ErrCount > 0 then
        SysUtils.Abort;    // Отмена внесенных изменений
    end;
    Database.Commit;
  except
    Database.Rollback
  end;
end;
```

Хотя данный метод работает достаточно хорошо, он, тем не менее, не обеспечивает возможности повторного использования кода, а такую возможность было бы неплохо иметь. Для этого достаточно выполнить следующие действия.

1. Поместите свойства `Delta` каждого набора данных клиента в массив типа `Variant`.
2. Поместите провайдеры каждого набора данных клиента в массив типа `Variant`.
3. Используйте все свойства `Delta` в одной транзакции.
4. Обеспечьте предотвращение ошибок доступа в пакетах данных, возвращенных на предыдущем этапе, и обновите данные.

Данные рекомендации реализованы в утилите, код которой представлен в листинге 21.4.

Листинг 21.4. Модуль, содержащий процедуры утилиты и необходимую степень абстракции

```
unit CDSUtil;

interface

uses
  DbClient, DbTables;

function RetrieveDeltas(const cdsArray :
                array of TClientDataset): Variant;
function RetrieveProviders(const cdsArray :
                array of TClientDataset): Variant;
procedure ReconcileDeltas(const cdsArray :
                array of TClientDataset; vDeltaArray: OleVariant);
procedure CDSApplyUpdates(ADatabase : TDatabase;
                var vDeltaArray: OleVariant;
                const vProviderArray: OleVariant; Local: Boolean);

implementation

uses
  SysUtils, Provider, Midas, Variants;

type
  PArrayData = ^TArrayData;
  TArrayData = array[0..1000] of Olevariant;
```

```
{ На входе свойство Delta принимает значение CDS.Delta. На выходе
Delta будет содержать пакеты данных, содержащие все записи, которые
не удалось занести в базу данных. Помните, что в Delphi необходимо
указать имя провайдера, поэтому оно передается в первом элементе
AProvider. }
  procedure ApplyDelta(AProvider: OleVariant;
                       var Delta : OleVariant; Local: Boolean);
var
  ErrCount : integer;
  OwnerData: OleVariant;
begin
  if not VarIsNull(Delta) then begin
    { ScktSrvr не поддерживает раннее связывание, а
      TLocalAppServer не поддерживает приведение
      типа IAppServerDisp }
    if Local then
      Delta := (IDispatch(AProvider[0]) as
↳IAppServer).AS_ApplyUpdates(AProvider[1],
↳Delta, 0, ErrCount, OwnerData)
    else
      Delta := IAppServerDisp(IDispatch(AProvider[0]))
↳.AS_ApplyUpdates(AProvider[1], Delta, 0, ErrCount, OwnerData);
    if ErrCount > 0 then  // Отмена внесенных изменений в
      SysUtils.Abort;     // вызывающей процедуре.
  end;
end;

{ Вызов со стороны сервера }
procedure CDSApplyUpdates(ADatabase : TDatabase;
                          var vDeltaArray: OleVariant;
                          const vProviderArray: OleVariant;
                          Local: Boolean);
var
  i : integer;
  LowArr, HighArr: integer;
  P: PArrayData;
begin
  { Все обновления в одной транзакции. Если на каком либо этапе
    произойдет ошибка, то будет передано исключение, приводящее к
    отмене транзакции. }
  ADatabase.Connected := true;
  ADatabase.StartTransaction;
  try
    LowArr:=VarArrayLowBound(vDeltaArray, 1);
    HighArr:=VarArrayHighBound(vDeltaArray, 1);
    P:=VarArrayLock(vDeltaArray);
    try
      for i:=LowArr to HighArr do
        ApplyDelta(vProviderArray[i], P^[i], Local);
    finally
      VarArrayUnlock(vDeltaArray);
    end;
```

```
    ADatabase.Commit;
  except
    ADatabase.Rollback;
  end;
end;

{ Вызовы со стороны клиентов }
function RetrieveDeltas(const cdsArray:
                        array of TClientDataset): Variant;
var
  i: integer;
  LowCDS, HighCDS: integer;
begin
  Result := NULL;
  LowCDS := Low(cdsArray);
  HighCDS := High(cdsArray);
  for i := LowCDS to HighCDS do cdsArray[i].CheckBrowseMode;
  Result:=VarArrayCreate([LowCDS, HighCDS], varVariant);
  { Задать массив Variant с изменениями (или NULL, если
    изменения отсутствуют) }
  for i:=LowCDS to HighCDS do begin
    if cdsArray[i].ChangeCount>0 then Result[i]:=cdsArray[i].Delta
    else Result[i]:=NULL;
  end;
end;

{ Необходимо вернуть имя провайдера и имя сервера приложения. Позже
имя провайдера (ProviderName) используется для вызова
AS_ApplyUpdates в функции CDSApplyUpdates. }
function RetrieveProviders(const cdsArray :
                           array of TClientDataset): Variant;
var
  i: integer;
  LowCDS, HighCDS: integer;
begin
  Result:=NULL;
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);

  Result:=VarArrayCreate([LowCDS, HighCDS], varVariant);
  for i:=LowCDS to HighCDS do
    Result[i]:=VarArrayOf([cdsArray[i].AppServer,
                           cdsArray[i].ProviderName]);
end;

procedure ReconcileDeltas(const cdsArray :
                          array of TClientDataset;
                          vDeltaArray: OleVariant);
var
  bReconcile: boolean;
  i: integer;
  LowCDS, HighCDS: integer;
begin
```

```
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);

  { Если предыдущий этап завершился ошибкой, то выполняется
    устранение ошибок пакетов данных. }
  bReconcile:=false;
  for i:=LowCDS to HighCDS do
    if not VarIsNull(vDeltaArray[i]) then begin
      cdsArray[i].Reconcile(vDeltaArray[i]);
      bReconcile:=true;
      break;
    end;

  { При необходимости обновить наборы данных }
  if not bReconcile then
    for i:=HighCDS downto LowCDS do begin
      cdsArray[i].Reconcile(vDeltaArray[i]);
      cdsArray[i].Refresh;
    end;
end;

end.
```

В листинге 21.5 предыдущий пример модифицирован с помощью модуля `CDSUtil`.

Листинг 21.5. Предыдущий пример, но с использованием модуля `CDSUtil.pas`

```
procedure TForm1.btnApplyClick(Sender: TObject);
var
  vDelta: OleVariant;
  vProvider: OleVariant;
  arrCDS: array[0..1] of TClientDataset;
begin
  arrCDS[0]:= cdsMaster;  // Установить массив ClientDataset
  arrCDS[1]:= cdsDetail;

  vDelta:=RetrieveDeltas(arrCDS);                  // Этап 1
  vProvider:=RetrieveProviders(arrCDS);            // Этап 2
  DCOMConnection1.ApplyUpdates(vDelta, vProvider); // Этап 3
  ReconcileDeltas(arrCDS, vDelta);                 // Этап 4
end;

procedure TServerRDM.ApplyUpdates(var vDelta,
                                  vProvider: OleVariant);
begin
  CDSApplyUpdates(Database1, vDelta, vProvider);   // Этап 3
end;
```

Этот модуль можно использовать и в двух- или трехуровневых приложениях. Для того чтобы применяемый подход перенести с двух- на трехуровневую модель, вместо вызова функции `CDSApplyUpdates` в клиентской части на сервере необходимо экспорти-

ровать функцию, в которой вызывается метод CDSApplyUpdates. Весь остальной код клиента остается без изменений.

Пример использования данного метода можно найти на прилагаемом CD в каталоге \MDCDS раздела, посвященного настоящей главе.

Примеры из реальной жизни

Теперь, изучив основные принципы построения приложений DataSnap, ознакомимся с тем, как можно применить эту технологию для решения практических задач.

Объединения

Существенное влияние на построение приложений реляционных баз данных оказывают способы организации связей между таблицами. Зачастую удобнее работать с *представлениями* (view), созданными на базе высоко нормализованных данных. В таком случае представления существенно облегчают и упрощают отображение данных, по сравнению с их реальной структурой. Но при обновлении данных в подобных *объединениях* (join) необходимо соблюдать дополнительные меры предосторожности.

Обновление одной таблицы

Применение обновлений к запросу на объединение представляет собой специальный случай в программировании баз данных, и приложения DataSnap не являются исключением. Проблема заключается в самом запросе на объединение. Хотя некоторые запросы на объединение предоставляют данные, которые могут быть обновлены автоматически, тем не менее существуют запросы, не допускающие автоматического извлечения, редактирования и обновления исходных данных. В Delphi задача разрешения обновлений запросов на объединение ложится на плечи разработчика.

При использовании объединений, требующих обновления лишь одной таблицы, большинство необходимых действий Delphi может выполнить самостоятельно. При записи в базу данных информации из одной таблицы необходимо выполнить следующее.

1. Добавьте *постоянные* (persistent) поля в объединенный компонент TQuery.
2. Для каждого поля компонента TQuery, которое необходимо обновить, установите TField.ProviderFlags=[].
3. Чтобы сообщить приложению DataSnap, какую именно таблицу нужно обновить, поместите в обработчик события DataSetProvider.OnGetTableName код, приведенный ниже. Это новое событие упрощает способ задания имени таблицы, хотя в предыдущих версиях Delphi то же самое можно было сделать с помощью события DataSetProvider.OnGetDataSetProperties:

```
procedure TJoin1Server.prvJoinGetTableName(Sender: TObject;
                    DataSet: TDataSet; var TableName: String);
begin
  TableName := 'Emp';
end;
```

По завершении указанных действий имя таблицы сохраняется в компоненте ClientDataSet. Теперь при вызове метода ClientDataSet1.ApplyUpdates() прило-

жению DataSnap будет известно имя таблицы, используемое по умолчанию, и ему не придется выполнять его поиск.

Можно воспользоваться и другим способом — компонентом TUpdateSQL, выполняющим обновление лишь требуемой таблицы. Благодаря этой возможности в Delphi во время процесса согласования можно использовать метод TQuery.UpdateObject. Такой подход наиболее точно соответствует процессу, применяемому в традиционных приложениях клиент/сервер.

НА ЗАМЕТКУ

Не все объекты класса TDataset обладают свойством UpdateObject. Но для них все равно можно использовать этот подход, поскольку изменения относятся к TUpdateSQL. Достаточно просто определить операторы SQL для каждого действия (удаления, вставки, изменения) и использовать код, подобный следующему:

```
procedure TForm1.DataSetProvider1BeforeUpdateRecord(Sender: TObject;
                  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
                  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  UpdateSQL1.DataSet := DeltaDS;
  UpdateSQL1.SetParams(UpdateKind);
  ADOCommand1.CommandText := UpdateSQL1.SQL[UpdateKind].Text;
  ADOCom-
mand1.Parameters.Assign(UpdateSQL1.Query[UpdateKind].Params);
  ADOCommand1.Execute;
  Applied := true;
end;
```

Пример использования такого подхода можно найти на прилагаемом CD в каталоге \Join1 раздела, посвященного настоящей главе.

Обновление нескольких таблиц

В более сложных случаях, когда редактируется и обновляется несколько таблиц, для обновления придется написать свой собственный код. Для решения этой проблемы можно использовать два подхода.

- Прежний подход заключался в использовании метода DataSetProvider. BeforeUpdateRecord() для разбиения пакета данных и применения обновлений к исходным таблицам.
- Нынешний подход заключается во внесении обновлений с помощью свойства UpdateObject.

Если для объединения нескольких таблиц используется кэширование обновлений, то для каждой обновляемой таблицы потребуется настроить один компонент TUpdateSQL. Поскольку свойство UpdateObject может быть присвоено только одному компоненту TUpdateSQL, то все свойства TUpdateSQL.DataSet необходимо связать с объединенным набором данных программно, в TQuery.OnUpdateRecord. После этого нужно вызвать метод TUpdateSQL.Apply, чтобы связать все параметры и выполнить исходный оператор SQL. В рассматриваемом случае набор данных содержится в свойстве Delta. Этот набор данных передается в качестве параметра в обработчик события TQuery.OnUpdateRecord.

Все, что осталось сделать, — так это присвоить значения свойствам SessionName и DatabaseName, что позволит осуществить обновление в том же самом контексте, что и другие транзакции, а также связать свойство DataSet со свойством Delta, которое передается событию. Полученный в результате код обработчика события TQuery.OnUpdateRecord приведен в листинге 21.6.

Листинг 21.6. Объединение с использованием компонента TUpdateSQL

```
procedure TJoin2Server.JoinQueryUpdateRecord(DataSet: TDataSet;
      UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  usqlEmp.SessionName := JoinQuery.SessionName;
  usqlEmp.DatabaseName := JoinQuery.DatabaseName;
  usqlEmp.Dataset := Dataset;
  usqlEmp.Apply(UpdateKind);

  usqlFTEmp.SessionName := JoinQuery.SessionName;
  usqlFTEmp.DatabaseName := JoinQuery.DatabaseName;
  usqlFTEmp.Dataset := Dataset;
  usqlFTEmp.Apply(UpdateKind);

  UpdateAction := uaApplied;
end;
```

Поскольку все действия выполнены внутри архитектуры DataSnap, то весь процесс обновления может быть запущен при обращении к методу DataSnap ClientDataSet1.ApplyUpdates(0);.

Пример использования такого подхода можно найти на прилагаемом CD в каталоге \Join2 раздела, посвященного настоящей главе.

Приложения DataSnap в Web

Несмотря на введение Kylix, Delphi остается жестко привязанным к платформе (Windows или Linux). Следовательно, любой тип клиента должен выполняться на том типе машины, для которого он был написан. Но это не всегда желательно. Например, может понадобиться обеспечить простой доступ к базе данных через Internet, а поскольку ранее было создано приложение сервера, которое помимо обеспечения бизнес-правил функционирует в качестве брокера, то было бы полезно использовать именно его и не создавать заново уровни бизнес-правил и доступа к данным для каждой среды исполнения.

Простой HTML

Настоящий раздел посвящен вопросам адаптации приложения сервера и создания нового уровня представления, на котором используется простой язык HTML. Прежде чем приступить к изучению этого раздела, необходимо ознакомиться с материалом главы 31, "Компоненты WebBroker открывают двери в Internet", предыдущего издания *Delphi 5 Руководство разработчика*, находящегося на прилагаемом CD. При использовании такого подхода в архитектуру приложения вводится еще один уровень. По

отношению к приложению сервера модуль WebBroker функционирует как клиент, преобразующий данные в формат HTML для отображения их в окне броузера. При этом теряются некоторые преимущества работы в интегрированной среде разработки Delphi (такие, например, как возможность использования многочисленных элементов управления). Но если доступ к данным в формате HTML необходим, то это неизбежно.

Создав приложение WebBroker и модуль `WebModule`, достаточно поместить в эту форму компоненты `TDispatchConnection` и `TClientDataSet`. После заполнения необходимых свойств можно воспользоваться соответствующими методами для преобразования данных в формат HTML и передачи их клиенту.

Одним из эффективных приемов является добавление компонента `TDataSetTableProducer`, связанного с компонентом `TClientDataSet`. В этом случае пользователь сможет щелкнуть на ссылке и перейти на страницу редактирования, на которой можно будет модифицировать и обновлять данные. В листингах 21.7 и 21.8 представлен пример реализации этого приема.

Листинг 21.7. Код HTML для редактирования и обновления данных

```
<form action="<#SCRIPTNAME>/updaterecord" method="post">
<b>EmpNo: <#EMPNO></b>
<input type="hidden" name="EmpNo" value=<#EMPNO>>
<table cellspacing="2" cellpadding="2" border="0">
<tr>
    <td>Last Name:</td>
    <td><input type="text" name="LastName" value=<#LASTNAME>></td>
</tr>
<tr>
    <td>First Name:</td>
    <td>
    <input type="text" name="FirstName" value=<#FIRSTNAME>>
    </td>
</tr>
<tr>
    <td>Hire Date:</td>
    <td>
    <input type="text" name="HireDate" size="8" value=<#HIREDATE>>
    </td>
</tr>
<tr>
    <td>Salary:</td>
    <td>
    <input type="text" name="Salary" size="8" value=<#SALARY>>
    </td>
</tr>
<tr>
    <td>Vacation:</td>
    <td>
    <input type="text" name="Vacation" size="4" value=<#VACATION>>
    </td>
</tr>
</table>
```

```
<input type="submit" name="Submit"value="Apply Updates">
<input type="Reset">
</form>
```

Листинг 21.8. Код для редактирования и внесения обновлений

```
unit WebMain;

interface

uses
  Windows, Messages, SysUtils, Classes, HTTPApp, DBWeb, Db,
  DBClient, MConnect, DSProd, HTTPProd;

type
  TWebModule1 = class(TWebModule)
    dcJoin: TDCOMConnection;
    cdsJoin: TClientDataSet;
    dstpJoin: TDataSetTableProducer;
    dsppJoin: TDataSetPageProducer;
    ppSuccess: TPageProducer;
    ppError: TPageProducer;
    procedure WebModuleBeforeDispatch(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                  var Handled: Boolean);
    procedure WebModule1waListAction(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                  var Handled: Boolean);
    procedure dstpJoinFormatCell(Sender: TObject; CellRow,
                CellColumn: Integer; var BgColor: THTMLBgColor;
                var Align: THTMLAlign; var VAlign: THTMLVAlign;
                var CustomAttrs, CellData: String);
    procedure WebModule1waEditAction(Sender: TObject;
                Request: TWebRequest; Response: TWebResponse;
                var Handled: Boolean);
    procedure dsppJoinHTMLTag(Sender: TObject; Tag: TTag;
                const TagString: String; TagParams: TStrings;
                var ReplaceText: String);
    procedure WebModule1waUpdateAction(Sender: TObject;
                Request: TWebRequest; Response: TWebResponse;
                var Handled: Boolean);
  private
    { Закрытые объявления }
    DataFields: TStrings;
  public
    { Открытые объявления }
  end;

var
  WebModule1: TWebModule1;
```

```
implementation

{$R *.DFM}

procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
begin
  with Request do
    case MethodType of
      mtPost: DataFields:=ContentFields;
      mtGet: DataFields:=QueryFields;
    end;
end;

function LocalServerPath(sFile: string = ''): string;
var
  FN: array[0..MAX_PATH- 1] of char;
  sPath: shortstring;
begin
  SetString(sPath, FN, GetModuleFileName(hInstance, FN,
            SizeOf(FN)));
  Result := ExtractFilePath( sPath ) + ExtractFileName( sFile );
end;

procedure TWebModule1.WebModule1waListAction(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
begin
  cdsJoin.Open;
  Response.Content := dstpJoin.Content;
end;

procedure TWebModule1.dstpJoinFormatCell(Sender: TObject; CellRow,
                  CellColumn: Integer; var BgColor: THTMLBgColor;
                  var Align: THTMLAlign; var VAlign: THTMLVAlign;
                  var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and (CellColumn = 0) then
    CellData := Format('<a href="%s/getrecord?empno=%s">%s</a>',
                       [Request.ScriptName, CellData, CellData]);
end;

procedure TWebModule1.WebModule1waEditAction(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                  var Handled: Boolean);
begin
  dsppJoin.HTMLFile := LocalServerPath('join.htm');
  cdsJoin.Filter := 'EmpNo = ' + DataFields.Values['empno'];
  cdsJoin.Filtered := true;
  Response.Content := dsppJoin.Content;
end;
```

```
procedure TWebModule1.dsppJoinHTMLTag(Sender: TObject; Tag: TTag;
                  const TagString: String; TagParams: TStrings;
                  var ReplaceText: String);
begin
  if CompareText(TagString, 'SCRIPTNAME') = 0 then
    ReplaceText:= Request.ScriptName;
end;

procedure TWebModule1.WebModule1waUpdateAction(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                  var Handled: Boolean);
var
  EmpNo, LastName, FirstName, HireDate, Salary, Vacation: string;
begin
  EmpNo:=DataFields.Values['EmpNo'];
  LastName:=DataFields.Values['LastName'];
  FirstName:=DataFields.Values['FirstName'];
  HireDate:=DataFields.Values['HireDate'];
  Salary:=DataFields.Values['Salary'];
  Vacation:=DataFields.Values['Vacation'];

  cdsJoin.Open;
  if cdsJoin.Locate('EMPNO', EmpNo, []) then begin
    cdsJoin.Edit;
    cdsJoin.FieldByName('LastName').AsString:=LastName;
    cdsJoin.FieldByName('FirstName').AsString:=FirstName;
    cdsJoin.FieldByName('HireDate').AsString:=HireDate;
    cdsJoin.FieldByName('Salary').AsString:=Salary;
    cdsJoin.FieldByName('Vacation').AsString:=Vacation;
    if cdsJoin.ApplyUpdates(0)=0 then
      Response.Content:=ppSuccess.Content
    else
      Response.Content:=pPError.Content;
  end;
end;

end.
```

Обратите внимание: данный метод требует написания большого объема кода вручную. Поэтому здесь реализован не полный набор возможностей приложений DataSnap, в частности отсутствует разрешение конфликтов доступа. При реальном использовании этого подхода пример можно усовершенствовать, сделав его более надежным.

СОВЕТ

При создании модуля `WebModule` и приложения сервера использовалась концепция учета состояния. Поскольку протокол HTTP не зависит от состояния, нельзя гарантировать, что по окончании соединения значения свойств останутся прежними.

СОВЕТ

WebBroker — это один из способов предоставить данные Web-броузерам. Используя WebSnap можно существенно расширить возможности приложения, поскольку, кроме всего прочего, он позволяет осуществлять поддержку сеансов и работу со сценариями.

Чтобы запустить этот пример, убедитесь, что предварительно скомпилировали и зарегистрировали приложение из примера `Join2`. Затем скомпилируйте Web-приложение (версии CGI или ISAPI) и поместите исполняемый файл в каталог Web-сервера, обладающий правом выполнять сценарии (script-capable). Коду в каталоге сценариев необходим также файл `join.htm`, поэтому скопируйте туда и его. Теперь остается набрать в командной строке броузера адрес **`http://localhost/scripts/WebJoin.exe`** и просмотреть результаты работы этого примера.

Пример использования такого подхода можно найти на прилагаемом CD в каталоге `\WebBrok` раздела, посвященного настоящей главе.

Компоненты InternetExpress

С помощью компонентов InternetExpress можно расширить функциональные возможности простого модуля `WebModule` и улучшить разрабатываемое клиентское приложение. Это возможно благодаря использованию в них таких открытых стандартов, как XML и JavaScript. Используя компоненты InternetExpress, можно создавать серверные приложения DataSnap, взаимодействующие лишь с броузером и не нуждающееся ни в элементах управления ActiveX, ни в какой-либо предварительной установке или настройке специального программного обеспечения. Достаточно лишь простого обращения броузера к Web-серверу.

При использовании компонентов InternetExpress на Web-сервере потребуется запустить дополнительное программное обеспечение. В данном примере будет использоваться приложение ISAPI (Internet Services Application Programming Interface — интерфейс прикладных программ служб Internet), но эту роль могут выполнять приложения CGI (Common Gateway Interface — общий шлюзовой интерфейс) или ASP (Active Server Pages — активные страницы сервера). Web-брокер получает запросы от броузера и передает их на сервер приложений. Это легко осуществить, поместив компоненты InternetExpress в приложение Web-брокера.

В рассматриваемом примере используется стандартное приложение DataSnap, в котором имеются наборы данных `Customers`, `Orders` и `Employees`. Наборы данных `Customers` и `Orders` связаны взаимоотношениями типа вложения (более подробная информация по этой теме приведена далее в настоящей главе), а набор данных `Employees` будет использоваться в качестве таблицы подстановок (lookup table). Воспользуйтесь прилагаемым исходным кодом для определения сервера приложений. После создания и регистрации сервера приложений можно будет сосредоточиться на построении приложения Web-брокера, которое будет взаимодействовать с сервером приложений.

Создайте новое приложение ISAPI, выбрав в меню File пункты New и Web Server Application в хранилище объектов. Поместите в модуль `WebModule` компонент `TDCOMConnection`. Он будет обеспечивать связь с сервером приложений. Поэтому задайте в качестве значения его свойства `ServerName` идентификатор `ProgID` сервера приложений.

Затем поместите в `WebModule` компонент `TXMLBroker`, расположенный во вкладке InternetExpress палитры компонентов, и установите значения его свойств `RemoteServer` и `ProviderName` равными `CustomerProvider`. Компонент `TXMLBroker` функционирует аналогично компоненту `TCLientDataSet`. Он будет использоваться

для получения пакетов данных с сервера приложений и передачи их броузеру. Основным отличием между пакетами данных компонентов TXMLBroker и TClientDataSet является то, что компонент TXMLBroker преобразует пакеты данных DataSnap в формат XML. Добавим также в WebModule компонент TCLientDataSet и свяжем его с провайдером Employees на сервере приложений. Позже этот компонент будет использоваться как источник данных для выборки.

Компонент TXMLBroker отвечает за соединение с приложением сервера и за навигацию по страницам HTML. Для настройки поведения приложения InternetExpress можно воспользоваться множеством параметров. Например, можно ограничить количество записей, передаваемых клиенту, или задать количество допустимых ошибок во время обновления.

Теперь необходимо определить способ передачи данных броузеру. Используя компонент TInetXPageProducer, можно применить технологию WebBroker для отображения страниц HTML в броузере. Однако компонент TInetXPageProducer также допускает визуальное создание Web-страниц в редакторе Web Page Editor.

Дважды щелкните на компоненте TInetXPageProducer, и на экране раскроется окно редактора Web-страниц. Этот визуальный редактор позволяет определить, какие элементы будут содержаться на данной Web-странице. Одной из самых интересных особенностей компонентов InternetExpress является их абсолютная открытость. Можно создавать свои собственные компоненты, которые будут использоваться в редакторе Web-страниц в соответствии с четко определенными правилами. Примеры компонентов InternetExpress содержатся в каталоге <DELPHI>\DEMOS\MIDAS\INTERNETEXPRESS\INETXCUSTOM directory.

СОВЕТ

Компонент TInetXPageProducer имеет свойство IncludePathURL. Содержащееся в нем значение существенно влияет на работоспособность всего приложения InternetExpress. Установите для него значение, которое соответствует виртуальному каталогу с файлами сценариев JavaScript, использующимися в приложении. Например, если файлы находятся в каталоге c:\inetpub\wwwroot\jscript, то свойство IncludePathURL должно иметь значение /jscript/.

В редакторе Web-страниц щелкните на кнопке Insert, и на экране появится диалоговое окно Add Web Component (рис. 21.7). В списке данного диалогового окна содержится перечень компонентов Web, которые могут быть добавлены на страницу HTML. Содержимое этого списка зависит от родительского компонента, выделенного в текущий момент в верхней левой области диалогового окна. Например, добавление в корневой узел Web-компонента DataForm позволит конечным пользователям просматривать и редактировать информацию из базы данных, представленную в виде некоторой формы.

Выделив в редакторе Web-страниц элемент DataForm, можно вновь щелкнуть на кнопке Insert. Обратите внимание: перечень доступных компонентов в этом случае будет отличаться от списка, представленного на предыдущем этапе. После выбора компонента FieldGroup во вкладке предварительного просмотра появится предупреждение, сообщающее о том, что свойству XMLBroker компонента FieldGroup не присвоено никакого значения. Заданные в редакторе свойств значения свойства XMLBroker сразу же отобразятся в окне предварительного просмотра редактора Web-

страниц. Если продолжить редактировать свойства или добавлять компоненты, состояние страницы HTML будет постоянно изменяться (рис. 21.8).

Рис. 21.7. *Диалоговое окно Add Web Component редактора Web-страниц*

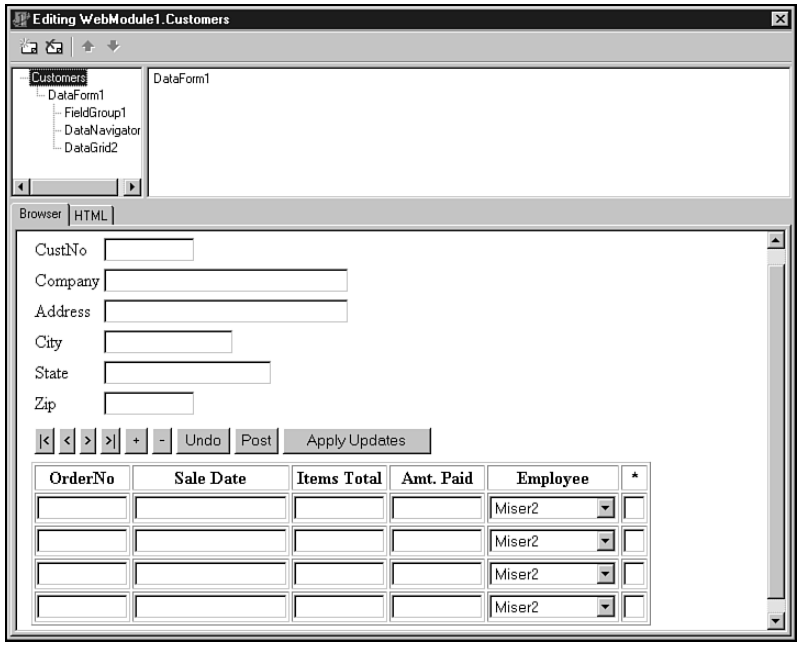

Рис. 21.8. *Редактор Web-страниц в процессе разработки страницы HTML*

Стандартные компоненты Web обладают обширными возможностями настройки. С помощью свойств легко изменить заголовки полей, тип выравнивания и цвет, добавить собственный код HTML и даже использовать *листы стилей* (style sheet). Более того, если компонент не до конца удовлетворяет требованиям, то можно создать его потомок и ис-

пользовать его в редакторе Web-страниц. Возможности среды разработки ограничиваются лишь воображением разработчика.

При использовании динамической библиотеки ISAPI ее необходимо поместить в виртуальный каталог, допускающий исполнение сценариев. Следует также переместить файлы JavaScript, содержащиеся в каталоге `<DELPHI>\SOURCE\WEBMIDAS`, в корректное местоположение на Web-сервере и модифицировать свойство `TInetXPageProducer.IncludePathURL` таким образом, чтобы в качестве его значения использовался адрес URL файлов JavaScript. После выполнения всех перечисленных действий страница будет готова к просмотру.

Для доступа к странице необходим броузер, поддерживающий JavaScript. Укажите в броузере адрес `http://localhost/inetx/inetxisapi.dll` — и данные будут отображены, как показано на рис. 21.9.

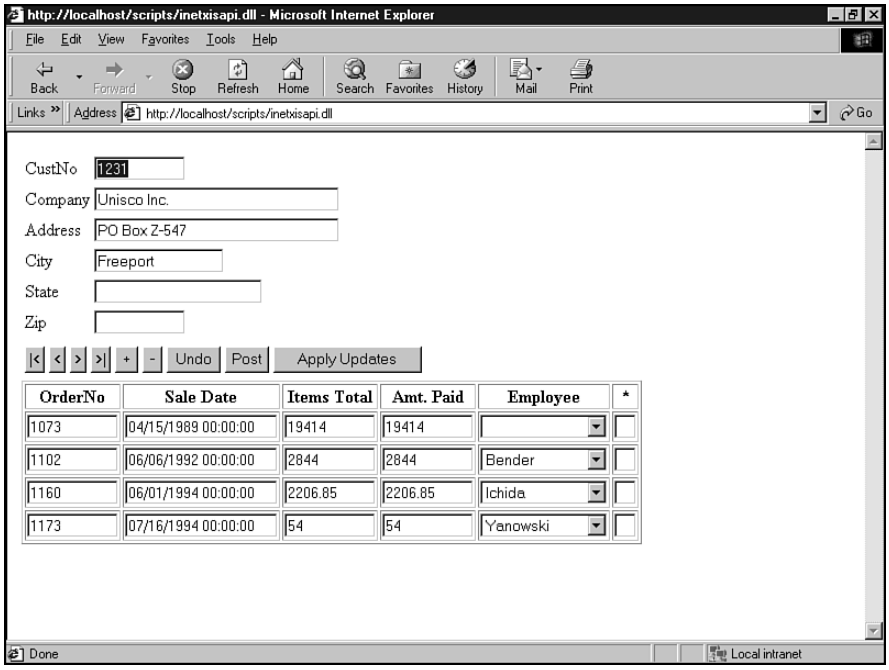

Рис. 21.9. *Web-страница приложения InternetExpress в окне Internet Explorer*

И, наконец, в процессе применения обновлений можно предотвращать ошибки доступа подобно тому, как это было в автономном приложении DataSnap. такую возможность применяют, если значение свойства `TXMLBroker.ReconcileProducer` равно `TPageProducer`. При возникновении ошибки данному свойству будет присвоено значение свойства `Content` компонента `TPageProducer`, которое затем будет возвращено конечному пользователю.

После установки пакета `InetXCustom.dpk`, содержащегося в каталоге `<DELPHI>\DEMOS\MIDAS\INTERNETEXPRESS\INETXCUSTOM`, станет доступен компонент `TReconcilePageProducer`, представляющий собой специализированный тип компонента

TPageProducer. Этот компонент создает код HTML, который функционирует аналогично диалоговому окну Reconciliation Error в стандартном приложении DataSnap (рис. 21.10).

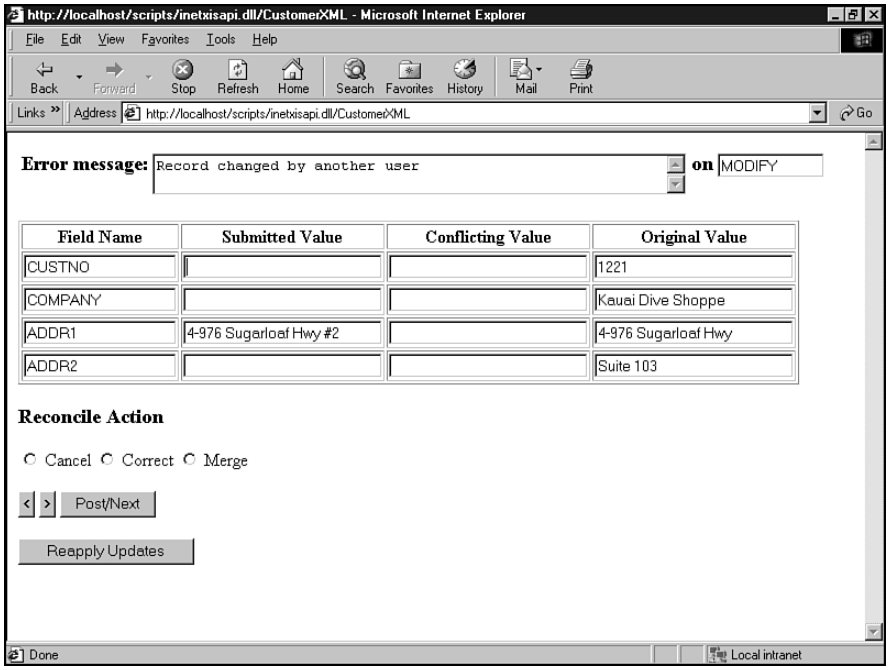

Рис. 21.10. *Страница HTML, созданная компонентом TReconcilePage-Producer*

Пример использования такого подхода можно найти на прилагаемом CD в каталоге \InetX раздела, посвященного настоящей главе.

Дополнительные возможности наборов данных клиента

Управление компонентом TClientDataSet осуществляется при помощи множества разнообразных параметров. В данном разделе будут рассмотрено применение компонента TClientDataSet, упрощающего код в сложных приложениях.

Двухуровневые приложения

Теперь, изучив, как в трехуровневом приложении поставить в соответствие компоненту ClientDataSet провайдера, а значит и данные, можно задаться вопросом; как выполнить подобные действия в двухуровневом приложении? Ведь зачастую создать подобное приложение достаточно просто. Для этого существует четыре возможности.

- Присвоить данные во время выполнения.
- Присвоить данные во время разработки.

- Назначить провайдера во время выполнения.
- Назначить провайдера во время разработки.

При работе с компонентом `ClientDataSet` существует две основные возможности: либо использовать свойство `AppServer`, либо обращаться к данным напрямую. В первом случае между компонентами `TClientDataSet` и `TDataSetProvider` нужно установить связь, что позволит им взаимодействовать друг с другом. Если же потребуется работать непосредственно с самими данными, то в распоряжении программиста окажется эффективный механизм локального хранения данных. При этом для получения данных нет необходимости обращаться к компоненту `TDataSetProvider`.

Для получения компонентом `TCLientDataSet` данных непосредственно от компонента `TDataSet` во время выполнения программы используйте код, приведенный в листинге 21.9.

Листинг 21.9. Код для получения данных непосредственно от компонента `TDataSet`

