По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-054-5, название «Платформа .NET. Основы» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

.NET Framework

Essentials

Second Edition

Thuan Thai & Hoang Q. Lam

O'REILLY®

Платформа .NET

Основы

Второе издание

Туан Тай, Хонг К.Лэм

Санкт-Петербург
2003

Туан Тай, Хонг К. Лэм

Платформа .NET. Основы, 2-е издание

Перевод Л. Фрейдина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Королев*
Редактор	*В. Овчинников*
Корректура	*С. Беляева*
Верстка	*Н. Гриценко*

Тай Т., Лэм Х. К.
Платформа .NET. Основы. – Пер. с англ. – СПб: Символ-Плюс, 2003. – 336 с., ил.
ISBN 5-93286-054-5

Перед вами краткое введение в платформу Microsoft .NET, призванное помочь разработчикам перейти от традиционного программирования для Windows к созданию приложений в среде .NET. В книге подробно описаны общеязыковая среда выполнения Common Language Runtime (CLR) и набор базовых классов, радикально упрощающих разработку крупномасштабных приложений. Рассмотрен механизм языковой интеграции и приведено описание цикла компонентной и корпоративной разработки с использованием .NET Framework. Кроме того, обсуждаются основы ключевых технологий .NET: работа с данными (ADO.NET) и XML, веб-службы (Web Services), веб-формы (Web Forms, ASP.NET) и Windows Forms.

Книга предназначена в основном для разработчиков, имеющих опыт в программировании COM, создании объектно-ориентированных, компонентных и корпоративных Windows-приложений с помощью языков Visual Basic, Visual C++, Java™ и C/C++, но может быть полезна всем желающим изучать Microsoft .NET Framework.

ISBN 5-93286-054-5
ISBN 0-596-00302-1 (англ)

© Издательство Символ-Плюс, 2003
Authorized translation of the English edition © 2002 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 14.10.2002. Формат 70x100 1/16 . Печать офсетная.
Объем 21 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 7

1. Обзор .NET 13

Microsoft .NET 14
Платформа .NET 15
Цели разработки .NET Framework 16
.NET Framework 22

2. Common Language Runtime 25

Среда CLR 25
Исполняемые файлы CLR 26
Метаданные 31
Сборки и манифесты 39
Промежуточный язык IL 46
CTS и CLS 49
Выполнение в среде CLR 54
Заключение 60

3. Программирование в среде .NET 61

Общая модель программирования 61
Языковая интеграция 78
Заключение 83

4. Работа с .NET-компонентами 84

Варианты развертывания 84
Распределенные компоненты 95
Службы COM+ в .NET 99
Организация пула объектов 106
Очереди сообщений 113
Заключение 116

5. Данные и XML 117

Архитектура ADO.NET 118
Достоинства ADO.NET 119
Компоненты сущностей 122
Управляемые поставщики 136
Наборы данных и XML 148
Заключение 160

6. Веб-службы 161

Веб-службы на практике 161
Структура Web Services 164
Поставщик веб-службы 175
Клиенты веб-служб 180
Веб-службы и безопасность 199
Заключение 201

7. Веб-формы 202

ASP 202
ASP.NET 204
Пространство имен System.Web.UI 205
Синтаксис веб-форм 214
Разработка приложения ASP.NET 222
ASP.NET и веб-службы 238
Привязка данных и применение шаблонов 241
Управление состоянием и масштабируемость 247
Заключение 253

8. Windows Forms 254

Введение в Windows Forms 254
Пространство имен System.Windows.Forms 256
Разработка Windows Forms 262
Windows Forms и веб-службы 287
Заключение 288

A. Языки .NET 289

B. Основные сокращения 291

C. Общие типы данных 297

D. Основные утилиты 304

Алфавитный указатель 314

Предисловие

Данная книга представляет собой краткое введение в Microsoft .NET Framework и предназначена для того, чтобы помочь программистам осуществить переход от традиционного программирования для Windows к миру программирования в среде .NET. Microsoft .NET Framework включает Common Language Runtime (CLR, общеязыковая среда выполнения) и набор базовых классов, радикально упрощающих разработку крупномасштабных приложений и сервисов. Книга детально описывает CLR, так что вы сможете активно применять ее новые возможности. В ней показано, как в действительности работает языковая интеграция, и приведено описание цикла компонентной и корпоративной разработки с использованием .NET Framework. Кроме того, имеется введение в четыре ключевых технологии .NET: работу с данными (ADO.NET) и XML, веб-сервисы (Web Services), веб-формы (Web Forms, ASP.NET) и Windows Forms.

Для подготовки рукописи, создания всех примеров и рисунков в этой книге мы использовали последнюю версию Microsoft Visual Studio .NET и .NET Framework SDK. Хотя мы старались сделать все возможное, чтобы техническое содержимое книги было актуальным, возможно, что кое-что с момента написания книги изменилось. Чтобы все время быть в курсе изменений, регулярно посещайте сайты *http://msdb.microsoft.com/net*, *http://www.gotdotnet.com* и страницу этой книги на сайте O'Reilly, *http://www.oreilly.com/catalog/dotnet_frmess2/*.

Наша аудитория

Хотя данная книга предназначена для всех желающих узнать о Microsoft .NET Framework, она нацелена на разработчиков с опытом создания Windows-приложений с помощью Visual Studio 6 и языков Visual Basic и Visual C++. Разработчики на Java™ и C/C++ также хорошо подготовлены для представленного здесь материала. Чтобы получить от книги максимум, вам следует иметь опыт в разработке объектно-ориентированных, компонентных, корпоративных и веб-приложений. Опыт в программировании COM также будет весьма полезным.

Об этой книге

Для тех, кто не знаком с каждой из описанных технологий, мы построили эту книгу, основываясь на кратком курсе, который Туан (Thuan) читал в нескольких компаниях с августа 2000 года, таким образом, что каждая глава базируется на знаниях из предыдущей главы. Чтобы дать вам общий обзор, приведем краткие аннотации глав и приложений этой книги.

Глава 1 «Обзор .NET» содержит краткое описание платформы Microsoft .NET. Она рассказывает о целях разработки .NET Framework и знакомит вас с компонентами .NET Framework.

В главе 2 «Common Language Runtime» мы приподнимем «капот» и заглянем внутрь CLR. Здесь рассматривается богатая библиотека времени выполнения CLR, а также другие средства.

В главе 3 «Программирование в .NET» содержится введение в .NET-программирование. Вы изучите простую программу, использующую концепцию объектно-ориентированной и компонентной разработки на четырех различных языках: Managed C++, VB.NET, C# и IL. Вы также сможете испытать преимущества языковой интеграции.

Глава 4 «Работа с .NET-компонентами» демонстрирует простоту компонентной и корпоративной разработки в .NET. Кроме рассмотрения средств развертывания компонентов, вы также исследуете полноценные программы, использующие преимущества транзакций, пула объектов, безопасности на основе ролей и очереди сообщений – и все в одной главе.

В главе 5 «Данные и XML» описана архитектура ADO.NET и ее преимущества. Кроме возможности отсоединения, что способствует улучшению масштабируемости, набор данных ADO.NET также тесно интегрирован с XML, что улучшает возможности взаимодействия между платформами. Глава знакомит вас с объектами .NET для доступа к данным, а также с пространством имен XML.

В главе 6 «Веб-службы» описано следующее поколение программных компонентов, к которым можно осуществлять доступ через Интернет. В этой главе мы обсудим протоколы, поддерживаемые веб-службами, а также опишем, как публиковать и находить их. Вы узнаете, каким образом XML, используемый в связке с HTTP, ломает закрытую природу текущей модели компонентно-ориентированной разработки программного обеспечения и обеспечивает лучшие возможности межсетевого взаимодействия.

В главе 7 «Веб-формы» дано введение в среду ASP.NET, которая теперь поддерживает объектно-ориентированное и управляемое событиями программирование, в отличие от традиционной разработки ASP.

В этой главе центральное место занимают веб-формы и серверные элементы управления. Кроме того, вы узнаете, как создавать собственные серверные элементы управления, выполнять привязку данных к различным элементам управления .NET, и изучите средства управления состоянием в ASP.NET.

Глава 8 «Windows Forms» переносит традиционное программирование на основе форм на шаг в будущее с помощью классов пространства имен System.Windows.Forms. Аналогично Win32-приложениям, Windows Forms лучше всего использовать для так называемых мощных или «толстых» клиентов, однако с новой, упрощенной процедурой инсталляции в .NET и приходом веб-сервисов Windows Forms стали пригодными для более широкого круга приложений.

Приложение A «Языки .NET» содержит список ссылок на веб-сайты с информацией о языках, использующих CLR, включая некоторые находящиеся в зачаточном состоянии проекты с открытым исходным кодом.

Приложение B «Основные сокращения» содержит список часто используемых сокращений, применяемых в литературе о .NET и на презентациях.

Приложение C «Общие типы данных» содержит несколько списков наиболее часто используемых в .NET типов данных. В этом приложении также проиллюстрировано применение некоторых классов коллекций.

В приложении D «Основные утилиты» приведен обзор наиболее важных утилит, предоставляемых .NET SDK для облегчения разработки в среде .NET.

Теперь, когда вы знаете, о чем эта книга, мы должны объяснить, чего в ней нет. В этой книге мы не обращаемся к маркетинговым аспектам .NET или других компонентов платформ .NET, таких как .NET Enterprise Servers, .NET Building Block Services или операционных систем .NET. Кроме того, не рассмотрен недавно объявленный сервис HailStorm, а также деятельность Microsoft по обеспечению доступности .NET Framework на многочисленных типах устройств.

Требования к читателю

В этой книге предполагается, что вы разработчик Windows- и веб-приложений, свободно разбирающийся в объектно-ориентированном и компонентном программировании. Также предполагается, что вы имеете базовые знания по XML. Хотя знание COM не является обязательным требованием, если вы имеете опыт в программировании COM, то в еще большей степени оцените эту книгу и .NET Framework.

Соглашения, используемые в этой книге

В этой книге приняты следующие соглашения о шрифтовом оформлении.

Курсив используется для:

- имен каталогов, файлов и программ;
- интернет-адресов, таких как имена доменов и URL;
- новых терминов там, где они определяются.

`Моноширинный шрифт` применяется для:

- командных строк и параметров, которые требуют дословного написания;
- прямых цитат и конкретных имен методов из примеров кода, а также конкретных значений атрибутов и параметров в коде;
- тегов XML-элементов.

`Моноширинный жирный шрифт` применяется для:

- вводимых пользователем данных в коде, которые должны печататься дословно;
- мест в коде, на которые мы бы хотели обратить внимание читателя.

`Моноширинный курсив` применяется для элементов кода, которые следует заменить на конкретные значения.

В примерах синтаксиса кода мы иногда будем использовать обозначение `[`*`value`*`]+` для одного или более экземпляров значения value и `[`*`value`*`]*` для обозначения нуля или более экземпляров значения value.

Как с нами связаться

Мы, насколько могли, аккуратно протестировали и сверили информацию из этой книги, однако вы можете обнаружить произошедшие изменения (или даже допущенные нами ошибки!). Просьба сообщать нам о любых найденных ошибках, а также о предложениях по будущим изданиям, написав по адресу:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (FAX)

Вы также можете присылать нам и электронные сообщения. Чтобы включить свой адрес в список рассылки или запросить каталог, отправьте письмо по адресу

info@oreilly.com

Чтобы задать технические вопросы или прислать комментарии к книге, отправьте письмо по адресу

bookquestions@oreilly.com

У нас есть веб-сайт этой книги, где мы поместим примеры, найденные ошибки и любые планы будущих изданий. Эта страница доступна по адресу

http://www.oreilly.com/catalog/dotnetfrmess2/

Дополнительную информацию по этой и другим книгам можно найти на веб-сайте издательства O'Reilly

http://www.oreilly.com

Дополнительная информация о среде .NET в целом приведена на .NET-центре издательства O'Reilly *http://dotnet.oreilly.com* и на *.NET DevCenter по адресу http://www.oreillynet.com/dotnet/*.

Благодарности

Сотрудники издательства O'Reilly не устают поражать нас предоставляемой ими поддержкой. Я хотел бы поблагодарить Джона Осборна (John Osborn), продлившего договор для написания этой книги, за постоянную поддержку проекта. Мы также хотим поблагодарить Нэнси Котари (Nancy Kotary) за выполненную трудную работу, позволившую обеспечить выход книги в установленные жесткие сроки. Нэнси провела большую работу по рецензированию наших материалов и координации проекта. Без Джона и Нэнси эта книга никогда не была бы написана. Спасибо сотрудникам O'Reilly, занимающимся производством и дизайном, за воплощение этой книги в реальность: Эмме Колби (Emma Colby), Татьяне Диаз (Tatiana Diaz), Дэвиду Футато (David Futato), Колину Горману (Coleen Gorman), Роберту Романо (Robert Romano), Майку Сьерра (Mike Sierra), Элли Волькхаузен (Ellie Volckhausen) и Джо Визда (Joe Wizda).

Спасибо Брайану Джепсону (Brian Jepson), внесшему значительный вклад в эту книгу с самого начала проекта. Брайан проделал, безуслов-

но, выдающуюся работу по чтению, тестированию и обеспечению соответствия технического содержимого каждой главы последнему релизу. Мы также хотим поблагодарить Деннис Анжелин (Dennis Angeline) и Брэда Меррила (Brad Merrill) из Microsoft за ответы на технические вопросы по CLR и языкам программирования.

Хоанг (Hoang) хочет поблагодарить своих родителей и семью за их поддержку и понимание того, что он выпал из их жизни на несколько месяцев. Мама и папа, ваши постоянные усилия, чтобы дети оказались там, где они сейчас, никогда не могут быть оплачены. Хоанг также благодарит свою жену, ВанДу (VanDu), источник своего вдохновения. Не надо недооценивать ее вклад в эту книгу. И, наконец, последнее, но не менее важное, персональная *благодарность* Туану (Thuan), который все время подталкивал меня к долгожданному завершению книги.

1

Обзор .NET

Корпорация Microsoft объявила об инициативе .NET в июле 2000 года. Платформа .NET – это новая среда разработки с новым интерфейсом программирования для доступа к службам и API Windows, интегрированная с рядом технологий, вышедших из Microsoft в конце 1990-х годов. В .NET объединены службы компонентов COM+; среда веб-разработки ASP; приверженность XML и объектно-ориентированному дизайну; поддержка новых протоколов веб-сервисов, таких как SOAP, WSDL и UDDI; а также ориентация на Интернет.

Платформа состоит из четырех отдельных групп продуктов:

Средства разработки

Набор языков, включающий C# и VB.NET; набор инструментальных средств разработки, в том числе Visual Studio.NET; всеобъемлющая библиотека классов для построения веб-сервисов, веб- и Windows-приложений; а также среда Common Language Runtime (CLR, общеязыковая среда выполнения) – для исполнения объектов, построенных в рамках этой структуры.

Специализированные серверы

Семейство серверов .NET Enterprise Servers, ранее известных как SQL Server 2000, Exchange Server 2000, BizTalk 2000 и т. д., обеспечивающих хранение реляционных баз данных, функции электронной почты и B2B-коммерции.

Веб-сервисы

Предоставление коммерческих веб-сервисов, в частности инициатива .NET My Services (ранее называвшаяся HailStrom); за плату разработчики могут применять эти сервисы для построения приложений, требующих идентификации пользователей.

Устройства

Новые устройства, поддерживающие .NET, от мобильных телефонов до игровых приставок.

Microsoft выделяет значительные ресурсы для разработки и поддержания успеха .NET и связанных с ней технологий: корпорация делает ставку на .NET как на следующий шаг в развитии компьютеризации.

Microsoft .NET

Microsoft потратила последние четыре года на создание платформы Microsoft .NET, представленной на конференции PDC 2000 в Орландо, штат Флорида. Хотя основной стратегией .NET было создание программного обеспечения в виде сервисов, .NET – это нечто большее. Кроме того, что она ориентирована на работу со Всемирной паутиной, Microsoft .NET соответствует следующим современным тенденциям программной индустрии:

Распределенные вычисления

Упрощает разработку надежных приложений клиент-сервер. Современные технологии распределенных вычислений требуют значительного сближения разных разработчиков и обеспечения возможности взаимодействия разных платформ. Microsoft .NET предоставляет архитектуру взаимодействия, использующую открытые стандарты Интернета, включая HTTP (Hypertext Transfer Protocol), XML (Extensible Markup Language) и SOAP (Simple Object Access Protocol).

Компонентное представление

Упрощение интеграции программных компонентов, созданных разными разработчиками. Технология COM (Component Object Model) воплотила в реальность программный «plug-and-play», однако разработка и развертывание COM-компонентов слишком сложны. Microsoft .NET предоставляет более простой способ построения и развертывания компонентов.

Корпоративные службы

Позволяет разрабатывать масштабируемые корпоративные приложения, не программируя управление транзакциями, средства безопасности или организацию пулов. Microsoft .NET продолжает поддерживать корпоративные службы, сокращающие время разработки и усилия, затрачиваемые на построение крупномасштабных приложений.