```
function GetData(ADataset: TDataset): OleVariant;
begin
  with TDatasetProvider.Create(nil) do
  try
    Dataset:= ADataset;
    Result:= Data;
  finally
    Free;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ClientDataset1.Data:= GetData(ADOTable1);
end;
```

При использовании такого метода в Delphi 6 требуется больше усилий и кода, чем в предыдущих версиях, в которых было достаточно свойству `ClientDataSet1.Data` присвоить свойство `Table1.Provider.Data`. Однако функция `GetData()` позволяет скрыть дополнительный код.

Для того чтобы из компонента `TDataSet` извлечь данные с помощью компонента `TClientDataSet` в процессе разработки, выберите в контекстном меню компонента `TClientDataSet` пункт Assign Local Data. Затем задайте компонент `TDataSet`, в котором содержатся необходимые данные, и они будут помещены в свойство `TClient-DataSet.Data`.

СОВЕТ

Если сохранить файл в этом состоянии и сравнить размер файла DFM с размером данных до выбора этого пункта, то можно заметить, что размер файла DFM увеличился. Это связано с тем, что в файле DFM Delphi сохраняет все метаданные и записи, ассоциированные с компонентом TDataSet. Эти данные будут переданы в файл DFM только в том случае, если значение свойства Active компонента TClientDataSet установлено равным True. Объем файла также можно сократить, выбрав в контекстном меню компонента TClientDataSet пункт Clear Data.

Для того чтобы воспользоваться всей гибкостью, предоставляемой провайдером, необходимо обратиться к свойству AppServer. Нужное значение этому свойству можно присвоить во время выполнения. Указанное можно осуществить в методе FormCreate примерно так:

```
ClientDataset1.AppServer:=TLocalAppServer.Create(Table1);
ClientDataset1.Open;
```

И, наконец, установить значение свойства AppServer можно во время разработки. Если значение свойства RemoteServer компонента TCLientDataSet не установлено, то свойству TClientDataSet.ProviderName можно присвоить значение TDataSetProvider.

Основным недостатком использования свойства TClientDataset.ProviderName является то, что оно не может быть связано с провайдерами, расположенными в другой форме или модуле данных во времени разработки. Поэтому в Delphi 6 появился компонент TLocalConnection. Он самостоятельно обнаруживает и обеспечивает доступ ко всем найденным компонентам TDatasetProvider того же самого владельца. Чтобы использовать этот метод связи с провайдерами, назначьте свойству ClientDataset.RemoteServer компонент LocalConnection во внешней форме или модуле данных DataModule. По завершении этих действий будет получен список провайдеров для данного компонента LocalConnection в свойстве ClientDataset.ProviderName.

Основным отличием между использованием компонента TDataSet и компонента TClientDataSet является то, что при использовании компонента TClientDataSet в качестве посредника между запросами к данным компонента TDataSet применяется интерфейс IAppServer. Это позволяет манипулировать свойствами, методами, событиями и полями компонента TClientDataSet, но не компонента TDataSet. Можно считать, что данный компонент содержится в отдельном приложении и, следовательно, им нельзя управлять непосредственно из кода. Поместите все серверные компоненты в отдельный модуль DataModule. Размещение компонентов TDatabase, TDataSet и TCDSProvider в отдельном модуле DataModule позволяет заранее подготовить создаваемое приложение к предстоящей установке в многоуровневой среде. Такая организация приложения обладает еще одним преимуществом: при ее использовании модуль DataModule можно рассматривать как объект, доступ к которому для клиента будет затруднен. Это также будет способствовать подготовке приложения к переносу в многоуровневую среду, поскольку затруднит разработчикам создание таких связей, которые впоследствии будут препятствовать переносу.

Классические ошибки

При создании многоуровневого приложения типичной ошибкой является размещение на уровне представления излишней информации об уровне данных. Как правило, проверки лучше производить на уровне представления, но способ выполнения такой проверки определяет удобство ее применения в многоуровневом приложении.

Например, при передаче динамических операторов SQL от клиента серверу необходимо, чтобы клиентская часть была четко синхронизирована с уровнем данных. По этой причине возрастает количество перемещений данных, которые должны быть скоординированы во всем приложении. При изменении структуры одной таблицы на уровне данных потребуется обновить все клиентские приложения, где используются динамические операторы SQL, чтобы они могли передавать на сервер корректные операторы. Это, естественно, ограничивает преимущества "тонких" клиентских приложений.

В качестве другого примера можно привести ситуацию, когда клиентское приложение пытается управлять временем жизни транзакции, вместо предоставления такой возможности приложению уровня бизнес-правил. Зачастую это реализуется с помощью публикации методов `BeginTransaction()`, `Commit()` и `RollBack()` экземпляра класса `TDataBase` на сервере и последующего вызова их из клиентской части приложения. При этом код клиента становится более сложным для сопровождения, а главное — нарушается принцип, согласно которому лишь уровень представления должен отвечать за взаимодействие с уровнем данных. Никогда не следует полагаться на такой подход. Наоборот, обновления необходимо передавать на уровень бизнес-правил, который и будет производить обновление данных в транзакции.

Установка приложений DataSnap

После создания приложения DataSnap последним барьером, который нужно преодолеть, является его установка. В настоящем разделе обсудим, что следует сделать для безболезненной установки приложения, созданного при использовании технологии DataSnap.

Проблемы лицензирования

С момента появления технологии DataSnap в Delphi 3 многих пользователей постоянно интересует вопрос предоставления лицензий. Распространение приложений, созданных с применением этой технологии, усложняется огромным количеством нюансов. В данном разделе описываются общие требования, определяющие необходимость приобретения лицензии DataSnap. Единственный законный документ, связанный с лицензированием, содержится в файле `DEPLOY.TXT`, находящемся в корневом каталоге Delphi. И, наконец, последней инстанцией, которая может ответить на вопросы, возникшие в конкретной ситуации, является региональный отдел продаж компании *Borland*. Дополнительные руководства и примеры можно найти по адресу `http://www.borland.com/ midas/papers/licensing/` или на Web-сайте `http://www.xapware.com/ddg`.

Информация в этом документе подготовлена таким образом, чтобы можно было получить ответы на вопросы, возникающие в некоторых общих случаях использования техно-

логии DataSnap. Здесь также содержится информация о ценах и условиях приобретения лицензии.

В процессе принятия решения о приобретении лицензии DataSnap основным критерием является вопрос о том, будут ли пакеты данных DataSnap передаваться за пределы компьютера. Если это так, то лицензию приобрести необходимо. В противном случае (как, например, в случае одно- и двухуровневых приложений, рассмотренных ранее) можно использовать технологию DataSnap без лицензии.

Настройка DCOM

Настройка протокола DCOM является и наукой, и искусством. Полная и безопасная настройка DCOM определяется очень многими аспектами, поэтому в данном разделе познакомимся лишь с основами этой "черной магии".

После регистрации приложение сервера можно настроить с помощью утилиты `DCOMCNFG` корпорации *Microsoft*. Данная утилита входит в комплект поставки системы Windows NT, а для компьютеров под управлением Windows 9x ее потребуется установить отдельно. Сразу же следует отметить, что в утилите `DCOMCNFG` существует много ошибок. Наиболее примечательной из них является то, что утилита может быть запущена лишь на тех компьютерах под управлением Windows 9x, на которых применяется режим совместного использования ресурсов на уровне пользователей. А это, безусловно, требует наличия домена, что не всегда возможно или желательно в одноранговых сетях (например в сети из двух компьютеров под управлением Windows 9x). В результате многие пользователи склонны думать, что для запуска утилиты настройки протокола DCOM необходим компьютер под управлением Windows NT.

Если утилита `DCOMCNFG` запущена, то можно выделить зарегистрированное приложение сервера и щелкнуть на кнопке Properties, чтобы вывести на экран информацию о сервере. Лучше всего начинать изучение утилиты `DCOMCNFG` со вкладки Identity. Для зарегистрированного объекта сервера по умолчанию используется режим Launching User. Корпорация *Microsoft* при всем своем желании не могла бы принять худшего решения.

При создании сервера подпрограммами DCOM обычно используется контекст безопасности пользователя, указанного на странице Identity. Если выбран режим Launching User, то для каждого отдельного зарегистрированного пользователя будет запущен новый процесс. Во многих случаях пользователи выбирают режим создания экземпляров `ciMultiple`, а потом удивляются, почему создается несколько копий сервера. Например, если с сервером соединяется пользователь A, а затем — пользователь B, то для пользователя B будет запущен новый процесс. Кроме того, если пользователь регистрируется на машине под именем, отличным от используемого сервером в данный момент, то его графический интерфейс будет недоступен этому пользователю. Это обусловлено концепцией системы Windows NT, известной под названием *станций Windows* (Windows stations). Осуществлять запись в графический пользовательский интерфейс на станции Windows может лишь пользователь имеющий статус `Interactive User`. Таким пользователем является тот, кто зарегистрирован на сервере в текущий момент. В общем, при настройке сервера приложения никогда не используйте режим Launching User.

Следующим интересным моментом является имеющийся во вкладке Identity режим Interactive User. Он означает, что пользователь, создавший сервер, будет работать в кон-

тексте пользователя, зарегистрированного на сервере в текущий момент времени. При этом можно взаимодействовать с сервером визуально. Но, к сожалению, большинство системных администраторов запрещают бездействовать, зарегистрировавшись на компьютере Windows NT. Кроме того, если зарегистрировавшийся пользователь решит закончить работу, то приложение сервера перестанет работать нормально.

Во вкладке Identity остался еще один режим — This User. Если выбран этот режим, то все клиенты будут создавать одно приложение сервера, а также пользоваться правами доступа и контекстом пользователя, определенного во вкладке Identity. Это также означает, что для запуска сервера компьютер Windows NT не потребует регистрации пользователя. Побочным эффектом такого подхода является то, что при использовании данного параметра не будет отображаться графический пользовательский интерфейс сервера. Однако, чтобы обеспечить правильную работу приложения сервера, этот режим следует предпочесть всем остальным.

После корректной настройки объекта сервера обратите внимание на вкладку Security. Убедитесь, что пользователь, который будет запускать этот объект, имеет соответствующие права доступа. Удостоверьтесь также в том, что доступ к серверу предоставлен и пользователю `SYSTEM`, в противном случае возникнут ошибки.

В процессе настройки протокола DCOM существует множество нюансов. Для получения самой свежей информации об этом протоколе, особенно в случае его использования совместно с Windows 9x, Delphi и DataSnap, посетите Web-сайт по адресу `http://www.DistribuCon.com/dcom95.htm`.

Файлы, необходимые для установки приложения

Требования к установке приложения DataSnap изменяются с выходом каждой новой версии Delphi.

Для установки приложения DataSnap в Delphi 6 необходим минимальный набор файлов. Все они указаны в приведенном ниже списке.

Для установки серверной части приложения выполните следующие действия (имеется в виду сервер COM; от других серверов он отличатся несущественно):

1. Скопируйте приложение сервера в каталог, обладающий достаточными правами доступа на разделе NTFS или набором прав общего доступа при установке на Win9x.
2. Установите свой уровень доступа к данным достаточным для того, чтобы приложение сервера могло взаимодействовать в качестве клиента с RDBMS (например BDE, MDAC, специфическими для клиента библиотеками баз данных и т.д.).
3. Скопируйте файл `MIDAS.DLL` в каталог `%SYSTEM%`. Для компьютеров, работающих под управлением Windows NT, таким каталогом по умолчанию является `C:\Winnt\System32`, а для компьютеров, на которых установлена система Windows 9x, — `C:\Windows\System`.
4. Запустите приложение сервера, чтобы зарегистрировать его как объект COM.

Для установки клиентской части выполните следующие действия.

1. Скопируйте клиентское приложение в каталог вместе со всеми другими внешними файлами, используемыми клиентом (например пакетами времени выполнения, библиотеками DLL, элементами управления ActiveX и т.д.).
2. Скопируйте файл `MIDAS.DLL` в каталог `%SYSTEM%`. Обратите внимание, что Delphi 6 может статически скомпоновать библиотеку `MIDAS.DLL` в создаваемое приложение, сделав таким образом данный этап ненужным. Для этого достаточно добавить модуль `MidasLib` в раздел `uses` и перекомпилировать приложение. Из-за статической компановки размер файла EXE заметно увеличится.
3. Дополнительно. Если в клиентском приложении задается свойство `ServerName` компонента `TDispatchConnection` или используется раннее связывание, то необходимо зарегистрировать файл библиотеки типов сервера (TLB). Для этого воспользуйтесь утилитой из каталога `<DELPHI>\BIN\TREGSVR.EXE` (при желании указанное можно осуществить и программно).

Соглашения по установке приложений в Internet (брандмауэры)

Для установки приложения в локальной сети не существует никаких ограничений. Можно выбрать любой тип соединения, лучше всего соответствующий данному приложению. Однако при установке приложения в Internet может возникнуть множество препятствий, среди которых в первую очередь следует отметить *брандмауэры* (firewall).

Протокол DCOM является далеко не самым дружественным по отношению к брандмауэрам. Он требует открыть на брандмауэре несколько портов. Большинство системных администраторов не торопятся предоставлять целый диапазон портов, поскольку это подталкивает хакеров к активным действиям. При использовании компонента `TSocketConnection` ситуация несколько улучшается, поскольку позволяет открыть лишь один порт. Но некоторые системные администраторы могут отказаться и от этого на том основании, что при этом нарушается система безопасности.

Компонент `TWebConnection` является производным от компонента `TSocketConnection` и позволяет преобразовать трафик DataSnap в поток протокола HTTP, использующего самый открытый порт в мире — порт HTTP (по умолчанию порт 80). На самом деле этот компонент поддерживает даже протокол SSL, благодаря чему можно воспользоваться безопасными соединениями и избежать проблем с брандмауэрами. В конце концов, если какая-либо корпорация не предоставит возможности соединения по протоколу HTTP, то связаться с ней не удастся никакими другими средствами.

Это небольшое чудо осуществляется за счет расширения ISAPI компании *Borland*, с помощью которого трафик HTTP можно преобразовать в поток DataSnap и наоборот. В этом отношении динамическая библиотека ISAPI будет работать точно так же, как и библиотека `ScktSrvr` для соединений типа сокетов. Расширение ISAPI (библиотека `httpsrvr.dll`) должно находиться в каталоге, допускающем выполнение кода. Например, в случае IIS4 для этого файла по умолчанию будет использоваться каталог `C:\Inetpub\Scripts`.

Еще одним преимуществом использования компонента `TWebConnection` является поддержка буферизации объектов (pooling). Это позволяет сэкономить ресурсы сервера, затрачиваемые на создание объектов при подключении к ним клиентов. Более того, механизм буферизации позволяет установить максимально возможное количество таких объектов. После достижения максимально допустимого количества объектов оче-

редному клиенту будет послано сообщение об ошибке. В этом сообщении содержится информация о том, что сервер занят и не может обработать поступивший запрос клиента. Такой подход гораздо более гибок, чем простое создание случайного числа потоков для каждого клиента, который хочет соединиться с сервером.

Следующим шагом в этом направлении будет создание модуля удаленных данных в качестве Web-службы. Применение модуля данных SOAP не только обеспечивает все преимущества компонента `TWebConnection`, но и позволяет клиенту использовать промышленный стандарт протокола SOAP. Кроме того, это обеспечивает для сервера приложений возможность использовать такие системы, как .Net, Sun ONE и другие совместимые с SOAP промышленные стандарты.

Если в приложении DataSnap используется буферизация модуля удаленных данных, то в его методе `UpdateRegistry` необходимо вызывать методы `RegisterPooled` и `UnRegisterPooled` (пример реализации метода `UpdateRegistry` содержится в листинге 21.1). Вызов метода `RegisterPooled` осуществляется приблизительно так:

```
RegisterPooled(ClassID, 16, 30);
```

В данной строке приложению DataSnap сообщается, что в буфере может располагаться 16 объектов и любой экземпляр объекта будет удален, если он не был активен в течение 30 минут. Если объекты удалять не нужно, то присвойте этому параметру нулевое значение.

В клиентскую часть приложения не нужно вносить никаких существенных изменений. Просто используйте компонент `TWebConnection` вместо компонента `TDispatchConnection`, заполните его соответствующие свойства и клиент будет готов взаимодействовать с сервером по протоколу HTTP. При использовании компонента `TWebConnection` вместо задания имени или адреса компьютера, на котором установлен сервер, следует указать полный адрес URL файла `httpsrvr.dll`. На рис. 21.11 представлен типичный фрагмент установки с использованием компонента `TWebConnection`.

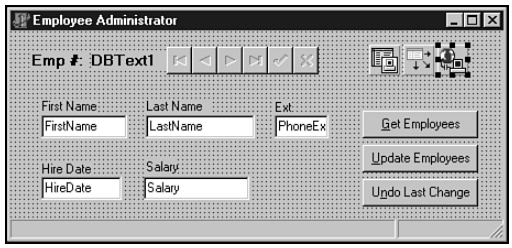

Рис. 21.11. *Установка компонента `TWebConnection` во время разработки*

Еще одним преимуществом использования транспортного протокола HTTP является то, что операционная система типа Windows NT Enterprise позволяет объединять серверы в кластеры. При этом для серверной части приложения обеспечивается реальная отказоустойчивость и балансировка загрузки. Более подробная информация о кластеризации приведена на Web-странице по адресу:

```
http://www.microsoft.com/ntserver/ntserverenterprise/exec/overview/
clustering
```

Ограничения, обусловленные использованием компонента `TWebConnection` относительно невелики, при этом они с лихвой окупаются увеличением количества клиентов, которые могут обращаться к приложению сервера. Эти ограничения заключаются в том, что на компьютере клиента необходимо установить файл `wininet.dll`, а при использовании компонента `TWebConnection` нельзя применять обратные вызовы.

Резюме

В настоящей главе представлены краткие сведения о технологии DataSnap. Здесь обозначены лишь основные возможности этой технологии, а рассмотрение всех ее возможностей выходит за рамки одного раздела. Даже изучив все нюансы DataSnap, можно постоянно совершенствовать свои знания и расширять возможности создаваемых приложений, используя C++Builder и JBuilder. В среде JBuilder с использованием все той же технологии DataSnap и концепций, изученных в данной главе, можно реализовать доступ к серверу приложения с различных платформ и, тем самым, достичь истинного совершенства.

Технология DataSnap быстро развивается, и каждый программист теперь может выбрать средства создания многоуровневых приложений. Однажды ощутив подлинную мощь приложений DataSnap, вы уже никогда не вернетесь к разработке обычных приложений баз данных.

ЧАСТЬ

VI

Программирование для Internet

В ЭТОЙ ЧАСТИ...

ГЛАВА

22

Разработка приложений ASP

В ЭТОЙ ГЛАВЕ...

В настоящей главе рассказывается о том, что представляют собой *активные страницы сервера* (ASP — Active Server Page) и *активные объекты сервера* (ASO — Active Server Object), а также о том, как Delphi 6 может помочь в создании и установке активных объектов сервера.

Понятие активного объекта сервера

Подобно *общему шлюзовому интерфейсу* (CGI — Common Gateway Interface), *интерфейсу прикладных программ служб Internet* (ISAPI — Internet Services Application Programming Interface) и *интерфейсу прикладных программ сервера Netscape* (NSAPI — Netscape Server Application Programming Interface), поддерживаемых WebBroker, ASP представляет собой серверное решение Web-приложений. Это означает, что активные страницы сервера и активные объекты сервера можно разместить на Web-сервере и обеспечить клиентам возможность, соединившись с ним, загружать эти страницы и объекты. Данная глава посвящена в основном активным объектам сервера, создаваемым в Delphi 6, но используемым в активных страницах.

В Delphi 5 был внедрен новый мастер, который позволяет создавать активные объекты сервера. Эти объекты могут быть использованы в страницах ASP для динамического формирования кода HTML при каждой загрузке страниц с сервера. В настоящей главе изложено, что представляют собой объекты активных страниц сервера, как они взаимодействуют с интерфейсами CGI, ISAPI и COM, а также то, как они могут быть использованы в контексте активных страниц сервера. Затем внимание будет уделено изучению важнейших аспектов создания активных объектов сервера, представляющих собой серверные компоненты, а также различиям между операционными системами (такими как Windows NT версии 4 и Windows 2000) и различиям между информационными серверами Internet (Web) (такими как IIS версии 3, 4 и 5). Все указанные факторы влияют на способ взаимодействия с активными объектами сервера.

В качестве примера создадим простой активный объект сервера и сценарий шаблона, добавим несколько методов и приспособим объект и сценарий для выполнения необходимых задач. Затем установим и зарегистрируем объект на Web-сервере. И в заключении рассмотрим, как установить новые версии активных объектов сервера, как их проверить и отладить.

Активные страницы сервера

Прежде чем приступить к созданию собственных активных объектов сервера, имеет смысл ознакомиться с основами технологии и синтаксисом активных страниц сервера, которые служат операционной средой для активных объектов сервера. Активные страницы сервера позволяют использовать язык сценариев, который интерпретируется Web-сервером, а не Web-броузером. Указанное означает, что для проверки работоспособность исходного кода, приведенного в листингах и примерах настоящей главы, необходимо установить Web-сервер. Авторы использовали *Internet Information Server* (IIS), версия 4, от *Microsoft* на платформе Windows NT 4, а также IIS 5 на Windows 2000, но *Personal Web Server* (PWS) на Windows 95 или 98 работает точно также. В то время как обычные страницы HTML имеют расширение `.htm` или `.html`, файлы страниц ASP имеют расширение `.asp`. Для того, чтобы Web-сервер мог выполнять страницы ASP, их необходимо поместить в каталог, обладающий правами на

выполнение сценариев. При установке Web-сервера *Microsoft* по умолчанию создается каталог `Scripts`. Но даже если такого каталога не существует, то можно создать новый виртуальный каталог и присвоить ему право выполнять сценарии. Для этого в Windows NT запустите диспетчер служб Internet (или консоль управления Microsoft), перейдите к Web-службе и добавьте новый виртуальный каталог (например по имени `Scripts` или `cgi-bin`). Удостоверьтесь, что возможность исполнения сценариев, параметр Scripting, разрешена.

Сценарий на сервере можно изменять безо всякой перекомпиляции кода или перезапуска Web-сервера. Операторы сценариев находятся между дескрипторами `<%` и `%>`, а в качестве *языка активных сценариев* (active scripting language) используются JavaScript и VBScript, которые нетрудно понять и изучить.

В качестве дополнительной поддержки активные страницы сервера обладают рядом встроенных объектов, которыми они могут воспользоваться для обмена данными со средой броузера и сервера.

Чаще всего используются два объекта:

- `Request` (запрос) — предназначен для ввода данных пользователем. Объект `Request` способен обращаться к введенным в форме переменным и проверять их значения.
- `Response` (ответ) — используется для формирования ответа пользователю. Объект `Response` обладает методом `write`, который применяются для создания результирующего кода HTML.

В качестве небольшого примера рассмотрим сценарий ASP, который проверяет вводимое значение переменной `Name`, и если введено слово `Bob`, то объект `Response` выведет на экран приветствие `"Hello, Bob!"` (Привет, Боб!), в противном случае — сообщение `"Hello, User!"` (Привет, пользователь!):

```
<%
  if Request("Name") = "Bob" then
      Response.Write("Hello, Bob!")
  else
      Response.Write("Hello, User!")
  end if
%>
Если этот код ASP содержится на странице по имени test.asp, то для
его вызова может быть использована следующая форма HTML:
<FORM ACTION="test.asp" METHOD=POST>
Name: <INPUT TYPE=text NAME=Name>
<P>
<INPUT TYPE=submit>
</FORM>
```

НА ЗАМЕТКУ

Для доступа к исходной переменной по имени `Name` можно использовать переменные объекта ASP `Request`.

Помните, что активные страницы могут быть выполнены только Web-сервером, причем только в том случае, если Web-сервер поддерживает обработку ASP. Это озна-

чает, что используемый для обращения к странице URL должен быть адресом выполняющегося (а не остановленного) Web-сервера. Поэтому если файл `test.asp` находится в каталоге `\cgi-bin`, то не стоит обращаться к нему как к файлу по адресу `file:///d:/www/cgi-bin/test.asp`, поскольку это не будет обращением к Web-серверу и на экране будет отображено (в лучшем случае) лишь содержимое самого файла ASP, его исходный код. Но URL `http://localhost/cgi-bin/test.asp` обращается уже к Web-серверу (хоть и локальной машины), и на экран будет выдан результат выполненная активной страницы сервера.

На первый взгляд, все кажется просто. Кроме того, разработчику Delphi необязательно создавать все Web-приложение для Internet, используя лишь сервер-ориентированные сценарии ASP. Учтите вопросы эффективности, ведь сценарии ASP не подлежат компиляции, их приходится интерпретировать. Одним из главных преимуществ ASP считается возможность изменять их исходный код на лету, безо всякой необходимости повторной компиляции и переустановки. Однако по мере увеличения и усложнения Web-сайтов это преимущество обернулось недостатком производительности, связанной с необходимостью применения интерпретатора. К счастью, язык сценариев ASP позволяет создавать и использовать специальные *активные объекты сервера COM* (Active Server COM Objects), которые располагаются на сервере. Поскольку это откомпилированные бинарные объекты, они выполняются быстрее, а следовательно, обладают большей эффективностью. Вот где проявляется преимущество Delphi 6 версии Enterprise, которая позволяет создавать активные объекты сервера.

Мастер активных объектов сервера

Delphi 6 Enterprise содержит мастера, существенно ускоряющие создание активных объектов сервера. Активные объекты можно создавать и в Delphi 6 Professional, но тогда вручную придется выполнять бо́льшие объемы работ. Пользователи, будьте благоразумны, если время создания приложения является критическим фактором, то рассмотрите возможность перехода на версию Enterprise.

Хранилище объектов (Object Repository) Delphi 6 Enterprise содержит во вкладке ActiveX мастера, позволяющие создавать новые активные объекты сервера. Чтобы создать новый активный объект сервера (упоминаемый далее как *объект ASP*), следует закрыть все открытые проекты (если они есть) и начать новый проект *библиотеки ActiveX* (ActiveX Library), которая будет содержать создаваемый объект ASP. Для этого необходимо выполнить следующие действия.

1. Запустите Delphi 6 и закройте проект по умолчанию.
2. В меню File (Файл) выберите пункты New (Новый) и Other (Другой), а затем во вкладке ActiveX хранилища объектов Delphi 6 (рис. 22.1) выберите пиктограмму ActiveX Library (Библиотека ActiveX).
3. Сохраните проект библиотеки как файл **`D6ASP.dpr`**.

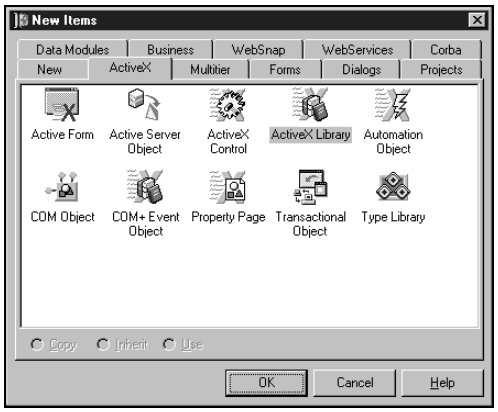

Рис. 22.1. *Пиктограмма ActiveX Library во вкладке ActiveX*

СОВЕТ

Для тех, кому надоело закрывать проект по умолчанию при каждом запуске Delphi, будет полезно узнать, что существует простой способ сделать так, чтобы Delphi запускался, не загружая новый проект. Для этого используется параметр командной строки `-np`. Перейдите в группу программ, содержащую ярлыки Delphi 6, и измените способ запуска.

Для этого щелкните правой кнопкой мыши на панели задач и выберите в контекстном меню пункт Properties (Свойства). Перейдите во вкладку Start Menu Programs (Настройка меню) и щелкните на кнопке Advanced (Дополнительно). Теперь рассмотрим пункты меню Start (Пуск). Перейдите в группу программ All Users (Все пользователи), которая будут содержать группу ярлыков Borland Delphi 6. Выберите элемент Delphi 6, щелкните на нем правой кнопкой мыши и в появившемся контекстном меню выберите параметр Properties. Перейдите во вкладку Shortcut (Ярлык) и добавьте в окне редактирования Target (Объект), справа от текущего значения, параметр **`-np`**, чтобы командная строка выглядела в результате примерно так:

```
"C:\Program Files\Borland\Delphi6\Bin\delphi32.exe " -np
```

Кроме того, это хорошее место, чтобы пересмотреть значение параметра Start In (Рабочая папка), определяющего исходный каталог для запуска Delphi 6.

Теперь, сохранив только что созданную библиотеку ActiveX (под именем `D6ASP.dpr`), в нее можно добавлять активные объекты сервера. Для этого достаточно выбрать во вкладке ActiveX хранилища объектов Delphi 6 пиктограмму Active Server Object (Активный объект сервера), как показано на рис. 22.1.

Это вызовет диалоговое окно Delphi 6 New Active Server Object (Новый активный объект сервера), показанное на рис. 22.2. Для тех, кто видит такое окно впервые (особенно при отсутствии опыта работы с объектами COM и ASP), приведем ряд пояснений.

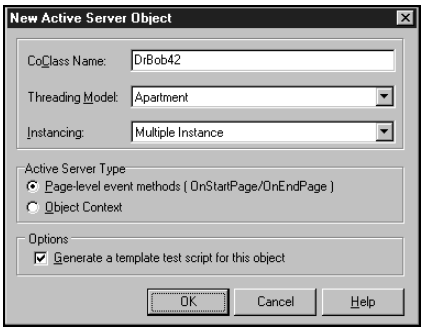

Рис. 22.2. *Диалоговое окно* New Active Server Object

В поле редактирования CoClass Name указывается внутреннее имя объекта COM. Обычно здесь можно ввести любое имя. Например, в настоящей главе использовано имя `DrBob42`. Это приведет к тому, что именем класса, производного от `TASPObject`, будет `TDrBob42`, а реализуемый интерфейс, соответственно, `IDrBob42`. Параметр Threading Model (Потоковая модель) установлен по умолчанию в состояние `Apartment`, а Instancing (Способ создания экземпляра) — в состояние `Multiple Instance` (Несколько экземпляров). Это подходит для большинства случаев, поэтому изменять данные установки не приходится почти никогда.

Параметр потоковой модели может принимать следующие значения:

- `Single` (Одиночная) — все запросы клиентов обрабатываются в отдельном потоке. Это не самый лучший выбор, ибо другие запросы должны ожидать завершения обработки запроса первого клиента.
- `Apartment` (Раздельная) — запрос каждого клиента обрабатывается в собственном потоке, изолированном от других потоков. (Ни один из потоков не имеет доступа к состоянию другого.) Данные экземпляра класса должны быть защищены объектами синхронизации потоков, в противном случае при использовании глобальных переменных можно столкнуться с проблемами совместного доступа к данным. Это наиболее предпочтительная потоковая модель.
- `Free` (Свободная) — доступ к экземпляру класса одновременно могут осуществлять многие потоки. Данные экземпляра класса никак не защищены, поэтому необходимо самостоятельно принимать меры предотвращения проблем многопользовательской среды.
- `Both` (Комбинированная) — комбинация значений `Apartment` и `Free`. В основе этого подхода лежит свободная потоковая модель, за одним исключением: обратный вызов осуществляется в том же потоке. (Таким образом, параметры в функциях обратного вызова защищены от проблем многопоточной обработки.)
- `Neutral` (Нейтральная) — используется, в основном, в условиях COM+. Объекты COM применяют преимущественно модель `Apartment`. Клиентские запросы получают доступ к экземплярам объекта в разных потоках, а ответственность за предотвращение конфликтов обращений принимает на себя COM. Тем не менее решать проблемы, связанные с использованием глобальных перемен-

ных (см. главу, посвященную многопоточному режиму), равно как и проблемы защиты данных в промежутках между вызовами методов, придется самостоятельно.

Параметр Instancing предлагает выбор из трех вариантов. Обратите внимание: если активный объект сервера зарегистрировать как внутренний сервер процесса (in-process), то выбор этого параметра не имеет значения. (Внутренний и внешний сервера процесса рассматриваются позднее.) Но все же необходимо знать, какие значения может принимать этот параметр:

- `Internal Instance` (Внутренний экземпляр) — такой экземпляр объекта COM существует только в рамках собственной библиотеки DLL.
- `Single Instance` (Единственный экземпляр) — приложение может иметь только один клиентский экземпляр.
- `Multiple Instance` (Множественные экземпляры) — одно приложение (библиотека ActiveX) может создать более одного экземпляра объекта COM.

Кроме того, в этом диалоговом окне можно выбрать тип сервера (Active Server Type). Он зависит от версии установленного сервера IIS. Для IIS 3 или IIS 4 используются такие методы событий уровня страницы, как `OnStartPage` и `OnEndPage`, в то время как серверы IIS 4 и IIS 5 могут использовать и методы контекста объекта. Например, для манипулирования данными экземпляра активного объекта сервера можно применить *сервер транзакций Microsoft* (MTS — Microsoft Transaction Server) или COM+.

Delphi 6 учитывает большую часть этих различий, следовательно, для данного примера можно оставить параметр Page-Level Event Methods (Методы событий уровня страницы), устанавливаемый по умолчанию. Точно так же можно поступить и в отношении активных объектов сервера, если выбран параметр Object Context (Контекст объекта). Не забывайте, что параметр необходимо выбрать соответствующий (или, по крайней мере, поддерживаемый) установленному Web-серверу.

Последний параметр диалогового окна New Active Server Object используется для создания очень простых тестовых сценариев HTML для данного активного объекта сервера. Для тех, кто не знаком с ASP или языком сценариев ASP, это подходящий момент, чтобы начать их изучение. Пример состоит всего лишь из двух строк, но их вполне достаточно, чтобы продемонстрировать вызов методов созданного активного объекта сервера с помощью сценария.

Обычно в этом диалоговом окне не приходится изменять ничего, за исключением имени объекта в поле CoClass Name.

Редактор библиотеки типов

Итак, был создан активный объект сервера, а также его собственная библиотека типов. Остановимся на редакторе библиотеки типов (Type Library Editor) Delphi 6 для активного объекта сервера `DrBob42`, представленном на рис. 22.3.

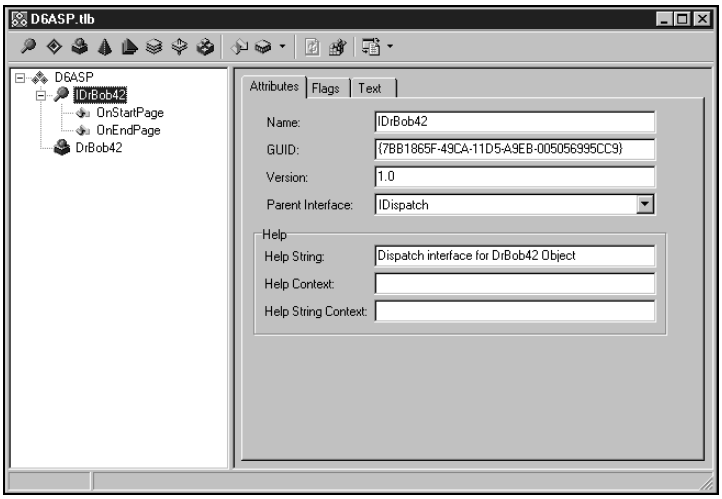

Рис. 22.3. *Редактор библиотеки типов для интерфейса* `IDrBob42`

Еще раз сохраните файлы проекта (меню File пункт Save All). При первом сохранении возникнет запрос об имени модуля `Unit1` (модуль, собственно содержащий активный объект сервера). Здесь этот модуль назван `DrBob42ASP.pas`. Затем последует запрос на сохранение файла `DrBob42.asp`, который содержит шаблон HTML файла ASP. Первоначально такой файл имеет следующее содержание:

```
<HTML>
<BODY>
<TITLE> Testing Delphi ASP </TITLE>
<CENTER>
<H3> You should see the results of your Delphi Active Server method
below </H3>
</CENTER>
<HR>
<%
  Set DelphiASPObj = Server.CreateObject("D6ASP.DrBob42")
  DelphiASPObj.{ Здесь укажите имя метода }
%>
<HR>
</BODY>
</HTML>
```

Как было сказано ранее в настоящей главе, код ASP располагается внутри дескриптора `<% .. %>`, в отличие от обычных дескрипторов текста HTML `< .. >`. В одном дескрипторе ASP может находиться весь код сценария (в данном случае две строки). Первая строка создает экземпляр объекта `DrBob42` из библиотеки ActiveX `D6ASP`, а вторая вызывает безымянный метод.

Второй особенностью, которую можно заметить на рис. 22.3, является наличие методов `OnStartPage` и `OnEndPage` интерфейса `IDrBob42`. Это последствие выбора параметра Page-Level Event Methods в диалоговом окне New Active Server Object. (Если бы был выбран переключатель Object Context, то они бы отсутствовали, в чем

нетрудно убедиться из листинга 22.2.) Реализация этих методов содержится в автоматически созданном модуле DrBob42ASP, который содержит приведенный в листинге 22.1 исходный код активного объекта сервера.

Листинг 22.1. DrBob42ASP — исходный код активного объекта сервера

```
unit  DrBob42ASP;

{$WARN SYMBOL_PLATFORM OFF}

interface
uses
  ComObj, ActiveX, AspTlb, D6ASP_TLB, StdVcl;

type
  TDrBob42 = class(TASPObject, IDrBob42)
  protected
    procedure OnEndPage; safecall;
    procedure OnStartPage(const AScriptingContext:
                          IUnknown); safecall;
  end;

implementation

uses  ComServ;

procedure TDrBob42.OnEndPage;
begin
  inherited OnEndPage;
end;

procedure TDrBob42.OnStartPage(const AScriptingContext: IUnknown);
begin
  inherited OnStartPage(AScriptingContext);
end;

initialization
  TAutoObjectFactory.Create(ComServer, TDrBob42, Class_DrBob42,
                            ciMultiInstance, tmApartment);
end.
```

Прежде чем начать добавлять другие методы, посмотрим, как будет выглядеть активный объект сервера, созданный с установленным переключателем Object Context. К счастью, в одну библиотеку ActiveX можно включить несколько активных объектов сервера. Поэтому еще раз вызовем окно New Active Server Object и, указав в поле CoClass Name имя нового класса **Micha42**, выберем, на этот раз, параметр Object Context. Сохраним полученный код в файле Micha42ASP.pas, а соответствующий файл ASP — в файле Micha42.asp.

Исходный код представлен в листинге 22.2. Отличия небольшие: не хватает только событий OnEndPage и OnStartPage. Но важнее всего то, что класс объекта TDrBob42 происходит от TASPObject, в то время как класс объекта TMicha42 происходит от TASPMTSObject.

Это одно из главных преимуществ объектов ASP в Delphi: для создания надежных и быстродействующих объектов не обязательно знать подробности их реализации. Начиная с данного момента, можно приступить к добавлению новых функциональных возможностей в модули DrBob42ASP и Micha42ASP. Такие объекты будут вести себя одинаково, несмотря на то, что использовались совершенно разные технологии.

Листинг 22.2. Micha42ASP — исходный код активного объекта сервера

```
unit Micha42ASP;

{$WARN SYMBOL_PLATFORM OFF}

interface
uses
  ComObj, ActiveX, AspTlb, D6ASP_TLB, StdVcl;

type
  TMicha42 = class(TASPMTSObject, IMicha42)
  end;

implementation

uses  ComServ;

initialization
  TAutoObjectFactory.Create(ComServer, TMicha42, Class_Micha42,
                            ciMultiInstance, tmApartment);
end.
```

Новые методы

Настало время добавить новые методы в интерфейс IDrBob42 (или IMicha42), которые могут быть вызваны из внешнего мира (как правило, из Web-страниц .asp).

Наряду с методами OnEndPage и OnStartPage (в объектах TDrBob42), можно определить еще несколько специальных методов. Например, используя библиотеку типов, можно добавить в интерфейс IDrBob42 метод по имени Welcome. (Щелкните правой клавишей мыши на узле IDrBob42 и выберите в контекстном меню пункты New и Method).

Данный метод может быть использован для отображения на экране динамического сообщения с приветствием. Добавив такой метод и обновив реализацию, можно писать программный код метода TDrBob42.Welcome. Для этого необходимо иметь представление о внутренних объектах ASP и их функциональной поддержке средствами Delphi 6. Подобно сценариям ASP, объекты ASP в Delphi имеют доступ к специальным объектам Request и Response.

Объект ASP Response

Объект ASP `Response` (Ответ) — это внутренний объект, доступный внутри методов активного объекта сервера. Данный объект используется всякий раз, когда необходимо организовать динамический вывод. Объект `Response` обладает рядом свойств и методов, позволяющих управлять его содержимым. Наиболее важным из них является метод `Write`. Он получает в качестве аргумента переменную типа `OleVariant` (как можно заметить в окне интерактивной подсказки на рис. 22.4) и обеспечивает вывод аргумента в строго определенном месте динамически создаваемого кода сценария ASP, в котором соответствующий вызов ограничен дескрипторами `<%` и `%>`.

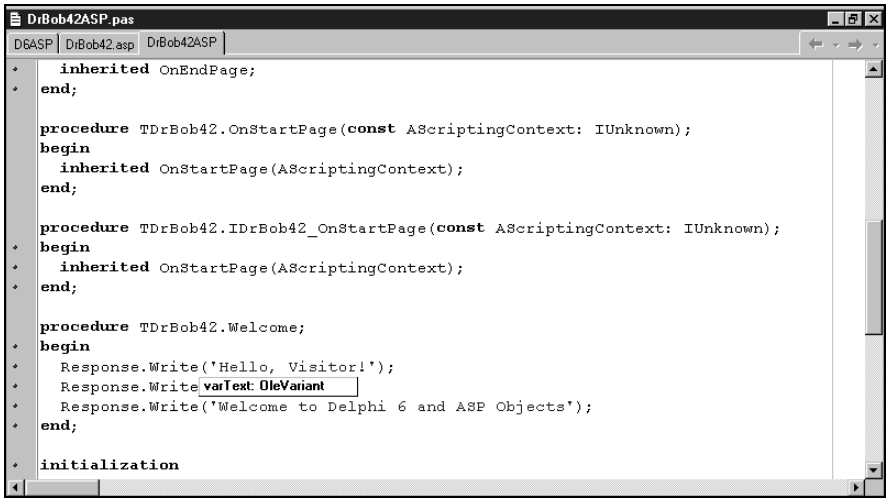

Рис. 22.4. *Редактор кода*

Чтобы вывести на экран приветствие, внесите код листинга 22.3 в реализацию метода `TDrBob42.Welcome`.

Листинг 22.3. Реализация метода `Welcome`

```
procedure TDrBob42.Welcome;
begin
  Response.Write('Hello, Visitor!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
end;
```

В файл DrBob42.ASP достаточно внести лишь одно изменение внутри дескриптора ASP (теперь известно имя метода — Welcome). Новый дескриптор ASP будут выглядеть так:

```
<%
  Set DelphiASPObj = Server.CreateObject("D6ASP.DrBob42")
  DelphiASPObj.Welcome
%>
```

Обратите внимание, что сценарию ASP не нужно удалять или освобождать переменную `DelphiASPObj`: это будет сделано автоматически, когда объект выйдет из области видимости. Кроме метода `Welcome`, в интерфейс `IDrBob42` можно добавить любое количество других методов, а затем вызывать их в сценарии ASP аналогичным образом. Но сначала давайте проверим работу только что созданного активного объекта сервера и позаботимся о его дальнейшем расширении.

Первый запуск

Теперь все готово для первого пробного запуска активного объекта сервера в составе активной страницы. Осталось только зарегистрировать библиотеку активного объекта сервера `D6ASP.dll` и разместить страницу `DrBob42.asp` в соответствующем каталоге (с правами исполнения сценариев ASP).

Ранее уже упоминались параметры внутреннего и внешнего процессов выполнения объектов ASP. *Внутренний процесс* (in-process) означает, что объект ASP будет загружаться и выполняться внутри Web-сервера, а выгружаться будет только тогда, когда Web-сервер завершит свою работу. *Внешний процесс* (out-of-process) означает, что объект ASP будет загружаться и выгружаться при каждом обращении клиента к серверу. Объекты внутреннего процесса в общем выполняются лучше, зато объекты внешнего процесса легче отлаживать. Зарегистрировать активные объекты сервера можно двумя способами: либо как внутренний сервер, либо как внешний.

Чтобы зарегистрировать библиотеку `D6ASP.dll` как внутренний сервер, содержащий активные объекты, выберите в меню Run Delphi 6 пункт Register ActiveX Server.

Чтобы отменить регистрацию этого сервера, выберите в меню Run пункт Unregister ActiveX Server.

На рис. 22.5 представлено сообщение, подтверждающее успешную регистрацию сервера ActiveX `D6ASP`.

Рис. 22.5. *Сервер ActiveX зарегистрирован*

К регистрации сервера ActiveX как внешнего вернемся далее в настоящей главе. Но сначала закончим проверку.

Зарегистрировав на машине разработчика библиотеку `D6ASP.dll` как сервер ActiveX, переместим файл `DrBob42.asp` в каталог, обладающий правом исполнения сценариев ASP. Например в каталог `WWWRoot/Scripts`.

Результат обращения к URL `http://localhost/scripts/DrBob42.asp` представлен на рис. 22.6.

Используя метод `Response.Write`, можно динамически поместить любой текст в любое место, где внутри Web-страницы `DrBob42.asp` осуществляется вызов метода `Welcome`.

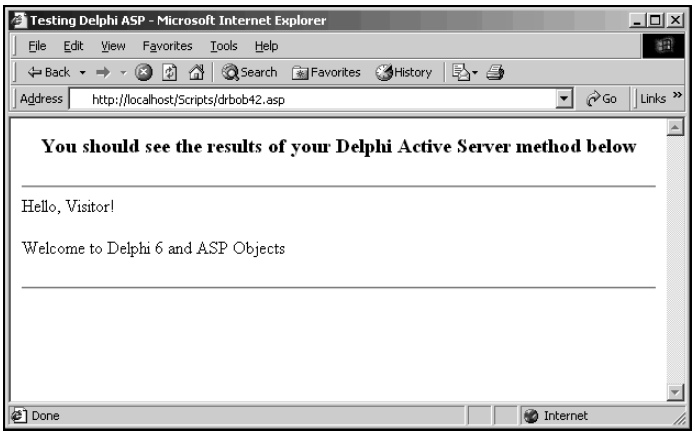

Рис. 22.6. *Активный объект сервера в действии*

Объект ASP Request

Прежде чем продолжать, рассмотрим другой очень важный внутренний объект ASP `Request` (Запрос). Подобно объекту `Response`, `Request` доступен внутри методов (интерфейса) активных объектов сервера. Объект `Request` применяется для доступа ко всей вводимой информации. Существует три способа ввода: с помощью переменных форм, передаваемых методом `POST`, с помощью переменных "жирных" ("fat") URL, передаваемых методом `GET`, и с помощью cookies. Каждый из них обладает свойством по имени `Items` (Элементы), представляющим собой набор строк для хранения содержимого объектов ASP `Response` или `Request`.

Модифицированный метод `Welcome`, получающий значение `Name` из исходной формы, используется в сценарии ASP, представленном в листинге 22.4.

Листинг 22.4. Определение модифицированного метода `Welcome`

```
procedure TDrBob42.Welcome;
var
  Str: String;
begin
  Str := Request.Form.Item['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
end;
```

Та же самая методика применяется для объектов `QueryString` и `Cookies`.

Перекомпиляция активных объектов сервера

Те, кто попробовал снова скомпилировать проект D6ASP, вероятно, получили от Delphi такое сообщение об ошибке: `Could not create output file D6ASP.dll.` (`Невозможно создать результирующий файл D6ASP.dll.`) Такое сообщение появилось потому, что активный объект сервера DrBob42, расположенный внутри библиотеки D6ASP.dll, все еще используется и кэшируется Web-сервером. Поэтому при попытке перекомпилировать данный активный объект сервера компоновщик выдаст сообщение об ошибке: `The file is still in use.` (`Файл все еще используется.`) Можно попробовать завершить работу IIS, но и это не поможет. Завершение работы службы публикации в Internet (World Wide Web Publishing Service) тоже окажется бесполезным. Фактически придется завершить работу всех служб администрирования IIS (IIS Admin Service) прежде, чем ASP.DLL и все активные объекты сервера будут удалены из памяти и появится возможность их перекомпилировать. Обратите внимание, что завершение работы служб администрирования IIS (показанное в диалоговом окне на рис. 22.7), означает завершение работы и всех связанных с ними служб и средств (WWW, FTP и так далее). Конечно, это вовсе не то, что можно было бы сделать на реальном Web-сервере.

При попытке завершить работу службы IIS Admin появится диалоговое окно Stopping, показанное на рис. 22.8, информирующее о зависимых службах, которые также должны завершить работу.

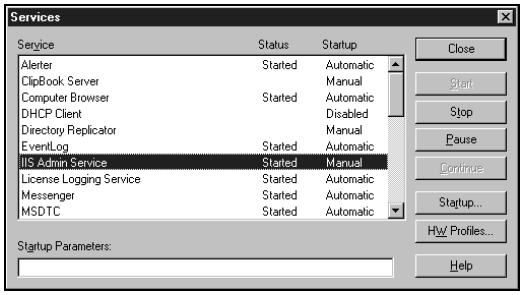

Рис. 22.7. *Служба IIS Admin*

Рис. 22.8. *Диалоговое окно Stopping, завершающее работу службы World Wide Web Publishing*

Чтобы упростить процесс перезапуска служб NT при перекомпиляции и переустановке активного объекта сервера, можно воспользоваться командным файлом RESTART.BAT примерно следующего содержания:

```
net stop "World Wide Web Publishing Service"
net stop "IIS Admin Service"
net start "World Wide Web Publishing Service"
```

СОВЕТ

На машине, используемой только для разработки, можно явно отменить кэширование Web-сервером активных объектов. Естественно, этого не следует делать на рабочих машинах, поскольку произойдет превращение активных объектов сервера в приложения, работающие даже медленнее обычного CGI — ведь их придется загружать для *каждого* клиентского запроса. Это эквивалентно активному объекту внешнего сервера, который вообще никогда не используется, за исключением специфической ситуации разработки.

Повторный запуск активных страниц сервера

Несмотря на то, что активные страницы можно загружать вполне самостоятельно, тем не менее зачастую более эффективным является их использование в комплекте с формой HTML. Для рассматриваемого примера можно воспользоваться следующей страницей HTML:

```
<HTML>
<HEAD>
<TITLE>Dr.Bob's ASP Example</TITLE>
</HEAD>
<BODY BGCOLOR=FFFFCC>
<FONT FACE="Verdana"SIZE=2>
<FORM ACTION="drbob42.asp" METHOD=POST>
Name: <INPUT TYPE=text NAME=Name>
<P>
<INPUT TYPE=submit>
</FORM>
</BODY>
</HTML>
```

Загруженная в окне Internet Explorer как ресурс `http://localhost/cgi-bin/drbob42.htm` эта страница выглядит так, как показано на рис. 22.9. Здесь в поле редактирования уже введено имя, и все готово для щелчка на кнопке Submit Query (Послать запрос).

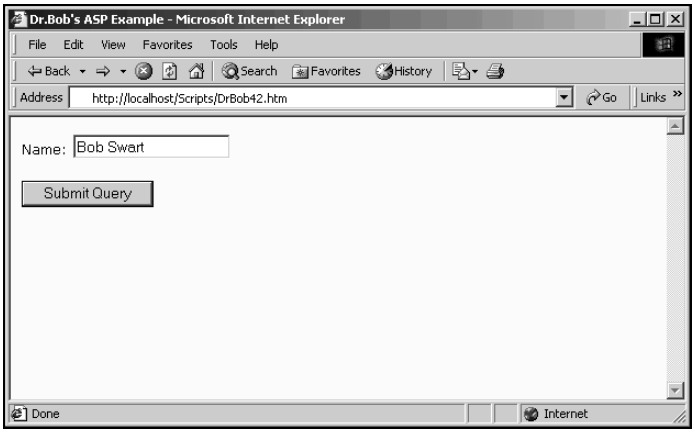

Рис. 22.9. *Форма* `DrBob42.htm` *в окне Internet Explorer*

После того как поле редактирования будет заполнено именем пользователя, щелчок на кнопке Submit Query отправит серверу запрос на загрузку активной страницы. Это приведет к созданию экземпляра активного объекта сервера `DrBob42` и вызову его метода `Welcome`, как и указано в сценарии ASP. Динамически созданный результат представлен на рис. 22.10.

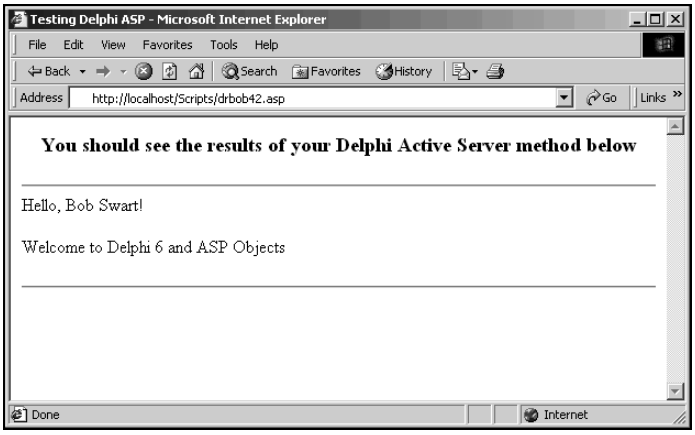

Рис. 22.10. *Результат работы* `DrBob42.asp` *в окне Internet Explorer*

Данный простой пример построен на применении в ASP всего лишь двух объектов: `Request` и `Response`, но и этого вполне достаточно, чтобы понять сам принцип. Теперь перейдем к изучению таких внутренних объектов ASP, как `Session` (Сеанс), `Server` (Сервер) и `Application` (Приложение), а затем вернемся к поддержке Delphi специфических приложений Web-сервера (таких как компоненты WebBroker), большинство из которых также применяются в комбинации с активными объектами сервера.

Объекты ASP Session, Server и Application

Кроме объектов `Request` и `Response`, ASP обладает также доступом к внутренним объектам `Session`, `Server` и `Application`. Фактически это одно из главных преимуществ ASP перед CGI и ISAPI: активный объект сервера может обращаться к информации сеанса и приложения без каких бы то ни было дополнительных усилий со стороны программиста (используя cookies, скрытые поля или переменные URL).

Возвращаясь к предыдущему примеру, напомним, что класс `TDrBob42` происходит от класса `TASPObject`. В дополнение к свойствам `Request` и `Response`, этот класс обладает свойствами `TASPObject.Session`, `Server` и `Application`. Такие свойства обеспечивают прямой доступ к внутренним объектам ASP `Session`, `Server` и `Application` соответственно. Эти свойства можно использовать в составе методов класса

TDrBob42, например для хранения имен всех посетителей Web-сайта, что может быть реализовано следующим образом (обратите внимание на вторую строку):

```
<%
Set DelphiASPObj = Server.CreateObject("D6ASP.DrBob42")
Session.Value("Name") = "Bob Swart"
DelphiASPObj.Welcome
%>
```

Чтобы сделать это значение постоянным (постоянным для всех активных страниц сервера, посещаемых тем же самым пользователем в течении того же самого сеанса), можно воспользоваться объектом Session активного объекта сервера, а также другими объектами Form, QueryString или Cookies (см. листинг 22.5).

Листинг 22.5. DrBob42ASP — исходный код активного объекта сервера

```
procedure TDrBob42.Welcome;
var
  Str: String;
begin
  Str := Session.Value['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome back to Delphi 6 and ASP Objects');
end;
```

Объект Session использует для хранения своего состояния cookies, поэтому удостоверьтесь, что на установленном Web-сервере применение cookies разрешено.

Активные объекты сервера и базы данных

Теперь создадим немного более полезный пример и используем базу данных или ее таблицу для демонстрации возможности запроса и доступа к данным таблицы на сервере и представления результатов в составе активной страницы. Чтобы воспользоваться этими функциональными возможностями, необходимо сначала добавить новый модуль данных, выбрав в меню File пункты New, Data Module.

Присвойте свойству Name модуля данных новое значение, например DataModuleASP, и сохраните этот новый модуль в файле DataMod.pas. Затем, чтобы сохранить весь проект, выберите в меню File пункт Save All. Теперь главный файл проекта D6ASP.dpr содержит в разделе uses ссылку на новый модуль данных. Внутри этого модуля данных можно разместить компонент TClientDataSet из вкладки Data Access (Доступ к данным), это самый простой и наиболее гибкий способ обеспечить доступ к набору данных. В принципе, чтобы расширить данный пример, можно заменить его на другой компонент набора данных.

Перетащите компонент TClientDataSet из панели инструментов в модуль данных. Чтобы обеспечить его источником данных, щелкните на кнопке с многоточием рядом со свойством FileName, перейдите в каталог C:\Program Files\Common Files\Bor-

land Shared\Data — там находятся таблицы из демонстрационной базы данных DBDEMOS, MyBase XML, а также бинарный файл ClientDataSet. Для этого примера выберите файл biolife.xml.

Теперь, когда модуль данных готов, необходимо позаботиться о его совместном использовании в среде многопоточного режима. Наилучший способ создать модуль данных внутри активного объекта сервера, когда в этом возникает необходимость, заключается либо в использовании событий BeginPage и EndPage, либо (что даже еще проще) непосредственно внутри метода Welcome.

Но, чтобы фактически воспользоваться модулем DataMod, нужно добавить его в раздел uses модуля DrBob42ASP. Затем необходимо перенести код листинга 22.6 внутрь метода Welcome и можно свободно создавать, использовать и удалять модули данных.

Листинг 22.6. DrBob42ASP — исходный код активного объекта сервера

```
procedure TDrBob42.Welcome;
var
  Str: String;
  DM: TDataModuleASP;
begin
  Str := Request.Form.Item['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
  try
    DM := TDataModuleASP.Create(nil);
    // использование DM...
  finally
    DM.Free
  end
end;
```

Для представления в броузере информации из набора данных следует передать все данные из компонента ClientDataSet в создаваемую таблицу HTML, которая будет содержать столбцы Common_Name (Название) и Notes (Замечания) всех рыб, перечисленных в справочнике dataset. Для этого достаточно всего лишь нескольких дополнительных строк кода, создающего динамический HTML (см. листинг 22.7).

Листинг 22.7. DrBob42ASP — исходный код активного объекта сервера

```
procedure TDrBob42.Welcome;
var
  Str: String;
  DM: TDataModuleASP;
begin
  Str := Request.Form.Item['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
  try
```