Смещение парадигмы Web

Внесены изменения в веб-технологии, упрощающие разработку приложений для Интернета. За последние несколько лет разработка веб-приложений сместилась от коммуникации (TCP/IP) к представлению (HTML) и к программируемости (XML и SOAP). Ключевой

задачей Microsoft .NET является обеспечение возможности продажи и распространения программного обеспечения как сервиса.

Факторы зрелости

Видно, какие уроки извлекла программная индустрия из разработки крупномасштабных корпоративных и веб-приложений. Коммерческое веб-приложение должно обеспечивать открытость, масштабируемость, доступность и управляемость. Microsoft .NET достигает всех этих целей.

Хотя здесь перечислены основные концепции Microsoft .NET, важно также отметить, что Microsoft .NET использует в своем ядре открытые стандарты Интернета (HTTP, XML и SOAP) для передачи объектов от одной машины к другой через Интернет. Фактически в .NET существует двухстороннее соответствие между XML и объектами. Например, класс может быть описан в виде определения XML-схемы (XSD); объект может быть преобразован в формат XML и обратно; метод может быть задан в XML-формате, называемом Web Services Description Language (WSDL, язык описания веб-сервисов); а вызов метода выражен в XML-формате SOAP.

Платформа .NET

Платформа .NET состоит из пяти основных компонентов, как показано на рис. 1.1. На самом нижнем уровне находится операционная система (ОС), которая может быть одной из нескольких Windows-платформ, включая Windows XP, Windows 2000, Windows Me и Windows CE. Как часть стратегии .NET, Microsoft обещает предоставить программное обеспечение для других .NET-устройств, чтобы способствовать появлению нового поколения «умных» устройств.

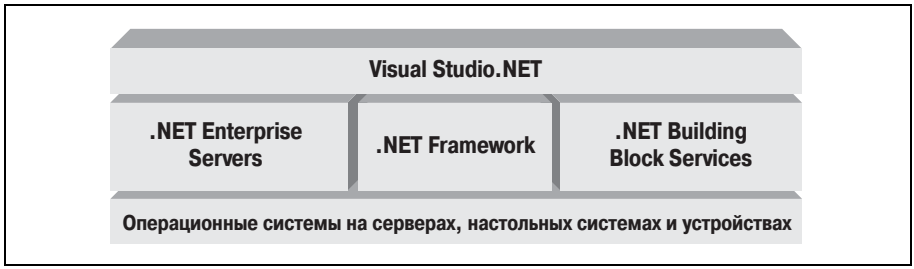


Рис. 1.1. *Платформа Microsoft .NET*

Поверх операционной системы функционирует серия продуктов .NET Enterprise Servers, сокращающих время, требуемое для разработки крупномасштабных бизнес-систем. Эти серверные продукты включают Application Center 2000, BizTalk Server 2000, Commerce Server 2000, Exchange Server 2000, Host Integration Server 2000, Internet Security Acceleration Server 2000 и SQL Server 2000.

Поскольку веб-сервисы могут активно применяться через Интернет, Microsoft планирует предоставить сервисы, являющиеся строительными блоками, которые разработчики приложений смогут использовать за плату. Примером такого сервиса является Microsoft Passport, позволяющий установить единое имя пользователя и пароль на всех сайтах, поддерживающих аутентификацию через Passport. В марте 2001 года Microsoft объявила об еще одном наборе веб-сервисов под кодовым названием HailStorm, которые теперь называются .NET My Services. В этом продукте заключен набор сервисов – строительных блоков, поддерживающих персонализацию, полностью основанную на последовательном опыте пользователя.[1] Microsoft планирует добавить и дополнительные сервисы, такие как календарь, каталог и поисковый сервис. Сторонние поставщики также создают собственные веб-сервисы.

На верхнем уровне архитектуры .NET находится совершенно новое средство разработки под названием Visual Studio.NET (VS.NET), которое делает возможным быструю разработку веб-сервисов и других приложений. Преемник Visual Studio 6.0, VS.NET, представляет собой интегрированную среду разработки (Integrated Development Environment, IDE), поддерживающую четыре различных языка и такие средства, как межъязыковая отладка и редактор XML-схем (XML Schema Editor).

А в центре .NET располагается Microsoft .NET Framework – основной предмет рассмотрения этой книги и новая инфраструктура разработки и исполнения, которая изменит создание бизнес-приложений на платформе Windows. Она включает общеязыковую среду выполнения Common Language Runtime (CLR) и общую структуру классов, которые могут использоваться всеми языками .NET.

Цели разработки .NET Framework

При разработке .NET Framework Microsoft преследовала много практических и исключительно амбициозных целей. В этом разделе мы обсудим главные задачи разработки Microsoft .NET Framework, включая улучшенную поддержку компонентов, межъязыковую интеграцию, обеспечение взаимодействия приложений через киберпространство, простую разработку и развертывание, улучшенную надежность и повышенную безопасность.

Компонентная инфраструктура

До появления технологии COM разработчики, использующие средства Microsoft, не имели простого способа интеграции бинарных библиотек

[1] Дополнительную информацию о .NET My Services можно будет найти в планируемой к публикации книге «.NET My Services Essentials» Калберта и Мэрфи (Culbert and Murphy, O'Reilly).

без ссылки на их исходный код или без его изменения. С введением COM программисты смогли интегрировать бинарные компоненты в свои приложения, аналогично аппаратным компонентам plug-and-play в вашем настольном компьютере. Хотя технология COM была хороша, неприятные детали ее применения были источником массы неприятностей для разработчиков и администраторов.

Хотя COM позволяет интегрировать бинарные компоненты, разработанные с использованием любого языка, эта технология требует соблюдения правил идентификации, времени жизни и двоичной структуры COM. Необходимо также писать код «заплаток», требуемых для создания COM-компонента, таких как DllGetClassObject, CoRegisterClassObject и т. д.

Понимая, что эти требования приводят к частому повторному написанию схожего кода, .NET решается избавиться от них. В мире .NET все классы готовы к использованию на двоичном уровне. Нет необходимости в написании дополнительного «заплаточного» кода для поддержки компонентного представления объекта в .NET Framework. Вы просто пишете класс .NET, который затем становится частью сборки (мы обсудим это в главе 2) и поддерживает plug-and-play.[1]

Кроме предоставления среды, облегчающей разработку, .NET устраняет трудности создания COM-компонентов. В частности, в .NET не требуется изменять реестр для регистрации компонентов и исключается необходимость в обязательном дополнительном коде, присутствующем во всех COM-компонентах, включая код для поддержки интерфейса IUnknown, фабрик классов, времени жизни компонента, регистрации, динамического связывания и прочего.

«Компонент» – неудачное слово, так как один человек может использовать его в отношении объекта, а другой – в отношении двоичного модуля. Чтобы исключить противоречия, в этой книге термин «COM-компонент» (или просто «компонент») означает двоичный модуль, такой как DLL или EXE.

Языковая интеграция

COM поддерживает *языковую независимость* (*language independence*), означающую, что вы можете разрабатывать COM-компонент на любом

[1] COM все еще играет свою роль в .NET Framework. Фактически, если вы воспользуетесь утилитой *dumpbin.exe* для получения дампа исполняемого файла (PE), созданного компиляторами, доступными в предварительном релизе или версии Beta 1 .NET SDK, то увидите некоторые остатки COM, в частности, упоминание чего-то с именем `COM+ Header`. Дополнительную информацию можно найти в разделе «Формат портируемого исполняемого файла .NET» в главе 2.

языке. При условии, что ваш компонент соответствует правилам, изложенным в спецификации COM, можно создать его экземпляр и использовать его в своих приложениях. Поддерживая таким образом двоичное взаимодействие, технология COM не поддерживает *языковую интеграцию* (*language integration*). Другими словами, вы не можете использовать код COM-компонентов, написанных кем-то другим, не можете также расширить класс, заключенный в COM-компоненте, перехватывать исключения, вызываемые кодом COM-компонента, и т. д.

Microsoft .NET поддерживает не только языковую независимость, но и языковую интеграцию. Это означает, что невозможно наследовать от классов, перехватывать исключения и использовать преимущества полиморфизма между разными языками. .NET Framework делает это возможным с помощью системы общих типов – спецификации под названием Common Type System (CTS), которую должны поддерживать все .NET-компоненты. Например, все в .NET является объектом определенного класса, производного от корневого класса System.Object. CTS поддерживает общую концепцию классов, интерфейсов, делегатов (обеспечивающих поддержку обратных вызовов), ссылочных типов и типов значений. В базовых классах .NET представлено большинство базовых системных типов, таких как классы для поддержки целых значений, строк и манипуляций с файлами. Поскольку компилятор любого языка должен соответствовать минимальному набору правил, оговоренных в Common Language Specification (CLS), и генерировать код, согласующийся с CTS, различные языки в .NET могут общаться между собой. Мы рассмотрим CTS и CLS в главе 2.

Взаимодействие компонентов через Интернет

COM поддерживает распределенные вычисления через свой протокол взаимодействия Distributed COM (DCOM). Проблема с DCOM состоит в том, что он вставляет IP-адрес хоста в буфер Network Data Representation (NDR), таким образом, DCOM не работает через брандмауэры и программное обеспечение, осуществляющее трансляцию сетевых адресов (Network Address Translation, NAT). Кроме того, средства динамической активации, согласования протоколов и сборки мусора закрыты, сложны и дороги. Решением является открытый, простой и экономичный протокол распределенных вычислений. .NET Framework использует поддержанный компьютерной отраслью протокол SOAP, базирующийся на широко распространенных стандартах XML и HTTP.

Упрощение разработки

Те, кто разрабатывал программное обеспечение для Windows-платформ с момента их появления, видел все, начиная от Windows API и заканчивая Microsoft Foundation Classes (MFC), Active Template Library (ATL), системными COM-интерфейсами и несчетным числом обо-

лочек, таких как Visual InterDev, Visual Basic, JScript и других языков сценариев. Каждый раз тому, кто собирался разрабатывать что-нибудь при помощи другого компилятора, приходилось изучать новый API или библиотеку классов, поскольку между всеми этими библиотеками или интерфейсами не существовало никакого соответствия или унификации.

Решение .NET представляет собой набор базовых классов и позволяет работать с ними любому языку. Такая среда исключает необходимость в изучении нового API при каждом «переключении» между языками. Другими словами, наверняка проще разобраться с десятком методов определенного класса, чем изучать тысячу функций API.

Упрощение развертывания приложений

Представьте себе следующий сценарий: ваше Windows-приложение, использующее три разделяемые DLL-библиотеки, прекрасно работает многие месяцы, однако перестает работать на следующий день после того, как вы установили другой программный пакет, переписавший первую DLL, ничего не сделавший со второй DLL и записавший дополнительную копию третьей в другой каталог. Если вы хоть раз встречались с такой трудной (но при этом вполне реальной) проблемой, вы попали в *ад DLL* (*DLL Hell*). И если вы спросите в группе бывалых разработчиков, знаком ли им ад DLL, вы увидите у них гримасу досады, и не из-за заданного вами вопроса, а из-за того, что они действительно испытали боль и страдание.

Чтобы избежать DLL Hell (по крайней мере для системных DLL), Windows 2000 сохраняет системные DLL в кэше. Если вы устанавливаете приложение, которое переписывает системные DLL, Windows 2000 заменит новые системные DLL исходными версиями из кэша.

Microsoft .NET еще более уменьшает проблему DLL Hell. В среде .NET ваш исполняемый файл будет использовать ту общую DLL, с которой он был собран. Это гарантируется тем, что общая DLL регистрируется в чем-то вроде кэша Windows 2000, который называется глобальным кэшем сборок (Global Assembly Cache, GAC). Кроме выполнения этого требования общая DLL должна иметь уникальные хеш-код, открытый ключ, региональный код и номер версии. Если эти требования выполнены и ваша общая DLL зарегистрирована в GAC, ее физическое имя перестает иметь значение. Другими словами, если у вас есть две версии DLL, обе под именем *MyDll.dll*, то они могут располагаться и исполнятся на одной и той же системе, не вызывая проблем. Это становится возможным потому, что исполняемый файл, использующий одну из этих DLL, жестко привязывается к DLL при компиляции.

Кроме искоренения DLL Hell, .NET также исключает необходимость настройки реестра, относящейся к компонентам. Разработчик COM расскажет вам, что половина сложностей в изучении COM состоит в

необходимости знать специфичные для COM записи реестра, за которые отвечает программист. Microsoft .NET хранит все ссылки и зависимости сборок в специальной секции, называющейся *манифестом* (*manifest*). Кроме того, сборки могут быть как закрытыми, так и общими. Закрытые сборки находятся по логическим путям или конфигурационным XML-файлам, а общие сборки регистрируются в GAC; в обоих случаях система находит требуемые вам файлы на этапе выполнения приложения. Если файлы отсутствуют, вы получите исключение, сообщающее о том, что случилось.

И наконец, в .NET возвращается концепция установки и удаления приложений без каких-либо усилий. Эта концепция является противоположностью тому, с чем вам приходилось иметь дело в мире COM. Для установки COM-приложения вы должны были зарегистрировать все компоненты после копирования их на свой компьютер. Если вы выполнили этот этап неверно, ничего не будет работать, и вам останется только рвать на себе волосы. Аналогично, чтобы удалить приложение, вам придется аннулировать регистрацию ваших компонентов (удалить записи в реестре) перед тем, как удалить файлы. И опять же, если вы неправильно выполните этот этап, в реестре останутся следы, которые будут существовать вечно.

В отличие от COM, так же как в DOS, для установки приложения в .NET вам достаточно просто выполнить команду `xcopy`, чтобы скопировать файлы из одного каталога на CD в другой каталог на вашем компьютере, и приложение сможет запуститься автоматически.[1] Аналогично, для удаления приложения с вашего компьютера достаточно просто удалить каталог.

Надежность

В индустрии коммерческого программного обеспечения существует много языков программирования и платформ, однако в немногих из них делается попытка предоставить как надежный язык программирования, так и устойчивую библиотеку времени выполнения или инфраструктуру. Наиболее успешным языком, из тех, что мы видели среди коммерческого ПО, является язык Java™ и Java Virtual Machine™, которые были с радостью приняты сообществом разработчиков ПО. Microsoft позиционирует .NET как следующий большой шаг вперед.

В Microsoft .NET безопасность типов является обязательной. В отличие от C++, любой класс в .NET является производным от прародителя всех классов, Object, поддерживающего средства идентификации типов во время выполнения, средства построения дампа содержимого и прочие возможности. CLR должна распознать и проверить типы до

[1] Это верно для закрытых, но не для общих. Дополнительные детали рассмотрены в главе 4.

того, как они смогут быть загружены и выполнены. Это сокращает возможности возникновения элементарных программистских ошибок и предотвращает переполнение буферов, ослабляющее безопасность.

Традиционные языки программирования не предоставляют общего механизма обработки ошибок. C++ и Java поддерживают обработку исключений, однако многие другие языки о вас не заботятся, вынуждая изобретать собственные средства. Microsoft .NET поддерживает исключения на уровне CLR, предоставляя согласованный механизм обработки ошибок. Другими словами, исключения работают во всех языках, совместимых с .NET.

Программируя на C++, вы должны освобождать все расположенные в куче объекты, которые были ранее созданы. Если этого не сделать, выделенные вашей системой ресурсы никогда не будут возвращены обратно, даже если они больше не требуются. И если это серверное приложение, оно не будет надежным, поскольку накопленные в памяти неиспользуемые ресурсы со временем могут привести к краху системы. Аналогично, Java во время выполнения .NET отслеживает и выполняет сборку мусора для всех созданных в памяти объектов, которые более не требуются.

Безопасность

В старые добрые времена DOS разработчики Microsoft, создавая приложения, мало беспокоились о безопасности, так как приложения работали на одиночном настольном компьютере с одним потоком выполнения. Как только разработчики начали создавать клиентские и серверные приложения, все усложнилось: к серверам могли обращаться несколько пользователей, а клиент и сервер обменивались друг с другом важной информацией. Проблема еще более усилилась в среде Интернета, так как существует опасность незаметно загрузить и выполнить на своей машине злонамеренный апплет.

Чтобы смягчить эти проблемы, .NET предоставляет ряд средств безопасности. Windows NT и Windows 2000 защищают ресурсы с помощью списков управления доступом и идентификаторов безопасности, однако не предоставляют инфраструктуру безопасности для проверки прав доступа к отдельным частям исполняемого кода. В отличие от традиционной поддержки безопасности, при которой защищается только доступ к исполняемому файлу, .NET идет дальше и позволяет защитить доступ к определенным частям исполняемого кода. Например, чтобы воспользоваться преимуществами декларативной проверки безопасности, вы можете указать в реализации ваших методов атрибуты безопасности и не писать дополнительный код. Чтобы воспользоваться императивными проверками, в методе пишется код для явного осуществления проверки безопасности. .NET предоставляет много других функций безопасности, чтобы усложнить несанкционированное проникновение в ваше приложение и систему.