```
    Response.Write('<P>');
    DM := TDataModuleASP.Create(nil);
    with DM.ClientDataSet1 do
    try
      Open;
      First;
      Response.Write('<TABLE BORDER=1><TR><TD>Common_Name</TD>');
      Response.Write('<TD>Notes</TD></TR>');
      while not Eof do begin
        Response.Write('<TR><TD>');
        Response.Write(FieldByName('Common_Name').AsString);
        Response.Write('</TD><TD>');
        Response.Write(FieldByName('Notes').AsString);
        Response.Write('</TD></TR>');
        Next
      end;
      Close;
    finally
      Response.Write('</TABLE>')
    end;
  finally
    DM.Free
  end
end;
```

Результат представлен на рис. 22.11.

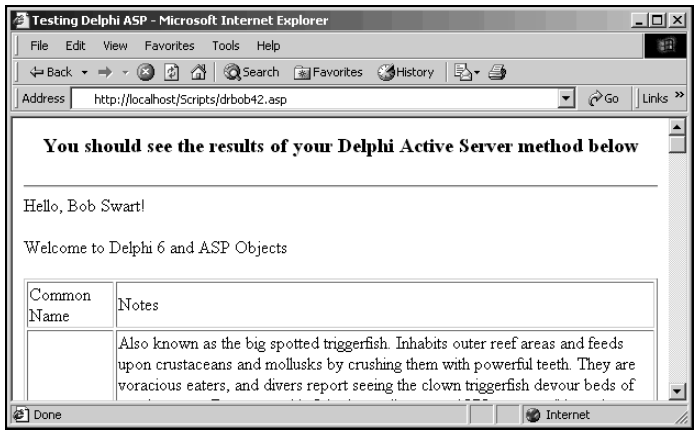

Рис. 22.11. *Динамический вывод HTML активного объекта сервера*

В состав Delphi уже входит ряд полезных компонентов и методов, позволяющих создавать хорошо отформатированный динамический HTML. В NetCLX компоненты, способные порождать код HTML, называются *PageProducers* (генераторы страниц). Вместо того, чтобы тратить время на изучение HTML, можно использовать генераторы страниц, которые динамически создадут необходимые файлы HTML.

Активные объекты сервера и поддержка NetCLX

Если сравнить активные объекты сервера с архитектурой NetCLX, то нельзя не заметить множества совпадений. И те, и другие используют объекты `Request` и `Response` в качестве основного средства обмена данными с клиентами. Но, с точки зрения разработчика, NetCLX получает большую поддержку со стороны компонентов `PageProducer` и `TableProducer`, которые были написаны специально для использования внутри Web-модулей. К счастью, область применения этих компонентов (генераторов HTML) не ограничивается модулями Web; их можно применять везде, где необходимо динамически создать объект класса `TDataSetTableProducer`, назначить ему набор данных и передать полученный в результате HTML обратно, используя метод `Response.Write`. Компонент `TDataSetTableProducer` можно перенести в создаваемый модуль данных и даже настраивать его во время разработки!

Фактически, применение в составе активного объекта сервера компонентов, генерирующих код HTML и написанных первоначально для NetCLX, не составляет особой сложности. Единственным исключением являются компоненты `TQueryTableProducer` и `TSQLQueryTableProducer`, которые на самом деле ожидают на входе объект `Request` NetCLX, а не одноименный объект ASP. Весе остальные генераторы страниц применяются в исходном состоянии, что и демонстрирует следующий пример.

Перенесите компонент `TDataSetTableProducer` в модули данных и назначьте его свойству `DataSet` значение `ClientDataSet`, использованное в предыдущем примере. Чтобы настроить параметры компонента `DataSetTableProducer`, удостоверитесь, что компонент `ClientDataSet` содержит реальные данные. Обычно свойство `Active` компонента `ClientDataSet` временно устанавливают в состояние `True` (впоследствии ему возвращают значение `False`). Щелкните на кнопке с многоточием рядом со свойством `Columns` компонента `DataSetTableProducer` (или, щелкнув правой кнопкой мыши на самом компоненте `DataSetTableProducer`, выберите в появившемся контекстном меню пункт Response Editor), что позволит вывести на экран окно редактора свойства `Columns`, показанного на рис. 22.12.

Вернемся к методу `Welcome`. Вызов метода `DataSetTableProducer.Content`, представленный в листинге 22.8, позволяет осуществить динамический вывод необходимого HTML (это существенно короче и намного проще самостоятельной организации аналогичного вывода).

Листинг 22.8. `DrBob42ASP` — исходный код активного объекта сервера

```
procedure TDrBob42.Welcome;
var
  Str: String;
  DM: TDataModuleASP;
begin
  Str := Request.Form.Item['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
  try
```

```
    Response.Write('<P>');
    DM := TDataModuleASP.Create(nil);
    Response.Write(DM.DataSetTableProducer1.Content);
  finally
    DM.Free
  end
end;
```

После перекомпиляции нового активного объекта сервера и перезагрузки файла `DrBob42.htm`, что позволяет обновить активную страницу, можно увидеть результат, показанный на рис. 22.13.

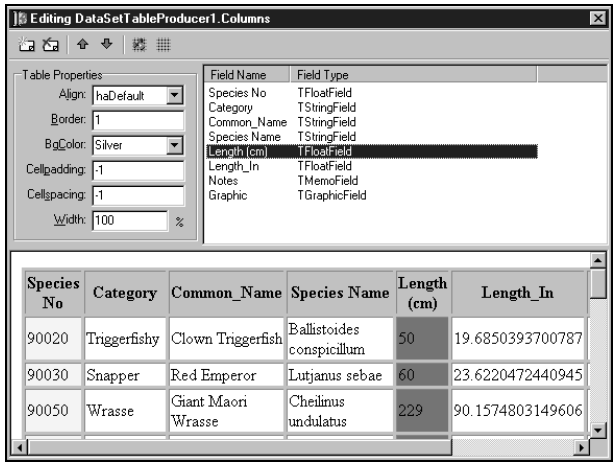

Рис. 22.12. *Компонент `DataSetTableProducer` в окне* Response Editor

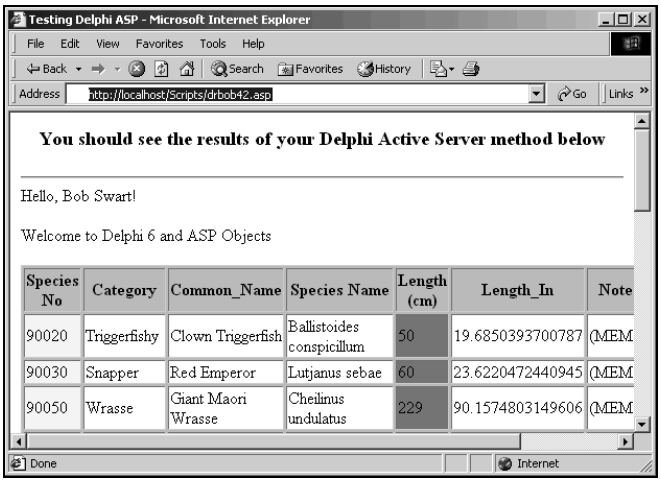

Рис. 22.13. *Результат выполнения `DrBob42.asp` NetCLX в окне Internet Explorer*

Отладка активных объектов сервера

Как уже было сказано, активные объекты сервера похожи на библиотеки DLL ISAPI: после их загрузки необходимо завершить работу всех служб Web-сервера, чтобы выгрузить их, поскольку активные объекты сервера загружает библиотека `ASP.DLL`, которая сама по себе является библиотекой DLL ISAPI. Но преимуществом ASP является то, что сами активные страницы можно изменять сколько угодно и безо всякой перекомпиляции, перезагрузки и перезапуска. Пока не изменятся функциональные возможности активного объекта сервера, *сценарии* ASP можно модифицировать беспрепятственно. Разумеется, проверка правильности функционирования активных объектов сервера представляет собой задачу, которую время от времени придется выполнять.

Когда дело доходит до отладки активных объектов сервера, применяется несколько подходов. Для вывода на экран строк, посланных из активного объекта сервера, можно воспользоваться простым окном сообщения или окном отладки. Однако, чтобы получить любое из этих сообщений, следует сначала установить владельца активного объекта сервера, способного на самом деле взаимодействовать с рабочим столом. В частности, для службы IIS Admin Service нужно установить взаимодействие с параметром `Desktop` в аплете (диалоговом окне) Services диалогового окна Control Panel Services (Службы панели управления), как показано на рис. 22.14.

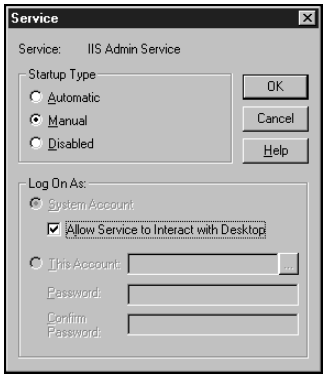

Рис. 22.14. *Флажок Allow Service to interact with Desktop (Разрешить службе взаимодействовать с рабочим столом)*

Установив этот параметр, можно использовать почти все средства взаимодействия активных объектов сервера с рабочим столом. Это весьма поверхностно, но иногда может оказаться достаточно эффективным.

Отладка активных объектов сервера с помощью MTS

Существует более легкий способ управления активными объектами сервера, который заметно повышает практические возможности отладки активных объектов сервера, написанных в Delphi (или в C++Builder). Решение, как это видно из названия данного раздела, заключается в использовании сервера MTS в качестве хоста (оболочки) активного объекта сервера.

Первый шаг подразумевает отмену регистрации активного объекта сервера, созданного в настоящей главе. Для этого в меню Run выберите пункт Unregister ActiveX Server.

После успешной отмены регистрации активного объекта сервера его можно зарегистрировать снова, но на сей раз как объект MTS. Для этого в меню Run выберите пункт Install MTS Objects (Установить объекты MTS). В Windows 2000 этот пункт меню будет называться Install COM+ Object (Установить объект COM+). Возникшее в результате диалоговое окно показано на рис. 22.15.

В диалоговом окне Install MTS Objects выберите объект `DrBob42`, и щелкните на его флажке. Возникнет диалоговое окно, показанное на рис. 22.16, которое запрашивает имя пакета, в который необходимо включить объект `DrBob42`. Можно выбрать существующий пакет или создать новый, например `DelphiDebugPackage`. В Windows 2000 диалоговое окно COM+ работает аналогично.

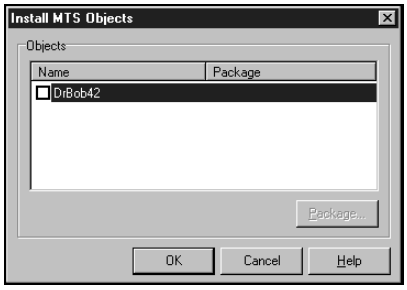

Рис. 22.15. *Установить объекты MTS (или COM+)*

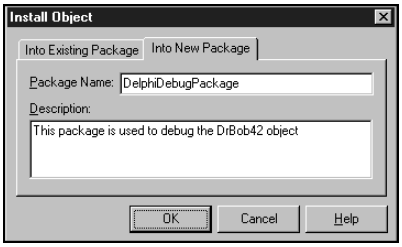

Рис. 22.16. *Установить объект `DelphiDebugPackage`*

Теперь можно щелкнуть на кнопке OK в диалоговом окне Install Object, а затем еще раз — в диалоговом окне Install MTS Objects.

Установив `DrBob42` как объект MTS, можно приступить к отладке активного объекта сервера непосредственно в интегрированной среде разработки Delphi. Чтобы загрузить активный объект сервера в IDE Delphi, необходимо приложение, которое будет содержать этот объект. Действия, которые следует предпринять, начиная с этого момента, различаются в Windows NT и Windows 2000. Сначала рассмотрим действия, необходимые для Windows NT, а затем — для Windows 2000.

Отладка в Windows NT 4

Для Windows NT с установленным пакетом Option Pack нужно указать MTS как ведущее приложение[1] (Host Application). По умолчанию MTS будет уже выполняться, поэтому сначала придется завершить его работу.

СОВЕТ

Никогда не делайте этого на настоящем сервере. Этот подход приемлем только на машине разработчика, на которой можно позволить себе время от времени завершать работу MTS (при отладке разрабатываемых активных объектов сервера, которые содержатся в MTS).

[1] *Приложение, которое будет содержать объект.* – Прим. ред.

Чтобы завершить работу MTS, запустите приложение Internet Service Manager, которое входит в состав Option Pack Windows NT 4. Раскройте в консоли MMC папку Microsoft Transaction Server и отыщите в папке My Computer папку Packages Installed (Установленные пакеты), в которой и содержит только что установленный пакет `DelphiDebugPackage`.

Если щелкнуть правой кнопкой мыши на пиктограмме My Computer, то можно завершать работу всех процессов сервера (рис. 22.17). Это необратимое действие. Проверить завершение работы системы можно, просмотрев список процессов диспетчера задач (Task Manager) Windows NT. Там больше не должно быть процесса `mtx.exe`.

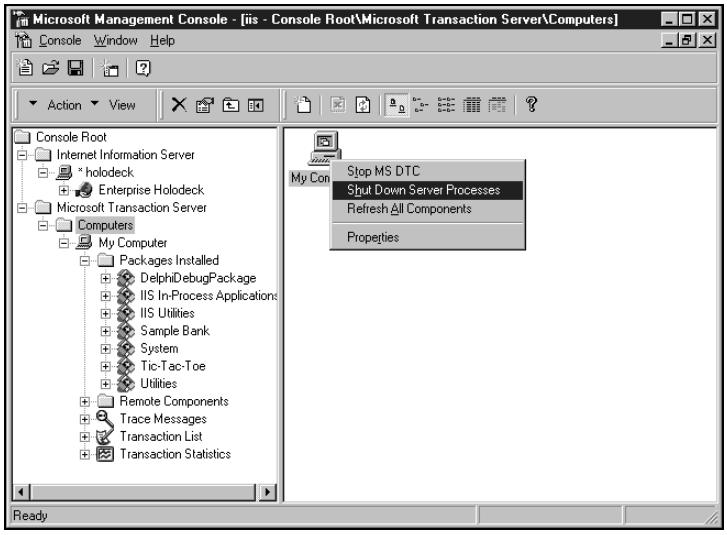

Рис. 22.17. *Завершение работы процессов сервера*

Теперь остается лишь сделать MTS ведущим приложением в диалоговом окне Delphi 6 Run Parameters, что позволит использовать его в качестве контейнера для создаваемого активного объекта сервера `DrBob42`. На машине авторов это был `c:\winnt\system32\mtx.exe`. Здесь также необходимо определить пакет, который содержит объект `DrBob42`. Для этого в раскрывающемся списке Parameters выберите параметр `/p:"DelphiDebugPackage"`. В данном случае объект `DrBob42` представляет собой `DelphiDebugPackage` (Пакет отладки Delphi). В результате диалоговое окно Run Parameters будет выглядеть так, как показано на рис. 22.18.

Теперь в коде можно устанавливать контрольные точки. Для этого достаточно щелкнуть мышью в поле справа от текста кода или нажать кнопку <F5> в то время, когда курсор находится на строке кода. Нажав клавишу <F9>, можно запустить на исполнение активный объект сервера `DrBob42` в отладчике, как содержащийся внутри MTS. Но ничего не случится, поскольку MTS выполняется, а наш активный объект сервера так и не вызван броузером. Теперь необходимо запустить Internet Explorer (или другой броузер) и обратиться к Web-странице `DrBob42.asp`, которая и загрузит активный объект сервера.

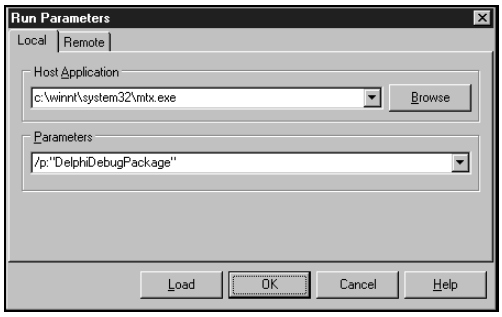

Рис. 22.18. *Диалоговое окно* Run Parameters

Указанные действия приводят к срабатыванию контрольной точки. После этого появится возможность применить интегрированный отладчик Delphi к загруженному активному объекту сервера.

Чтобы завершить сеанс отладки в среде Delphi, достаточно завершить работу MTS (точно так же, как и в начале сеанса отладки).

Отладка в Windows 2000

Работая в Windows 2000, `mtx.exe` можно больше не использовать просто потому, что в Windows 2000 сервер MTS интегрирован в операционную систему. Тем не менее можно воспользоваться файлом `dllhost.exe` для загрузки идентификатора процесса (ProcessID) активного объекта сервера. Данный метод будет работать и в Windows NT, но это окажется несколько сложнее, поэтому сначала был описан метод с применением сервера MTS в качестве контейнера.

Теперь как содержащее приложение можно использовать `dllhost.exe`, которое необходимо указать в диалоговом окне Run Parameters. В раскрывающемся списке Parameters должен быть указан идентификатор процесса пакета `DelphiDebugPackage`, содержащего активный объекта сервера `DrBob42`. Эту информацию можно получать с помощью диспетчера служб Internet (Internet Service Manager) в консоли MMC (Microsoft Management Console) Windows NT или службе компонентов (Component Services) Windows 2000.

Идентификатор процесса пакета `DelphiDebugPackage` в данном примере — `{50AE66A2-349B-11D5-A9F0-005056995CC9}`. Его можно скопировать в текстовом поле Package ID диалогового окна, представленного на рис. 22.19. Это удобнее всего, поскольку в противном случае такую строку придется набирать вручную. Диалоговое окно Run Parameters с установленным идентификатором в поле Parameters приведено на рис. 22.20.

Теперь, удостоверившись, что контрольная точка установлена, нажмите клавишу <F9>, чтобы запустить (и отладить) активный объект сервера `DrBob42`. Как и в прошлый раз, ничто не произойдет до тех пор, пока объект ASP не будет вызван броузером. Придется запустить Internet Explorer (или другой броузер) и загрузить Web-страницу `DrBob42.asp`, которая, в свою очередь, загрузит активный объект сервера, что приведет к срабатыванию контрольной точки. После этого появится возможность применить интегрированный отладчик Delphi к загруженному активному объекту сервера.

Обратите внимание, что до завершения сеанса отладки в среде Delphi следует завершить работу пакета `DelphiDebug Package` в MTS (точно так же, как и в начале сеанса отладки с использованием MTS).

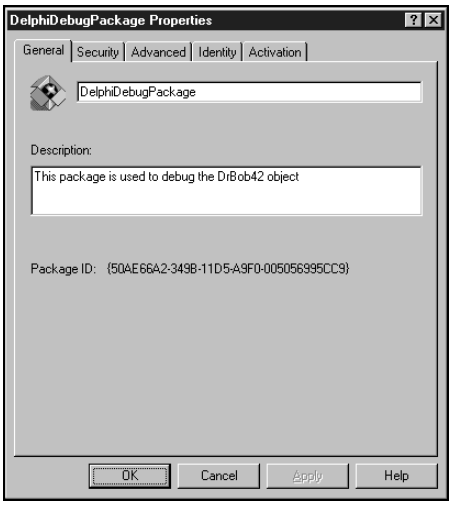

Рис. 22.19. *Идентификатор процесса пакета* `DelphiDebugPackage`

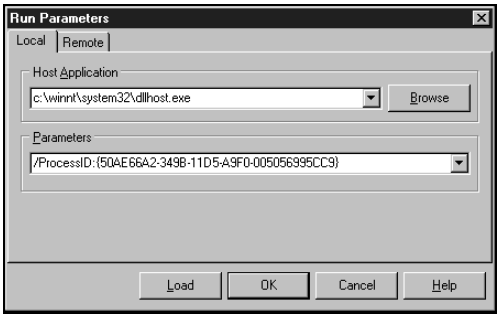

Рис. 22.20. *Диалоговое окно* Run Parameters *с установленным идентификатором*

Резюме

В настоящей главе рассматривалось, что представляют собой активные страницы сервера, какую роль в них играют активные объекты сервера и как можно использовать среду разработки Delphi 6 при создании активных объектов сервера. Здесь также описано использование внутренних объектов (таких как `Request` и `Response`) и то, как можно использовать их для работы с базами данных в активном объекте сервера, как можно комбинировать активные объекты сервера и компоненты NetCLX. И, в заключение, рассматривалась отладка активных объектов сервера в Delphi 6 под Windows NT и Windows 2000.

ГЛАВА

23

Разработка приложений WebSnap

В ЭТОЙ ГЛАВЕ...

Delphi 6 представляет новую среду разработки Web-приложений под названием WebSnap, созданную на базе *инструментов ускоренной разработки приложений* (RAD — Rapid Application Development). Встроенный в WebBroker и InternetExpress, WebSnap представляет собой новый этап программирования для тех разработчиков Delphi, которые хотят использовать свой любимый инструмент и для создания Web-приложений. Он поддерживает все стандартные механизмы Web-приложений, в том числе управление сеансами, регистрацию (login) пользователя, отслеживание предпочтений пользователя и создание сценариев. Естественно, Delphi 6 применяет при разработке Web-сайтов инструменты ускоренной разработки приложений (RAD), позволяя просто и быстро создавать надежные Web-приложения, динамически управляющие базами данных.

Возможности WebSnap

WebSnap не является абсолютно новой технологией и не предназначен для замены WebBroker и InternetExpress. WebSnap используется вместе с этими технологиями и позволяет относительно просто интегрировать уже существующий код в новое приложение. В следующих разделах перечислены возможности WebSnap.

Несколько Web-модулей

В предыдущих версиях Delphi приложения WebBroker и InternetExpress вынуждены были выполнять все действия в едином Web-модуле. Существования нескольких Web-модулей не допускалось. Чтобы добавить модули данных, их приходилось создавать во время выполнения вручную, а не автоматически. WebSnap устраняет это ограничение и позволяет Web-приложению иметь любое количество Web-модулей и модулей данных. Приложения WebSnap состоят из нескольких модулей, и каждый модуль представляет собой отдельную Web-страницу. Это позволяет нескольким разработчикам независимо работать над различными частями приложения и не беспокоиться о совместимости кода.

Серверные сценарии

WebSnap позволяет полностью интегрировать серверные сценарии в состав приложения, а также очень легко создавать мощные объекты сценариев, которые можно использовать для создания и управления HTML в приложении. Компонент класса `TAdapter` и все его производные обладают возможностью работать со сценариями, а следовательно, все они могут быть вызваны серверными сценариями, создавать HTML и клиентские сценарии JavaScript для использования в приложении.

Компоненты класса TAdapter

Компоненты класса `TAdapter` устанавливают взаимодействие между приложением и серверным сценарием. Серверные сценарии способны взаимодействовать с приложением только через адаптеры, гарантируя, что сценарий не произведет несанкционированных изменений данных в приложении и не вызовет функций, не предна-

значенных для открытого употребления. Можно создать специализированные классы, производные от TAdapter, которые смогут необходимым образом манипулировать содержимым, а также сделать его видимым и доступным во время разработки. Компоненты класса TAdapter способны содержать данные и выполнять с ними определенные действия. Например, экземпляр класса TDataSetAdapter способен отображать записи из набора данных, а также выполнять такие стандартные действия, как навигация, добавление, модификация и удаление.

Разнообразие методов доступа

WebSnap обеспечивает несколько способов обработки обращений к HTTP. К содержимому Web-страницы можно обратиться по ее имени, с помощью компонента TAdapter либо используя WebBroker. Это обеспечивает необходимую гибкость и позволяет избрать метод отображения создаваемых Web-страниц на основании конкретных условий. Возможно, страницу необходимо отобразить в ответ на щелчок на кнопке Submit или потребуется создать список ссылок, напоминающий меню и позволяющий осуществлять навигацию по сайту.

Компоненты генераторов страниц

WebBroker обладает компонентом класса TPageProducer (генератор страниц), предназначенным для управления HTML, его модификации и обновления содержимого на основании специальных дескрипторов. InternetExpress расширил эту концепцию классом TMidasPageProducer, а WebSnap дополнил ее рядом новых мощных элементов управления, способных обращаться как к содержимому экземпляров класса TAdapter, так и к данным XSL/XML. Наиболее мощным из новых классов, производных от TPageProducer, является TAdapterPageProducer, способный создавать текст HTML на основании действий и значений полей компонентов класса TAdapter.

Управление сеансом

Приложения WebSnap обладают автоматическим, встроенным управлением сеансом; теперь можно отслеживать все действия пользователя при обращении к HTTP. Поскольку сам протокол HTTP не способен сохранять текущее состояние, создаваемые Web-приложения должны сами следить за пользователями, оставляя на компьютере клиента что-нибудь, способное идентифицировать каждого пользователя. Обычно для этого используют cookies, строки URL или скрытые поля элементов управления. WebSnap обеспечивает полную поддержку сеансов, что позволяет отслеживать информацию пользователей легко и просто. Для этого WebSnap использует компонент SessionsService, который содержит значения идентификаторов сеансов всех пользователей, поэтому задача отслеживания обращений отдельных пользователей не составляет труда. Вообще-то, это довольно-таки сложно управляемая служба, но WebSnap инкапсулирует все подробности и обеспечивает доступ к информации сеанса как из серверного сценария, так и непосредственно из кода Web-приложения.

Служба регистрации (login)

Создаваемые Web-приложения зачастую нуждаются в системе безопасности, осуществляющей аутентификацию пользователей. WebSnap автоматизирует этот процесс с помощью специализированного компонента — *адаптера регистрации* (login adapter). Данный компонент содержит функции, необходимые для правильной организации запроса и опознания пользователей согласно выбранной модели безопасности приложения. Он обладает как информацией, необходимой для регистрации, так и возможностями управления сеансом WebSnap, что позволяет ему обеспечить проверку прав для каждого запроса. Кроме того, компоненты регистрации обеспечивают автоматическую проверку периода действия информации регистрации (гарантируя, что время действия пароля не истекло). Можно сделать так, чтобы пользователи, которые пытаются получить несанкционированный доступ, автоматически перенаправлялись на страницу регистрации или в другое место.

Отслеживание пользователя

Чаще всего функции контроля сеанса используют для отслеживания предпочтений пользователей приложения. WebSnap содержит компоненты, которые позволяют легко установить контроль над информацией пользователя и использовать ее на сайте. Информацию пользователя можно сохранять и получать впоследствии при необходимости. Кроме того, можно сохранять информацию о правах доступа пользователя и его предпочтениях при просмотре сайта (так поступают в приложениях электронной коммерции, имитируя корзинку с покупками).

Управление HTML

Зачастую в динамическом Web-приложении весьма затруднительно осуществлять управление HTML. Содержимое HTML может представлять собой набор ресурсов и файлов различных типов, размещенных в разных местах или вовсе создаваемых динамически. WebSnap предоставляет средства для управления этим процессом, в том числе службу расположения файлов.

Службы загрузки файлов

Обычно управление загрузкой файлов требует большого количества специального кода. WebSnap обеспечивает простое решение на базе адаптера, управляющего многокомпонентными формами, необходимыми для загрузки файлов. Используя встроенные функциональные возможности компонента `TAdapter`, можно легко и быстро обеспечить загрузку файла в приложении WebSnap.

Создание приложения WebSnap

Как обычно, наилучшим методом изучения новой технологии является разработка приложения с ее использованием. Для начала создадим (как это принято) версию приложения WebSnap "`Hello World`".

Проект приложения

Сначала необходимо добавить в IDE Delphi панель инструментов WebSnap. Для этого щелкните правой кнопкой мыши в области заголовка окна панели инструментов IDE и выберите в появившемся меню панель инструментов Internet (рис. 23.1). В результате в главное окно IDE будет добавлена панель инструментов, которая позволит создавать приложения WebSnap, а также добавлять формы и Web-модули.

Рис. 23.1. *Панель инструментов* Internet

Теперь щелкните на кнопке с изображением руки, держащей земной шар, и на экране появится диалоговое окно, представленное на рис. 23.2.

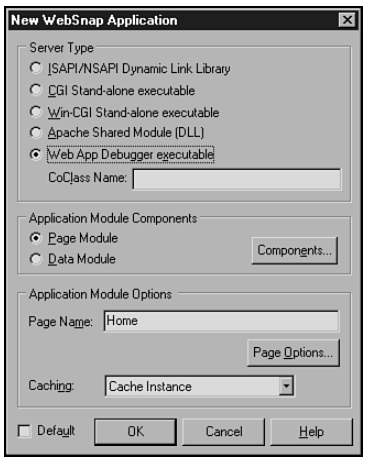

Рис. 23.2. *Диалоговое окно New WebSnap Application*

Диалоговое окно на рис. 23.2 позволяет установить ряд параметров создаваемого приложения WebSnap. Первый параметр, Server Type, — это тип сервера, на котором будет выполняться создаваемое приложение. Выбрать можно один из пяти типов:

- ISAPI/NSAPI Dynamic Link Library (Динамически подключаемая библиотека ISAPI/NSAPI) — этот параметр позволяет создать проект, выполняющийся под IIS (или под сервером Netscape с установленным соответствующим адаптером ISAPI). В результате компиляции проекта будет создана библиотека DLL, выполняющаяся в той же самой области памяти, что и Web-сервер. Обычно Web-сервером приложений ISAPI оказывается Internet Information Server от *Microsoft* (IIS), хотя DLL ISAPI способны выполнять и другие Web-серверы.
- CGI Standalone executable (Автономная исполняемая программа CGI) — этот параметр позволяет создать проект, в результате компиляции которого получится *консольное приложение* (console executable), осуществляющее чтение и запись в стандартные порты ввода и вывода. Полученное приложение будет соответствовать спецификации CGI, которую поддерживают практически все Web-серверы.

- Win-CGI Standalone executable (Автономная исполняемая программа CGI Windows) — этот параметр позволяет создать проект исполняемой программы CGI Windows, способной взаимодействовать с Web-сервером через текстовые файлы `.INI`. Приложения Win-CGI достаточно редки и не рекомендованы для применения.
- Apache Shared Module (DLL) (Совместно используемый модуль Apache (DLL)) — этот параметр позволяет создать проект, выполняющийся на Web-сервере Apache. Более подробная информация по данной теме приведена на Web-сайте `http://www.apache.org`.
- WebApp Debugger Executable (Исполняемая программа для отладчика Web-приложений) — при выборе этого параметра создается приложение, которое будет выполняться в отладчике Web-приложений Delphi (рис. 23.3). Web-приложение станет внешним сервером COM и отладчик Web-приложений (WebApp Debugger) сможет запустить и контролировать его. Это позволяет воспользоваться всеми преимуществами отладчика Delphi для отладки Web-приложения. Кроме того, данный параметр позволяет избежать проблем с перезапуском Web-сервера при загрузке и выгрузке отлаживаемого приложения. Все это делает отладку приложения простой и удобной.

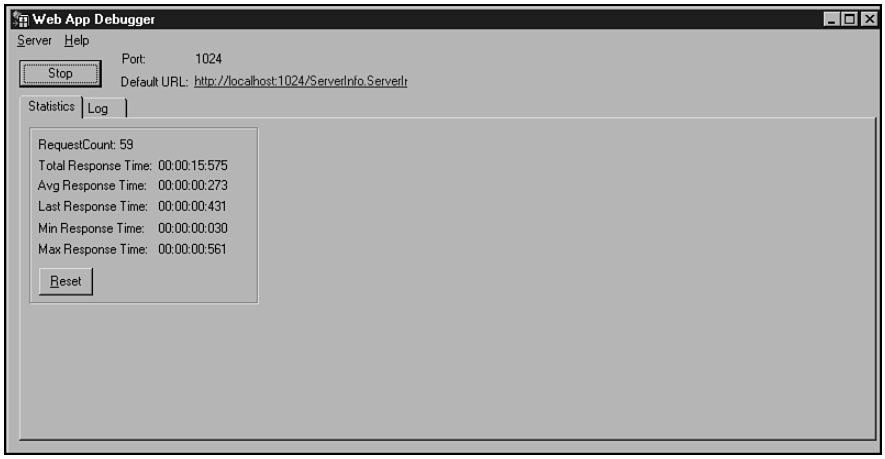

Рис. 23.3. *WebApp Debugger значительно упрощает отладку создаваемых Web-приложений*

НА ЗАМЕТКУ

Отладчик Web-приложений можно запустить в меню Tools IDE. Чтобы работать правильно, необходимо сначала зарегистрировать приложение, поэтому найдите в каталоге `<Delphi Dir>\bin` файл `serverinfo.exe` и запустите его один раз, чтобы он смог себя зарегистрировать. Отладчик Web-приложений — это COM-ориентированная программа, которая выступает в качестве Web-сервера проверяемого приложения. При создании исполняемой программы для отладчика Web-приложений новый проект будет содержать форму и Web-модуль. Форма будет выступать в качестве контейнера для сервера COM и выполнения приложения при

первом запуске для регистрации. После этого отладчик Web-приложений сможет управлять через Web-броузер загруженным в него приложением. Поскольку приложение Delphi представляет собой исполняемую программу и его расширение не ассоциировано с Web-сервером, вполне возможно установить в нем контрольные точки и запустить его в IDE Delphi. Теперь, при обращении из броузера, код будет выполнен до первой контрольной точки, а затем сработает отладчик Delphi. Далее отладка приложения происходит как обычно.
Для доступа к приложению через броузер запустите отладчик Web-приложений, и щелкните на гиперссылке Default URL (URL по умолчанию). Это загрузит Web-приложение, которое перечислит все приложения, зарегистрированные на сервере. Теперь можно выбрать необходимое приложение и запустить его. Параметр View Details (Подробно) позволит просмотреть более подробную информацию обо всех приложениях, а также удалить их из системного реестра, когда они будут уже не нужны. Будьте внимательны, и не удаляйте приложение `ServerInfo`, поскольку после повторной установки его придется регистрировать снова.

Для простого приложения, которое будет создано здесь в качестве примера, выберите параметр WebApp Debugger. Это позволит отлаживать приложение по мере его создания.

Следующий параметр мастера позволяет выбирать тип создаваемого модуля и различные компоненты, которые он будут содержать. Если выбирать параметр Page Module, то будет создан Web-модуль, который в приложении будет представлять собой страницу. Если выбирать параметр Data Module, то будет создан модуль данных, применяющийся в приложении WebSnap. Он сможет выполнять те же функции, что и модули данных в традиционном приложении клиент/сервер. Для данного приложения выберем параметр Page Module.

Теперь щелкнем на кнопке Components (Компоненты) и рассмотрим диалоговое окно, показанное на рис. 23.4.

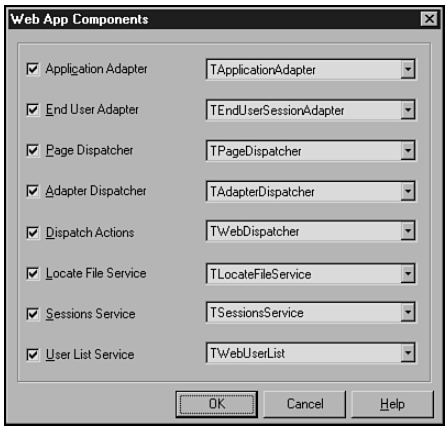

Рис. 23.4. *Диалоговое окно Web App Components позволяет выбрать компоненты, которые будут включены в создаваемый модуль*

Здесь можно выбрать следующие компоненты:

- Application Adapter (Адаптер приложения) — этот компонент манипулирует полями и действиями, доступными с помощью объекта серверного сценария приложения. Чаще всего используется свойство `Title` (Заголовок) этого компонента.

- End User Adapter (Адаптер конечного пользователя) — этот компонент манипулирует информацией о текущем пользователе приложения, например об идентификаторе сеанса, имени пользователя, правах пользователя, а также информацией о предпочтениях пользователя. Он также будет отвечать за регистрацию пользователя и действия, связанные с выходом из системы.
- Page Dispatcher (Диспетчер страниц) — этот компонент создает и посылает запросы HTTP, сделанные по именам страниц. Можно создать ссылки (`HREF`) или действия, которые обращаются к конкретным страницам, а диспетчер страниц получит соответствующий ответ.
- Adapter Dispatcher (Диспетчер адаптера) — диспетчер адаптера обрабатывает все запросы, которые приходят в результате действий адаптера. Обычно это результаты, переданные формами HTML.
- Dispatcher Actions (Диспетчер действий) — этот параметр добавляет в приложение компонент `TWebDispatcher`. Пользователи WebBroker помнят этот компонент. Он обрабатывает запросы из приложения на основании URL точно так же, как это делает приложение WebBroker. Данный компонент можно использовать для того, чтобы добавить свои собственные специальные действия в создаваемое приложение, как это делается в WebBroker.
- Locate File Service (Служба поиска файлов) — события этого компонента вызываются всякий раз, когда Web-модулю необходим ввод HTML. Можно добавить обработчики событий, которые позволят обойти стандартный механизм поиска и получить HTML почти из любого источника. Чаще всего этот компонент используется для захвата содержимого страниц и шаблонов для создания стандартных страниц.
- SessionsService (Служба сеансов) — этот компонент манипулирует сеансами пользователей, позволяя поддержать состояние отдельных пользователей между запросами HTTP. Служба сеансов может хранить информацию о пользователях и автоматически завершать их сеансы по истечении определенного времени бездействия. Любую необходимую информацию, специфическую для данного сеанса, можно добавить в свойство `Session.Values`, представляющее собой индексированную строку как массив вариантов. По умолчанию сеансы контролируются с помощью cookies, хранимых на машине пользователя, но можно создать класс, применяющий для этого и другие способы, например переменные URL или скрытые поля.
- User List Service (Служба списка пользователей) — этот компонент содержит список пользователей, обладающих правом доступа к приложению и информацией о них.

НА ЗАМЕТКУ

Каждый из этих параметров имеет раскрывающийся список, позволяющий выбирать компонент, который будет выполнять определенную роль. Можно создать свой собственный компонент, который после регистрации в WebSnap появится в диалоговом окне и станет доступным для выбора. Например, можно создать компонент сеанса, который хранит информацию в переменных URL, а не в cookies.

В данном примере установим всё флажки, а для параметра End User Adapter выберем в раскрывающемся списке компонент `TEndUserSessionAdapter`. Этот компонент автоматически ассоциирует идентификатор сеанса (session ID) с конечным пользователем. Затем щелкните на кнопке OK.

Следующим параметром мастера является имя страницы. Назовите главную страницу **`Home`** и щелкните на кнопке Page Options (Параметры страницы). Появится диалоговое окно, представленное на рис. 23.5.

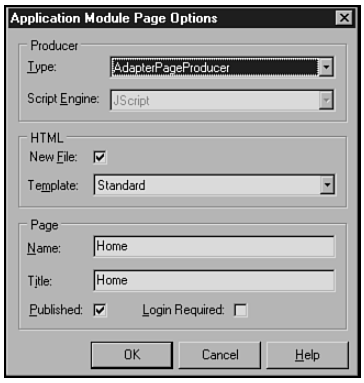

Рис. 23.5. *Диалоговое окно Application Module Page Options позволяет установить параметры страницы разрабатываемого Web-модуля*

Это диалоговое окно позволяет настроить компонент `PageProducer` создаваемого Web-модуля и ассоциированного с ним HTML. Здесь представлен ряд параметров. Первый из них — тип генератора страниц. WebSnap содержит несколько стандартных генераторов страниц, способных по-разному создавать и обрабатывать HTML. По умолчанию установлен простой `PageProducer`. При необходимости можно сменить тип сценария сервера. В раскрывающемся списке Script Engine можно выбрать любой из поддерживаемых Delphi типов сценариев: JScript или VBScript (XML/XSL обсуждается несколько позднее.) В данном случае оставим установленный по умолчанию `JScript`.

С каждым Web-модулем ассоциирована страница HTML. Следующий параметр (Template) позволяет выбирать, какой именно HTML необходим. По умолчанию Delphi устанавливает стандартную страницу с простым навигационным меню на базе сценария. Но можно создать и свои собственные шаблоны HTML, зарегистрировать их в WebSnap, а затем выбирать их в этом списке. Как это сделать — рассматривается далее в настоящей главе, а пока оставим значение, установленное по умолчанию, — `Standard`.

Присвоим странице имя **`Home`** (заголовок (Title) будет автоматически заполнен тем же именем). Удостоверитесь, что флажок Published (Публикуемая) установлен, а флажок Login Required (Необходима регистрация) сброшен. Публикуемая страница отображается в списке страниц приложения и будет доступна объектом сценариев страниц. Это необходимо для навигационного меню на базе сценария, использующего объект сценариев страниц.

Теперь можно щелкнуть на кнопке OK, а затем еще раз в окне мастера. Мастер самостоятельно создаст приложение, и новый Web-модуль будет выглядеть так, как показано на рис. 23.6.

Не будем пока обсуждать элемент управления класса `TWebAppComponents`. Он представляет собой "материнскую плату" для всех остальных компонентов. Поскольку большинство компонентов приложений WebSnap работают совместно, компонент `WebAppComponents` является тем связующим звеном, которое объединяет их вместе и позволяет взаимодействовать друг с другом. Его свойства представляют собой другие компоненты, заполненные специфическими параметрами, обсуждавшимися ранее в настоящей главе.

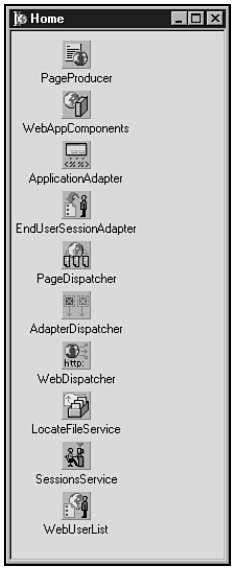

Рис. 23.6. *Web-модуль демонстрационного приложения, созданного мастером WebSnap*

Теперь проект можно сохранить. Соблюдая порядок именования проектов в книге, назовем Web-модуль (`Unit2`) `wmHome`, форму (`Unit1`) — `ServerForm`, а сам проект — `DDG6Demo`.

Осмотрев панель редактора кода, можно заметить теперь новую, незнакомую возможность. Обратите внимание на вкладки в нижней части. Каждый Web-модуль (поскольку он представляет собой страницу в Web-приложении) имеет ассоциированный файл HTML, который может содержать серверный сценарий. Вторая вкладка в нижней части демонстрирует эту страницу (рис. 23.7). Поскольку в мастере был выбран шаблон стандартной страницы HTML, она содержит серверный сценарий, который приветствует пользователя при входе, а также простое меню навигации, которое будет автоматически сформировано на основании всех опубликованных страниц приложения. По мере добавления страниц в приложение это меню будет увеличиваться, что позволит пользователям свободно перемещаться по всем страницам приложения. Стандартный код HTML, устанавливаемый по умолчанию, представлен в листинге 23.1.

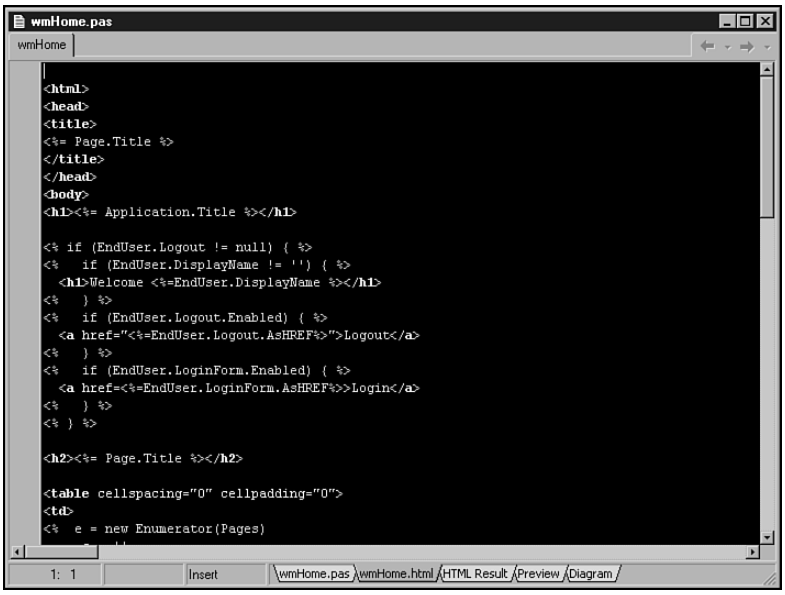

Рис. 23.7. *Страница HTML, ассоциированная с Web-модулем*

Листинг 23.1. Код HTML по умолчанию

```
<html>
<head>
<title>
<%= Page.Title %>
</title>
</head>
<body>
<h1><%= Application.Title %></h1>

<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%    } %>
<%    if (EndUser.Logout.Enabled) { %>
  <a href="<%=EndUser.Logout.AsHREF%>">Logout</a>
<%   } %>
<%   if (EndUser.LoginForm.Enabled) { %>
  <a href=<%=EndUser.LoginForm.AsHREF%>>Login</a>
<%   } %>
<% } %>

<h2><%= Page.Title %></h2>

<table cellspacing="0" cellpadding="0">
<td>
<%  e = new Enumerator(Pages)
    s = ''
    c = 0
    for (; !e.atEnd(); e.moveNext())
    {
      if (e.item().Published)
      {
        if (c>0) s += ' | '
        if (Page.Name != e.item().Name)
          s += '<a href="' + e.item().HREF + '">' +
               e.item().Title + '</a>'
        else
          s += e.item().Title
        c++
      }
    }
    if (c>1) Response.Write(s)
%>
</td>
</table>

</body>
</html>
```

Этот код содержит как обычные дескрипторы HTML, так и серверный сценарий JScript.

Обратите внимание на подсветку HTML в окне IDE. (Цвета можно устанавливать в меню Tools, пункт Editor Options, вкладка Color.) Кроме того, можно установить свой собственный внешний редактор HTML, например HomeSite, и точно так же обращаться к нему через IDE. Сменить редактор HTML можно, выбрав в меню Tools пункты Environment Options и Internet. Выберите в списке пункт HTML, а затем, щелкнув на кнопке Edit, выберите соответствующее действие редактирования, которое будет выполняться внешним редактором. Теперь, если щелкнуть правой кнопкой мыши в окне редактора кода на странице HTML, то можно будет выбирать пункт HTML Editor и вызвать свой редактор.

Следующая вкладка, после вкладки просмотра HTML, демонстрирует результат выполнения сценария HTML. Следующая вкладка демонстрирует внешний вид HTML в окне Internet Explorer. Обратите внимание, что не все сценарии выполняются и отображаются в этом представлении, поскольку часть кода зависит от значений времени выполнения. Но это позволяет хотя бы ориентировочно понять, как будет выглядеть страница, не прибегая к запуску броузера.

Расширение функциональных возможностей приложения

Теперь давайте добавим в приложение немного кода, чтобы оно могло что нибудь делать. Сначала перейдем к Web-модулю Home и выберем адаптер Application. Присвоим свойству Application.Title значение **Delphi Developers Guide 6 WebSnap Demo Application** (Демонстрационное приложение WebSnap Delphi 6 Руководство разработчика). Обратите внимание, что это немедленно отобразится во вкладке предварительного просмотра, поскольку в разделе <BODY> HTML содержит следующий серверный сценарий:

```
<h1><%= Application.Title %></h1>
```

Указанное заставит объект сценария приложения отобразить значение Application.Title в коде HTML.

Затем перейдите в редактор кода и выберите страницу HTML для модуля Home. Переместите курсор ниже дескриптора </table> и добавьте подробное описание страницы, которое разъяснит пользователю задачу проекта. Код этого примера находится на прилагаемом CD.

```
<P>
<FONT SIZE="+1" COLOR="Red">
Welcome to the Delphi 6 Developers Guide WebSnap Demonstration
Application!
</FONT> <P>
This application will demonstrate many of the new features in
Delphi 6 and WebSnap. Feel free to browse around and look at the
code involved. There is a lot of power, and thus a lot to learn, in
WebSnap, so take your time and don't try to absorb it all at once.
<P>
```

(Добро пожаловать в демонстрационное приложение WebSnap Delphi 6 Руководство разработчика! Данное приложение демонстрирует большинство новых возможностей WebSnap и Delphi 6. Не стесняйтесь внимательно просмотреть этот код. В WebSnap много полезного и интересного, поэтому запаситесь терпением и не рассчитывайте изучить все и сразу.)

Естественно, этот новый код также немедленно отобразится в панели предварительного просмотра HTML.

Теперь осталось только доказать, что созданный проект фактически является приложением броузера. Запустим проект. Сначала появится пустая форма. Это сервер COM. Если запуск прошел нормально, то ее можно сразу закрыть, а затем из меню Tools запустить отладчик Web-приложений. Щелкните в окне отладчика на гиперссылке Default URL (это обращение к `DDG6DemoApp.DDG6TestApp`), найдите приложение в списке окна броузера и щелкните на кнопке Go. Броузер должен отобразить страницу, как показано на рис. 23.8.

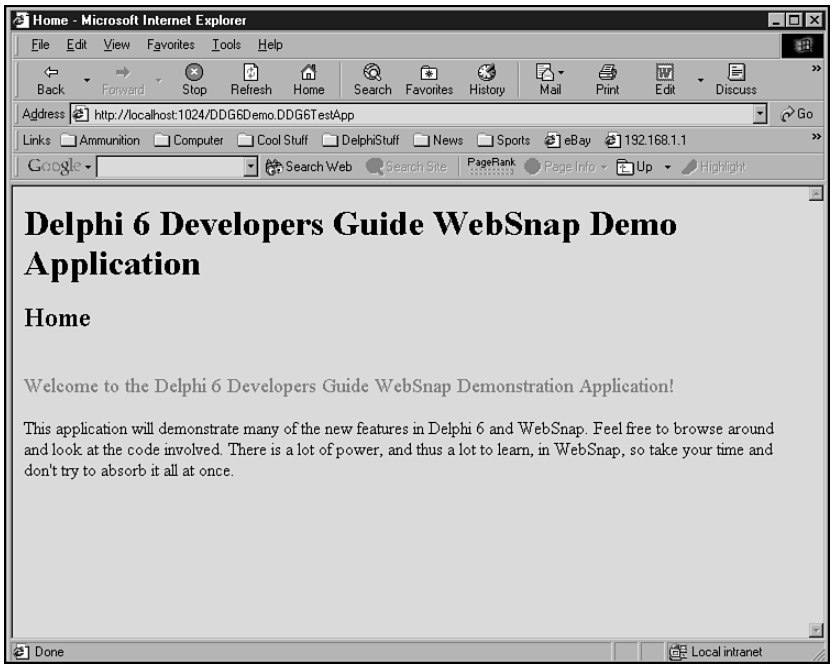

Рис. 23.8. *Результат загрузки первой страницы, добавленной в демонстрационное приложение*

Итак, вполне очевидно, что это действительно Web-приложение!

Меню навигации

Теперь добавим следующую страницу, которая продемонстрирует меню навигации. Перейдите в главное меню IDE и выберите вторую кнопку в панели меню Internet (ту, что с небольшим земным шаром и листом бумаги). Она вызовет мастер нового модуля страницы WebSnap (New WebSnap Page Module Wizard), который похож на диалоговое

окно предыдущего мастера. Оставьте все параметры со значениями, установленными по умолчанию, кроме поля редактирования Name. Назовите эту страницу **Simple**. В результате будет создан Web-модуль с одним генератором страниц. Обратите внимание: страница HTML, ассоциированная с ним, имеет то же самое содержимое, что и первая страница, рассмотренная выше. Сохраните блок как `wmSimple.pas`.

Установка параметров экземпляров и кэширования

Мастер нового модуля страницы WebSnap содержит в нижней части два параметра, которые определяют способ обработки каждого экземпляра Web-модуля. Первый параметр — Creation (Создание). Web-модули могут создаваться либо всегда (Always), либо по требованию (On Demand). Экземпляры Web-модулей, создаваемых по требованию, будут созданы лишь тогда, когда для этого будет передан специальный запрос. Такой подход выбирают для редко используемых страниц. Для страниц, которые должны быть созданы непосредственно после запуска приложения, выберите параметр Always. Вторым параметром является Caching Option (Параметр кэширования). Он определяет то, что случается с Web-модулем, когда он завершит обработку запроса. Если выбрать Cache Instance, то каждый создаваемый Web-модуль будет кэшироваться, а при завершении обработки запроса оставаться в пуле кэшируемых экземпляров, готовых для повторного использования. Следует заметить, что при повторном использовании данные полей будут в том же самом состоянии, в котором они были при завершении последнего запроса. Если выбрать Destroy Instance, то после завершения обработки запроса каждый экземпляр Web-модуля будет удален.

Теперь добавьте в страницу HTML простое сообщение ниже таблицы в стандартной странице. Затем скомпилируйте и запустите приложение через отладчик Web-приложений, как и прежде. Если с прошлого раза там еще оставалась страница, то достаточно щелкнуть на кнопке Refresh (Обновить) броузера.

На сей раз при запуске приложения обратите внимание на появившееся меню навигации. Это меню является результатом следующего серверного сценария:

```
<%
e = new Enumerator(Pages)
s = ''
c = 0
for (; !e.atEnd(); e.moveNext())
{
  if (e.item().Published)
  {
    if (c>0) s += ' | '
    if (Page.Name != e.item().Name)
      s += '<a href="' + e.item().HREF + '">' +
           e.item().Title + '</a>'
    else
      s += e.item().Title
    c++
  }
}
if (c>1) Response.Write(s)
%>
```

Данный код просто перебирает объекты сценариев страниц, создавая меню имен страниц. Если найденная страница не является текущей, то на нее создается ссылка. Таким образом, на текущую страницу ссылки не будет, а в меню отобразятся все остальные страницы, вне зависимости от того, какая из страниц является текущей в данный момент. Это довольно простое меню, обычно пользовательские меню специального приложения существенно сложнее.

НА ЗАМЕТКУ

При переключении между двумя страницами обратите внимание на возникающую на заднем плане форму приложения при каждом запросе. Это происходит потому, что отладчик Web-приложений при каждом запросе вызывает данное приложение как объект COM, запускает его, получает ответ в виде кода HTTP и завершает работу приложения.

Теперь можно сделать часть приложения закрытой для неавторизованного доступа. Сначала добавим страницу, которая требует обязательной регистрации пользователя при входе. Щелкните в панели инструментов Internet на кнопке New WebSnap Page и назовите страницу "**LoggedIn**". Затем выберите флажок LoginRequired. Щелкните на кнопке OK, и будет создана Web-страница, которую сможет просматривать только авторизованный пользователь. Сохраните страницу как `wmLoggedIn.pas`. После этого добавьте в страницу код, уведомляющий пользователя об успешном входе на страницу. Данное приложение находится на прилагаемом CD.

```
<P>
<FONT COLOR="Green"><B>Congratulations! </B></FONT> <BR>
You are successfully logged in! Only logged in users are granted
access to this page. All others are sent back to the Login page.
<P>
```

(Поздравляю! Вы успешно вошли! К этой странице разрешен доступ только проверенным пользователям. Все остальные возвращаются на страницу регистрации.)

Единственным различием между страницами `LoggedIn` и `Simple` является параметр в коде, который регистрирует страницу в диспетчере приложения. Каждая страница WebSnap имеет раздел инициализации, который выглядит примерно так:

```
initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory(
                      TWebPageModuleFactory.Create(TLoggedIn,
                      TWebPageInfo.Create([wpPublished,
                      wpLoginRequired], '.html'), ecrOnDemand,
                      caCache));
```

Этот код с помощью объекта `WebRequestHandler` регистрирует страницу, которая управляет всеми остальными страницами и предоставляет, при необходимости, содержащийся в них HTML. Объект `WebRequestHandler` способен создавать, кэшировать и уничтожать экземпляры Web-модулей. Приведенный выше код страницы `LoggedIn` обладает параметром `wpLoginRequired`, указывающим на то, что доступ к странице могут получить только проверенные пользователи. По умолчанию мастер новых страниц добавляет этот параметр в закомментированном виде. Если впоследствии понадобится защитить страницу паролем, то достаточно будет раскомментировать этот параметр и перекомпилировать приложение.

Процесс регистрации

Допустим, необходимо создать страницу, которая проверяет пользователей при входе. Это подразумевает определенные служебные действия на домашней странице (Home page).

Сначала создадим новую страницу по имени Login. Затем выберем генератор страниц типа TAdapterPageProducer. Но на сей раз страница не будет публикуемой (флажок Publish сброшен) и не требующей регистрации пользователя для просмотра содержимого! Отмена параметра Publish сделает страницу доступной для использования, но она больше не будет частью объекта сценариев страниц и, таким образом, не отобразится в меню навигации. Сохраните ее как wmLogin. Теперь перейдите на вкладку WebSnap палитры компонентов и перетащите в модуль компонент TLoginAdapter.

Компонент TAdapterPageProducer является специализированным генератором страниц, способным отображать и обрабатывать соответствующие поля и элементы управления HTML компонента TAdapter. В разрабатываемом демонстрационном приложении генератор TAdapterPageProducer будет отображать поля редактирования Username и Password, которые пользователь должен будет заполнить, чтобы получить доступ. Разобравшись в работе компонента WebSnap TAdapterPageProducer, разработчики начинают применять его повсеместно, поскольку он очень удобен для отображения информации и выполнения действий компонента TAdapter, а также создания форм HTML на основании полей класса TAdapter.

Поскольку класс TLoginFormAdapter имеет все поля, необходимые для создания страницы регистрации (login), сделать ее будет очень просто и без всякого дополнительного кода. Компонент этого класса позволяет добавлять пользователей, создавать страницы регистрации, осуществлять проверку прав пользователей на доступ к странице, и все это без единой строки кода.

Прежде чем преступить к управлению процессом регистрации, необходимо создать несколько пользователей. Перейдите к Web-модулю Home и дважды щелкните на компоненте WebUserList. Это компонент управления пользователями и паролями. Он позволяет легко добавлять пользователей и их пароли. Щелкните на кнопке New и добавьте двух различных пользователей. Введите для каждого пользователя пароли. В демонстрационном приложении, находящемся на прилагаемом CD, созданы два пользователя: ddg6 и user. Их пароли совпадают с именами, как показано на рис. 23.9.

Рис. 23.9. *Редактор компонента WebUserList в процессе добавления пользователей*

Выберите тип адаптера EndUserSessionAdapter, а также установите свойство LoginPage в состояние Login (Регистрация), то есть страницу, которая имеет элементы управления для регистрации пользователей. Затем вернитесь к Web-модулю Login и дважды щелкните на компоненте TAdapterPageProducer. Появится окно Web-дизайнера, представленное на рис. 23.10.

Выберите в верхнем левом окне элемент AdapterPageProducer и щелкните на кнопке New Component (Новый компонент). Выберите элемент AdapterForm и щелк-

ните на кнопке OK. Затем выберите AdapterForm1 и снова щелкните на кнопке New Component. Выберите AdapterErrorList, AdapterFieldGroup и AdapterCommandGroup. После этого установите свойство Adapter всех этих трех компонентов как LoginFormAdapter1. Затем выберите AdapterFieldGroup и добавьте два объекта AdapterDisplayField. Установите свойство FieldName первого из них как UserName, а второго как Password. Выберите AdapterCommandGroup и установите его свойство DisplayComponent как AdapterFieldGroup1. В результате форма должна выглядеть так, как показано на рис. 23.10. Если закрыть эту форму и перейти в редактор кода, то можно заметить, что теперь она содержит элементы управления регистрацией.

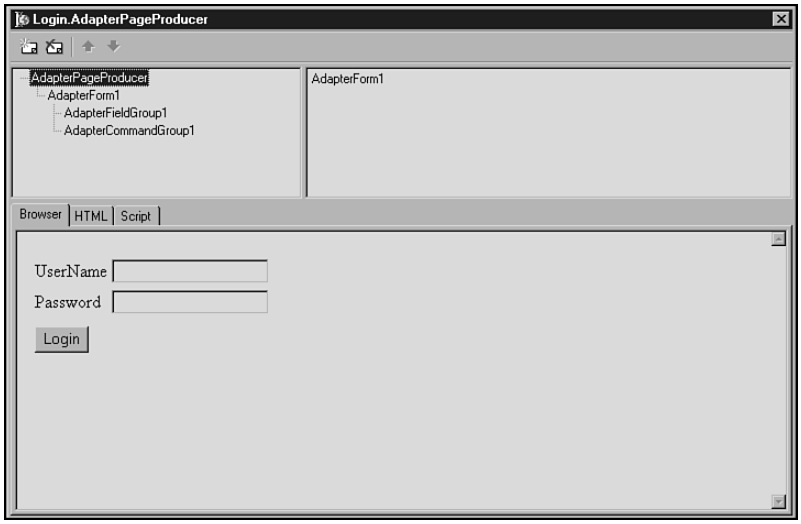

Рис. 23.10. *Компонент TAdapterPageProducer, содержащий компонент LoginFormAdapter, в окне Web-дизайнера*

Вот и все, что нужно было сделать. Запустите приложение и оставьте его в таком состоянии. Приложение нельзя закрывать, поскольку вся информация о регистрации пользователей хранится в объектах сеанса, расположенных в памяти, и если их удалить, то приложение "забудет" зарегистрированных пользователей. Запустите приложение в броузере, а затем попробуйте перейти к защищенной странице. Это перенаправит броузер на страницу регистрации (Login). Введите имя пользователя и пароль, а затем щелкните на кнопке Login. Если введено допустимое имя и пароль, то броузер перейдет на защищенную страницу, содержащую соответствующее приветствие. С этого момента любая страница, требующая аутентификации, будет предоставлена беспрепятственно. Если имя или пароль будут введены неправильно, то страница регистрации появится вновь. Интереснее всего то, что все это реализовано без единой строки кода Object Pascal.

Пробуйте сделать следующее: выйдите из защищенной страницы, щелкнув на ссылке Logout (Выход), а затем попытайтесь войти на нее снова, но с недопустимым именем пользователя или паролем. В этом случае появится сообщение об ошибке. Таково действие компонента AdapterErrorList. Он автоматически обрабатывает ошибки регистрации и самостоятельно отображает их.

Обратите внимание, что при навигации по страницам приложения после успешной регистрации пользователя приложение помнит его имя и отображает его в заголовке каждой страницы. Это результат работы следующего серверного сценария файла HTML для Web-модулей:

```
<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
```

Управление данными предпочтений пользователя

Следующим немаловажным элементом является поддержка информации о предпочтениях пользователя. Большинство динамических приложений позволяют пользователем управлять способом отображения информации на странице в довольно широких приделах: начиная с порядка расположения покупок в корзинке и до цветовых предпочтений пользователя. Конечно, WebSnap существенно упрощает эту задачу, но на сей раз придется написать несколько строк кода.

Сначала добавим в проект новую страницу. Выберем для нее тип генератора `TAdapterPageProducer` и затребуем регистрацию пользователей для ее просмотра. (Как это сделать с помощью панели инструментов и возникающего в результате мастера — рассматривалось ранее.) Сохраним файл как `wmPreferenceInput`. Добавим к Web-модулю `TAdapter` и переименуем содержимое свойства `Adapter` из `Adapter1` в `PrefAdapter`, как показано на рис. 23.11.

Рис. *23.11. Web-модуль `PreferenceInput`, который хранит предпочтения пользователей*

Дважды щелкните на компоненте `PrefAdapter`, а затем добавьте два компонента `AdapterField` и один `AdapterBooleanField`. Назовите компоненты `AdapterField` `FavoriteMovie` и `PasswordHint`, а `AdapterBooleanField` как `LikesChocolate`. (Обратите внимание: при переименовании этих компонентов изменяются значения `DisplayLabel` и `FieldName`.) Кроме того, в коде HTML значение `DisplayLabel` можно изменить на более осмысленное.

Компонент `PrefAdapter` будет содержать значения пользовательских предпочтений, а также обеспечивать доступ к ним из других страниц. Компоненты класса `TAdapter` содержат сценарии, которые могут самостоятельно хранить, управлять и манипулировать информацией, но выполнение этих действий требует определенного кода. Каждый из трех созданных компонентов `AdapterField` должен обладать возможностью получить в сценарии свои значения, поэтому каждый из них поддерживает событие `OnGetValue`. Поскольку эта информация должна быть доступна при всех обращениях данного пользователя, ее целесообразно хранить в свойстве `Session.Values` (значения сеанса). Переменная `Session.Values` представляет собой

индексированную строку как массив вариантов, поэтому в нем можно хранить практически все, что угодно на протяжении действия текущего сеанса.

Класс TAdapter позволяет также выполнять действия над своими данными. Чаще всего, для этого применяется кнопка Submit в форме HTML. Выберите компонент PrefAdapter, перейдите в инспектор объектов и дважды щелкните на свойстве Actions. Добавьте одно действие и назовите его SubmitAction. Измените значение DisplayLabel его свойства на Submit Information. Затем перейдите к вкладке Events (События) в инспекторе объектов и добавьте в обработчик события OnExecute код, представленный в листинге 23.2.

Листинг 23.2. Обработчик события OnExecute

```
procedure TPreferenceInput.SubmitActionExecute(Sender: TObject;
                                               Params: TStrings);
var
  Value: IActionFieldValue;
begin
  Value := FavoriteMovieField.ActionValue;
  if Value.ValueCount > 0 then begin
    Session.Values[sFavoriteMovie] := Value.Values[0];
  end;