.NET Framework

Теперь, когда вы знакомы с основными задачами, поставленными перед .NET Framework, давайте кратко рассмотрим ее архитектуру. Как видно на рис. 1.2, .NET Framework располагается поверх операционной системы, которая может быть одной из различных видов Windows[1], и состоит из нескольких компонентов. (К более подробному обсуждению каждого из этих компонентов мы приступим с главы 4, как и сказано в предисловии.) .NET по существу является системным приложением, работающим в Windows.

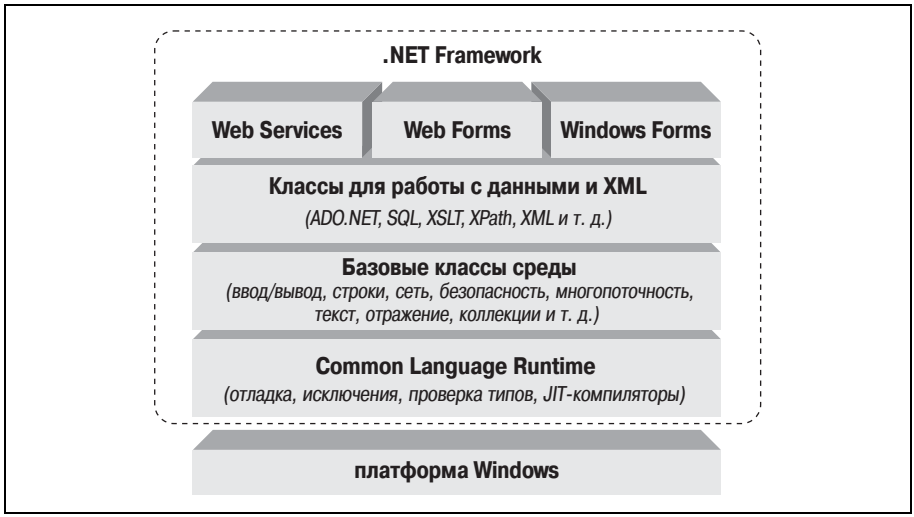


***Рис. 1.2.** .NET Framework*

Наиболее важным компонентом .NET Framework является нечто под названием Common Language Runtime (CLR), или общеязыковая среда выполнения. Если вы Java-программист, рассматривайте CLR как .NET-эквивалент виртуальной Java-машины (JVM). Если вы не знаете Java, считайте CLR сердцем и душой архитектуры .NET. На верхнем уровне CLR активизирует объекты, выполняет для них проверку безопасности, располагает их в памяти, исполняет их и выполняет сборку мусора.

Концептуально CLR и JVM схожи в том, что обе являются инфраструктурами времени выполнения, абстрагирующими различия в платформах, на которых они работают. Однако если JVM в данный момент поддерживает только язык Java, то CLR поддерживает все язы-

[1] В действительности операционной системой может быть (потенциально) любой вид Unix или иной операционной системы. Это возможно вследствие архитектуры CLR, которая обсуждается в главе 2.

ки, которые представимы на общем промежуточном языке Common Intermediate Language (CIL). JVM исполняет байт-код, поэтому технически также может поддерживать много различных языков. Однако, в отличие от байт-кода Java, IL никогда не интерпретируется. Другим концептуальным отличием между этими двумя инфраструктурами является то, что код Java с помощью JVM может работать на многих платформах, тогда как код .NET работает только на платформе Windows с CLR (на момент написания этой книги). Microsoft представила общеязыковую инфраструктуру Common Language Infrastructure (CLI), являющуюся подмножеством CLR, в ECMA, так что сторонние поставщики теоретически смогут реализовать CLR для платформы, отличной от Windows. Дополнительную информацию о сторонних поставщиках см. в приложении A.

На рис. 1.2 уровнем, расположенным над CLR, является набор базовых классов среды. Этот набор похож на наборы классов в STL, MFC, ATL или Java. Классы поддерживают элементарную функциональность ввода/вывода, манипуляции со строками, управление безопасностью, функциональность отражений и коллекций, а также другие функции.

Над базовыми классами среды находится набор классов, расширяющих базовые поддержкой управления данными и XML. Классы работы с данными поддерживают работу с постоянными данными – информацией, хранящейся в базах данных. Эти классы включают классы работы с SQL (Structured Query Language), позволяя вам манипулировать хранилищами данных через стандартный SQL-интерфейс. Аналогично SQL-классам, набор классов под названием ADO.NET позволяет вам манипулировать постоянными данными. Наряду с классами работы с данными, .NET Framework поддерживает несколько классов, позволяющих вам манипулировать XML-данными, выполнять XML-поиск и XML-преобразования.

Классы в трех различных технологиях (Web Services, Web Forms и Windows Forms) расширяют базовые классы и классы работы с данными и XML. Web Services включают ряд классов, поддерживающих разработку облегченных распределенных компонентов, которые будут работать даже через брандмауэры и программное обеспечение NAT. Эти компоненты поддерживают plug-and-play через киберпространство, так как веб-сервисы применяют стандартные протоколы HTTP и SOAP.

Web Forms включает набор классов, позволяющих вам быстро разрабатывать веб-приложения с графическим интерфейсом пользователя (GUI). Если сейчас вы разрабатываете веб-приложения с помощью Visual Interdev, можете рассматривать Web Forms как средство, позволяющее использовать для создания графических веб-интерфейсов подход «drag-n-drop», аналогичный используемому для разработки пользовательских интерфейсов в Visual Basic. Просто поместите управляющие элементы на вашу веб-форму, два раза щелкните мышью на элементе управления и пишите код реакции на соответствующее событие.

Windows Forms поддерживают набор классов, позволяющий вам разрабатывать обычные графические Windows-приложения. Можно рассматривать все эти классы как улучшенную версию MFC, т. к. они поддерживают более простую разработку графических интерфейсов и предоставляют общий, совместимый интерфейс, который можно использовать во многих языках.

В следующей главе мы рассмотрим внутреннее устройство CLR и то, как в ней поддерживаются и исполняются .NET-компоненты, официально называемые в .NET *сборками* (*assemblies*).

2

Common Language Runtime

Наиболее важный компонент .NET Framework – это общеязыковая среда выполнения Common Language Runtime (CLR). CLR управляет кодом и исполняет код, написанный на языках .NET, и является фундаментом всей архитектуры .NET, подобно виртуальной машине Java. CLR активизирует объекты, выполняет для них проверку безопасности, располагает в памяти, исполняет их и выполняет сборку мусора.

В этой главе мы приведем описание среды CLR, исполняемых файлов (с примерами на нескольких языках), метаданных, сборок, манифестов, CTS и CLS.

Среда CLR

CLR – это инфраструктура, лежащая в основе .NET. Ее возможности охватывают все задачи, о которых мы говорили в главе 1. В отличие от таких библиотек программного обеспечения, как MFC или ATL, CLR построена с чистого листа. CLR управляет исполнением кода в .NET Framework.

Сборка (*assembly*) – это основная единица для развертывания и контроля версий, состоящая из манифеста, набора из одного или более модулей и необязательного набора ресурсов.

На рис. 2.1 показаны две части среды .NET: нижняя – это CLR, а верхняя – исполняемые файлы CLR или переносимые исполняемые файлы (Portable Executable, PE), являющиеся сборками .NET или единицами развертывания. CLR – это механизм времени выполнения, загружающий требуемые классы, выполняющий оперативную (just-in-time)

компиляцию необходимых методов, проверку безопасности и много других функций времени выполнения. Исполняемые файлы CLR, показанные на рис. 2.1, являются EXE- или DLL-файлами, состоящими в основном из метаданных и кода.

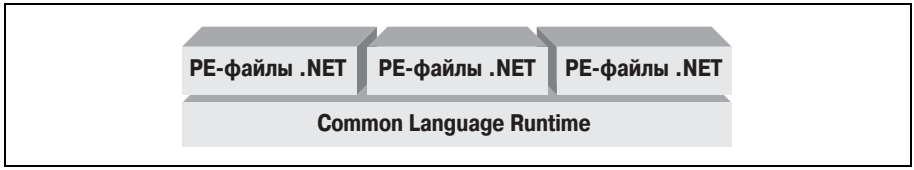


Рис. 2.1. *Среда CLR*

Исполняемые файлы CLR

Исполняемые файлы Microsoft .NET отличаются от типичных исполняемых файлов Windows тем, что они содержат не только код и данные, но еще и метаданные (см. далее в этой главе разделы «Метаданные» и «Промежуточный язык IL»). В данном разделе мы начнем с кода нескольких .NET-приложений и обсудим PE-формат, используемый в .NET.

Hello, World: Managed C++

Начнем с простого приложения *Hello, World*, написанного на Managed C++ – расширении языка C++, используемом в Microsoft .NET. Managed C++ содержит несколько новых ключевых слов, специфических для .NET, которые позволяют программам, написанным на C++, воспользоваться новыми возможностями .NET, включая сборку мусора. Вот версия нашей программы, написанная на Managed C++:

```
#using <mscorlib.dll>
using namespace System;

void main()
{
  Console::WriteLine(L"C++ Hello, World!");
}
```

Как видите, это обычная программа на C++ с дополнительной директивой `#using` (выделенной жирным шрифтом). Если вы работали с компилятором Microsoft Visual C++, поддерживающим средства для технологии COM, вам может быть знакома директива `#import`. Если `#import` конвертирует информацию о типах для генерации классов-оболочек для COM-интерфейсов, то директива `#using` делает доступными все типы из указанной DLL, аналогично директиве `#include` в C или C++. Однако в отличие от `#include`, импортирующей типы C или C++, `#using` импортирует типы любой сборки .NET, написанной на любом языке .NET.

Первый и единственный оператор в методе main() не требует объяснений – он обозначает вызов статического метода (или метода уровня класса) WriteLine(), класса Console. Символ `L`, предшествующий строке символов, сообщает компилятору C++ о необходимости преобразовать строку символов в строку Unicode. Как вы могли догадаться, класс Console – это тип, расположенный в *mscorlib.dll*, и его метод принимает один строковый параметр.

Вы также можете заметить, что код сообщает компилятору об использовании типов из пространства имен System, на что указывает оператор `using namespace`. Это позволяет нам применять идентификатор Console вместо полного обозначения класса System::Console.

Скомпилируем эту простую программу с помощью нового компилятора C++ с интерфейсом командной строки, поставляемого с .NET SDK:

```
cl hello.cpp /CLR /link /entry:main
```

Параметр командной строки `/CLR` исключительно важен, т. к. он сообщает компилятору C++ о необходимости сгенерировать PE-файл для среды .NET, а не обыкновенный PE-файл Windows.

После выполнения этой команды компилятор C++ сгенерирует исполняемый файл *hello.exe*. Когда вы запустите *hello.exe*, CLR загрузит, проверит и исполнит его.

Hello, World: C#

Поскольку .NET серьезно относится к языковой интеграции, мы проиллюстрируем ту же программу, используя новый язык C#, разработанный в Microsoft специально для .NET. Заимствовавший синтаксис у Java и C++, C# – это простой и объектно-ориентированный язык, который Microsoft использовала для написания большей части базовых классов и утилит .NET. У тех, кто программирует на Java (или C++), не будет проблем с пониманием кода C#. Вот *Hello, World* на C#:

```
using System;

class MainApp
{
  public static void Main()
  {
    Console.WriteLine("C# Hello, World!");
  }
}
```

C# схож с Java в том, что в нем нет концепции заголовочного файла: определения классов и их реализации хранятся в одном файле *.cs*. Другое сходство с Java состоит в том, что Main() – это открытая статическая функция определенного класса, что видно из кода. Этим он отличается от C++, в котором метод main() является глобальной функцией.

Ключевое слово `using` работает здесь аналогично `using namespace` в предыдущем примере, сообщая компилятору C#, что мы хотим использовать типы из пространства имен System. Вот как компилируется эта программа C#:

```
csc hello.cs
```

В приведенной выше команде `csc` – это имя компилятора C#, поставляемого в составе .NET SDK. Результатом выполнения этой команды также будет исполняемый файл *hello.exe*, который можно запустить как обычный EXE, однако он работает под управлением CLR.

Hello, World: VB.NET

И наконец, для полноты картины, вот та же самая программа на языке Visual Basic.NET (VB.NET):

```
Imports System

Public Module modmain
  Sub Main()
    Console.WriteLine ("VB Hello, World!")
  End Sub
End Module
```

Если вы VB-программист, вас ждет сюрприз. Синтаксис языка немного изменился, однако, к счастью, в этих изменениях воспроизведены возможности других объектно-ориентированных языков, таких как C# и C++. Посмотрите внимательно на этот фрагмент кода, и вы увидите, что каждая его строка может быть транслирована в эквивалентную строку C#. В то время как в C# применяются ключевые слова `using` и `class`, в VB.NET аналогичную роль играют, соответственно, ключевые слова `Import` и `Module`. Вот как нужно компилировать эту программу:

```
vbc /t:exe /out:Hello.exe Hello.vb
```

Microsoft также предоставляет для VB.NET компилятор с интерфейсом командной строки, *vbc*. Параметр `/t` указывает на тип создаваемого PE-файла. В этом случае, поскольку мы указали имя EXE-файла, результатом выполнения этой команды будет файл *hello.exe*.

Во всех трех версиях этой программы *Hello, World*, класс Console и метод WriteLine() оставались неизменными. Таким образом, не имеет значения, на каком языке вы программируете, и если вы знаете, как сделать что-либо с помощью одного языка, то вы знаете, как это делать и в другом. Это большое изменение по сравнению с традиционным программированием в Windows, в котором, если вы знаете, как осуществляется запись в файл на C++, вы не обязательно знаете, как это сделать в VB, Java или Cobol.

Переносимый исполняемый файл .NET

Исполняемый файл Windows, EXE или DLL, должен соответствовать формату, который называется форматом *PE* и является производным от формата Microsoft Common Object File Format (COFF). Для обоих этих форматов существуют и свободно доступны полные спецификации. ОС Windows знает, как загружать и исполнять DLL- и EXE-файлы, т. к. понимает формат PE-файла. Поэтому любой компилятор должен генерировать исполняемый файл Windows в соответствии со спецификацией PE/COFF.

Стандартные PE-файлы Windows делятся на две основных секции. Первая включает в себя заголовки PE/COFF со ссылками на содержимое внутри PE-файла. Кроме секции заголовков в PE-файле имеется несколько секций двоичных образов, включающих секции `.data`, `.rdata`, `.rsrc` и `.text`. Это стандартные секции типичного исполняемого файла Windows, однако компилятор Microsoft C и C++ позволяет добавлять в PE-файл собственные секции с помощью директивы компилятора `pragma`. Например, можно создать собственные секции, содержащие зашифрованные данные, которые сможете читать только вы сами. Пользуясь этой возможностью, Microsoft добавила несколько новых секций в обычный PE-файл специально для поддержки функциональности CLR. Среда CLR понимает и управляет новыми секциями. Например, она читает эти секции и определяет, каким образом загружать классы и выполнять ваш код.

Как показано на рис. 2.2, разделы, добавленные Microsoft к обычному формату PE, – это заголовок CLR и раздел данных CLR. Заголовок CLR содержит информацию, указывающую, что PE-файл является исполня-

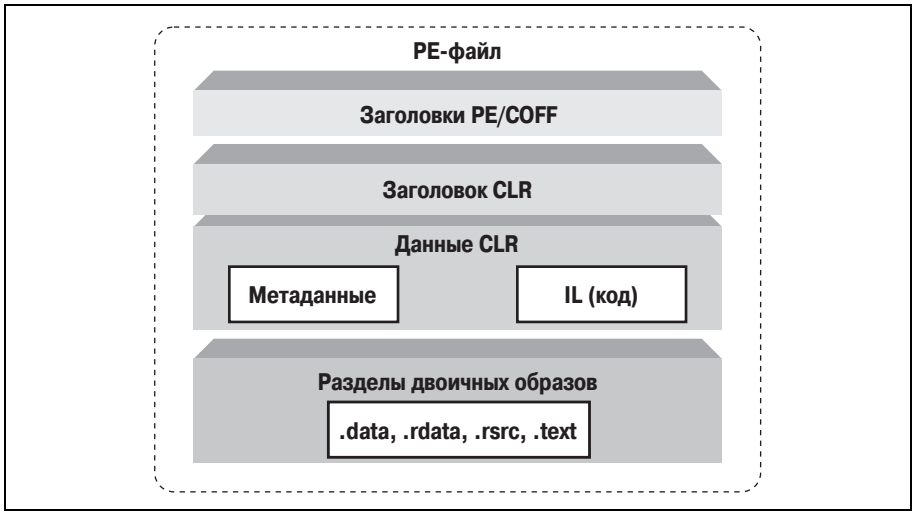


Рис. 2.2. *Формат PE-файла .NET*

емым файлом .NET, а раздел данных CLR содержит метаданные и IL-код, которые вместе определяют, как будет выполняться программа.