  Value := PasswordHintField.ActionValue;
  if Value.ValueCount > 0 then begin
    Session.Values[sPasswordHint] := Value.Values[0];
  end;

  Value := LikesChocolateField.ActionValue;
  if Value <> nil then begin
    if Value.ValueCount > 0 then begin
      Session.Values[sLikesChocolate] := Value.Values[0];
    end;
  end else begin
    Session.Values[sLikesChocolate] := 'false';
  end;
end;
```

Когда пользователь щелкает на кнопке Submit, этот код получает значения из входных полей HTML и помещает их в переменные сеанса для последующего использования в компоненте AdapterField.

Конечно, доступ к этим значениям может понадобиться сразу, как только они будут установлены, поэтому каждое поле адаптера возвращает свое значение из объекта SessionsService. Для каждого поля адаптера установите обработчик события OnGetValue так, как это показано в листинге 23.3.

Листинг 23.3. Обработчики события OnGetValue

```
...
const
```

```
  sFavoriteMovie = 'FavoriteMovie';
  sPasswordHint = 'PasswordHint';
  sLikesChocolate = 'LikesChocolate';
  sIniFileName = 'DDG6Demo.ini';
...

procedure TPreferenceInput.LikesChocolateFieldGetValue(Sender:
                                    TObject; var Value: Boolean);
var
  S: string;
begin
  S := Session.Values[sLikesChocolate];
  Value := S = 'true';
end;

procedure TPreferenceInput.FavoriteMovieFieldGetValue(Sender:
                                    TObject; var Value: Variant);
begin
  Value := Session.Values[sFavoriteMovie];
end;

procedure TPreferenceInput.PasswordHintFieldGetValue(Sender:
                                     TObject; var Value: Variant);
begin
  Value := Session.Values[sPasswordHint];
end;
```

Теперь необходим способ отобразить элементы управления, которые фактически получат данные от пользователя. Для этого применяется генератор `TAdapterPageProducer`. Все происходит точно так же, как и со страницей Login: двойной щелчок на компоненте `TAdapterPageProducer` снова вызовет Web-дизайнер. Создайте новую форму `AdapterForm`, добавьте `AdapterFieldGroup` и `AdapterCommandGroup`. Установите значение свойства `Adapter` компонента `AdapterFieldGroup` в состояние `PrefAdaper`, а также свойство `DisplayComponent` компонента `AdapterCommandGroup` в состояние `AdapterFieldGroup`. Затем щелкните правой кнопкой мыши на компоненте `AdapterFieldGroup` и выберите в появившемся меню пункт Add All Fields (Добавить все поля). Для каждого из созданных в результате полей используйте инспектор объектов, чтобы заполнить их свойства `FieldName` соответствующими значениями. Кроме того, можно изменить значение свойства `Caption` (Заголовок) на более осмысленное, чем установлено по умолчанию. Затем выберите компонент `AdapterCommandGroup` и, щелкнув на нем правой кнопкой мыши, выберите в появившемся меню пункт Add All Commands (Добавить все команды). Установите свойство `ActionName` возникающего в результате элемента `AdapterActionButton` в состояние `SubmitAction`. И, в заключение, установите свойство `AdapterActionButton.PageName` в состояние `PreferencesPage`.

Если в Web-дизайнере что-нибудь сделано неправильно, то во вкладке Browser появится сообщение об ошибке. Сообщение информирует обо всех свойствах, которые необходимо установить для правильного взаимодействия с HTML.

После того как все будет сделано и страница HTML будет выглядеть правильно, останется лишь запустить приложение, чтобы новая страница появилась в меню. Если войти на страницу теперь, то можно заметить, что элементы управления ввода готовы принять данные пользовательских предпочтений, которые можно изменить не щелкая на кнопке Submit (поскольку ее просто нет).

Теперь с помощью панели инструментов создадим страницу для отображения предпочтений пользователя и назовем ее `PreferencesPage`. Эта страница будет защищенной и публикуемой. (Как обычно, все это можно сделать с помощью мастера.) Сохраним новый модуль как `wmPreferences`.

Затем перейдем к HTML этой страницы и в области ниже таблицы, содержащей меню навигации, добавим следующий сценарий:

```
<P>
Favorite Movie: <%
  = Modules.PreferenceInput.PrefAdapter.FavoriteMovieField.Value
%>
<BR>
Password Hint: <%
  = Modules.PreferenceInput.PrefAdapter.PasswordHintField.Value
%>

<BR>
<%
s = ''
if (Modules.PreferenceInput.PrefAdapter.LikesChocolateField.Value)
  s = 'You like Chocolate'
else
  s = 'You do not like chocolate'
Response.Write(s);
%>
```

Теперь, когда приложение скомпилировано и запущено, можно ввести свои предпочтения и щелкнуть на кнопке Submit, приложение запомнит и отобразит введенные предпочтения на странице `Preferences`. К значениям, полученным в этом сценарии, можно обращаться из любой другой страницы точно так же, как если бы они были установлены в ней. Значения хранятся в объектах `Session` на протяжении всех обращений HTTP и могут быть в любой момент возвращены с помощью сценария из компонента `Adapter`.

НА ЗАМЕТКУ

Как можно заметить, с каждой из страниц в приложении WebSnap ассоциирован файл HTML. Но поскольку такие файлы существуют вне приложения, их можно редактировать, сохранять и обновлять в окне броузера без перекомпиляции приложения. Это означает, что саму страницу можно модифицировать без перезагрузки Web-сервера. Во время разработки с серверными сценариями также можно экспериментировать свободно, безо всякой необходимости перекомпилировать приложение. Далее в настоящей главе рассматриваются альтернативные способы сохранения и получения HTML.

Хранение данных между сеансами

Теперь осталась только одна проблема: установленные пользователем данные не сохраняются между сеансами. Информация о предпочтениях будет утеряна, если пользователь выйдет из приложения. Но если сохранить эти значения в конце сеанса и прочитать их в начале следующего, то можно передать данные между сеансами. Создаваемое демонстрационное приложение считывает необходимые данные в обработчике события `LoginFormAdapter.OnLogin`, а затем сохраняет их в обработчике события `SessionService.OnEndSession`. Код обработчиков этих событий приведен в листинге 23.4.

Листинг 23.4. Обработчики событий `OnLogin` и `OnEndSession`

```
procedure TLogin.LoginFormAdapter1Login(Sender: TObject;
                                        UserID: Variant);
var
  IniFile: TIniFile;
  TempName: string;
begin
  // Данные сеанса захватываются здесь
TempName:= Home.WebUserList.UserItems.FindUserID(UserId).UserName;
  // WebContext.EndUser.DisplayName;
  Home.CurrentUserName := TempName;

  Lock.BeginRead;
  try
   IniFile := TIniFile.Create(IniFileName);
   try
    Session.Values[sFavoriteMovie]:= IniFile.ReadString(TempName,
                                     sFavoriteMovie, '');
    Session.Values[sPasswordHint]:= IniFile.ReadString(TempName,
                                     sPasswordHint, '');
    Session.Values[sLikesChocolate]:= IniFile.ReadString(TempName,
                                     sLikesChocolate, 'false');
   finally
     IniFile.Free;
   end;
  finally
    Lock.EndRead;
  end;
end;

procedure THome.SessionsServiceEndSession(ASender: TObject;
      ASession: TAbstractWebSession; AReason: TEndSessionReason);
var
  IniFile: TIniFile;
begin
  // Сохранение данных о предпочтениях
  Lock.BeginWrite;
  if FCurrentUserName <> '' then begin
    try
```

```
      IniFile := TIniFile.Create(IniFileName);
      try
        IniFile.WriteString(FCurrentUserName, sFavoriteMovie,
                            ASession.Values[sFavoriteMovie]);
        IniFile.WriteString(FCurrentUserName, sPassWordHint,
                            ASession.Values[sPasswordHint]);
        IniFile.WriteString(FCurrentUserName, sLikesChocolate,
                            ASession.Values[sLikesChocolate]);
      finally
        IniFile.Free;
      end;
    finally
      Lock.EndWrite
    end;
  end;
end;
```

Эти обработчики событий хранят данные в файле .INI, но нет никаких причин, по которым нельзя было бы сохранить данные в базе или любым другим способом.

Переменная Lock является глобальной переменной типа TMultiReadExclusive-WriteSynchronizer, которая создается в разделе инициализации страницы Home. Поскольку осуществлять чтение и запись в файл .INI могут несколько сеансов одновременно, этот компонент позволяет синхронизировать эти процессы. Добавьте в раздел интерфейса модуля wmHome следующее объявление:

```
var
  Lock: TMultiReadExclusiveWriteSynchronizer;
А затем добавьте в него разделы инициализации (initialization) и
завершения (finalization):
Initialization
  ...
  Lock := TMultiReadExclusiveWriteSynchronizer.Create;
finalization
  Lock.Free;
```

Этот код использует также функцию IniFileName, которая объявлена следующим образом:

```
const
  sIniFileName = 'DDG6Demo.ini';

...
function IniFileName: string;
begin
  Result := ExtractFilePath(GetModuleName(HInstance)) +
                            sIniFileName;
end;
```

Добавьте данный код в модуль wmHome и можно считать, что создание полнофункционального Web-приложения завершено. Приложение способно не только проверять пользователей при входе, но и подстраиваться под их предпочтения, а также запоминать эту информацию и использовать ее при следующем посещении пользователем данных страниц.

Обработка изображений

Практически каждое Web-приложение содержит графику. Графика не только украшает приложение но и способна обладать функциональными возможностями. Естественно, WebSnap позволяет использовать в создаваемых приложениях рисунки и графику так же легко и просто, как и все остальное. Как и следовало ожидать, WebSnap позволяет использовать рисунки и графику из любого доступного источника: из файла, ресурса, потоковой базы данных и так далее. Любые графические данные, которые можно поместить в поток, можно использовать в приложении WebSnap.

Используя панель инструментов Internet, добавьте новую страницу в проект приложения. Используйте генератор типа `TAdapterPageProducer`, создайте публикуемую и защищенную страницу. Назовите страницу `Images` и сохраните ее под именем `wmImages`. Затем перейдите к Web-модулю `Images` и, добавив компонент `TAdapter`, сохраните его под именем `ImageAdapter`. И, наконец, дважды щелкнув на элементе `ImageAdapter`, добавьте два поля типа `TAdapterImageField`. Каждый из них представит свой способ отображения изображений.

Сначала отобразим рисунок на основании его URL. Выберите первый элемент `AdaperImageField` и установите в его свойстве `HREF` полный адрес URL, указывающий на рисунок расположенный на локальном сервере Internet или где-нибудь в другом месте. Например, если хочется просмотреть хронологию изменения биржевых котировок *Borland* за последний год, установите значение свойства `HREF` равным `http://chart.yahoo.com/c/1y/b/borl.gif`.

Дважды щелкните в Web-модуле `Images` на компоненте `TAdapterPageProducer`, добавьте новую форму `AdapterForm`, а затем добавьте в нее `AdapterFieldGroup`. Установите свойство `AdapterFieldGroup` этого нового адаптера в состояние `ImageAdapter`. Затем вновь щелкните правой кнопкой мыши на компоненте `AdapterFieldGroup` и выберите в появившемся меню пункт Add All Fields. Установите поле `ReadOnly` компонента `AdapterImageField` в состояние `True`, это отобразить рисунок на странице. Если установить его в состояние `False`, то вместо рисунка будет отображено поле редактирования и кнопка, позволяющие выбрать имя файла. В данном случае необходимо изображение, поэтому установите это свойство в состояние `True`. При просмотре изображения в первый раз обратите внимание на небольшой заголовок рисунка. Обычно от него стараются избавиться, поэтому установим в свойстве `Caption` пробел. (Обратите внимание: заголовок не может быть пуст.) Теперь просмотрим диаграмму в окне Web-дизайнера, как показано на рис. 23.12.

НА ЗАМЕТКУ

Если к изображению необходимо обратиться по относительной ссылке, чтобы видеть рисунок и во время разработки, то каталог файлов изображений следует добавить в путь поиска (Search Path) отладчика Web-приложений.

С выводом на экран изображения на основании его URL все ясно. Но иногда нужно получать изображение из потока. Компонент `AdapterImageField` обеспечивает поддержку и для этого случая. Выберите второй компонент `AdapterImageField` формы `ImageAdapter` и откройте инспектор объектов. Перейдите к странице событий и дважды щелкните на событии `OnGetImage`. Поместите файл изображения `.JPG`

в каталог приложения (демонстрационное приложение на CD использует файл `athena.jpg`) и создайте свой обработчик события, похожий на следующий:

```
procedure TImages.AdapterImageField2GetImage(Sender: TObject;
                      Params: TStrings; var MimeType: String;
                      var Image: TStream; var Owned: Boolean);
begin
  MimeType := 'image\jpg';
  Image := TFileStream.Create('athena.jpg', fmOpenRead);
end;
```

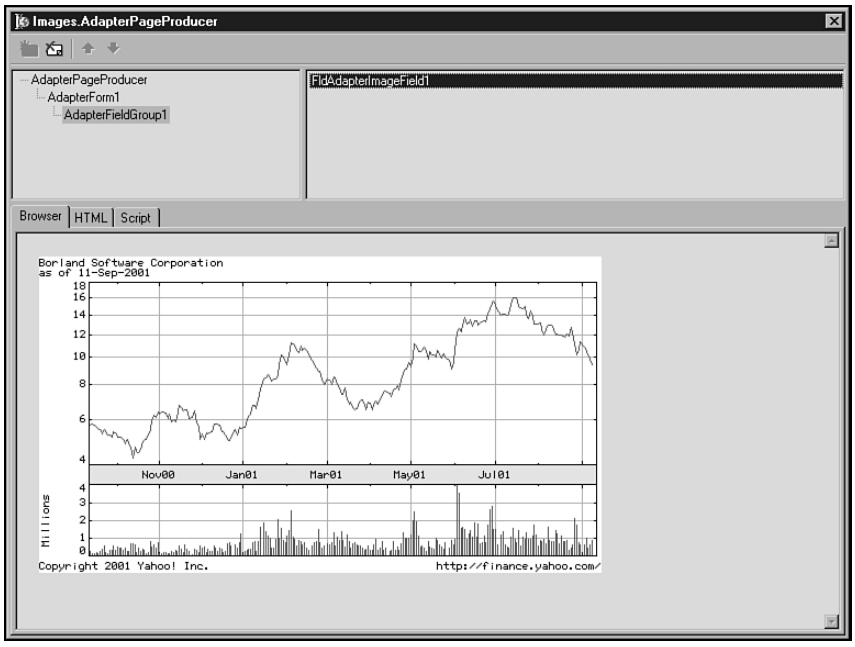

Рис. 23.12. *Web-дизайнер с графическим изображением поля* `ImageAdapter-Field`

Это совсем простой код. `Image` — это переменная потока, которая создается и заполняется изображением. Конечно, приложению должен быть известен тип загружаемого изображения, его необходимо указать в параметре `MimeType`. Решение с помощью класса `TFileStream` достаточно просто, но позволяет получать изображение из любого источника, например поля `BlobStream` базы данных или созданного динамически и переданного в память в потоке. Теперь при запуске приложения на экране должно появиться выбранное изображение JPG (справа или ниже диаграммы биржевых котировок).

Отображение данных

Конечно, реальное приложение должно выполнять задачи более сложные, чем примеры, рассмотренные до сих пор. Возникнет необходимость отображать данные из базы либо в виде таблицы, либо запись за записью. Естественно, WebSnap существенно упрощает и эту задачу, позволяя создать мощные приложения баз данных с минимумом кода. Используя экземпляры класса `TDatasetAdapter` и встроенные в него поля и действия, можно легко отображать данные, а также добавлять, модифицировать и удалять записи в любой базе данных.

Фактически метод отображения набора данных в форме очень прост. Добавьте в демонстрационное приложение новый модуль, но на сей раз создайте его как Web-модуль данных, используя третью кнопку панели инструментов Internet. Это очень простой мастер, поэтому оставьте все значения установленными по умолчанию. Затем добавьте компоненты `TDatasetAdapter` из вкладки WebSnap и `TTable` из вкладки BDE палитры компонентов. Присвойте компоненту `TTable` базу данных `DBDEMOS`, а затем таблицу `BioLife`. Установите свойство `Dataset` компонента `DatasetAdapter1` в состояние `Table1`. И, в заключение, установите свойство `Table1.Active` в состояние `True`, чтобы таблицу можно было открыть. Назовите Web-модуль данных `BioLife` и сохраните его как `wdmBioLife`.

НА ЗАМЕТКУ

Данное приложение использует простую таблицу BDE Paradox, но компонент `TDatasetAdapter` способен отобразить данные любого объекта производного от класса `TDataset`. Обратите внимание — это не самая хорошая идея использовать экземпляры класса `TTable` в Web-приложении безо всякой поддержки сеанса. В демонстрационном приложении это сделано только для простоты, чтобы не отвлекать внимание от возможностей WebSnap при работе с данными.

Теперь, для разнообразия, используем для установки свойств компонентов объект Treeview (Древовидное представление). Если объект Treeview невидим, то выберите в меню View пункт Object Treeview. Выберите `DatasetAdapter`, щелкните правой кнопкой мыши на узле `Actions` и в появившемся контекстном меню выберите пункт Add All Actions. Затем подключите компонент `TTable` к `TAdapterDataset` с помощью его свойства `Dataset`. Выберите узел `Fields` и щелкнув на нем правой кнопкой мыши выберите в появившемся контекстном меню пункт Add All Fields. Сделайте то же самое для `TTable`, подключив все поля набора данных к Web-модулю данных. После этого, поскольку для операций с базами данных WebSnap создает серверы без состояния, необходимо указать первичный ключ набора данных, чтобы обеспечить клиентам возможность перемещения и манипулирования данными. Все указанное WebSnap сделает автоматически, после того как будет указан первичный ключ. Для этого выберите в древовидном представлении поле Species_No Field и добавьте значение `pfInKey` в его свойство `ProviderFlags`.

Затем добавьте в приложение обычную страницу. Сделайте ее защищенной с генератором типа `TAdapterPageProducer` по имени `Biolife`. Сохраните модуль как `wmBioLife`. Поскольку в этой странице будут отображены данные, добавим в раздел `uses` модуля `wmBioLife` имя модуля `wdmBioLife`. Теперь выберите Web-модуль `BioLife` и щелкните правой кнопкой мыши на компоненте `AdapterPageProducer`.

Снова щелкните правой кнопкой мыши на узле WebPageItems, расположенном непосредственно ниже его, и в появившемся контекстном меню выберите пункт New Component и AdapterForm. Выберите AdapterForm и, щелкнув правой кнопкой мыши, добавьте компоненты AdapterErrorList и AdapterGrid. Установите свойство Adapter обоих компонентов в состояние DatasetAdapter. Щелкните правой кнопкой мыши на компоненте AdapterGrid и в появившемся контекстном меню выберите пункт Add All Columns. Затем выберите узел Actions, расположенный ниже DatasetAdapter и, щелкнув правой кнопкой мыши, выберите пункт Add All Actions. Затем выберите узел Fields и, щелкнув правой кнопкой мыши, добавьте все поля. Теперь все свойства установлены правильно и можно отображать данные.

Перейдите к Web-модулю BioLife и дважды щелкните на компоненте AdaperPageProducer. Появится Web-дизайнер с загруженными реальными данными. Если этого не произошло, удостоверьтесь в возможности открыть таблицу, а также проверьте правильность установки всех свойств Adapter компонентов в DatasetAdapter. Поле Notes делает таблицу слишком длинной, поэтому выберите в верхней левой панели дизайнера компонент AdapterGrid, а в верхней правой — элемент ColNotes — и удалите его. Теперь все должно выглядеть так, как показано на рис. 23.13.

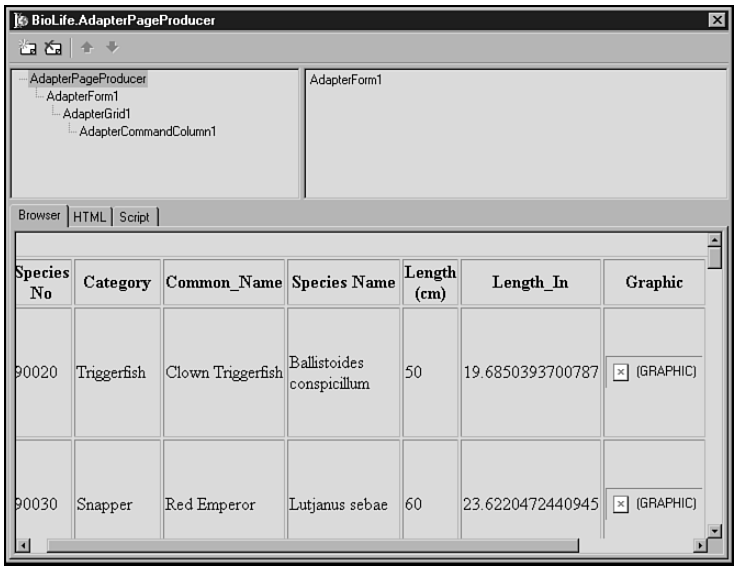

Рис. 23.13. *Таблица BioLife, представленная компонентом TAdapterPageProducer в окне Web-дизайнера. Таблица HTML создана компонентом TDatasetAdapter*

Во время разработки графика не отображается, но она появится во время выполнения. Теперь, если откомпилировать и запустить приложение, можно будет просмотреть все данные страницы BioLife. И все это — без единой строки программного кода, написанной вручную.

Конечно, простой просмотр данных не очень полезен. В реальном приложении, вероятно, понадобится работать с отдельными записями. Естественно, в WebSnap это

предусмотрено. Перейдите в Web-дизайнер и выберите компонент AdapterGrid. Щелкните на нем правой кнопкой мыши и добавьте адаптер AdapterCommandColumn. Затем щелкните правой кнопкой мыши на нем и выберите команды DeleteRow, EditRow, BrowseRow и NewRow, как показано на рис. 23.14.

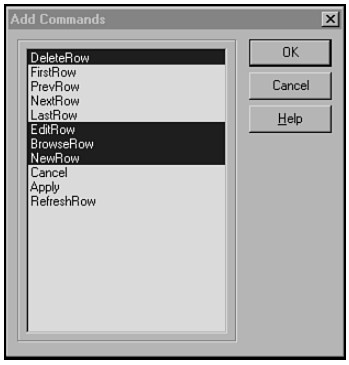

Рис. 23.14. *Используя диалоговое окно Add Commands, можно выбрать действия, выполняемые над отдельными строками набора данных*

Щелкните на кнопке OK. Затем расположите в ячейке таблицы соответствующие кнопки в вертикальном порядке, установив свойство DisplayColumns компонента AdapterCommandColumn в состояние 1. Теперь набор кнопок управления в окне Web-дизайнера должен выглядеть так, как на рис. 23.15.

Кнопки установлены. Теперь, для того чтобы они могли отображать и выполнять какие-либо действия с отдельной записью, необходима новая страница. Добавьте ее в проект как защищенную с генератором TAdapterPageProducer. Назовите ее BioLifeEdit и сохраните модуль как wmBioLifeEdit. Чтобы обеспечить доступ к данным, добавьте в раздел uses модуль wdmBioLife.

Щелкните дважды на компоненте TAdapterPageProducer в новом Web-модуле и добавьте форму AdapterForm. Затем добавьте компоненты AdapterErrorList, AdapterFieldGroup и AdapterCommandGroup. Щелкните правой кнопкой мыши на компоненте AdapterFieldGroup и добавьте все поля, а затем в компоненте AdapterCommandGroup добавьте все команды. Окно Web-дизайнера должно выглядеть примерно так, как показано на рис. 23.16.

Теперь, чтобы использовать эту страницу для редактирования отдельных записей, вернемся к модулю wmBioLife, в котором установлена таблица, и, используя клавишу SHIFT, в объекте TreeView выделим все четыре кнопки в AdapterCommandColumn. Присвойте их свойству страниц EditBioLife имя страницы, которая отобразит отдельную запись. Теперь при щелчке на кнопке в таблице будет отображаться страница EditBioLife. Если запись необходимо только просмотреть, то данные будут представлены в виде простого текста. Но если ее нужно отредактировать, то данные будут отображены в полях редактирования. Графическое поле позволит добавить в базу данных новый рисунок, обеспечив поиск и просмотр его файла. Перемещаться по набору данных также можно с помощью кнопок. И снова все это сделано без единой строки программного кода или сценария, написанного вручную.

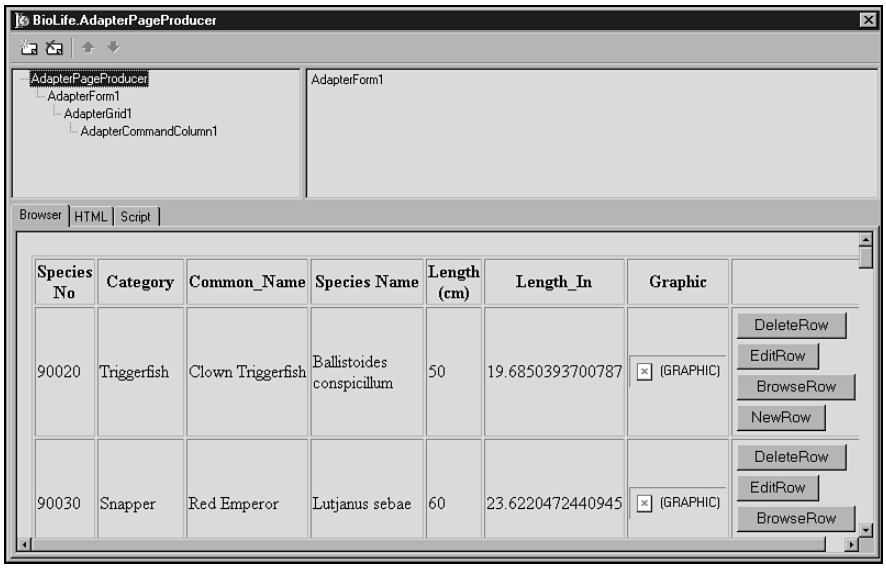

Рис. 23.15. *Демонстрационное приложение с кнопками управления*

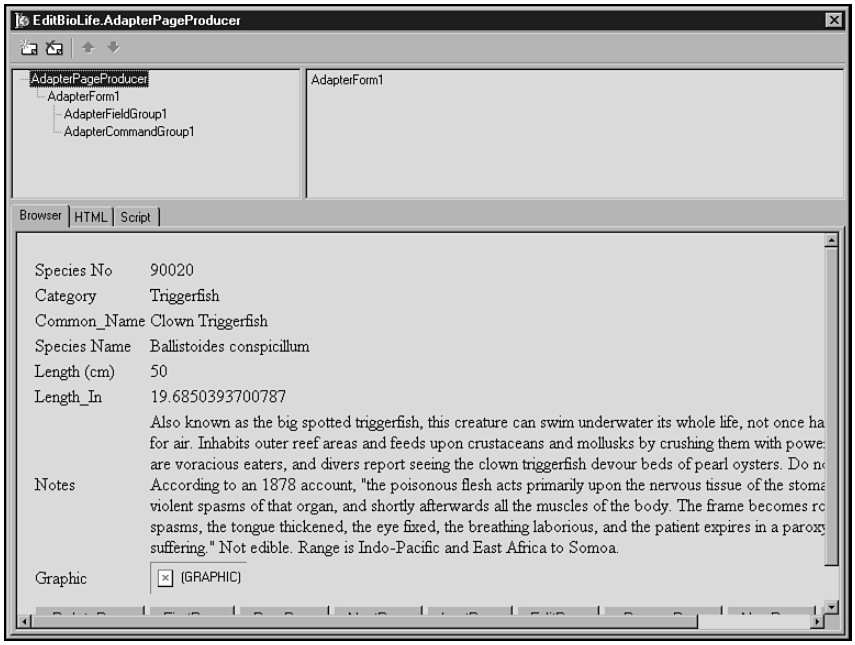

Рис. 23.16. *Страница* `BioLifeEdit` *со всеми полями и действиями в окне Web-дизайнера*

НА ЗАМЕТКУ

В принципе, можно было обеспечить представление и поля `Notes`. По умолчанию, когда страница находится в режиме редактирования, для него используется элемент управления `TextArea`, небольшего размера и не обеспечивающий перенос текста по словам. Можно выбрать компонент `FldNotes` и откорректировать его свойства `TextAreaWrap`, `DisplayRows` и `DisplayWidth` в соответствии с конкретными условиями.

Преобразование приложения в DLL ISAPI

До сих пор приложение выполнялось под управлением отладчика Web-приложений, облегчая его проверку и отладку. Но это, конечно, не метод установки реального приложения. Обычно для этого применяют приложение в виде DLL ISAPI, чтобы оно могло постоянно находиться в памяти и содержать всю информацию, необходимую для обеспечения работы сеансов связи.

Преобразовать приложение из ориентированного на отладчик Web-приложений в сервер ISAPI достаточно просто. Для этого создайте новый, пустой проект ISAPI и удалите из него все установленные по умолчанию модули. Затем добавьте в него все модули Web-приложения версии отладчика, за исключением форм. Скомпилируйте и запустите приложение. Все очень просто. Фактически можно иметь два проекта, которые одновременно используют те же самые Web-модули: один проект — для отладки, а другой — для установки. Большинство демонстрационных приложений в каталоге `WebSnap` находящемся на прилагаемом CD, сделаны именно так — они состоят из двух проектов: один — для отладчика, другой для ISAPI. При установке новой DLL ISAPI убедитесь, что все необходимые файлы HTML установлены в том же самом каталоге, что и файл библиотеки DLL.

Дополнительные возможности

Все, что рассматривалось до сих пор, было лишь предварительной подготовкой. Созданное приложение WebSnap способно управлять пользователями, информацией сеансов этих пользователей, а также работать с данными. WebSnap обладает значительно большими возможностями и позволяет приложению выполнять существенно более сложные задачи. Этот раздел представляет ряд дополнительных возможностей, которые позволят существенно улучшить разрабатываемое приложение WebSnap.

Компонент LocateFileServices

Разработка Web-приложений с помощью WebSnap обычно требует координации различных ресурсов. HTML, серверные сценарии, код Delphi, информация базы данных и графика — все они должны быть увязаны в единое приложение. Чаще всего большая часть этих ресурсов встроена в файл HTML. WebSnap позволяет обеспечить поддержку HTML независимо от реализации динамических Web-страниц. Это означает, что файлы HTML можно редактировать независимо от бинарного файла Web-приложения. Но по умолчанию файлы HTML должны находиться в том же самом каталоге, что и бинарный файл. Указанное не всегда удобно и не всегда возможно. Обычно файлы HTML стараются разместить отдельно от бинарного кода. Либо ре-

сурсы HTML, применяемые в приложении, могут принадлежать другому приложению, например базе данных.

WebSnap дает возможность получить HTML из любого необходимого источника. Компонент `LocateFileService` позволяет получать HTML из любого места расположения, включая файлы и потоки (фактически из любого объекта, производного от класса `TStream`). Возможность доступа к HTML из экземпляров классов производных от `TStream` означает, что существует возможность получить HTML из любого источника, который можно загрузить в объект класса `TStream`.[1]

Например, HTML можно создать динамически из потока файла `.RES`, встроенного в бинарный файл приложения. Простое демонстрационное приложение поможет понять, как это можно сделать. Естественно, для этого необходима страница HTML. С помощью любого текстового редактора (или вашего любимого редактора HTML), выберите файл `wmLogin.html` в качестве шаблона и сохраните его в каталоге создаваемого демонстрационного приложения как `embed.html`. Затем добавьте в файл какой-нибудь текст, чтобы его можно было узнать при загрузке (например уведомление об успешной загрузке встроенного файла `.RES`).

Затем, безусловно, необходимо внедрить данный код HTML в создаваемое приложение. В Delphi это можно сделать с помощью файлов ресурсов `.RC`, автоматически подключаемых в состав приложения. Использовав Notepad (Блокнот) или любой другой текстовый редактор, создайте текстовый файл. Сохраните его под именем `HTML.RC` в том же самом каталоге, где находятся исходные файлы демонстрационного приложения, и добавьте его в проект. Затем добавьте в этот файл следующий текст:

```
#define HTML 23  // Идентификатор ресурса HTML
EMBEDDEDHTML HTML embed.html
```

При подключении в проект, Delphi скомпилирует файл RC в файл RES и включит его в состав приложения.

Когда текст HTML окажется в составе приложения, создайте новую страницу с генератором класса `TPageProducer` и назовите ее `Embedded`. Сохраните файл как `wmEmbedded`, перейдите к странице `Home` и выберите компонент `LocateFileServices`. Перейдите на вкладку Events инспектора объектов и дважды щелкните на событии `OnFindStream`. В результате будет создан обработчик события, похожий на этот:

```
procedure THome.LocateFileServiceFindStream(ASender: TObject;
        AComponent: TComponent; const AFileName: String;
        var AFoundStream: TStream; var AOwned, AHandled: Boolean);
begin

end;
```

Основными параметрами здесь являются `AFileName` и `AFoundStream`. Именно их используют для доступа к HTML, расположенного во встроенных ресурсах. Заполните обработчик события следующим кодом:

```
procedure THome.LocateFileServiceFindStream(ASender: TObject;
        AComponent: TComponent; const AFileName: String;
        var AFoundStream: TStream; var AOwned, AHandled: Boolean);
```

[1] *Короче говоря, все, что только можно загрузить, можно использовать в составе HTML.* – Прим. ред.