Если вы хотите убедиться, что исполняемый файл .NET содержит обе эти секции, воспользуйтесь утилитой *dumpbin.exe*, которая показывает содержимое исполняемого файла Windows в читабельном текстовом виде. Например, выполнив в командной строке команду:

```
dumpbin.exe hello.exe /all
```

вы получите следующие данные (для краткости мы показали лишь основные элементы, которые хотели проиллюстрировать):

```
Microsoft (R) COFF/PE Dumper Version 7.00.9344.1
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file hello.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES  [MS-DOS/COFF HEADERS]
  14C machine (x86)
   3 number of sections
   . . .

OPTIONAL HEADER VALUES  [PE HEADER]
  10B magic # (PE32)
  . . .

SECTION HEADER #1  [SECTION DATA]
  . . .
  Code
  Execute Read

RAW DATA #1
  . . .

  clr Header:
    . . .

  Section contains the following imports:
    mscoree.dll
      402000 Import Address Table
      4022B8 Import Name Table
        . . .

          0 _CorExeMain
```

Взглянув на этот текстовый дамп PE-файла .NET, можно увидеть, что PE-файл начинается с заголовков MS-DOS и COFF, которые должны содержаться в любой программе. За этими заголовками следует PE-заголовок для поддержки 32-разрядных программ Windows. Сразу за PE-заголовком находится первый раздел данных исполняемого файла. В PE-файле среды .NET это раздел (SECTION HEADER #1, как пока-

зано выше), в котором хранится заголовок и данные CLR. Заметьте, что он помечен как `Core` и `Execute Read`, благодаря чему загрузчик ОС и CLR знают, что эта секция содержит код, который должен исполняться CLR во время запуска программы.

Можно заметить, что в CLR-заголовке импортируется функция с именем `_CorExeMain`, реализованная в *mscoree.dll*, ядре механизма исполнения CLR.[1] На время написания этих строк Windows 98, 2000 и Me имели загрузчик, который знал, как загружать только стандартные PE-файлы. Чтобы избежать внесения значительных изменений в эти операционные системы и при этом обеспечить в них запуск .NET-приложений, Microsoft обновила загрузчики для всех этих платформ. Новые загрузчики знают, как проверить наличие заголовка CLR и, если такой заголовок существует, исполняют `_CorExeMain`, таким образом, не только передавая управление CLR, но и отдаваясь в ее руки. Как вы можете догадаться, ваша функция Main() будет в конечном итоге вызвана из CLR.

Рассмотрев содержимое заголовка CLR, перейдем к содержимому данных CLR, включающих метаданные и код, которые, бесспорно, являются наиболее важными элементами в .NET.

Метаданные

Метаданные (*metadata*) – это данные о ресурсах, или «данные о данных», предназначенные для чтения машиной. Это могут быть подробные сведения о содержимом, формате, размере или других характеристиках источника данных. В .NET метаданные включают определения типов, сведения о версии, ссылки на внешние сборки и другую стандартизованную информацию.

Для того чтобы две системы, два компонента или два объекта могли взаимодействовать друг с другом, по крайней мере один из них должен что-то знать о другом. В COM это «что-то» – спецификация интерфейса, реализуемая поставщиком компонента и используемая его клиентами. Спецификация интерфейса содержит прототипы методов с полными сигнатурами, включая определения типов для всех параметров и возвращаемых значений.

Только разработчики C/C++ могли легко модифицировать и использовать определения типов на языке Interface Definition Language (IDL, язык описания интерфейса), что было недоступно для разработчиков

[1] Мы советуем вам запустить *dumpbin.exe* и просмотреть данные экспорта *mscoree.dll*. Вы обнаружите также `_CorDllMain`, `_CorExeMain`, `_CorImageUnloading` и другие интересные экспортируемые функции. Интересно отметить, что эта DLL представляет собою внутрипроцессный COM-сервер, свидетельствуя о том, что .NET создана с привлечением технологии COM.

на VB и других языках и, что более важно, для утилит и связующего программного обеспечения (middleware). Поэтому в Microsoft придумали нечто отличное от IDL, что мог бы использовать каждый, под названием *библиотека типов* (*type library*). В COM библиотеки типов позволяют утилите или среде разработки читать, разбирать и создавать классы-оболочки, наиболее подходящие и удобные для данного разработчика. Библиотеки типов также позволяют механизмам времени выполнения, таким как библиотеки времени выполнения VB, COM, MTS или COM+, определять типы во время выполнения приложений и предоставлять необходимые заплатки или промежуточную поддержку для их использования приложениями. Например, библиотеки типов поддерживают динамические вызовы и позволяют исполняющей системе COM предоставлять универсальный маршалинг[1] для межконтекстных вызовов.

Библиотеки типов в COM исключительно богаты возможностями, однако многие разработчики критикуют их за отсутствие стандартизации. Команда .NET создала новый механизм получения информации о типах. Вместо термина «библиотека типов» в .NET для названия такой информации используется термин «метаданные».

Библиотеки типов «на стероидах»

Если считать, что библиотеки типов – это заголовочные файлы C++ «на стероидах», то метаданные – это библиотеки типов «на стероидах». В .NET метаданные – это общий механизм, применяемый и средой времени выполнения .NET, и компиляторами, и утилитами, или «диалект», на котором они «общаются». В Microsoft .NET метаданные служат для описания всех типов, используемых и предоставляемых данной сборкой .NET. В этом смысле метаданные описывают сборку в деталях, включая ее идентификатор (комбинация имени сборки, версии, культуры и открытого ключа), типы, на которые она ссылается, экспортируемые типы и требования безопасности для исполнения. Предоставляя намного больше информации, чем библиотеки типов, метаданные включают описания сборки и модулей, классов, интерфейсов, методов, свойств, полей, событий, глобальных методов и т. д.

Метаданные предоставляют достаточно информации для любой среды выполнения, утилиты или программы, чтобы выяснить буквально все, что требуется для интеграции компонентов. Давайте посмотрим на краткий список компонентов, использующих метаданные в среде .NET, просто чтобы доказать, что метаданные действительно подобны библиотекам типов «на стероидах».

---

[1] В COM *универсальный маршалинг* (*universal marshaling*) – это обычный способ преобразования всех типов данных. Универсальный маршалер может использоваться для преобразования любых типов данных, поэтому вам не требуется писать собственный код посредников и заглушек.

CLR

CLR использует метаданные для верификации, обеспечения безопасности, межконтекстного маршалинга, распределения памяти и исполнения. CLR активно полагается на метаданные для поддержки этих средств времени выполнения, которые мы обсудим в свое время.

Загрузчик классов

Компонент CLR, загрузчик классов, использует метаданные для обнаружения и загрузки классов .NET. Это становится возможным потому, что в метаданных содержится подробная информация по конкретному классу, – о том, где этот класс находится, в той же ли он сборке, внутри или вне определенного пространства имен или в подчиненной сборке где-то в сети.

JIT-компиляторы

JIT-компиляторы применяют метаданные для компиляции кода Microsoft Intermediate Language (IL). IL – это промежуточное представление, вносящее значительный вклад в поддержку языковой интеграции, но не являющееся ни кодом VB, ни байт-кодом, который должен интерпретироваться. В .NET JIT компилирует IL перед исполнением в двоичный код и делает это с использованием метаданных.

Инструментальные средства

Инструментальным средствам метаданные нужны для поддержки интеграции. Например, инструментальные средства разработчика могут, основываясь на метаданных, генерировать процедуры-оболочки, обеспечивающие взаимодействие компонентов .NET и COM. Такие средства, как отладчики, профайлеры, браузеры объектов, могут при помощи метаданных обеспечивать более обширную поддержку разработки. Один из примеров – средства IntelliSense, поддерживаемые в Microsoft Visual Studio.NET. Как только вы набрали имя объекта и точку, данное средство выводит список методов и свойств, из которого можно выбрать требуемое. Благодаря этому не приходится искать имена методов или свойств и синтаксис их вызова в заголовочных файлах или документации.

Как и CLR, любое приложение, инструментальное средство или утилита, способные читать метаданные из сборки .NET, могут использовать эту сборку. Классы-отражения в Microsoft .NET Framework помогут исследовать PE-файл .NET и узнать все о типах данных, которые используются и предоставляются сборкой. В CLR применяется тот же набор классов-отражений для просмотра и предоставления функций времени выполнения, включая управление памятью, средства безопасности, проверки типов, отладки, удаленного управления и т. д.

Метаданные гарантируют возможность межъязыкового взаимодействия, основного элемента .NET, т. к. все языки должны использовать

одни и те же типы, чтобы сгенерировать корректный PE-файл для .NET, которая не сможет во время выполнения поддерживать функции управления памятью, безопасности, распределения памяти, проверки типов, отладки и т. д. без метаданных. Вот почему метаданные – крайне важная часть .NET, столь важная, что мы спокойно можем утверждать, что без метаданных не было бы .NET.

Получение метаданных

В этом разделе мы представим важное инструментальное средство .NET, дизассемблер IL (*ildasm.exe*), позволяющий увидеть метаданные и код IL из заданного PE-файла .NET. Например, если запустить *ildasm.exe* и открыть PE-файл .NET *hello.exe*, созданный ранее в этой главе, то получится что-то похожее на рис. 2.3:

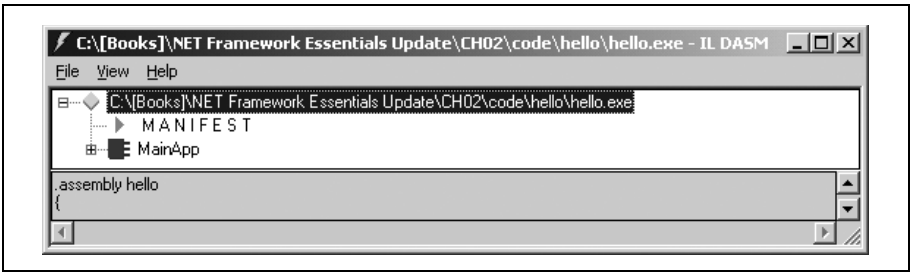


***Рис. 2.3.** Утилита ildasm.exe*

Утилита *ildasm.exe* показывает метаданные PE-файла в виде дерева, так что легко можно перейти от сборок к классам, методам и т. д. Чтобы увидеть полное содержимое PE-файла .NET, можно нажать <Ctrl>+<D> и вывести содержимое в текстовый файл.[1] Вот пример дампа, полученного с помощью программы *ildasm.exe* (мы покажем только те части, которые имеют значение для нашего обсуждения):

```
.assembly extern mscorlib
{
}
.assembly hello
{
}
.module hello.exe
// MVID: {6181DB3C-D7B0-4897-B8F7-013540F29313}
.class private auto ansi beforefieldinit MainApp
       extends [mscorlib]System.Object
```

[1] Утилита *ildasm.exe* также поддерживает и интерфейс командной строки. Чтобы узнать параметры командной строки, выполните команду `ildasm /h`. Также стоит отметить, что если вы хотите в точности узнать, какие типы определены и используются в исполняемом файле, нажмите <Ctrl>+<M> в графическом интерфейсе *ildasm.exe*, и вы увидите эти подробности.

```
{
  .method public hidebysig static
          void Main() cil managed
  {
  } // конец метода MainApp::Main

  .method public hidebysig specialname rtspecialname
          instance void .ctor() cil managed
  {
  } // конец метода MainApp::.ctor

} // конец метода MainApp
```

Как видите, в этом дампе приведена полная информация о типах и зависимостях .NET-сборки. Первая инструкция IL, `.assembly extern`, говорит нам о том, что данный PE-файл ссылается на внешнюю сборку (т. е. использует ее) под названием *mscorlib*; вторая описывает нашу сборку под названием `hello`. Содержимое блоков `.assembly`, все вместе называемое *манифестом*, мы обсудим позднее. После манифеста располагается инструкция, сообщающая нам об имени модуля, *hello.exe*, который имеет GUID (глобально уникальный идентификатор).

Далее следует определение класса на языке IL, начинающееся с IL-инструкции `.class`. Заметьте, что этот класс, MainApp, является производным от класса System.Object, родительского для всех классов среды .NET. Хотя мы явно не писали на Managed C++, C# или VB.NET о том, что наследуем MainApp от System.Object, компилятор автоматически добавил это указание за нас, поскольку System.Object представляет собой неявного родителя всех классов, для которых не указан базовый класс.

В этом классе вы видите два метода. Если первый метод, Main(), – это статический метод, который мы написали ранее, то второй, .ctor(), был сгенерирован автоматически. Main() выступает в качестве основной точки входа вашего приложения, а .ctor() – это конструктор, позволяющий любому создавать экземпляр класса MainApp.

Как видно из этого примера, при наличии PE-файла .NET мы можем исследовать все содержащиеся в нем метаданные. Важно помнить, что это можно сделать без наличия исходного кода или заголовочных файлов. Если вы можете сделать это сами, представьте, какие замечательные возможности может предложить CLR или утилита стороннего разработчика, просто разумно использующая метаданные. Правда, теперь каждый сможет увидеть ваш код, если только вы не применяете различные приемы (например, шифрование) для защиты своих авторских прав.

Просмотр и генерация метаданных

Чтобы загрузить и просмотреть сборку .NET для определения поддерживаемых ею типов, применяется набор классов, предоставляемых .NET Framework. В отличие от функций API, эти классы инкапсулиру-

ют несколько методов, предоставляющих простой интерфейс для просмотра метаданных и манипуляции ими. В среде .NET все вместе эти классы называются *Reflection API* и включают классы пространств имен System.Reflection и System.Reflection.Emit. Классы пространства имен System.Reflection позволяют просматривать метаданные .NET-сборки, как это показано в следующем примере:

```
using System;
using System.IO;
using System.Reflection;

public class Meta
{
  public static int Main()
  {
    // Сначала загрузим сборку.
    Assembly a = Assembly.LoadFrom("hello.exe");

    // Получаем список модулей, поддерживаемых сборкой.
    Module[] m = a.GetModules();

    // Получаем список типов первого модуля.
    Type[] types = m[0].GetTypes();

    // Исследуем первый тип.
    Type type = types[0];
    Console.WriteLine("В типе [{0}] имеются следующие методы:", type.Name);

    // Исследуем методы, поддерживаемые этим типом.
    MethodInfo[] mInfo = type.GetMethods();
    foreach ( MethodInfo mi in mInfo )
    {
      Console.WriteLine("  {0}", mi);
    }

    return 0;
  }
}
```

Посмотрев на эту простую программу на C#, вы заметите, что сначала мы сообщаем компилятору о своем желании использовать классы пространства имен System.Reflection, чтобы иметь возможность посмотреть метаданные. В Main() мы загружаем сборку по ее физическому имени *hello.exe*, поэтому убедитесь, что при запуске программы этот PE-файл находится в том же каталоге. Далее мы запрашиваем у загруженного объекта сборки массив модулей, содержащихся в нем. Из массива модулей мы извлекаем массив типов, поддерживаемых модулями, а из массива типов мы затем извлекаем первый тип. Для *hello.exe* первым и единственными типом оказывается MainApp. Получив этот тип или класс, мы выполняем цикл по всем предоставляемым им методам. Если вы скомпилируете и выполните эту простую программу, то увидите следующий результат:

```
В типе [MainApp] имеются следующие методы:
  Int32 GetHashCode()
  Boolean Equals(System.Object)
  System.String ToString()
  Void Main()
  System.Type GetType()
```

Хотя мы писали только функцию Main(), наш класс на самом деле поддерживает еще четыре метода, что ясно видно по результату работы программы. Здесь нет никакого фокуса, т. к. MainApp унаследовал реализации этих методов от System.Object, являющегося корневым для всех классов .NET.

Как видите, работать с классами System.Reflection не трудно, и они позволяют просматривать метаданные. Те, кто имеет опыт применения интерфейсов библиотек типов в COM, знают, что это можно делать и в COM, однако со значительно большими усилиями. Но при помощи интерфейса библиотек типов COM нельзя создать COM-компонент во время выполнения, – эта возможность отсутствует в COM и, к счастью, присутствует в .NET. На основе простых классов System.Reflection.Emit можно написать программу для динамической генерации .NET-сборки. При наличии System.Reflection.Emit любой может написать собственный .NET-компилятор.

Поддержка взаимодействия компонентов

Предоставляя общий формат для задания типов, метаданные обеспечивают возможность взаимодействия различных компонентов, утилит и систем времени выполнения. Как было показано выше, можно просматривать метаданные любой .NET-сборки. Во время выполнения также можно запросить объект о его типе, методах, свойствах, событиях и т. п. Утилиты могут делать то же самое. С Microsoft .NET SDK поставляется четыре полезных инструментальных средства, помогающих в обеспечении взаимодействия компонентов, в том числе утилита регистрации сборок (*RegAsm.exe*), утилита экспорта библиотек типов (*tlbexp.exe*), утилита импорта библиотек типов (*tlbimp.exe*) и утилита определения XML-схем (*xsd.exe*).