```
begin
  // разыскать файл Embedded
  if Pos('EMBEDDED', UpperCase(AFileName)) > 0 then begin
    AFoundStream := TResourceStream.Create(hInstance,
                                           'EMBEDDED', 'HTML');
    AHandled := True;     // далее можно не искать
  end;
end;
```

Параметр `AFileName` содержит имя файла HTML (без пути), который Delphi будет использовать по умолчанию. Его можно использовать для того, чтобы указать имя, если ресурс расположен в файле. При передаче в обработчик события параметр `AFoundStream` будет содержать значение `nil`, что позволяет создать новый поток, используя эту переменную. В данном случае переменной `AFoundStream` присваивается результат выполнения метода `Create` объекта `TResourceStream`, который захватывает HTML из ресурсов исполняемой программы. Установка параметра `AHandled` в состояние `True` гарантирует прекращение методом `LocateFileServices` дальнейшего поиска содержимого HTML.

Запустите приложение и убедитесь, что код HTML страницы `Embedded` обнаружен и отображен.

Загрузка файлов

В прошлом, одной из самых сложных задач при разработке Web-приложений была передача файла от клиента на сервер. Чаще всего для этого использовались недокументированные возможности спецификации HTTP, которые требовали очень тщательной побитовой передачи информации. Как и следовало ожидать, WebSnap решает и эту задачу легко и просто. WebSnap содержит все функции, необходимые для загрузки файла, внутри класса `TAdapter`, поэтому осуществить передачу файла ничуть не сложнее, чем загрузить его в поток.

Создадим в приложении, как обычно, новую страницу, которая будет передавать файлы от клиента на сервер. Назовем эту страницу `Upload` и разместим в ней генератор типа `TAdapterPageProducer`. Сохраним файл как `wmUpload`. Перенесем в форму компонент `TAdapter` и назовем его `AdapterFileField`. Это поле будет управлять всей загрузкой файлов, выбранных на клиенте. Кроме того, назначим адаптеру одно действие и назовем его `UploadAction`.

Теперь добавим в форму `AdapterForm` компоненты `AdapterErrorList`, `AdapterFieldGroup` и `AdapterCommandGroup`. Первые два подключим к `Adapter1`, а `AdapterCommandGroup` — к `AdapterFieldGroup`. Затем добавьте в `AdapterFieldGroup` все поля, а в `AdapterCommandGroup` все действиям. Измените название кнопки на Upload File (Загрузить файл). То, что должно получиться в результате в окне Web-дизайнера, представлено на рис. 23.17.

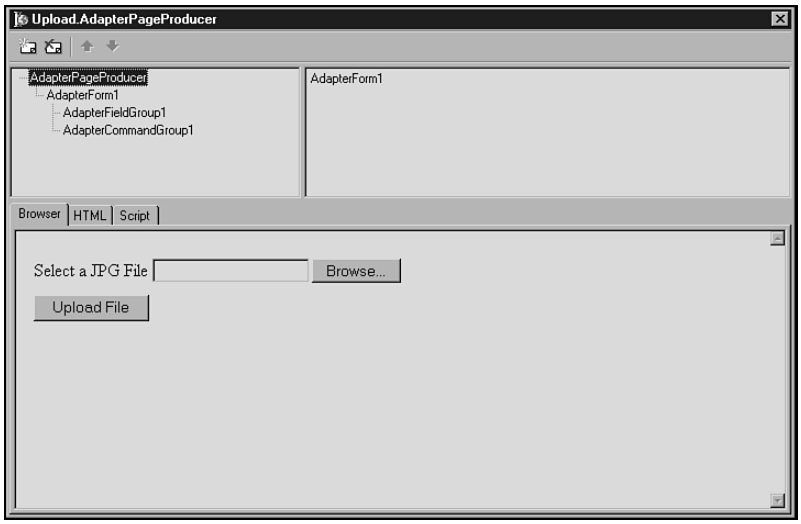

Рис. 23.17. *Страница `Upload` в окне Web-дизайнера. Кнопка* **Browse** *добавлена автоматически*

Код можно добавить в двух местах. В обработчике события `OnFileUpload` компонента `Adapter1.AdapterFileField` этот код может выглядеть подобно листингу 23.5.

Листинг 23.5. Обработчик события `OnFileUpload`

```
procedure TUpload.AdapterFileField1UploadFiles(Sender: TObject;
                                          Files: TUpdateFileList);
var
  i: integer;
  CurrentDir: string;
  Filename: string;
  FS: TFileStream;
begin
  // Здесь происходит загрузка файла
  if Files.Count <= 0 then
    raise Exception.Create('Not any files to be uploaded');
  for i := 0 to Files.Count - 1 do begin
    // Удостовериться, что это файл .jpg или .jpeg
    if (CompareText(ExtractFileExt(Files.Files[I].FileName),
          '.jpg') <> 0) and (CompareText(ExtractFileExt(
          Files.Files[I].FileName), '.jpeg') <> 0) then
      Adapter1.Errors.AddError(
                'You must select a JPG or JPEG file to upload');
    else begin
      CurrentDir := ExtractFilePath(GetModuleName(HInstance)) +
                    'JPEGFiles';
      ForceDirectories(CurrentDir);
      FileName := CurrentDir + '\' +
                  ExtractFileName(Files.Files[I].FileName);
```

```
      FS := TFileStream.Create(Filename,
                               fmCreate or fmShareDenyWrite);
      try
        // 0 = копировать все с самого начала
        FS.CopyFrom(Files.Files[I].Stream, 0);
      finally
        FS.Free;
      end;
    end;
  end;
end;
```

Сначала этот код проверяет, выбран ли файл, и если это так, то является ли он файлом `.jpg` или `.jpeg`. Удостоверившись, что так и есть, он получает имя файла и проверяет наличие каталога назначения. Если каталог существует, то файл передается в поток `TFileStream`. Все основные действия происходят внутри экземпляра класса `TUpdateFileList`, который управляет всеми таинственными процессами HTTP, необходимыми для загрузки файл от клиента на сервер.

Вторым местом для добавления кода является обработчик события `OnExecute` действия `UploadAction` компонента `Adapter1`. Вот этот код:

```
procedure TUpload.UploadActionExecute(Sender: TObject;
                                      Params: TStrings);
begin
  Adapter1.UpdateRecords;
end;
```

Он просто сообщает компоненту `Adapter1` о необходимости обновить свои записи и получить необходимые файлы.

Применение специальных шаблонов

Как уже, вероятно, заметил читатель, создавая в мастере новую страницу, можно выбрать для приложения либо стандартный шаблон HTML, либо пустой. Стандартный шаблон хорош для демонстрационного приложения в настоящей главе, но при разработке более сложных сайтов понадобится способ автоматически подключить свои собственные шаблоны HTML при добавлении страниц в приложение. WebSnap позволяет и это.

Создав и зарегистрировав в пакете времени разработки класс, производный от `TProducerTemplatesList`, появляется возможность добавлять новые шаблоны, которые впоследствии можно будет выбирать и использовать в мастере новых страниц. Демонстрационный пакет расположен в каталоге `<Delphi>\Demos\WebSnap\Producer Template`. Его можно просмотреть и, при необходимости, добавить в файл RC свои собственные шаблоны HTML или сценарии, содержащийся в пакете. Обратите внимание: для компиляции этого пакета необходимо сначала откомпилировать пакет `<Delphi>\Demos\WebSnap\Util\TemplateRes.dpk`. После компиляции и установки этих пакетов появится большое количество дополнительных шаблонов, которые можно будет выбирать в мастере новых страниц.

Специальные компоненты TAdapterPageProducer

Бо́льшая часть работы по отображению HTML в настоящей главе осуществлялась компонентом TAdapterPageProducer и компонентами, встроенными в него. Безусловно, в реальном приложении стандартных страниц HTML, использовавшихся до сих пор, будет явно недостаточно. WebSnap позволяет пользователям создавать свои собственные компоненты, которые можно будет затем подключать в TAdapterPageProducer. Это позволит не только разнообразить страницы HTML но и решить конкретные задачи.

Специальные компоненты TAdapterPageProducer должны быть производными от класса TWebContainedComponent и реализовать интерфейс IWebContent. Поскольку этим условиям должны удовлетворять все компоненты, имеет смысл использовать абстрактный класс, как показано в листинге 23.6.

Листинг 23.6. Абстрактный класс, производный от TWebContainedComponent

```
type

  Tddg6BaseWebSnapComponent = class(TWebContainedComponent,
                                    IWebContent)
  protected
    { IWebContent }
    function Content(Options: TWebContentOptions;
                     ParentLayout: TLayout): string;
    function GetHTML: string; virtual; abstract;
  end;
```

Этот класс реализован следующим образом:

```
function Tddg6BaseWebSnapComponent.Content(Options:
                                    TWebContentOptions;
                                    ParentLayout: TLayout): string;
var
  Intf: ILayoutWebContent;
begin
  if Supports(ParentLayout, ILayoutWebContent, Intf) then
    Result := Intf.LayoutField(GetHTML, nil)
  else
    Result := GetHTML;
end;
```

Абстрактный класс реализует функцию Content только потому, что функция GetHTML объявлена абстрактной. Функция Content в основном выясняет, является ли содержащий компонент LayoutGroup. Если это так, то функция Content помещает свое содержимое внутрь компонента LayoutGroup. В противном случае она просто возвращает содержимое GetHTML. Следовательно, производным компонентам достаточно лишь реализовать функцию GetHTML, возвращающую соответствующий код HTML, и можно их регистрировать для работы в составе TAdapterPageProducer.

Код на прилагаемом CD реализует два компонента, которые позволяют добавить содержимое HTML в TAdapterPageProducer в виде строки или файла. Код компонента Tddg6HTMLCode представлен в листинге 23.7.

Листинг 23.7. Компонент Tddg6HTMLCode

```
Tddg6HTMLCode = class(Tddg6BaseWebSnapComponent)
  private
    FHTML: TStrings;
    procedure SetHTML(const Value: TStrings);
  protected
    function GetHTML: string; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property HTML: TStrings read FHTML write SetHTML;
  end;

constructor Tddg6HTMLCode.Create(AOwner: TComponent);
begin
  inherited;
  FHTML := TStringList.Create;
end;

destructor Tddg6HTMLCode.Destroy;
begin
  FHTML.Free;
  inherited;
end;

function Tddg6HTMLCode.GetHTML: string;
begin
  Result := FHTML.Text;
end;

procedure Tddg6HTMLCode.SetHTML(const Value: TStrings);
begin
  FHTML.Assign(Value);
end;
```

Это очень простой класс. Он всего лишь обеспечивает публикацию свойства типа TString, который будет принимать любой код HTML, а затем помещать его в компонент TAdapterPageProducer в исходном состоянии. Функция GetHTML просто возвращает код HTML в виде строки. Можно создать компоненты, способные возвращать любой код HTML, включая изображения, ссылки, файлы и другое содержимое. Для этого все производные компоненты должны содержать переопределенный метод GetHTML(). Обратите внимание на необходимость регистрации функции в модуле, реализующем компонент. При создании компонентов удостоверьтесь, что они заре-

гистрированы в данном модуле, как в примере на прилагаемом CD. Использовать эти компоненты очень просто – достаточно установить их в пакет времени разработки и они появятся в Web-дизайнере компонента `TAdapterPageProducer` (рис. 23.18).

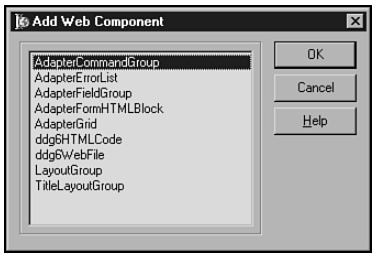

Рис. 23.18. *Компоненты `TadapterPageProducer` в окне Web-дизайнера*

Резюме

Это был краткий обзор возможностей WebSnap и весьма поверхностное перечисление того, на что он способен. Внимательно изучите все демонстрационные приложения в каталоге `<Delphi>\Demos\WebSnap`. Многие из них позволят расширить функциональные возможности стандартных компонентов WebSnap.

Вполне очевидно, что WebSnap – это очень мощная технология, но она требует определенных усилий для изучения. Но если уделить ей достаточно времени и сил, то очень скоро они окупятся многократно, позволив легко и просто создавать мощные дистанционно управляемые базы данных и динамические Web-сайты.

ГЛАВА

24

Разработка приложений беспроводной связи

В ЭТОЙ ГЛАВЕ...

Без сомнения, в прошлом десятилетии две технологии изменили нашу жизнь больше, чем все остальные, это Internet и мобильная связь. Несмотря на взлеты и падения Internet-ориентированного бизнеса, сам Internet (и в частности Web) постепенно вошел нашу жизнь. Это повлияло на то, как мы общаемся, делаем покупки, работаем и играем. Но самое главное, многие из нас теперь дрожат от мысли о том, что окажутся где-нибудь без своего испытанного мобильного телефона или электронной записной книжки (PDA — Personal Digital Assistant). Осознав их значение в повседневной жизни, кажется вполне естественным, что теперь эти две технологии объединяются с устройствами мобильной связи, протягивающими свои беспроводные щупальца к модемам Internet, чтобы обеспечить доступ к информации, без которой нам уже трудно обходиться.

По мере того, как мобильные средства связи становились не роскошью, а средством общения, перед разработчиками возникали новые задачи по созданию программного обеспечения, способного манипулировать этими аппаратными средствами и их инфраструктурой, чтобы удовлетворять постоянно растущий спрос на передачу данных. Кроме того, понадобились приложения, работающие с устройствами мобильной связи. С появлением на рынке десятков типов устройств мобильной связи, сетей и технологий основными вопросами для разработчиков станут такие: какую именно из беспроводных платформ выбрать; какой способ связи наиболее эффективен; какие технологии можно применить для использования мобильной связи для доступа к данным и приложениям; как обеспечить взаимодействие между всеми этими платформами и технологиями?

Настоящая глава ни в коем случае не является исчерпывающим практическим руководством по реализации всех технологий мобильной связи. Это потребовало бы нескольких томов. Но авторы будут считать свою задачу выполненной, если из данной главы читатель уяснит две вещи. Во-первых, настоящую главу можно рассматривать как каталог различных типов аппаратных средств, программного обеспечения и технологий, играющих определенную роль в понимании перспектив программирования в области мобильной связи. Во-вторых, разобраться в том, как некоторые из этих технологий мобильной связи можно реализовать с помощью Delphi.

Эволюция разработки: как это было

Прежде чем перейти к обсуждению самостоятельной разработки приложений, рассмотрим историю возникновения этой информационной технологии, ее становления и тенденций развития. Иногда следует оглянуться назад, чтобы увидеть грядущее. Здесь приведено весьма упрощенное описание состояния и тенденций новейших информационных технологий.

До восьмидесятых: сначала были динозавры

До революции PC в 80-х годах, которая принесла информационные технологии в массы, разрабатывать программное обеспечение приходилось для самых разнообразных систем: мейнфреймов, терминалов и уникальных систем, никак не совместимых друг с другом. Инструментальные средства разработки были крайне примитивны, что делало создание приложений занятием дорогим, трудоемким и требующим высочайшей квалификации.

Восьмидесятые: настольные приложения баз данных

После того как революция PC произошла, люди начали использовать новые возможности, которые появились на их рабочих местах, — настольные приложения баз данных типа dBASE, FoxPro и Paradox. Общие инструментальные средства разработки приложений также достигли зрелого возраста и их количество значительно увеличилось, что сделало разработку приложений относительно простой задачей. Наиболее популярными языками этого поколения были C, Pascal и BASIC. Королем персональных компьютеров был DOS, обеспечивающий приложения общей операционной платформой. В коммерческих предприятиях всех уровней локальные сети становились реальностью, обеспечивая централизованное хранение данных на файловых серверах.

Начало девяностых: архитектура клиент/сервер

Корпоративные сети считаются вещью само собой разумеющейся. Более того, сеть охватывает все удаленные филиалы. Основной задачей стала организация канала связи между устаревшими системами мейнфреймов и немасштабируемыми настольными базами данных, которые все еще были важны для бизнеса. Ответом стала архитектура клиент/сервер, а также концепция мощных баз данных от таких компаний, как *Oracle*, *Sybase* и *Informix*, совместимая с пользовательским интерфейсом PC. Эти системы позволили с любого рабочего места манипулировать всей мощью доступных серверов баз данных и выполнять любые специализированные задачи. Инструментальные средства разработки четвертого поколения, такие как Visual Basic и Delphi, упростили создание приложений больше, чем когда-либо прежде, а встроенная поддержка баз данных стала неотъемлемым элементом этих систем.

Девяностые: многоуровневые, Internet-ориентированные транзакции

Главной проблемой архитектуры клиент/сервер был выбор места расположения бизнес-логики: поместите ее на сервер баз данных — и в результате получите ограничение масштабируемости; разместите ее на клиенте — и сопровождение приложения превратится в настоящий кошмар. Эту проблему решили многоуровневые системы, способные размещать бизнес-логику на одном и более дополнительных уровнях, логически и/или физически отделенных от клиента и сервера. Это позволило создавать системы, обладающие на самом деле практически неограниченной масштабируемостью, и проложило путь к осуществлению сложных транзакций, которые могли обслуживать тысячи или даже миллионы клиентов через Internet. Инструментальные средства разработки распространились и на многоуровневый мир с такими технологиями, как CORBA, EJB и COM. Предприниматели быстро применили Internet для предоставления информации и различных услуг своим служащим, клиентам и партнерам. Это превратилось в настоящую индустрию, возникшую (и продолжающую бы-

стро расти) на базе возможности публиковать, манипулировать и обмениваться данными между машинами с помощью Internet.

Начало 2000-х: инфраструктура приложений простирается до устройств мобильной связи

Итак, каков итог безграничной доступности информации, обеспеченной Internet? Ответ очень прост: информационная зависимость. Доступность услуг и информации через Internet сделала нас зависимыми от нее во все больших аспектах повседневной жизни. Электронные записные книжки и мобильные телефоны частично удовлетворили информационный голод человечества, но им весьма далеко до компьютеров. Серверы приложений и инструментальные средства разработки, обладающие неограниченными возможностями в области управления передачей информации, обратились и к этим типам устройств. Потенциальный рынок для приложений устройств мобильной связи неудержимо растет, поскольку ожидаемое количество продаж этих устройств за несколько следующих лет затмевает количество всех персональных компьютеров, проданных до сих пор.

Мобильные беспроводные устройства

Между мобильными телефонами, PDA и интеллектуальными пейджерами нет никаких промежуточных устройств, способных объединить их функции и составить им конкуренцию. В то же время всем им далеко до компьютера. Тем, кому трудно отдать предпочтение одному средству связи, приходится все их носить на себе, становясь все более и более похожим на Бэтмэна. Но в недалеком будущем все эти устройства, вероятно, будут объединены в одно многофункциональное устройство. Подтверждением тому стали последние тенденции: мобильный телефон содержит теперь возможность приема и передачи простых текстовых сообщений, кроме того, мобильный телефон Kyocera Smartphone совместим с PalmOS, а Stinger с Microsoft Windows CE. В настоящем разделе рассматриваются некоторые лидеры в этой области.

Мобильные телефоны

Мобильные телефоны — это наиболее распространенное устройство мобильной связи. В настоящий момент мобильные телефоны из средств речевой связи перешли в область передачи данных. Большинство новых телефонов, выпущенных в продажу, обеспечивают обмен текстовыми сообщениями с помощью *службы коротких сообщений* (SMS — Short Message Service), а также просмотр специальных Web-страниц с помощью *протокола беспроводных приложений* (WAP — Wireless Application Protocol). Нынешние скорости передачи данных довольно низки (от 9,6 до 14,4 Кбит), но новые технологии обещают скорость до 2 Мбит на протяжении ближайших 2–3 лет.

Устройства PalmOS

Устройства, использующие операционную систему PalmOS от *Palm Computing*, лидируют на рынке PDA в течение нескольких последних лет. Некоторые из устройств

PalmOS обладают встроенной системой беспроводной связи (например серия Palm VII или Kyocera Smartphone), а другие могут быть дополнительно укомплектованы беспроводным модемом (например таким, как Novatel) или разъемом для мобильного телефона, совместимым с Palm. Некоторые крупные компании, в том числе *Handspring*, *Sony*, *Kyocera*, *Symbol*, *Nokia*, *Samsung* и *TRG*, купили у *Palm, Inc.* лицензию на использование операционной системы PalmOS в своих собственных устройствах. Преимуществом PalmOS является тот факт, что она установлена на подавляющем большинстве PDA, продаваемых в данный момент. Кроме того, с ней уже начало работать большое количество разработчиков и сторонних производителей.

Pocket PC

Compaq, *HP*, *Casio* и другие изготовители выпускают PDA на базе операционной системы *Microsoft* Pocket PC (ранее Windows CE). До сих пор ни одно из этих устройств не имеет встроенной мобильной связи, но они комплектуются беспроводными модемами подобно устройствам PalmOS. По сравнению со своим конкурентом PalmOS, компьютеры Pocket PC обещают быть немного более мощными, а также совместимыми со стандартом PC Cards (PCMCIA). Потенциально это обещает диапазон применения даже шире, чем у беспроводных сетей, обладающих более высокой пропускной способностью.

RIM BlackBerry

BlackBerry обеспечивает функциональные возможности, аналогичные PDA, но его размер и форма ближе к пейджеру. Укомплектованный внутренним беспроводным модемом и клавиатурой type-with-your-thumbs (для большего пальца), BlackBerry особенно хорошо подходит для задач мобильной электронной почты. Однако BlackBerry способен обеспечить просмотр Web через броузер стороннего производителя. Изучив BlackBerry, автор пришел к заключению, что благодаря встроенной интеграции с MS Exchange и Lotus Domino это устройство способно стать общей платформой для мобильной электронной почты, благодаря размеру его экрана и возможности мобильной связи он применим и для Web.

Технологии радиосвязи

Технологии радиосвязи обеспечивают соединение между устройствами мобильной связи, Internet и корпоративными локальными сетями.

GSM, CDMA и TDMA

Это первые технологии, применявшиеся как транспорт для мобильных телефонов. Их часто называют *2G*, поскольку они представляют собой второе поколение мобильных систем связи (*1G* был аналоговой службой). Большинство сетей в Соединенных Штатах построено на основании CDMA и TDMA, в то время как в остальной части мира ориентируются в основном на GSM. С точки зрения разработчика программного обеспечения, различия между этими технологиями незначительны, необходимо лишь знать наиболее существенные особенности этих стандартов, затрудняющие создание приложений или конфликтующие со стандартными телефона-

ми и сетями. В этих типах сетей скорость передачи данных не превышает 9,6–14,4 Кбит.

CDPD

Сотовая цифровая передача пакетов данных (CDPD — Cellular Digital Packet Data) — это технология, обеспечивающая пакет-ориентированную передачу данных по беспроводными сетям, позволяющая увеличить пропускную способность и предоставить функциональные возможности типа "Всегда готов!" ("always on"). Обычно в Соединенных Штатах CDPD применяется такими не коммерческими службами беспроводной связи PDA, как *GoAmerica* и *OmniSky*, их скорость достигает примерно 19.2 Кбит.

3G

3G, или третье поколение мобильных сетей, разработано на основе ряда других типов передачи информации и отличается достаточно высокой пропускной способностью (от 384 Кбит до 2 Мбит). Наиболее известными технологиями стандартов 3G являются EDGE и UMTS. Но, несмотря на существование этих технологий, операторы мобильной связи не очень торопятся реализовать сети3G, ни один из операторов не хочет быть первым, поскольку внедрение новой сетевой технологии потребует многомиллиардных затрат.

GPRS

Всеобщая служба передачи пакетов по радио (GPRS — General Packet Radio Service) рассматривает пути перехода на скорости от 2 Гбит на 3 Гбит, поэтому иногда говорят, что они обеспечивают *2.5* Гбит. GPRS осуществляет пакет-ориентированную передачу информации со скоростью 2 Гбит лишь в пределах специальной инфраструктуры, а она не так уж и велика. Обычная производительность в сетях GPRS составляет около 20–30 Кбит.

Bluetooth

Устройства, использующие технологию радиосвязи Bluetooth, уже доступны в рынке. Новая технология Bluetooth очень важна, потому, что работая на небольших расстояниях, позволяет создать сеть из устройств различных типов. Поскольку передатчики Bluetooth очень малы, обладают малой мощностью и относительно недороги, их можно использовать в любых устройствах мобильной связи, включая телефоны, PDA, ноутбуки и так далее. Большинство радиопередатчиков Bluetooth будет работать в примерно 10-метровом радиусе с относительно высокой пропускной способностью — 700 Кбит. Возможными приложениями для Bluetooth могут быть приложения синхронизации данных между PDA и компьютером, когда они находятся на расстоянии прямой связи или программное обеспечение ноутбука с подключением к Internet через мобильный телефон. Новый термин, *персональная локальная сеть* (PAN — Personal Area Network), используется для описания именно такой небольшей беспроводной сети, где все персональные устройства мобильной связи регулярно связываются друг с другом.

Вероятнее всего, Bluetooth заменит кабели последовательного порта, USB и IEEE 1394, подобно сетевой технологии Ethernet. В настоящий момент Bluetooth представляет собой только одно главное устройство, обслуживающее до семи одновременно работающих периферийных устройств.

802.11

В отличие от технологии Bluetooth, разработанной для поддержания беспроводной персональной сети малого радиуса действия, технология 802.11 предназначена для создания полнофункциональных локальных сетей. Текущее поколение этой технологии (802.11b или *WiFi*), обеспечивает пропускную способность до 11 Мбит, а версия известна как 802.11a до 45 Мбит. Радиус действия системы 802.11 составляет примерно 30 метров, для более дальней связи нужна специальная антенна. Системе 802.11 необходимо больше мощности, чем Bluetooth, и устройства ее большие по размеру. Радиопередатчик 802.11 можно установить на материнской плате стандартного PC, для ноутбуков он тоже может подойти, но для мобильных телефонов и большинства PDA он неприменим.

Одно немаловажное замечание: имейте в виду, что и Bluetooth, и 802.11 используют одинаковый диапазон радиочастот — 2.4 GHz, поэтому при размещении их в пределах одного пространства возможен конфликт. Хотя это и маловероятно, но полностью исключить возможность наведения помех друг на друга из-за факта совпадения частот обоих технологий нельзя.

Серверные технологии беспроводной передачи данных

Беспроводные системы используют, в основном, передовые технологии радиосвязи для передачи данных и обеспечения работоспособности устройств мобильной связи. Эти системы включают серверы, создающие содержимое, которое может быть передано мобильному клиенту и интерпретировано встроенным программным обеспечением.

SMS

Служба коротких сообщений (SMS) — это технология, используемая для передачи коротких (от 100 до 160 символов максимум) текстовых сообщений мобильным телефонам. Кроме ограничения длины сообщений, технология SMS имеет проблему совместимости между разными операторами мобильной связи, использующими различные протоколы передачи сообщений. Но SMS стал очень популярен в Европе из-за легкости в эксплуатации и широкой доступности.

Поскольку каждый оператор мобильной связи может использовать разные варианты SMS, методы для разработки приложений, поддерживающих SMS, также могут варьироваться в зависимости от конкретного оператора мобильной связи. Хотя GSM имеет преимущество перед другими сетями мобильных телефонов, с точки зрения разработчика приложений, поддержка SMS встроена в стандарт GSM. Конечно, это сомнительное удовольствие — посылать сообщения SMS мобильному клиенту с сервера, подключенного к Internet. Указанное связано с тем, что работать придется с сер-

верами SMS, расположенными на стороне оператора мобильной связи, который может вносить изменения не только в поддерживаемые стандарты, но и в лицензионные технологии.

Рекомендуем одно из двух направлений унификации поддержки серверов SMS. Первое подразумевает использование электронной почты; большинство операторов мобильной связи обеспечивает передачу сообщений SMS через электронную почту по определенному адресу, который содержит номер телефона получателя. Хоть это относительно простой подход с технической точки зрения, но он не универсален, а кроме того, дополнительный уровень повышает вероятность ошибок. Второй подход заключается в поддержке всего разнообразия инструментальных средств стороннего производителя, которые обеспечивают пересылку сообщений SMS в необходимых сетях. Эта технология предпочтительна, хоть она немного сложнее и дороже.

WAP

Протокол беспроводных приложений (WAP — Wireless Application Protocol) был принят как стандарт средств доступа к информации Internet через устройства мобильной связи. Стандарт WAP был утвержден на рынке по ряду причин. С одной стороны, его одобрили операторы мобильной связи и производители телефонов, поскольку он был изначально создан для работы с любыми беспроводными службами и сетевыми стандартами фактически на любых устройствах. Но опыт пользования WAP не был до конца положительным из-за ограничения размера экрана, возможности ввода данных и низкой пропускной способности мобильных устройств. Кроме того, поскольку WAP-сайты мало посещаемы и не приносят существенной коммерческой прибыли, нет и объективного стимула для разработки высококачественных WAP-сайтов. Содержимое для систем WAP разрабатывают на XML-ориентированном языке, известном как *язык беспроводной разметки* (WML — Wireless Markup Language).

Типичная архитектура приложения WAP представлена на рис. 24.1.

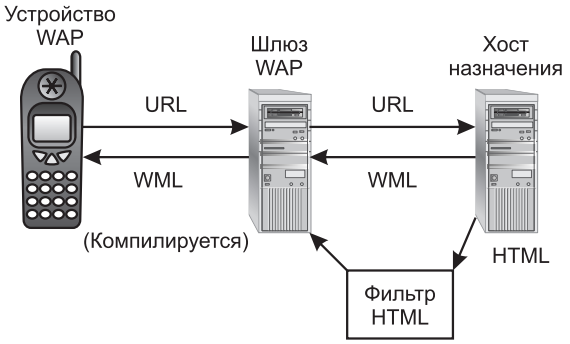

Рис. 24.1. *Архитектура приложения WAP*

Устройство мобильной связи, обычно телефон, обладает встроенным программным обеспечением, известным как *микроброузер* (micro-browser). Как и следует из названия, данный элемент программного обеспечения подобен Web-броузеру, но разработан для устройств типа мобильных телефонов с ограниченной памятью и возможностью обработки. Большинство мобильных телефонов, представленных на рынке в

настоящий момент, использует микроброузер OpenWave (прежде Phone.com). Кроме того, микроброузеры обычно разрабатывают так, чтобы они могли работать с языками WML или HDML, а не с HTML, как описано в следующем разделе.

Поскольку текущее поколение мобильных телефонов неспособно самостоятельно связаться с ресурсами Internet, шлюз WAP выступает в качестве посредника между устройством мобильной связи и обычным Internet. Большинство шлюзов WAP контролируется провайдерами услуг беспроводной связи и используют программное обеспечение, созданное такими компаниями как *OpenWave*, *Nokia* или *SAS*.

Хост назначения — это всего лишь старый добрый Web-сервер, который просто возвращает содержимое, но оформленное для WAP. Под оформлением следует понимать, что содержимое написано на языке WML или, что менее предпочтительно, динамически преобразовано из HTML в WML с помощью фильтра.

Главным преимуществом WAP является то, что его поддерживает значительное количество мобильных и беспроводных устройств. Кроме того, WAP, по существу, представляют собой минимально допустимый уровень функциональных возможностей, общий для всех устройств мобильной связи, что означает более широкую совместимость за счет снижения их возможностей. В связи с тем, что между сервером приложений и клиентским устройством находятся Web-сервер, шлюз WAP и микроброузер, разработчики WAP имеют область для приложения усилий по разработке приложений, способных обеспечить правильность функционирования всех этих элементов для большего количества разнообразных конечных пользователей.

К основным недостаткам WAP относят ограниченный размер экрана и низкие возможности процессора, а остальными недостатками является отсутствие полнофункциональной клавиатуры для ввода данных, низкая скорость приема и все еще относительно высокий тариф мобильной связи, ограничивающий время работы с приложениями WAP.

WML: язык WAP

Как уже говорилось, для обмена информацией WAP используется *язык беспроводной разметки* (WML — Wireless Markup Language). WML является неким подобием HTML, но, в отличие от него, обладает двумя дополнительными особенностями. Во-первых, он состоит из относительно небольшего набора дескрипторов и атрибутов, что, обеспечивает большую плотность информации, позволяет эффективнее использовать устройства с небольшой памятью и слабым процессором. Во-вторых, он происходит от *расширяемого языка разметки* (XML — Extensible Markup Language), поэтому его содержимое лучше оформлено и не так доступно для интерпретации броузером как HTML. Настоящая глава не является учебным пособием для начинающих программистов WAP, но некоторое представление об основах этого языка мы приведем.

Как всем, вероятно, известно, каждый файл `.html` представляет собой в окне броузера одну страницу HTML и содержит всю ее информацию. Язык WML, напротив, основан на представлении наборов карточек (card deck), в одном файле `.wml` находится набор, содержащий несколько карточек (card), каждая из которых представляет один экран информации. Таким образом, весь набор карточек WML может быть передан клиенту сразу, в отличие от обычного Web, которому приходится обращаться к серверу несколько раз, чтобы загрузить все страницы документа. Типичный файл `.wml` может выглядеть следующим образом:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.WAPforum.org/DTD/wml_1.1.xml">
<wml>
  <card>
    <do type="accept">
      <go href="#hello"/>
    </do>
    <p>Punch the Button</p>
  </card>
  <card id="hello">
    <p>Hello from WAP!</p>
  </card>
</wml>
```

Те, кто знаком с HTML и XML, вероятно, заметили, что этот код относительной прост. Заголовок документа, занимающий первые строки, стандартен для XML. Он описывает версию языка XML документа и расположение DTD, используемого для описания дескрипторов и атрибутов, содержащихся в нем. При загрузке этот код создаст набор из двух экранов: первый — с кнопкой OK, второй — со строкой приветствия.