Вы можете использовать утилиту регистрации .NET-сборок для регистрации сборки в реестре, чтобы ее могли использовать COM-клиенты. Утилита экспорта библиотек типов – это средство, генерирующее файл библиотеки типов (*.tlb*) на основании переданной ему .NET-сборки. Сгенерировав библиотеку типов, вы можете импортировать ее в VC++ или VB и использовать .NET-сборку точно так же, как и COM-компонент. Проще говоря, утилита экспорта библиотек типов «маскирует» .NET-сборку под COM-компонент. Следующая командная строка генерирует библиотеку типов с именем *hello.tlb*:

```
tlbexp.exe hello.exe
```

Microsoft также поставляет утилиту, по выполняемой функции противоположную *tlbexp.exe* – утилиту импорта библиотек типов; ее работа состоит в том, чтобы COM-компонент мог притвориться .NET-сборкой. Поэтому если вы разрабатываете .NET-приложение и хотите применять старый COM-компонент, обратитесь к утилите импорта библиотек типов для преобразования информации о типах, имеющийся в COM-компоненте, в эквивалентные данные для .NET. Например, вы можете сгенерировать .NET PE-файл с помощью следующей команды:

```
tlbimp.exe COMServer.tlb
```

В результате выполнения этой команды будет сгенерирована .NET-сборка в виде DLL-файла (например, *COMServer.dll*). На эту DLL можно ссылаться из .NET-кода так же, как на любую другую .NET-сборку. При исполнении .NET-кода все вызовы методов и свойств из этой библиотеки будут перенаправляться в исходный COM-компонент.

Имейте в виду, что утилита импорта библиотек типов не позволит заново импортировать библиотеку типов, ранее экспортированную утилитой экспорта библиотек типов. Другими словами, попытка использовать *tlbimp.exe* с файлом *hello.tlb*, сгенерированным *tlbexp.exe*, будет проигнорирована программой *tlbimp.exe*.

Другая впечатляющая утилита, поставляемая в .NET SDK, – это утилита определения XML-схем, которая позволяет вам преобразовывать XML-схему в класс C# и наоборот. В следующей XML-схеме:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 targetNamespace="urn:book:car"
 xmlns:t="urn:book:car">
  <element name="car" type="t:CCar"/>
  <complexType name="CCar">
    <all>
      <element name="vin" type="string"/>
      <element name="make" type="string"/>
      <element name="model" type="string"/>
      <element name="year" type="int"/>
    </all>
  </complexType>
</schema>
```

представлен тип, названный CCar. Чтобы преобразовать эту XML-схему в определение класса на C#, выполните следующую команду:

```
xsd.exe /c car.xsd
```

Параметр `/c` сообщает утилите о желании сгенерировать класс из данного XSD-файла. После выполнения этой команды в качестве результата будет получен файл *car.cs*, содержащий код на C# для этого типа.

Утилита определения XML-схем также может, используя .NET-сборку, сгенерировать определение XML-схемы (XSD), в которой представлены типы, содержащиеся в .NET-сборке. Например, в результате выполнения следующей команды будет получен XSD-файл:

```
xsd.exe somefile.exe
```

Заканчивая эту тему, хотим посоветовать вам самостоятельно опробовать рассмотренные утилиты, поскольку они предоставляют много ценных возможностей, которые мы не раскрыли в этой вводной книге.

Сборки и манифесты

Как мы только что видели, типы должны предоставлять свои метаданные, чтобы утилиты и программы могли обращаться к ним и пользоваться предоставляемыми ими сервисами. Но одних метаданных недостаточно. Чтобы упростить программный plug-and-play, а также настройку и установку компонента или программного обеспечения, нам также потребуются метаданные о компонентах, которые содержат эти типы. Сейчас мы поговорим о сборках .NET (единицах развертывания) и манифестах (метаданных, описывающих сборки).

Сборки и компоненты

В эпоху COM в документации Microsoft термин *компонент* (*component*) непоследовательно применялся для обозначения как класса COM, так и COM-модуля (DLL или EXE), заставляя читателей или разработчиков каждый раз рассматривать контекст, окружающий термин. В .NET Microsoft разрешила эту путаницу, введя новую концепцию *сборки* (*assembly*), являющейся компонентом программного обеспечения, поддерживающим plug-and-play, во многом аналогично аппаратному компоненту. Теоретически .NET-сборка примерно эквивалентна COM-модулю. Практически сборка может содержать несколько типов и физических файлов (в том числе файлов растровой графики, PE-файлов .NET и т. д.), необходимых для ее успешного запуска, или ссылаться на них. Кроме хранения IL-кода сборка является базовой единицей контроля версий, развертывания, управления безопасностью, одновременного исполнения версий, совместного и повторного использования кода, что мы обсудим далее.

Для справки: сборка – это логический DLL- или EXE-файл, а манифест – описание (метаданные) сборки (в том числе ее версия, используемые в ней другие сборки и т. д.).

Уникальная идентификация

Уникальность типов важна в RPC, COM и .NET. Наличие огромного числа идентификаторов GUID в COM (идентификаторы приложения,

библиотеки, класса и интерфейса) делало разработку и развертывание трудоемкими, т. к. всегда и везде в коде приходилось использовать эти магические цифры. В .NET ссылки на определенный тип основываются на понятном имени и его пространстве имен. Поскольку удобочитаемого имени и его пространства имен недостаточно для обеспечения глобальной уникальности, среда .NET гарантирует уникальность с помощью уникальных пар открытого и закрытого ключа. Все сборки, совместно используемые многими приложениями (и называемые *совместно используемыми* или *разделяемыми сборками* (*shared assemblies*)), должны содержать пару из открытого и закрытого ключа. Пары открытого и закрытого ключа применяются в криптографии с открытым ключом. Поскольку этот вид криптографии использует ассиметричное шифрование, создатель сборки может подписать сборку своим закрытым ключом, а любой другой – проверить эту цифровую подпись с помощью открытого ключа создателя сборки.

Для цифровой подписи сборки необходимо при создании последней использовать открытый и закрытый ключи. На этапе компиляции компилятор сгенерирует хеш-код файлов сборки, подпишет его с помощью закрытого ключа и сохранит получившуюся цифровую подпись в зарезервированном разделе PE-файла. Открытый ключ будет также сохранен в сборке.

Чтобы проверить цифровую подпись сборки, CLR нужен открытый ключ сборки для расшифровки ее цифровой подписи и получения исходного, рассчитанного ранее хеш-кода. Кроме того, CLR опирается на информацию из манифеста сборки для динамической генерации хеш-кода. Сгенерированное значение сравнивается с исходным значением хеш-кода. Эти значения должны совпадать, в противном случае предполагается, что сборка поддельная.

Теперь, когда мы знаем, как подписать и проверить подлинность сборки в среде .NET, поговорим о том, как CLR гарантирует, что данное приложение загружает именно ту сборку, с которой оно компилировалось. Когда вы или кто-то еще компилирует приложение, для работы которого требуется совместно используемая сборка, в манифест сборки приложения будет включен 8-байтовый хеш-код, полученный с помощью открытого ключа совместно используемой сборки.[1] При запуске приложение динамически вычисляет 8-байтовый хеш-код по открытому ключу сборки и сравнивает его со значением, хранящимся в

[1] Для генерации новой пары ключей разделяемой сборки применяется утилита .NET Strong Name (она же .NET Shared name) *sn.exe*. Прежде чем предоставить доступ к своей сборке, зарегистрируйте ее в глобальном кэше сборок (Global Assembly Cache, GAC) (см. раздел этой главы «Параллельное исполнение различных версий») с помощью утилиты глобального кэша .NET-сборок (*gacutil.exe*). GAC – это просто каталог под названием *Assembly*, расположенный в системном каталоге Windows (`%windir%`), который в Windows 2000 обычно называется *WINNT*.

манифесте сборки приложения. Если эти значения совпадают, CLR предполагает, что была загружена корректная версия сборки.

IL-код

Сборка содержит IL-код (см. раздел «Промежуточный язык IL» далее в этой главе), который исполняется CLR во время выполнения приложения. IL-код обычно использует или ссылается на типы, определенные в той же сборке, однако это могут быть и типы из других сборок. Если в первом случае ничего особенного не требуется, то во втором в сборке должны быть определены ссылки на другие сборки, что мы скоро увидим. Есть только одно предостережение: все сборки должны иметь как минимум одну точку входа, такую как DllMain(), WinMain() или Main(). Это правило необходимо соблюдать, т. к. при загрузке сборки CLR ищет одну из этих точек входа, чтобы начать исполнение сборки.

Контроль версий

В .NET существует четыре типа сборок:

Статические сборки

В .NET существуют PE-файлы, создаваемые при компиляции. Статические сборки могут быть созданы с помощью вашего любимого компилятора: *csc*, *cl* или *vbc*.

Динамические сборки

Это сборки в памяти, имеющие формат PE, динамически создаваемые во время выполнения с помощью классов пространства имен System.Reflection.Emit.

Закрытые сборки

Это статические сборки, используемые конкретным приложением.

Совместно используемые или разделяемые сборки

Это статические сборки, которые должны иметь уникальное разделяемое имя и доступны любому приложению.

Приложение использует закрытую сборку, ссылаясь на нее с использованием статического пути или через конфигурационный XML-файл приложения. Хотя CLR не навязывает определенной стратегии в отношении контроля версий (проверки на правильность версии) для закрытых сборок, для совместно используемых сборок CLR гарантирует, что приложение использует корректные сборки, с которыми оно компилировалось. Таким образом, приложение ссылается на совместно используемую сборку, а CLR гарантирует, что во время выполнения будет загружена корректная версия.

В .NET сборка является минимальной единицей, которой может быть присвоен номер версии, имеющий следующий формат:

<основная_версия>.<подверсия>.<номер_построения>.<ревизия>

Развертывание

Поскольку манифест сборки клиентского приложения (что мы кратко обсудим далее) содержит информацию о внешних ссылках – в том числе местоположение внешней сборки и версию сборки, используемую приложением, – больше не требуется хранить в реестре указания по активизации компонентов и маршалингу, как в COM. Основываясь на информации о версии и безопасности, записанной в манифесте приложения, CLR загрузит корректную совместно используемую сборку. CLR выполняет «ленивую» (отложенную) загрузку внешних сборок и извлечет их код по запросу, когда приложению потребуются типы из них. По этой причине можно уменьшать размер загружаемых приложений с помощью многочисленных небольших внешних сборок. Когда потребуется определенная внешняя сборка, среда времени выполнения загрузит ее автоматически, не требуя регистрации или перезагрузки компьютера.

Безопасность

Концепция идентификации пользователя является общей для всех платформ разработки и операционных систем, однако концепция *идентичности кода* (*code identity*), в соответствии с которой может идентифицироваться даже фрагмент кода, нова для индустрии коммерческого программного обеспечения. В среде .NET сама сборка представляет собой единицу идентификации кода, включающую такую информацию, как имя совместно используемой сборки, номер версии, информацию о культуре (для локализации приложений) и открытый ключ. Опираясь на эту концепцию, CLR может проверить, имеет ли сборка полномочия на доступ к системным ресурсам и на вызовы других сборок.

Для того чтобы соответствовать концепции идентичности кода, CLR поддерживает концепцию *управления доступом к коду* (*code access*). Другими словами, среда времени выполнения определяет возможность доступа к определенной сборке на основе набора разрешений. CLR проверяет эти разрешения и определяет, удовлетворять ли запросы на исполнение на уровне сборки. При создании сборки можно указать набор разрешений, которые клиентское приложение должно иметь для доступа к вашей сборке. Во время выполнения, если клиентское приложение имеет права на доступ к коду вашей сборки, оно может выполнять вызовы ее объектов, а в противном случае оно не сможет их использовать.

Одновременное исполнение версий

Мы говорили, что сборка – это единица контроля версий и развертывания, и кратко обсудили DLL Hell – проблему, сведение которой к ми-

нимуму является задачей среды .NET. CLR позволяет любым версиям одной и той же разделяемой DLL (совместно используемой сборки) работать одновременно, на одной системе и даже в одном процессе. Эта концепция известна как *одновременное исполнение версий* (*side-by-side execution*). Microsoft .NET осуществляет одновременное исполнение версий, применяя средства контроля версий и развертывания, присущие всем совместно используемым сборкам. Эта концепция позволяет устанавливать различные версии одной совместно используемой сборки на одной машине без DLL Hell или конфликтов версий. Следует только учесть, что сборки должны быть совместно используемыми или разделяемыми, т. е. что необходимо зарегистрировать их в GAC при помощи утилиты глобального кэша .NET-сборок (*gacutil.exe*). После регистрации различных версий одной совместно используемой сборки в GAC пользовательское имя сборки уже не имеет значения – имеет значение только информация, предоставляемая средствами контроля версий и развертывания .NET.

Вспомним, что при компиляции приложения, использующего определенную разделяемую сборку, информация о версии сборки присоединяется к манифесту приложения. Кроме того, 8-байтовый хеш-код открытого ключа совместно используемой сборки также присоединяется к манифесту приложения. Основываясь на этой информации, CLR может найти в точности ту совместно используемую сборку, которая необходима данному приложению, и даже убедиться, что указанный 8-байтовый хеш-код действительно совпадает с хеш-кодом совместно используемой сборки. Если CLR сможет идентифицировать и загружать в точности ту сборку, которая требуется, внедрение среды .NET будет означать, что конец «ада DLL» близок.

Совместное и повторное использование кода

Для того чтобы использовать свою сборку совместно с остальным миром, необходимо дать ей разделяемое (или строгое) имя и зарегистрировать ее в GAC. Точно так же, если требуется использовать или расширить определенный класс, находящийся в определенной сборке, необходимо не просто импортировать этот класс, а импортировать в приложение всю сборку. Следовательно, единицей совместного использования кода является сборка в целом.

Сборки оказались исключительно важным средством в .NET, т. к. являются существенной частью среды времени выполнения. В .NET сборка инкапсулирует все типы, определенные в ней. Например, две различные сборки, Personal и Company, могут определять и предоставлять один и тот же тип Car. Сам тип Car не имеет смысла, не будучи квалифицированным как [Personal]Car или [Company]Car. Таким образом, все типы определены только в масштабе содержащей их сборки, и потому CLR не может воспользоваться определенным типом, если не знает, к какой сборке относится этот тип. Фактически, если от-

сутствует манифест сборки, содержащий ее описание, CLR не выполнит программу.

Манифесты: метаданные сборки

Манифест сборки – это метаданные, целиком описывающие сборку, включая ее идентификатор, список файлов, относящихся к сборке, ссылки на внешние сборки, экспортируемые типы и ресурсы, а также требуемые разрешения. Короче говоря, в нем описаны все детали, необходимые для осуществления компонентного plug-and-play. Поскольку сборка содержит все эти детали, не требуется хранить данную информацию в реестре, как принято в мире COM.

В технологии COM при использовании определенного COM-класса необходимо сообщить COM-библиотеке идентификатор класса. COM-библиотека выполняет в реестре поиск COM-компонента, предоставляющего этот класс, загружает компонент, запрашивает компонент о получении экземпляра этого класса и возвращает ссылку на экземпляр. В .NET вместо поиска в реестре CLR обращается непосредственно к манифесту сборки, определяет, какие требуются внешние сборки, загружает именно ту сборку, которая необходима вашему приложению, и создает экземпляр класса.

Рассмотрим манифест приложения *hello.exe*, созданного нами ранее. Вспомните, что эту информацию мы получали при помощи утилиты *ildasm.exe*.

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:3200:0
}

.assembly hello
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module hello.exe
// MVID: {F828835E-3705-4238-BCD7-637ACDD33B78}
```

Как видите, этот манифест начинается с идентификации внешней или ссылочной сборки с именем `mscorlib`, на которую ссылается данное приложение. Ключевые слова `.assembly extern` сообщают CLR, что это приложение не реализует `mscorlib`, а лишь использует ее. Эта внешняя сборка необходима всем приложениям, и вы увидите ее в манифесте всех других сборок. Обратите внимание, что внутрь определения этой сборки компилятор вставил специальное значение под названием *publickeytoken*, представляющее собой основную информацию об издателе `mscorlib`. Компилятор генерирует значение для `.publickeytoken`, вычис-

ляя хеш-функцию открытого ключа, связанного со сборкой `mscorlib`. Также в блоке `mscorlib` можно заметить номер ее версии.[1]

Теперь, после рассмотрения первого блока `.assembly`, давайте изучим второй, в котором описана данная конкретная сборка. Понять, что этот блок манифеста описывает сборку вашего приложения, можно по отсутствию ключевого слова `extern`. Идентификатор этой сборки складывается из ее пользовательского имени (`hello`), информации о версии (`0.0.0.0`) и необязательной (в данном случае отсутствующей) информации о культуре. В данном блоке в первой строке задается хеш-алгоритм, применяемый для вычисления хеш-функции определенных частей содержимого сборки, результат которой будет зашифрован с использованием закрытого ключа. Однако поскольку эта простая сборка не будет совместно использоваться другими приложениями, шифрование не используется, и значение `.publickey` отсутствует.

Последнее, что мы обсудим – это ключевое слово `.module`, которое просто идентифицирует выходное имя файла этой сборки, *hello.exe*. Можно заметить, что модуль связан с GUID, т. е. вы будете получать различные значения GUID при каждом построении модуля. Следовательно, элементарной проверкой на точное соответствие модуля будет сравнение GUID двух модулей.

По причине простоты этого примера мы больше ничего не смогли получить из данного манифеста. В более сложной сборке мы получили бы все то же самое плюс значительно большее количество глубинных деталей о ее составе.