Кроме того, синтаксис WML поддерживает такие элементы, как события, таймеры, наборы полей, списки и изображения (хоть изображения поддерживают не все устройства). Ряд новых версий броузеров WAP поддерживает даже язык сценариев WMLScript. В данной книге нет полного описания языка WML, но более подробная информация о спецификации WML содержится по адресу `http://www.WAPforum.org`.

Если хочется потренироваться в разработке кода WML, то для начала желательно раздобыть эмулятор. Эмулятор можно получать у разработчика микроброузера по адресу `http://www.openwave.com`. Два других популярных эмуляторах можно найти непосредственно у лидеров мобильной связи `Nokia` и `Ericsson` по адресам `http://forum.nokia.com` или `http://www.ericsson.com/developerszone`. Применение эмулятора — очень хорошея идея, поскольку позволяет проверить работоспособность кода до перехода к реальным аппаратным средствам. Кроме того, эмуляторы обеспечивают намного более быструю и простую отладку по принципу "написал — проверил", позволяя сэкономить время на установке. Здесь также можно проверить работу финальной версии на разнообразных устройствах, поскольку индивидуальные характеристики этих устройств могут повлиять на внешний вид отображаемых карточек.

Безопасность WAP

Для обеспечения защищенных соединений спецификация WAP использует *беспроводной стек шифрования* (wireless encryption stack), известный как *уровень безопасности беспроводного транспорта* (WTLS — Wireless Transport Layer Security). Поскольку используемый в нынешнем поколении устройств мобильной связи *уровень защищенных сокетов* (SSL — Secure Sockets Layer) также зависит от конкретного ресурса, WTLS был разработан так, чтобы службы обеспечения шифрования и аутентификации располагались между мобильным устройством и шлюзом WAP. Поэтому шлюз WAP способен общаться с хостами Internet через стандартный протокол SSL. Несмотря на то, что и WTLS и SSL сами по себе защищены достаточно серьезно, потенциальной лазейкой

для взлома защиты остается шлюз WAP, где поток данных WTLS дешифруется и вновь шифруется для SSL. Архитектура WTLS приведена на рис. 24.2.

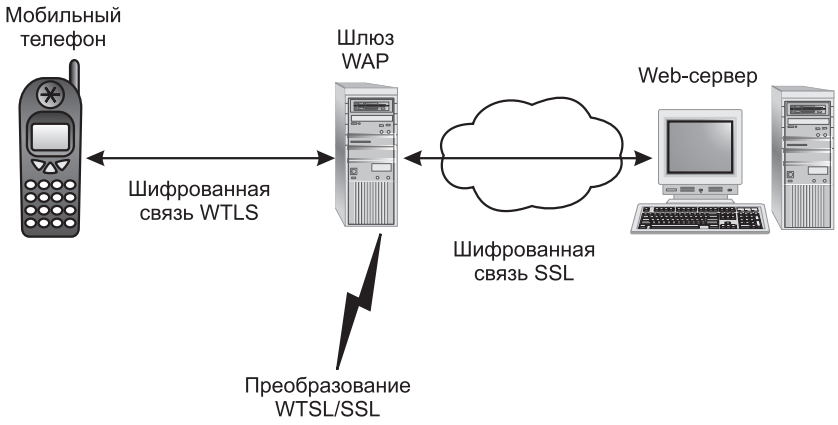

Рис. 24.2. *Уровень безопасности беспроводного транспорта WAP*

Простое приложение WAP

Создание приложения WAP в Delphi немного отличается от создания обычного Web-приложения. Возможно, создание приложения WAP даже проще, поскольку ограничения, свойственные WAP и его устройствам, позволяют создавать более простые серверные приложения, чем те, которые необходимы традиционным броузер-ориентированным приложениям. Но эта медаль имеет и обратную сторону: разработчику сложнее создать приложение, которые должно обеспечить конечным пользователям необходимый уровень сервиса вопреки этим ограничениям.

Для данного примера создадим сначала обычное приложение WebBroker, подобное тому, которое было создано в главе 23, "Разработка приложений WebSnap". Данное приложение имеет один Web-модуль, выполняющий одно действие. Это действие отмечено как установленное по умолчанию (рис. 24.3).

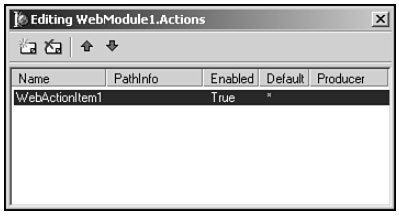

Рис. 24.3. *Простое приложение WebBroker для WAP*

В листинге 24.1 представлен исходный код главного модуля этого приложения, включая обработчик события `OnAction` — действия, заданного для Web-модуля по умолчанию.

Листинг 24.1. `Main.pas` — главный модуль проекта `SimpWap`

```
unit Main;

interface

uses
  SysUtils, Classes, HTTPApp;

type
  TWebModule1 = class(TWebModule)
    procedure WebModule1WebActionItem1Action(Sender: TObject;
                      Request: TWebRequest; Response: TWebResponse;
                      var Handled: Boolean);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  WebModule1: TWebModule1;

implementation

{$R *.DFM}

const
  SWMLContent = 'text/vnd.wap.wml';
  SWMLDeck =
    '<?xml version="1.0"?>'#13#10 +
    '<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"'#13#10 +
    '"http://www.WAPforum.org/DTD/wml_1.1.xml">'#13#10 +
    '<wml>'#13#10 +
    '  <card>'#13#10 +
    '    <do type="accept">'#13#10 +
    '      <go href="#hello"/>'#13#10 +
    '    </do>'#13#10 +
    '    <p>Punch the Button</p>'#13#10 +
    '  </card>'#13#10 +
    '  <card id="hello">'#13#10 +
    '    <p>Hello from WAP!</p>'#13#10 +
    '  </card>'#13#10 +
    '</wml>'#13#10;

procedure TWebModule1.WebModule1WebActionItem1Action(Sender:
                    TObject; Request: TWebRequest;
                    Response: TWebResponse; var Handled: Boolean);
begin
  Response.ContentType := SWMLContent;
  Response.Content := SWMLDeck;
end;

end.
```

При вызове действия срабатывает обработчик события `OnAction`, заполняя свойства `ContentType` и `Content` объекта `Response`. Свойство `ContentType` получает строковое значение, устанавливающее тип содержимого WML, а возвращено будет содержимое той же самой простой карточки WAP, которая уже рассматривалась ранее в настоящей главе.

НА ЗАМЕТКУ

Не забудьте присвоить свойству `ContentType` объекта `Response` строку, содержащую тип MIME возвращаемого содержимого. Она установит информацию о типе в заголовке HTTP. Если тип будет возвращен (а следовательно и установлен) неправильно, то содержимое будет, вероятно, искажено на устройстве назначения. Наиболее распространены следующие типы WAP:

- `text/vnd.wap.wml` — для кода WML.
- `text/vnd.wap.wmlscript` — для кода сценариев WML.
- `image/vnd.wap.wbmp` — для беспроводных растровых изображений.

На рис. 24.4 это простое приложение WAP представлено в действии.

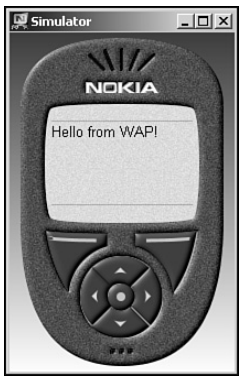

Рис. 24.4. *Приложение WAP в действии*

Сообщения об ошибках

Стандартный обработчик исключений для приложений WebSnap передает клиенту сообщение HTML, содержащие информацию об ошибке. Конечно, большинство устройств WAP будет неспособно понять сообщение в формате HTML, поэтому следует удостовериться, что информация о любых ошибках, которые могут произойти в приложении WAP, будут переданы клиенту в виде WML, а не HTML. Это можно сделать, заключив каждый обработчик события `OnAction` в блок `try..except`, который будет содержать функцию, предающую сообщению об ошибках соответствующий формат. Это показано в листинге 24.2.

Растровые изображения

Хоть WAP и не поддерживает изображений JPEG и GIF, принятых в Web, большинство устройств WAP поддерживает монохромные изображения в форме беспроводных растров (`.wbmp`). В листинге 24.2 Web-модуль содержит новую возможность — создание `wbmp`. Специальное действие генерирует солидного вида, но совершенно

случайный график для представления на дисплее мобильного устройства. Не будем вдаваться в подробности бинарного формата файлов .wbmp, но, как очевидно из кода, его создание не вызывает затруднений. Рис. 24.5 демонстрирует, как будет выглядеть график WBMP на дисплее мобильного телефона.

НА ЗАМЕТКУ

Не все WAP-броузеры, устройства и эмуляторы поддерживают изображения .wbmp. Удостоверьтесь, что используемое устройство обладает этой возможностью.

Листинг 24.2. `Main.pas`. Еще раз и с чувством

```
unit Main;

interface

uses
  SysUtils, Classes, HTTPApp;

type
  TWebModule1 = class(TWebModule)
    procedure WebModule1WebActionItem1Action(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
    procedure WebModule1GraphActionAction(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
  private
    procedure CreateWirelessBitmap(MemStrm: TMemoryStream);
    procedure HandleException(e: Exception;
                              Response: TWebResponse);
  end;

var
  WebModule1: TWebModule1;

implementation

{$R *.DFM}
const
  SWMLContent = 'text/vnd.wap.wml';
  SWBMPContent = 'image/vnd.wap.wbmp';
  SWMLDeck =
    '<?xml version="1.0"?>'#13#10 +
    '<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"'#13#10 +
    '"http://www.WAPforum.org/DTD/wml_1.1.xml">'#13#10 +
    '<wml>'#13#10 +
    '  <card>'#13#10 +
    '    <do type="accept">'#13#10 +
    '      <go href="#hello"/>'#13#10 +
    '    </do>'#13#10 +
    '    <p>Punch the Button</p>'#13#10 +
```

```
    '  </card>'#13#10 +
    '  <card id="hello">'#13#10 +
    '    <p>Hello from WAP!</p>'#13#10 +
    '  </card>'#13#10 +
    '</wml>'#13#10;

  SWMLError =
    '<?xml version="1.0"?>'#13#10 +
    '<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"'#13#10 +
    '"http://www.wapforum.org/DTD/wml_1.1.xml">'#13#10 +
    '<wml>'#13#10 +
    '  <card id="error" title="SimpWAP">'#13#10 +
    '    <p>Error: %s'#13#10 +
    '      <do type="prev" label="Back">'#13#10 +
    '        <prev/>'#13#10 +
    '      </do>'#13#10 +
    '    </p>'#13#10 +
    '  </card>'#13#10 +
    '</wml>'#13#10;

procedure TWebModule1.HandleException(e: Exception;
                                      Response: TWebResponse);
begin
  Response.ContentType := SWMLContent;
  Response.Content := Format(SWMLError, [e.Message]);
end;

procedure TWebModule1.WebModule1WebActionItem1Action(Sender:
                    TObject; Request: TWebRequest;
                    Response: TWebResponse; var Handled: Boolean);
begin
  try
    Response.ContentType := SWMLContent;
    Response.Content := SWMLDeck;
  except
    on e: Exception do
      HandleException(e, Response);
  end;
end;

procedure TWebModule1.WebModule1GraphActionAction(Sender: TObject;
                     Request: TWebRequest; Response: TWebResponse;
                     var Handled: Boolean);
var
  MemStream: TMemoryStream;
begin
  try
    MemStream := TMemoryStream.Create;
    try
      CreateWirelessBitmap(MemStream);
      MemStream.Position := 0;
      with Response do begin
        ContentType := SWBMPContent;
```

```
        ContentStream := MemStream;
        SendResponse;
      end;
    finally
      MemStream.Free;
    end;
  except
    on e: Exception do
      HandleException(e, Response);
  end;
end;

procedure TWebModule1.CreateWirelessBitmap(MemStrm:
                                                TMemoryStream);
const
  Header : Array[0..3] of Char = #0#0#104#20;
var
  Bmp: array[1..104,1..20] of Boolean;
  X, Y, Dir, Bit: Integer;
  B: Byte;
begin
  { очистить изображение }
  FillChar(Bmp,SizeOf(Bmp),0);
  { прорисовка осей X и Y }
  for X := 1 to 104 do Bmp[X, 20] := True;
  for Y := 1 to 20 do Bmp[1, Y] := True;
  { прорисовка случайных данных }
  Y := Random(20) + 1;
  Dir := Random(10);
  for X := 1 to 104 do begin
    Bmp[X,Y] := True;
    if (Dir > 4) then Y := Y+Random(2)+1
    else Y := Y - Random(2) - 1;
    if (Y > 20) then Y := 20;
    if (Y < 1) then Y := 1;
    Dir := Random(10);
  end;
  { создать данные WBMP }
  MemStrm.Write(Header, SizeOf(Header));
  Bit := 7;
  B := 0;
  for Y := 1 to 20 do begin
    for X := 1 to 104 do begin
      if Bmp[X,Y] = True then
        B := B or (1 shl Bit);
      Dec(Bit);
      if (Bit < 0) then begin
        B := not B;
        MemStrm.Write(B, SizeOf(B));
        Bit := 7;
        B := 0;
      end;
    end;
```

```
  end;
end;

initialization
  Randomize;
end.
```

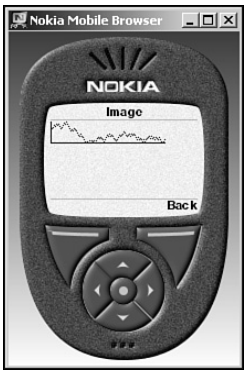

Рис. *24.5. Просмотр изображения WBMP в эмуляторе Nokia*

I-mode

I-mode (режим I) представляет собой частную технологию передачи содержимого Internet на мобильные телефоны, разработанные компанией *NTT DoCoMo* для Японии. В Японии i-mode очень популярен, количество абонентов превысило 20 миллионов и продолжает увеличиваться. I-mode существенно превосходит WAP: он обеспечивает богатую 256-цветную графику, цветные дисплеи телефонов и использует постоянное соединение TCP/IP. Кроме того, *DoCoMo* разработал специальную модель для применения в финансовой сфере, что позволяет теперь сайтам i-mode предоставлять целый сектор бизнес-услуг. Но то, что эта технология является частной, затрудняет ее распространение за пределы Японии. В ближайшие годы услуги i-mode, вероятно, появятся в Соединенных Штатах, Великобритании и континентальной Европе.

С точки зрения разработчика, программирование под телефоны i-mode ненамного более сложная задача, чем разработка содержимого Web, поскольку содержимое i-mode представляет собой разновидность HTML, известную как *компактный HTML* (cHTML – Compact HTML). Дескрипторы и синтаксис cHTML доступны на официальном сайте *DoCoMo* по адресу `http://www.nttdocomo.com/i/tagindex.html`. Обратите внимание, что в дополнение к использованию дескрипторов сугубо cHTML сайты i-mode должны обязательно поддерживать национальный набор символов S-JIS, использовать изображения в формате GIF и не должны содержать ни сценариев, ни кода Java.

PQA

Приложения запросов Palm (PQA – Palm Query Applications) представляют собой, по существу, обычные страницы HTML, чередующиеся с такими дополнениями, как сценарии и изображения. Они предназначены для представления на экране беспровод-

ного устройства PalmOS. Область применения беспроводных устройств PalmOS, типа Palm VIIx, а также устройств, оборудованных модемами Novatel, в настоящее время ограничена Северной Америкой. Подобно i-mode, PQA разработан на основании HTML и отличается от него рядом собственных дополнений. Разработчики, заинтересовавшиеся созданием PQA, могут загрузить соответствующую документацию (*Web Clipping Developer's Guide*) непосредственно с Web-сайта *Palm* по адресу `http://www.palmos.com/dev/tech/docs/`.

В принципе, PQA, как разновидность HTML, включает в себя все дескрипторы HTML 3.2, за исключением аплетов, JavaScript, вложенных таблиц, карт изображения (map), а также дескрипторов `VSPACE`, `SUB`, `SUP`, `LINK` и `ISINDEX`. PQA включает также несколько интересных дополнений к HTML. Из них наиболее известны метадескриптор `palmcomputingplatform`, а также дескрипторы `%zipcode` и `%deviceid`. Если документ HTML содержит метадескриптор `palmcomputingplatform`, то он обрабатывается прокси-сервером Palm.net производства *Palm Computing* (который выступает посредником между Web-сервером и устройством Palm, мало чем отличаясь от шлюза WAP). Кроме того, он означает, что документ оптимизирован для представления на портативном устройстве PalmOS и не требует дополнительной обработки по удалению недопустимого содержимого. Когда в запросе, посланном клиентом встречается дескриптор `%zipcode`, прокси-сервер Palm.net заменяет его дескриптором зип-кода (ZIP Code) места расположения устройства (причем делается это на основании информации о конкретной радиомачте, принявшей сигнал устройства). Дескриптор `%deviceid` аналогичен передаче серверу индивидуального идентификатора устройства PalmOS. Это очень удобно, поскольку PQA HTML не поддерживает cookies, зато аналогичное средство хранения состояния может быть реализовано с помощью дескриптора `%deviceid`.

Для преобразования Web, *Palm* применяет подходы, отличные от большинства других разработчиков. Вместо того, чтобы использовать нечто похожее на броузер для навигации по сайту и загрузки содержимого, PQA существует на устройстве PalmOS локально. PQA разработан так, чтобы загружать стандартный файл `.HTML` через специальный компилятор, который компонует HTML со ссылками на графические и другие сопутствующие файлы. Пользователи устанавливают PQA подобно обычному приложению PalmOS PRC. Высокой эффективности PQA добивается за счет частной установки приложения на локальное устройство и обращения в сеть только за необходимыми участками результирующих страниц.

Клиент PQA

Первым этапом разработки приложения PQA является создание фрагмента, который будет физически расположен на клиентском устройстве. Для этого достаточно разработать документ HTML и скомпилировать его с помощью утилиты PQA Builder tool производства *Palm Computing*. Пример документа HTML для PQA представлен в листинге 24.3.

Листинг 24.3. Документ HTML для PQA

```
<html>
<head>
<title>DDG PQA Test</title>
<meta name="palmcomputingplatform" content="true">
```

```
</head>
<body>
<p>This is a sample PQA for DDG</p>
<img src="image.gif">
<form method="post"
  action="http://128.64.162.164/scripts/pqatest.dll">
<input type="hidden" value="%zipcode" name="zip">
<input type="hidden" value="%deviceid" name="id">
<input type="submit">
</form>
</body>
```

Как можно заметить, этот простой документ HTML содержит ссылку на изображение, текст, форму с кнопкой Submit и скрытые поля, используемые для передачи на сервер зип-кода и уникального идентификатора устройства.

На рис. 24.6 представлен момент загрузки этого документа в компилятор PQA Builder.

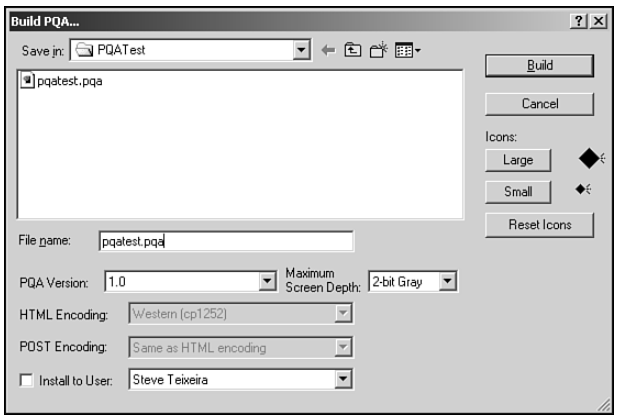

Рис. 24.6. *Компиляция с помощью PQA Builder*

В результате компиляции будет создан файл с расширением `.pqa`. Он будет содержать как сам документ HTML, так и все необходимые ему изображения. Данный файл может быть загружен устройством PalmOS, равно как и любым приложением PalmOS.

Сервер PQA

Серверная часть, аналогично WAP, является приложением WebSnap, которое обрабатывает запросы, возвращающиеся со страниц клиента. В отличие от WAP, использующего WML, PQA применяет для взаимодействия с сервером разновидность HTML, описанную ранее. Листинг 24.4 содержит главный модуль приложения WebBroker, выполняющего роль сервера для описанного ранее клиента PQA.

Листинг 24.4. `Main.pas` — главный модуль приложения `PQATest`

```
unit Main;

interface
```

```
uses
  SysUtils, Classes, HTTPApp;

type
  TWebModule1 = class(TWebModule)
    procedure WebModule1WebActionItem1Action(Sender: TObject;
                Request: TWebRequest; Response: TWebResponse;
                var Handled: Boolean);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  WebModule1: TWebModule1;

implementation

{$R *.DFM}

const
  SPQAResp =
    '<html><head><meta name="palmcomputingplatform" '+
    'content="true"></head>' #13#10 +
    '<body>Hello from a Delphi server<br>Your zipcode is: '+
    '%s<br>'#13#10 +
    'Your device ID is: %s<br><img '+
    'src="file:pqatest.pqa/image.gif"></body>'+
    '</html>';

procedure TWebModule1.WebModule1WebActionItem1Action(Sender:
                  TObject; Request: TWebRequest;
                  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := Format(SPQAResp,
                      [Request.ContentFields.Values['zip'],
                      Request.ContentFields.Values['id']]);
end;

end.
```

В ответ на запрос клиента сервер посылает зип-код и идентификатор клиента. Обратите внимание, что возвращаемый сервером код HTML ссылается на то же самое изображение, которое было скомпилировано в файл `.pqa` на клиентской стороне. Для этого используется синтаксис типа *`file:<pqaname>`*, что позволяет применять в PQA достаточно много графики, которая в скомпилированном виде хранится на клиенте, а обращение к ней осуществляется через сервер. Таким образом устраняется необходимость загружать по беспроводным модемам *всю* графику. Рис. 24.7 и 24.8 демонстрируют

это приложение в действии, до и после щелчка на кнопке Submit. Обратите внимание: в эмуляторе зип-код и идентификатор клиента имеют нулевые значения.

Рис. 24.7. *Приложение* `PQATest` *в эмуляторе PalmOS*

Рис. 24.8. *Приложение* `PQATest` *в эмуляторе PalmOS после щелчка на кнопке* Submit

Квалификация пользователя

В конечном счете именно квалификация пользователей окажется наиболее важный фактором, определяющим популярность той или иной мобильной системы. Кстати, популярностью у пользователей зачастую пользуются бесплатные возможности и технологии. Из-за свойственных мобильному миру ограничений, особенно в отношении возможности связи и размера устройства, здесь нет места ошибкам разработчика в попытке обеспечить пользователей необходимыми функциональными возможностями (если они на самом деле необходимы). Тут основное внимание следует уделять пользователю: установите, какая именно информация или услуга ему нужна, и попытайтесь реализовать ее настолько эффективно, насколько это возможно.

Сети с коммутацией каналов против сетей с коммутации пакетов

Рассматривая мобильный телефон как клиентскую платформу, можно заметить одну серьезную проблему, которая может иметь самые драматические последствия. Речь идет о применении сетей мобильных телефонов с *коммутацией каналов* (Circuit-Switched) или *коммутацией пакетов* (Packet-Switched). Сети с коммутацией каналов функционируют подобно модему с обычным телефонным каналом связи: сначала необходимо установить прямое соединение с хостом, а затем поддерживать его в течение всего времени обмена информацией. Сети с коммутацией пакетов ведут себя аналогично постоянному соединению локальной сети Ethernet: соединение всегда активно, и пакеты данных могут быть посланы и получены в любое время.

Неизбежная дополнительная трата времени на установление соединения в сетях с коммутацией каналов может стать главным препятствием к удовлетворению потреб-

ностей пользователей, особенно, если выполняемая задача рассчитана всего на несколько секунд. В конце концов, кто хочет ждать до 20 секунд, чтобы, соединившись, взаимодействовать с приложением в течение 3 секунд? Большинство современных мобильных сетей все еще используют коммутацию каналов, но уже появились сети, использующие технологию коммутации пакетов. К ним относятся сети i-mode от *NTT DoCoMo*, iDEN от *Nextel* и CDPD от *AT&T*.

Беспроводные сети — это не Web

Общепринятым заблуждением поставщиков устройств мобильной связи и беспроводных сетей является то, что они полагают, будто бы пользователи горят нетерпением заняться Web-серфингом на этих устройствах. С их крошечными экранами и ограниченной пропускной способностью устройства мобильной связи — чрезвычайно неудобное транспортное средство для Web-серфинга. Загрузка каждой страницы информации может занять несколько секунд, и это в условиях ограничений мобильной связи, а ввод данных может стать просто кошмаром, особенно на мобильном телефоне. Поэтому приложения для мобильной связи должны быть ориентированы на предоставление специфической информации и услуг, требующих минимального ввода данных со стороны пользователя, а также как можно меньшего количества обращений к серверу.

Серьезность фактора форм

При проектировании приложений мобильных устройств необходимо всегда помнить о том, что на экране оно должно выглядеть эстетично. Будь то приложение WAP, создаваемое для встроенного микроброузера, или специальное приложение для пользовательского интерфейса J2ME, разработчики вынуждены балансировать на грани проблем, связанных с достаточностью количества информации на любом экране, и сохранением читабельного и легко управляемого пользовательского интерфейса.

Ввод данных и методы навигации

С фактором форм связана и проблема ввода данных. Различные типы устройств ориентируются на разные механизмы ввода данных. Устройства PalmOS и ноутбуки используют комбинацию курсора и символьной клавиатуры; устройства BlackBerry имеют ограниченную клавиатуру; а мобильные телефоны вообще имеют только цифровую клавиатуру и несколько дополнительных кнопок. Это означает, что приложения, разработанные для одной мобильной платформы, могут оказаться неприемлемы для другой. Например, курсор PDA удобен для навигации по разным областям экрана, в то время как для BlackBerry лучше подходит ввод текстовых данных, а телефон лучше всего приспособлен для ввода коротких цифровых данных.

Мобильная коммерция

Подобно тому, как *электронная коммерция* (e-commerce) применяется при осуществлении коммерческих операций средствами электронной связи и Web, *мобильная коммерция* (m-commerce) осуществляет коммерческие операций средствами мобильной связи. Разработчики сайтов мобильной торговли должны понять, что мобильная

коммерция существенно отличается от электронной. Наиболее существенным отличием является то, что мобильному заказчику трудно перемещаться по спискам товаров. В случае мобильного телефона, например, слишком много времени отнимает ввод команд на клавиатуре, изображение товара отсутствует или имеет слишком низкое качество, а для подробного описания недостаточно места на экране.

Поэтому системы мобильной коммерции должны быть разработаны с учетом того, что пользователь знает, что они хочет купить; вопрос только в том, чтобы помочь ему расстаться с деньгами. Помните: если бы пользователь захотел купить телевизор или книгу, то нет никаких причин, по которым он не может ждать, пока доберется до магазина или полнофункционального компьютера, чтобы сделать покупку. Сам факт, что пользователь вынужден прибегнуть к услугам мобильной коммерции, свидетельствует о крайней важности и срочности покупки, и победят те поставщики от мобильной коммерции, которые смогут осознать и использовать этот фактор.

Например, одна восточноевропейская страна позволяет автомобилистам оплачивать время парковки с помощью мобильного телефона. На первый взгляд это кажется относительно простой задачей, но реальное приложение, удобное для обеих сторон, достаточно сложно. Автомобилист не должен заботиться о том, достаточно ли у него монет, чтобы оплатить парковку, а оператор парковки не должен заботиться об инкассации монет из сотен и тысяч парковочных автоматов и их транспортировки в банк. Полиция, контролируя парковку, может использовать устройство мобильной связи, подключенное к той же самой системе, чтобы точно узнать, сколько времени осталось до завершения оплаченного времени у каждой конкретной машины, и прикрепить на стекло штрафную квитанцию.

Резюме

За последние годы мир мобильных компьютеров существенно вырос, и за новыми тенденциями его развития становится трудно уследить. Теперь, надеюсь, изложено достаточно информации, чтобы можно было представить себе основные направлении развития средств мобильной связи и вероятные области разработки программного продукта для них. Кроме того, было продемонстрировано, что Delphi очень удобный инструмент, когда речь идет о создании приложений беспроводной связи следующего поколения.

Предметный указатель

P

Q

R

S

T

U

V

W

X

А

Б

В

Г

Д

Е

З

И

K

Л

М

H

O

П

P

C

T

У

Ф

Х

Ц

Ш

Э

Я

Научно-популярное издание

Стив Тейксейра, Ксавье Пачеко

Borland Delphi 6.
Руководство разработчика

Литературный редактор	*Е.В. Мельниченко*
Верстка	*А.В. Плаксюк*
Художественный редактор	*В.Г. Павлютин*
Корректоры	*Т.А. Корзун, Л.В. Коровкина* и *О.В. Мишутина*

Издательский дом "Вильямс"
101509, г. Москва, ул. Лесная, д. 43, стр. 1
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати

Подписано в печать 24.05.2002. Формат 70x100/16.
Гарнитура NewBaskerville. Печать офсетная.
Усл. печ. лист. 73,80. Уч.-изд. лист. 57,20.
Заказ . Тираж 5 000.

Отпечатано с диапозитивов в ФГУП "Печатный Двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, С.-Петербург, Чкаловский пр., 15.