Создание сборок

Сборка может быть либо *одномодульной* (*single-module*) либо *многомодульной* (*multi-module*). В одномодульной сборке все, что требуется собрать, размещено в одном EXE- или DLL-файле; примером может быть приложение *hello.exe*, разработанное нами раньше. Такую сборку легко создать, потому что компилятор сам заботится о ее создании.

Для создания многомодульной сборки, содержащей много модулей и файлов с ресурсами, есть несколько вариантов. Первый – применение утилиты связывания сборок Assembly Linker (*al.exe*), предоставляемой .NET SDK. Эта утилита формирует файл с манифестом сборки из одного или более IL-файлов или файлов ресурсов.

---

[1] Замечательные подробности описаны в файлах документации *Partition II Metadata.doc* и *Partition III CIL.doc*, поставляемых с .NET SDK. Если вы действительно хотите разобраться с языком IL и метаданными, прочитайте эти документы.

Использование сборок

Для того чтобы со сборкой можно было работать, ее необходимо сначала импортировать в код с помощью синтаксиса, зависящего от используемого языка. Например, в C# библиотека импортируется, как мы уже видели ранее в этой главе, следующим образом:

```
using System;
```

При создании сборки следует сообщить компилятору, что вы ссылаетесь на внешнюю сборку. Способ зависит от того, с каким компилятором вы работаете. Если это компилятор C#, это делается следующим образом:

```
csc /r:mscorlib.dll hello.cs
```

Ранее мы показали, как компилировать *hello.cs* без параметра `/r`, однако оба метода эквивалентны. Присутствие ссылки на *mscorlib.dll* неявно подразумевается, т. к. в ней содержатся все базовые классы среды .NET.

Промежуточный язык IL

В проектировании программного обеспечения концепция *абстракции* (*abstraction*) исключительно важна. Мы часто применяем абстракцию для скрытия сложности системы или прикладных сервисов, предоставляя потребителю простой интерфейс к ним. Сохраняя интерфейс неизменным, мы можем изменять скрытые внутренние детали, а различные клиенты смогут по-прежнему использовать тот же интерфейс.

По ходу развития языков программирования ученые предлагали различные воплощения уровней языковой абстракции, таких как *p-код* и *байт-код*. Генерируемый компилятором Pascal-P p-код представляет собою промежуточный язык, поддерживающий процедурное программирование. Генерируемый компиляторами Java байт-код – это промежуточный язык, поддерживающий объектно-ориентированное программирование. Байт-код – это языковая абстракция, позволяющая коду Java работать на различных платформах при условии, что на этих платформах имеется виртуальная Java-машина (JVM) для его исполнения.

Microsoft называет свой уровень языковой абстракции Common Intermediate Language (CIL, общий промежуточный язык). Так же как и байт-код, IL поддерживает все средства объектно-ориентированного программирования, включая абстракцию данных, наследование, полиморфизм и такие полезные концепции, как исключения и события. Кроме этих возможностей IL поддерживает и другие концепции, например свойства, поля и перечисления. Любой язык среды .NET может быть преобразован в IL, поэтому .NET поддерживает несколько

языков и, возможно, в дальнейшем будет поддерживать несколько платформ (при условии, что на целевых платформах будет CLR).

В поставляемом с .NET SDK файле *Partition III CIL.doc* описаны важные инструкции IL, которые должны использоваться компиляторами. Кроме этой спецификации .NET SDK включает и другой важный документ – *Partition II Metadata.doc*. Оба эти документа предназначены для разработчиков, пишущих компиляторы и утилиты, однако вам также следует прочитать их, чтобы еще лучше понять, какое место IL занимает в .NET. Конечно, можно создать .NET-сборку с помощью инструкций и средств IL, но это покажется вам очень трудоемким, т. к. его инструкции несколько непонятны. Однако тому, кто решит написать собственный IL-код, следует воспользоваться IL-ассемблером (*ilasm.exe*), чтобы преобразовать IL-код в PE-файл .NET.[1]

Но хватит теории: давайте взглянем на какой-нибудь IL-код. Вот отрывок из IL-кода программы *hello.exe*, написанной нами раньше:[2]

```
.class private auto ansi beforefieldinit MainApp
  extends [mscorlib]System.Object
{
  .method public hidebysig static
          void Main() cil managed
  {
    .entrypoint
    .maxstack  1
    ldstr "C# hello world!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
  } // end of method MainApp::Main

  .method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
  {
    .maxstack  1
    ldarg.0
    call instance void [mscorlib]System.Object::.ctor()
    ret
  } // end of method MainApp::.ctor

} // end of class MainApp
```

Проигнорировав странные детали синтаксиса, можно видеть, что IL концептуально схож с любым другим объектно-ориентированным

[1] Чтобы проверить эту утилиту с помощью IL-дизассемблера, загрузите PE-файл .NET и извлеките из него IL-код в текстовый файл. Сделав это, воспользуйтесь IL-ассемблером для преобразования текстового файла обратно в PE-файл .NET.

[2] Не компилируйте этот IL-код, он не является полным, т. к. мы убрали непонятные детали, чтобы сделать его проще для понимания. Если вы хотите увидеть полный IL-код, воспользуйтесь *ildasm.exe* для файла *hello.exe*.

языком. Ясно, что здесь есть класс под названием MainApp, являющийся производным от System.Object. Этот класс поддерживает статический метод Main(), содержащий код, выводящий текстовую строку на консоль. Хотя мы и не писали конструктор для этого класса, наш компилятор C# добавил конструктор по умолчанию для MainApp, чтобы он поддерживал создание объекта.

Поскольку длинное обсуждение IL выходит за рамки этой книги, обратимся к методу Main(), чтобы кратко рассмотреть его реализацию. Прежде всего, вы можете видеть следующую сигнатуру метода:

```
.method public hidebysig static
        void Main() cil managed
```

В этой сигнатуре объявлен метод, являющийся открытым (public, т. е. он может вызываться кем угодно) и статическим (т. е. является методом на уровне класса). Имя этого метода – Main(). Он содержит IL-код, который управляется или исполняется CLR. Атрибут `hidebysig` говорит о том, что этот метод скрывает такие же методы (с той же сигнатурой), определенные ранее в иерархии классов. Для большинства объектно-ориентированных языков это просто подразумевается по умолчанию. Разобравшись с сигнатурой метода, поговорим о самом теле метода:

```
{
  .entrypoint
  .maxstack 1
  ldstr "C# hello world!"
  call void [mscorlib]System.Console::WriteLine(string)
  ret
} // end of method MainApp::Main
```

Этот метод использует две директивы: `.entrypoint` и `.maxstack`. Директива `.entrypoint` указывает, что Main() – это первая и единственная точка входа для сборки. Директива `.maxstack` определяет максимальное количество слотов стека, необходимых для этого метода; в данном случае максимальное количество слотов стека, требуемое Main(), равно единице. Информация о стеке необходима для любого метода в IL, т. к. IL-инструкции являются стековыми, что позволяет компиляторам легче генерировать IL-код.

Кроме этих директив в методе использованы три IL-инструкции. Первая инструкция, `ldstr`, загружает строку литералов в стек, чтобы код, расположенный в том же блоке, мог ее использовать. Следующая IL-инструкция, `call`, вызывает метод WriteLine(), выбирающий строку из стека. IL-инструкция `call` предполагает наличие аргументов метода в стеке, при этом первый аргумент должен быть помещен в стек первым, второй аргумент – вторым и т. д. Кроме того, при использовании инструкции `call` для вызова метода необходимо указать его сигнатуру. Рассмотрев, например, сигнатуру метода WriteLine():

```
void [mscorlib]System.Console::WriteLine(string)
```

мы увидим, что WriteLine() – это статический метод класса Console. Класс Console принадлежит пространству имен System, являющемуся частью сборки *mscorlib*. Метод WriteLine() получает строку (это псевдоним объекта System.String) и возвращает `void`. Последнее, что мы отметим в отношении этого фрагмента IL-кода, это что IL-инструкция `ret` просто возвращает управление вызвавшей стороне.

CTS и CLS

Поняв важность метаданных и языка IL, давайте рассмотрим CTS и CLS. И CTS и CLS служат для обеспечения совместимости, возможности взаимодействия и интеграции различных языков.

Общая система типов CTS

Поскольку в среде .NET все языки рассматриваются как равные, класс, написанный на C#, должен быть эквивалентен классу, написанному на VB.NET, а интерфейс, определенный в Managed C++, должен быть полностью аналогичен интерфейсу, определенному в управляемом Коболе. Языки должны договориться о значениях этих концепций перед тем, как смогут взаимодействовать друг с другом. Чтобы воплотить в реальность языковую интеграцию, фирма Microsoft определила общую систему типов, которой должен придерживаться любой язык в .NET. В этом разделе мы выделим общие типы, имеющие одинаковую концептуальную семантику в любом языке среды .NET. Microsoft .NET поддерживает богатый набор типов, но мы ограничим наше обсуждение самыми важными из них, в частности размерными типами, ссылочными типами, классами, интерфейсами и делегатами.

Размерные типы

Вообще CLR поддерживает две различных категории типов: размерные типы и ссылочные типы. *Размерные типы* (*value types*) представляют значения, расположенные в стеке. Они не могут быть пустыми (null) и обязаны содержать какие-то данные. Такие данные передаются в функцию по значению, т. е. перед выполнением функции создается копия этого значения. Из этого следует, что исходное значение не меняется, независимо от того, что происходит с его копией в процессе вызова функции. Внутренние типы невелики по размеру и не занимают много памяти, поэтому затраты ресурсов на создание копии незначительны и предпочтительнее, чем падение производительности при управлении объектом и сборке мусора. Размерные типы включают примитивы, структуры и перечисления; примеры приведены в следующем листинге C#:

```
int i;                      // примитив
struct Point { int x, y; } // структура
enum State { Off, On }      // перечисление
```

Можно также создать размерный тип, породив производный класс от System.ValueType. Также необходимо отметить, что такой размерный тип является *изолированным* (*sealed*), т. е. если вы создали производный класс от System.ValueType, никто другой уже не сможет создать класс, производный от вашего класса.

Ссылочные типы

Хотя ссылочные типы требуют значительных ресурсов памяти, ссылочный тип предоставляет преимущества по сравнению с размерным. *Ссылочные типы* (*reference types*) (в том числе объекты, интерфейсы и указатели) называются так, потому что содержат ссылки на объекты, расположенные в куче, и могут иметь пустое значение (null). Эти типы передаются по ссылке, т. е. когда вы передаете такой объект в функцию, передается адрес или указатель на объект, а не копия объекта, как в случае размерного типа. Поскольку передается ссылка, вызывающая сторона увидит, что вызванная функция сделала с вашим объектом. Первое преимущество состоит в том, что ссылочный тип может использоваться как выходной параметр, а второе – в том, что не тратятся лишние ресурсы, т. к. не создается копия объекта. Если ваш объект велик (занимает большой объем памяти), ссылочные типы являются лучшим выбором. В .NET один из недостатков ссылочного типа заключается в том, что память для него выделяется в управляемой куче, поэтому требуется больше тактов процессора, т. к. CLR должна управлять и осуществлять сборку мусора для этого объекта. В .NET наиболее близкой к деструкции концепцией является финализация, но в отличие от деструкторов в C++, финализация недетерминирована. Другими словами, неизвестно, когда произойдет финализация, т. к. она выполняется при запуске сборщика мусора (по умолчанию, когда в системе заканчивается память). Поскольку финализация недетерминирована, еще один недостаток ссылочных типов состоит в том, что если объекты ссылочного типа занимают дорогие ресурсы, которые будут освобождены во время финализации, страдает производительность системы, т. к. ресурсы не будут освобождены до тех пор, пока объекты не будут удалены сборщиком мусора. Ссылочные типы включают классы, интерфейсы, массивы и делегаты; их примеры показаны в следующем листинге на C#:

```
class Car {}              // класс
interface ISteering {}    // интерфейс
int[] a = new int[5];     // массив
delegate void Process(); // делегат
```

Классы, интерфейсы и делегаты мы обсудим немного позднее.

Упаковка и распаковка объектов

Поддержка размерных типов включена в Microsoft .NET в целях повышения производительности, однако все в .NET в конечном счете яв-

ляется объектом. На самом деле все примитивные типы имеют соответствующие классы в .NET Framework. Например, `int` представляет собой псевдоним System.Int32, а System.Int32 оказывается производным от System.ValueType, т. е. это размерный тип. Размерные типы по умолчанию располагаются в стеке, но они всегда могут быть преобразованы в ссылочный тип, располагающийся в куче, и эта операция называется *упаковкой* (*boxing*). Следующий фрагмент кода показывает, как упаковать копию значения переменной `i`:

```
int i = 1;          // i - размерный тип
object box = i;     // box - ссылочный объект
```

Когда мы упаковываем значение, то получаем объект, для которого можно вызывать методы, свойства и события. Например, преобразовав целое в объект, как показано в этом фрагменте кода, вы можете вызывать методы, определенные в System.Object, в том числе ToString(), Equals() и т. д.

Операцией, обратной упаковке, естественно, является *распаковка* (*unboxing*), которая представляет собой преобразование объекта ссылочного типа, расположенного в куче, в эквивалентный ему размерный тип, как показано ниже:

```
int j = (int)box;
```

В этом примере оператор `cast` применяется для преобразования типа объекта под названием `box`, расположенного в куче, в целое число.

Классы, свойства, индексаторы

CLR предоставляет полноценную поддержку концепций объектно-ориентированного программирования (инкапсуляции, наследования и полиморфизма) и возможностей классов (методов, полей, статических членов, видимости, доступности, вложенных типов и т. п.). Кроме того, CLR поддерживает новые возможности, отсутствующие во многих традиционных объектно-ориентированных языках программирования, включая свойства, индексаторы и события.[1] События описаны в главе 8. А сейчас кратко поговорим о свойствах и индексаторах.

Свойства похожи на поля (переменные-члены), за исключением того, что для них имеются методы-установщики (setters) и методы-получатели (getters), как показано ниже:

```
using System;

public class Car
{
```

[1] Событие – это обратный вызов (callback), реализуемый через делегатов, о которых мы вскоре расскажем.

```
  private string make;
  public string Make
  {
    get { return make; }
    set { make = value; }
  }

  public static void Main()
  {
    Car c = new Car();
    c.Make = "Acura";  // используется установщик
    String s = c.Make; // используется получатель
    Console.WriteLine(s);
  }
}
```

Хотя, скорее всего, вы в первый раз видите такой синтаксис, этот пример прост и практически не требует пояснений, за исключением ключевого слова `value`. Это специальное ключевое слово, обозначающее первый и единственный аргумент метода-установщика.

Сходный со свойством синтаксически, индексатор аналогичен конструкции `operator[]` в C++, т. к. позволяет осуществлять доступ к содержимому объекта подобно обращению к массиву. Другими словами, индексатор позволяет обращаться к объекту так, как будто это обращение к массиву, что показано в следующем примере:

```
using System;

public class Car
{
  Car()
  {
    wheels = new string[4];
  }

  private string[] wheels;
  public string this[int index]
  {
    get { return wheels[index]; }
    set { wheels[index] = value; }
  }


  public static void Main()
  {
    Car c = new Car();
    c[0] = "LeftWheel";  // c[0] может быть как с правой, так
                         // и с левой стороны присваивания
    Console.WriteLine(c[0]);
 }
}
```

Еще одно замечание перед тем, как мы закончим эту тему: в отличие от C++, но аналогично Java классы в .NET не поддерживают множественное наследование.

Интерфейсы

Интерфейсы поддерживают в точности ту же концепцию, что и абстрактные базовые классы (ABC) в C++, содержащие только чисто виртуальные функции. ABC – это класс, в котором объявлена одна или несколько чисто виртуальных функций, и потому экземпляр такого класса невозможно создать. Для тех, кто знаком с COM или Java, скажем, что интерфейсы в .NET концептуально эквивалентны интерфейсу COM или Java. Их можно определить, но нельзя реализовать. Интерфейс должен реализовываться производным от него классом. Интерфейс может содержать методы, свойства, индексаторы и события. В .NET класс может быть производным от нескольких интерфейсов.

Делегаты

Одним из наиболее мощных средств языка C является его поддержка указателей на функции. Указатели на функции позволяют создавать программное обеспечение с обработчиками, которые могут быть реализованы кем-то другим. Действительно, указатели на функции позволяют многим людям создавать расширяемое и настраиваемое программное обеспечение. Microsoft .NET поддерживает типизированную версию указателей на функции, называемых *делегатами* (*delegates*). Вот пример, который, возможно, потребует от вас нескольких минут для погружения в него, однако как только вы в нем разберетесь, то поймете, что он на самом деле прост:

```
using System;
class TestDelegate
{
  // 1. Определяем прототип для обратного вызова
  delegate void MsgHandler(string strMsg);

  // 2. Определяем метод обратного вызова
  void OnMsg(string strMsg)
  {
    Console.WriteLine(strMsg);
  }

  public static void Main()
  {
    TestDelegate t = new TestDelegate();

    // 3. Привязка вашего метода обратного вызова
    MsgHandler f = new MsgHandler(t.OnMsg);

    // 4. Неявный вызов метода обратного вызова
    f("Hello, Delegate.");
  }
}
```

Первое, что мы делаем, это определяем прототип функции обратного вызова, и здесь важным ключевым словом является `delegate`, которое сообщает компилятору, что вам требуется объектно-ориентированный указатель на функцию. При этом подспудно компилятор генерирует вложенный класс, MsgHandler, производный от System.MulticastDelegate.[1] Широковещательный делегат поддерживает много приемников. Определив прототип, необходимо определить и реализовать метод с сигнатурой, соответствующей этому прототипу. Затем просто выполните привязку метода обратного вызова, передав функцию в конструктор делегата, как показано в этом листинге. И наконец, неявно вызовите ваш метод обратного вызова. Разобравшись с делегатами, заметим, что делегаты образуют основу событий, обсуждаемых в главе 8.

Общая языковая спецификация CLS

Задача среды .NET состоит в такой поддержке языковой интеграции, чтобы программы можно было писать на любом языке, и при этом они могли бы взаимодействовать друг с другом, пользуясь всеми преимуществами наследования, полиморфизма, обработки исключений и другими возможностями. Однако языки созданы неодинаковыми, и разные языки поддерживают разные возможности. Например, Managed C++ чувствителен к регистру, а VB.NET – нет. Чтобы составить из всего этого единое произведение, Microsoft опубликовала спецификацию Common Language Specification (CLS). В CLS определен набор базовых правил, обязательных для языковой интеграции. Поскольку Microsoft предоставила CLS, содержащую минимальные требования к языкам .NET, поставщики компиляторов могут, придерживаясь этой спецификации, разрабатывать компиляторы для .NET. Кроме создателей компиляторов, прочитать CLS также следует разработчикам приложений и использовать эти правила, чтобы гарантировать возможность межъязыкового взаимодействия.

Выполнение в среде CLR

Теперь, когда вы понимаете, из каких элементов состоит исполняемый файл .NET, поговорим о сервисах, предоставляемых CLR для поддержки управления и исполнения .NET-сборок. В CLR имеется много потрясающих компонентов, однако для краткости мы ограничим наше обсуждение только основными из них, показанными на рис. 2.4.

Основными компонентами CLR являются загрузчик классов, верификатор, JIT-компиляторы и средства поддержки исполнения, такие как управление кодом, безопасностью, сборка мусора, управление исклю-

[1] Если вы хотите это увидеть, воспользуйтесь *ildasm.exe* и посмотрите метаданные представленного нами примера *delegate.exe*.

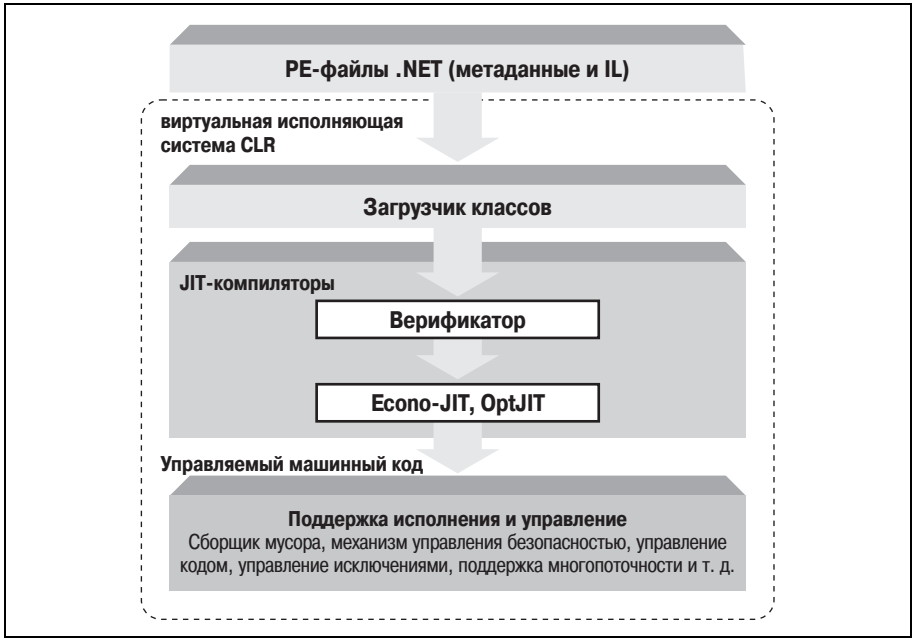


Рис. 2.4. *Основные компоненты CLR: виртуальная исполняющая система (VES)*

чениями, отладкой, маршалингом, потоками и т. д. Как видно из рис. 2.4, PE-файлы .NET расположены поверх CLR и исполняются в рамках виртуальной исполняющей системы (VES) CLR, содержащей основные компоненты среды времени выполнения. Ваши PE-файлы .NET должны пройти через загрузчик классов, верификатор типов, JIT-компиляторы и другие компоненты поддержки исполнения перед тем, как они будут выполнены.

Загрузчик классов

Когда запускается стандартное Windows-приложение, загрузчик ОС загружает его перед тем, как сможет выполнить. На момент написания этого текста загрузчики в существующих операционных системах Windows 98/Me/2000 распознают только стандартные PE-файлы Windows. В результате Microsoft пришлось предоставить обновленные загрузчики для каждой из этих операционных систем, поддерживающие среду времени исполнения .NET. Обновленные загрузчики ОС понимают формат PE-файлов .NET и могут обращаться с файлом должным образом.

При запуске .NET-приложения на одной из этих систем, имеющих обновленный загрузчик, загрузчик распознает .NET-приложение и потому передает управление CLR. После этого CLR находит точку входа, которой обычно является функция Main(), и передает ей управление,

чтобы начать работу приложения. Однако перед тем как сможет выполниться функция Main(), загрузчик классов должен найти класс, предоставляющий Main(), и загрузить этот класс. Кроме того, когда Main() создает объект определенного класса, снова подключается загрузчик класса. Короче говоря, загрузчик классов выполняет свою работу при первой ссылке на класс.

Загрузчик классов (*class loader*) загружает .NET-классы в память и готовит их для исполнения. Прежде чем он сможет это сделать, он должен найти требуемый класс. Загрузчик классов ищет его в нескольких разных местах, в том числе в конфигурационном файле приложения в текущем каталоге (*.config*), в GAC, в метаданных, являющихся частью PE-файла, в частности в манифесте. Информация, предоставляемая одним или несколькими источниками, является ключевой для поиска нужного класса. Вспомните, что класс ограничен определенным пространством имен, пространство имен ограничено определенной сборкой, а сборка ограничивается определенной версией. Поэтому два класса, оба имеющие имя Car, рассматриваются как различные классы, даже если информация о версиях их сборок совпадает.

После того как загрузчик классов нашел и загрузил необходимый класс, он кэширует информацию о типах этого класса, чтобы класс не пришлось загружать снова в процессе работы. Сохранив эту информацию в кэше, он позднее определит, сколько требуется выделить памяти для нового экземпляра класса. Когда необходимый класс загружен, загрузчик вставляет маленькую заглушку, вроде пролога функции, в каждый метод загруженного класса. Эта заглушка предназначена для того, чтобы отмечать состояние JIT-компиляции и для перехода между управляемым и неуправляемым кодом. На этом этапе, если загруженный класс ссылается на другие классы, загрузчик также попытается загрузить эти классы. Однако если классы, указанные в ссылках, уже были загружены, загрузчику ничего делать не надо. И наконец, загрузчик классов использует соответствующие метаданные для инициализации статических переменных и создания экземпляра загруженного класса.

Верификатор

Языки сценариев и интерпретируемые языки очень снисходительны к применению типов, позволяя писать код без явных объявлений переменных. Эта гибкость приводит к созданию кода, чрезмерно подверженного ошибкам и сложного в поддержке, и она часто виновна в таинственных программных сбоях. В отличие от языков сценариев и интерпретируемых языков, компилируемые языки требуют явного определения типов перед их использованием, позволяя компилятору обеспечить корректную работу с типами и спокойное выполнение кода.

Ключом к этому является безопасность типов, и это фундаментальная концепция верификации кода в .NET. Верификатор – это компонент

VES, функционирующий во время выполнения приложения для проверки кода на предмет безопасности типов. Заметьте, что верификация типов происходит во время выполнения, и в этом состоит фундаментальное различие между .NET и другими оболочками. Осуществляя проверку типов во время работы программы, CLR может предотвратить исполнение кода, не являющегося безопасным в отношении типов, и гарантирует, что код используется так, как это предполагалось. Короче говоря, безопасность типов означает улучшенную надежность.

Поговорим о месте, которое занимает верификатор внутри CLR. После того как загрузчик классов загружает класс, но перед тем, как фрагмент IL-кода будет исполнен, верификатор внедряется в проверяемый код. Верификатор отвечает за следующие проверки:

- являются ли метаданные корректными и действительными;
- безопасен ли IL-код в отношении типов, т. е. корректно ли используются сигнатуры типов.

Код, который будет исполнен, должен отвечать обоим критериям, т. к. JIT-компиляция будет выполнена только тогда, когда код и метаданные успешно пройдут верификацию. Кроме проверки на безопасность типов верификатор также выполняет элементарный анализ управляющей логики кода, чтобы обеспечить корректное использование типов в коде. Заметьте также, что, поскольку верификатор является частью JIT-компиляторов, он подключается только при вызове метода, а не при загрузке класса или сборки. Кроме того, обратите внимание, что верификация это необязательный этап, т. к. доверенный код никогда не проходит верификацию, а сразу передается для компиляции JIT-компилятору.

JIT-компиляторы

JIT-компиляторы играют важную роль в платформе .NET, т. к. все PE-файлы .NET содержат IL-код и метаданные, а не двоичный код. JIT-компиляторы преобразуют IL в двоичный код, чтобы он мог выполняться в целевой операционной системе. Для каждого метода, успешно проверенного на безопасность типов, JIT-компилятор CLR компилирует метод и преобразует его в управляемый машинный код. Требуется именно управляемый машинный код, т. к. только им могут управлять и только его могут исполнять компоненты поддержки выполнения в целевой операционной системе.

Одно из преимуществ JIT-компилятора заключается в том, что он может динамически создавать код, оптимизированный для целевой машины. Если перенести один и тот же PE-файл .NET с однопроцессорной машины на двухпроцессорную, JIT-компилятор на двухпроцессорной машине будет знать о втором процессоре и получит возможность выдать двоичный код, использующий преимущества второго процессора. Другое очевидное преимущество состоит в том, что можно

взять один и тот же PE-файл .NET и запустить его на абсолютно другой платформе, независимо от того, Windows ли это, Unix или другая операционная система, если только на этой платформе есть CLR.

По соображениям оптимизации JIT-компиляция происходит лишь при первом вызове метода. Вспомните, что при загрузке класса загрузчик классов добавляет в каждый метод заглушку. При первом вызове метода VES читает информацию из этой заглушки, которая сообщает о том, что код метода не был скомпилирован JIT-компилятором. При наличии этого признака JIT-компилятор компилирует метод и вставляет в эту заглушку адрес управляемого двоичного кода. При последующих вызовах этого же метода JIT-компиляция не требуется, т. к. каждый раз, когда VES обращается за информацией в заглушку, она видит адрес кода скомпилированного метода. Поскольку JIT-компилятор делает свое дело только при первом вызове метода, методы, не требующиеся во время выполнения приложения, никогда не будут скомпилированы JIT-компилятором.

Скомпилированный двоичный код занимает место в памяти до тех пор, пока процесс не закроется и сборщик мусора не очистит все ссылки и области памяти, связанные с процессом. Это значит, что в следующий раз при выполнении процесса или компонента JIT-компилятор снова проделает ту же работу.

Для того чтобы исключить накладные расходы, связанные с JIT-компиляцией во время выполнения, можно обратиться к специальной утилите ngen, компилирующей ваш IL-код во время установки. Применение ngen позволяет выполнить JIT-компиляцию кода один раз и кэшировать его на компьютере, чтобы избежать JIT-компиляции во время выполнения (этот процесс называется pre-JITting, или предварительной компиляцией). В том случае, если PE-файл был обновлен, необходимо снова выполнить предварительную компиляцию PE-файла. В противном случае CLR обнаружит обновление и динамически скомандует соответствующему JIT-компилятору скомпилировать сборку.

Поддержка и управление исполнением

Как видите, все компоненты CLR, рассмотренные нами до сих пор, тем или иным образом используют метаданные IL для успешного осуществления поддерживаемых ими функций. Кроме предоставления метаданных и генерации управляемого кода, JIT-компилятор должен генерировать управляемые данные, необходимые диспетчеру кода для поиска и разворачивания стековых фреймов.[1] *Диспетчер кода* (*code manager*) использует управляемые данные для управления исполнением

[1] Кстати, вы можете создать собственный JIT-компилятор или диспетчер кода для CLR, поскольку в ней поддерживается подключение подобных компонентов.

кода, в том числе перемещения по стеку, требуемые для обработки исключений, проверок безопасности и сборки мусора. Кроме диспетчера кода CLR также предоставляет несколько важных сервисов для поддержки и управления исполнением. Подробное обсуждение этих сервисов выходит за рамки данной книги, поэтому здесь мы лишь кратко перечислим некоторые из них:

Сборка мусора

В отличие от C++, где приходится вручную удалять все расположенные в куче объекты, CLR поддерживает автоматическое управление временем жизни всех объектов среды .NET. Сборщик мусора может определить, что на объекты больше нет ссылок и выполнить сборку мусора для освобождения неиспользуемой памяти.

Обработка исключений

До появления .NET не было согласованного метода обработки ошибок или исключений, что создавало много проблем в обработке ошибок и в оповещении о них. В .NET CLR поддерживает стандартный механизм обработки исключений, работающий во всех языках, позволяя всем программам использовать общий механизм обработки ошибок. Механизм обработки исключений в CLR интегрирован со структурной обработкой исключений (Windows Structured Exception Handling, SEH).

Поддержка безопасности

На этапе выполнения CLR осуществляет различные проверки безопасности, чтобы гарантировать, что код безопасен для исполнения и не нарушает никаких требований безопасности. Кроме поддержки безопасности доступа к коду механизм безопасности также поддерживает декларативные и императивные проверки. Декларативная безопасность не требует специального кода, однако разработчик должен указать свои требования по безопасности в атрибутах или в административной конфигурации. Императивная безопасность требует написания в методе кода, явно вызывающего проверки безопасности.

Поддержка отладки

CLR предоставляет богатую поддержку отладки и профилирования. Это API, который может применяться разработчиками компиляторов для создания отладчиков. Он содержит поддержку управления исполнением программы, точек останова, исключений, управляющей логики и т. п. Существует также API для поддержки профилирования работающих программ.

Поддержка взаимодействия кода

CLR поддерживает взаимодействие между управляемым (CLR) и неуправляемым (без CLR) «мирами». Средство *COM Interop* играет роль моста, соединяющего COM и CLR, обеспечивая COM-объекту

возможность использовать .NET-объект, и наоборот. Средство *Platform Invoke* (*P/Invoke*) позволяет вызывать функции Windows API.

Этот список никак нельзя считать исчерпывающим. Мы только хотим снова подчеркнуть, что как и загрузчик классов, верификатор, JIT-компиляторы и практически все, что имеет отношение к .NET, все эти средства поддержки и управления исполнением тем или иным образом используют метаданные, управляемый код и управляемые данные для выполнения своих функций.

Заключение

Как вы видели в этой главе, архитектура .NET направлена на поддержку языковой интеграции и компонентного представления приложений всеми разумными способами. Благодаря метаданным процесс программирования значительно облегчается, т. к. теперь нет необходимости беспокоиться о реестре при развертывании компонентов и о других странностях (таких как CoCreateInstanseEx, CLSID, IID, IUnknown, IDL и т. п.) для поддержки компонентного представления. Благодаря CTS, CLS, метаданным и IL теперь имеется реальная языковая интеграция. Microsoft предоставляет версии CLR для разных версий Windows, однако также работает над версией CLR с общедоступными исходными текстами, которая будет работать под FreeBSD и наверняка будет переносимой на другие Unix-подобные системы. Появились также реализации CLR не от Microsoft, включая DotGNU Portable.NET (дополнительную информацию можно найти на сайте *http://www.southern-storm.com.au/portable_net.html*) и Mono (см. *http://www.go-mono.com*). .NET, таким образом, представляет собой многоязычную и многоплатформенную архитектуру.

3

Программирование в среде .NET

Теперь, когда вы знаете, что такое .NET, поговорим о программировании для среды .NET. В этой главе рассмотрены общая модель программирования, предоставляемая .NET, основные языки и возможности, поддерживаемые в .NET, и языковая интеграция, позволяющая пользоваться преимуществами объектно-ориентированных средств даже между разными языками, что возможно в среде .NET.

Общая модель программирования

Без среды .NET программистам приходится выбирать среди богатства API или библиотек, поддерживающих системные службы. Например, для того чтобы написать графическое приложение для Windows, есть масса возможностей для выбора, в том числе Win32 API, MFC, ATL, VB и т. д. Выбрав библиотеку, необходимо изучить, как используются структуры, классы, функции, интерфейсы и прочие возможности, предоставляемые библиотекой. К сожалению, это знание не переносится напрямую в другую среду. Например, существует большая разница между кодом управления вводом/выводом в MFC и кодом управления вводом/выводом в VB.

Одна из задач .NET Framework состоит в унификации разработки приложений путем предоставления структуры общих классов для разработчиков, применяющих компиляторы, генерирующие IL-код. Эта общая среда крайне полезна: если вы знаете, как реализовать функциональность ввода/вывода в .NET с помощью своего любимого языка, то сможете легко перенести этот код на другой язык. Это становится возможным, поскольку пространства имен, классы, методы и т. д. имеют согласованное представление во всех языках. Например, мож-

но вывести строку текста на консоль одинаковым для всех .NET-языков образом, посредством метода WriteLine() объекта Console, как мы уже видели ранее в этой книге. Эта согласованная структура требует меньше времени на обучение и позволяет увеличить производительность работы программиста.

Поскольку полное обсуждение всего набора классов .NET Framework выходит за рамки этой книги (см. серию издательства O'Reilly по .NET – «In a Nutshell»), мы поговорим о классе System.Object и представим основные пространства имен .NET Framework, открыв вам двери для шага в этот мир.

System.Object

Каждый тип в .NET – это объект, т. е. он должен прямо или косвенно наследоваться от класса Object. Если при определении класса базовый класс не указывается, компилятор учитывает это требование при генерации IL-кода. Класс Object поддерживает общие возможности, наследуемые всеми .NET-классами и автоматически предоставляемые их клиентам. Класс Object предоставляет открытые методы, перечисленные в табл. 3.1, которые могут быть вызваны любым .NET-объектом во время выполнения.

Таблица 3.1. Открытые методы класса Object

Метод	Описание
`Equals()`	Сравнивает два объекта и определяет их эквивалентность (совпадение содержимого).
`ReferenceEquals()`	Сравнивает две ссылки на объект и определяет, ссылаются ли они на один и тот же объект.
`GetHashCode()`	Получает хеш-код объекта. В среде .NET хеш-коды играют роль дополнительного механизма определения уникальности объектов во время выполнения. Например, для того чтобы объекты использовались в качестве ключей хеш-таблицы, необходимо переопределить эту функцию и предоставить уникальные хеш-значения для каждого экземпляра вашего класса.
`GetType()`	Получает тип объекта во время выполнения. По типу конкретного объекта можно получить любую информацию об этом типе с помощью Reflection API, как объясняется в главе 2.
`ToString()`	Формирует строковое представление объекта. Обычно применяется для отладочных целей, по умолчанию выдает полностью квалифицированное имя класса.

Рассмотрим следующую программу, иллюстрирующую применение всех этих методов:

```
using System;
namespace Cpm
```

```
{
  class CPModel
  {
    public static void Main()
    {
      CPModel c = new CPModel();
      // Проверка на самоэквивалентность
      Console.WriteLine("Эквивалентность:\t" +
        c.Equals(c)
      );
      // Получаем хеш-код этого объекта
      Console.WriteLine("Хеш-код объекта:\t" +
        c.GetHashCode()
      );
      // Исспользуем тип для получения информации о методе
      Console.WriteLine("Метод объекта:\t" +
        c.GetType().GetMethods()[1]
      );
      // Преобразуем объект в строку
      Console.WriteLine("Имя объекта:\t" +
        c.ToString()
      );
    }
  }
}
```

Если скомпилировать и запустить эту программу на C#, то будет получен следующий результат:

```
Эквивалентность: True
Хеш-код объекта:3
Метод объекта:Boolean Equals(System.Object)
Имя объекта:    Cpm.CPModel
```

В строке, выделенной жирным шрифтом, показан второй метод класса CPModel. Вернувшись к коду программы, мы увидим, что для получения типа применяется метод GetType(), а затем с помощью метода GetMethods() извлекается массив методов, поддерживаемых этим типом. Из этого массива мы извлекаем второй метод, которым оказывается Equals() – метод, реализованный в System.Object.

Как видите, класс System.Object предоставляет механизмы для идентификации типа во время выполнения, для проверки эквивалентности и для инспектирования всех .NET-объектов.

Основные пространства имен

В табл. 3.2 приведен краткий список важнейших пространств имен и классов .NET Framework, предоставляющих поддержку практически для любого разрабатываемого вами приложения. Это пространства имен, которые вы станете использовать тем чаще, чем больше .NET-

приложений вы разработаете. За дополнительной информацией обратитесь к MSDN Online или к документации SDK, т. к. подробное обсуждение этих пространств имен и классов выходит за рамки книги.

Таблица 3.2. Важные пространства имен и классов среды .NET

Пространство имен	Описание
`System`	Включает базовые классы, которые будут использоваться почти всеми программами. Вот некоторые простые классы, относящиеся к этому пространству имен: Object, Char, String, Array и Exception. Это пространство имен включает и более сложные классы, такие как GC и AppDomain.
`System.IO`	Предоставляет набор классов для поддержки синхронного и асинхронного ввода/вывода потоков данных. Также предоставляет классы для операций с файловой системой, таких как создание, изменение и удаление файлов и каталогов. Некоторые из этих классов – FileStream, MemoryStream, Path и Directory.
`System.Collections`	Включает набор классов, позволяющих управлять коллекциями объектов. Вот некоторые из этих классов: ArrayList, DictionaryBase, Hashtable, Queue и Stack.
`System.Threading`	Включает набор классов, поддерживающих многопоточное программирование. Некоторые из этих классов – Thread, ThreadPool, Mutex и AutoResetEvent.
`System.Reflection`	Включает набор классов, поддерживающих динамическое связывание и просмотр типов. Некоторые из этих классов – Assembly, Module и MethodInfo.
`System.Security`	Включает набор классов и дочерних пространств имен, предоставляющих поддержку безопасности. Интересными дочерними пространствами имен являются Cryptography, Permissions, Policy и Principal.
`System.Net`	Включает набор классов и дочерних пространств имен, предоставляющих поддержку сетевого программирования. Эти классы включают IPAddress, Dns и HttpWebRequest.
`System.Data`	См. главу 5.
`System.Web.Services`	См. главу 6.
`System.Web.UI`	См. главу 7.
`System.Windows.Forms`	См. главу 8.

Имейте в виду, что, зная, как применять какой-либо из классов этих пространств имен, можно писать использующий их код на любом языке, т. к. имена методов и классов согласованы для всех языков среды .NET.

Базовые средства и языки

Поскольку одной из целей .NET является поддержка общей парадигмы прикладного программирования, в ней должны согласованно определяться и использоваться концепции программирования. В этом разделе мы рассмотрим базовые языки Microsoft .NET, в том числе Managed C++, VB.NET и C# и несколько базовых концепций программирования, которые поддерживаются всеми языками в .NET, в том числе:

Пространства имен

Смягчают конфликты имен.

Интерфейсы

Определяют методы и свойства, которые должны быть реализованы объектами, предоставляющими интерфейс.

Инкапсуляция

В объектно-ориентированных языках позволяет классу инкапсулировать все свои данные и поведение.

Наследование

Позволяет классу быть наследником родительского класса, так что он может использовать богатую функциональность, реализованную родительским классом, сокращая затрачиваемые на разработку усилия и ошибки программирования.

Полиморфизм

Позволяет разработчикам указывать или реализовывать в базовом классе манеру поведения, которая может быть перекрыта производным классом. Это очень мощное средство, благодаря которому разработчики могут выбрать во время выполнения правильную манеру поведения на основе типа объекта.

Обработка исключений

Позволяет писать понятный код, т. к. обеспечивает перехват всех ошибок по общей и понятной схеме, полностью противоположной девяти уровням вложенных условных конструкций.

Хотя это и не полный список поддерживаемых в .NET концепций, он включает все основные концепции, которые мы хотим рассмотреть в этом разделе. Мы также покажем вам примеры всех этих возможностей в Managed C++, VB.NET и C#. Эти концепции не представляют собой ничего нового, мы просто демонстрируем, как они реализованы во всех языках среды .NET.

Прежде чем начать, мы должны понять, что будут делать наши примеры. Прежде всего, мы создадим пространство имен под названием Lang, которое инкапсулирует интерфейс ISteering. Затем мы создадим два класса: Vehicle, который является абстрактным базовым классом, реализующим интерфейс ISteering, и Car, производный от Vehicle. Мы

создадим точку входа, которая создает экземпляр класса Car и использует его внутри блока try. Другие детали мы раскроем в процессе работы над примерами.

Код на Managed C++

Managed C++ – это язык программирования Microsoft C++, в который добавлены некоторые новые ключевые слова и возможности для поддержки программирования в среде .NET. Это позволяет применять для разработки управляемых объектов, т. е. объектов, работающих в CLR, язык C++. Managed C++ позволяет достичь производительности[1], присущей программам C++, и при этом воспользоваться преимуществами средств CLR.[2]

Взглянем теперь на пример, включающий все концепции, которые мы хотим рассмотреть. Как видно в следующем листинге, мы начинаем с создания нового пространства имен, Lang, в которое включено все, кроме main(). За исключением первых двух строк и специальных ключевых слов, этот код полностью соответствует стандарту языка C++:

```
#using <mscorlib.dll>
using namespace System;

namespace Lang
{
```

Далее мы определяем интерфейс с именем ISteering. Если вы программировали на C++, то сразу заметите, что в следующем фрагменте имеется два новых ключевых слова, __gc и __interface. Новое ключевое слово __interface позволяет объявить интерфейс, который в основном эквивалентен абстрактному базовому классу в C++. Другими словами, два прототипа методов здесь определяются, но не реализуются. Реализацию этих методов предоставляет класс, реализующий этот интерфейс:

```
__gc __interface ISteering
{
  void TurnLeft();
  void TurnRight();
};
```

Если вы знакомы с технологией COM, то знаете, что в COM программист должен сам управлять временем жизни своих объектов и компонентов. И хуже того, ему приходится полагаться на то, что клиенты

---

[1] Можно смешивать управляемый и неуправляемый код в программах на C++. Неуправляемый код будет работать быстрее.

[2] Однако если вы внимательно посмотрите на возможности и ключевые слова (__abstract, __box, __delegate, __gc, __nogc, __pin и т. д.), добавленные в Microsoft C++, то вряд ли захотите использовать Managed C++ для написания нового кода для CLR, особенно при наличии C#.

корректно договариваются и взаимодействуют с его COM-компонентами, иначе остающиеся ссылки никогда не будут освобождены. Managed C++ избавляется от этой проблемы путем добавления нового ключевого слова, __gc. Это ключевое слово сообщает CLR о необходимости осуществлять сборку мусора для ссылок на ваш интерфейс, когда они более не используются. Не считая этих двух ключевых слов, предыдущий фрагмент кода не требует больше никаких пояснений для программистов, имеющих опыт работы с C-подобными языками.

Теперь, когда у нас есть интерфейс, давайте его реализуем. Следующий фрагмент кода – это класс «Managed C++ (на что указывает ключевое слово __gc), реализующий интерфейс ISteering. Обратите внимание, что этот класс представляет собой абстрактный базовый класс, т. к. метод ApplyBrakes() – это чисто виртуальная (т. е. полиморфная) функция, как это следует из синтаксиса =0. Класс Vehicle не предоставляет реализации данного метода, однако производный класс должен его реализовать:

```
__gc class Vehicle : public ISteering
{
  public:

  void TurnLeft()
  {
    Console::WriteLine("Транспортное средство поворачивает налево.");
  }

  void TurnRight()
  {
    Console::WriteLine("Транспортное средство поворачивает направо.");
  }

  virtual void ApplyBrakes() = 0;
};
```

Поскольку Vehicle является абстрактным базовым классом и его экземпляр нельзя создать, мы должны создать класс, производный от Vehicle, который назовем Car. Как видно в следующем листинге, все в этом классе относится к C++, за исключением ключевого слова __gc. Заметьте, что функция ApplyBrakes() сначала выводит текстовое сообщение на консоль, а потом сразу же создает и вызывает исключение, сообщая обработчику исключений о поломке тормозов. Особенность заключается в том, что класс Exception представляет собой часть .NET Framework и принадлежит к пространству имен System. Это хорошо, т. к. данный класс работает абсолютно одинаково во всех языках, и больше нет необходимости в изобретении собственной иерархии исключений.

```
__gc class Car : public Vehicle
{
  public:

  void ApplyBrakes()
```

```
    {
      Console::WriteLine("Машина пытается остановиться.");
      throw new Exception("Поломка тормозов! ");
    }
};

} // Эта скобка обозначает конец описания пространства имен Lang.
```

Теперь у нас есть конкретный класс, и мы можем написать функцию main() для проверки класса Car. Заметьте, что мы добавили блок try, заключающий в себя наш фрагмент кода, чтобы мы могли обрабатывать все исключения в блоке catch. Посмотрев внимательно на следующий фрагмент кода, вы заметите, что новый экземпляр класса Car создается в управляемой куче, однако для ссылок на него используется указатель типа Vehicle. Затем мы просим транспортное средство повернуть налево – вызываем метод TurnLeft(). Здесь нет ничего удивительного, т. к. мы реализовали этот метод в классе Vehicle. Однако далее мы просим применить тормоз, но в классе Vehicle метод ApplyBrakes() не реализован. Поскольку этот метод виртуальный, будут использованы корректные значения vptr и vtbl[1], что приведет к вызову метода Car::ApplyBrakes. Конечно, Car::ApplyBrakes() генерирует исключительную ситуацию, перенося нас в блок catch. Внутри блока catch мы преобразуем перехваченное исключение в строку, которую и выводим на консоль. Мы можем это сделать, т. к. Exception – класс .NET Framework, а все классы среды являются производными от System.Object, в котором реализована элементарная функция ToString() для преобразования любого объекта в строку.

```
void main()
{
  try
  {
    Lang::Vehicle *pV = 0;  // идентификатор пространства имен
    pV = new Lang::Car();   // pV ссылается на Car
    pV->TurnLeft();         // использование интерфейса
    pV->ApplyBrakes();      // полиморфизм в действии
  }
  catch(Exception *pe)
  {
    Console::WriteLine(pe->ToString());
  }
}
```

Заметьте, что программисту не требуется освобождать объекты из управляемой кучи после завершения их использования, т. к. в .NET за него это сделает сборщик мусора.

---

[1] Многие компиляторы C++ используют переменные vtbl (таблица указателей на функции) и vptr (указатель на vtbl) для поддержки динамического связывания или полиморфизма.

В этом простом примере мы использовали Managed C++, чтобы проиллюстрировать все основные концепции .NET, включая пространства имен, интерфейсы, инкапсуляцию, наследование, полиморфизм и обработку исключений. Далее мы продемонстрируем, как вы можете перевести этот код на любой другой язык .NET, т. к. все эти языки поддерживают данные концепции. В частности, мы приведем тот же пример на VB.NET, C# и IL, только чтобы доказать, что эти концепции могут быть одинаково представлены в любом языке.NET.

Код на VB.NET

Корпорация Microsoft модернизировала язык Visual Basic и добавила в него полноценные средства объектно-ориентированного программирования. Новый язык, Visual Basic.NET, или VB.NET, позволяет делать все, что можно было делать в VB, хотя и значительно более простым образом. Если вы программируете на VB и знаете другие объектно-ориентированные языки, такие как C++ или Smalltalk, тогда вам понравится новый синтаксис, пришедший с VB.NET. Если вы программируете на VB без знаний других объектно-ориентированных языков, то сначала будете удивлены новым синтаксисом VB.NET, однако поймете, что он облегчает жизнь программиста.

VB.NET поддерживает быструю разработку программ (Rapid Application Development, RAD) в стиле Visual Basic и кроме того является модернизированным языком, дающим полный доступ к .NET Framework. Компилятор VB.NET генерирует метаданные и IL-код, делая этот язык равноправным с C# и Managed C++. В отличие от VB, в VB.NET отсутствует интерпретатор, что прекращает шумные обсуждения меньшей производительности VB по сравнению с другими языками.

Возможно, наиболее мощным средством является возможность писать интерфейсы и классы, которые выглядят очень похожими на те, что написаны на других языках .NET. Новый синтаксис позволяет вам выполнять наследование от базовых классов, реализовать интерфейсы, перекрывать виртуальные функции, создавать абстрактные базовые классы и т. д. Кроме того, поддерживается обработка исключений, выполняемая так же, как в C# и Managed C++, что намного облегчает обработку ошибок. И наконец, для VB.NET поставляется компилятор с интерфейсом командной строки, *vbc.exe*, представленный в главе 2.

Посмотрим, как перевести предыдущую программу, написанную на Managed C++, на VB.NET, чтобы вы могли увидеть поразительное концептуальное сходство. Начнем с определения пространства имен под названием Lang, что показано жирным шрифтом:

```
Imports System

Namespace Lang
```

Затем мы определим интерфейс ISteering, что осуществляется в VB.NET с использованием очень простого синтаксиса, особенно по

сравнению с Managed C++. В следующем фрагменте кода вы заметите, что вместо открывающей и закрывающей фигурных скобок, как в Managed C++, начало определения интерфейса обозначается ключевым словом VB.NET `Interface`, а конец – тем же ключевым словом, которое предваряется словом `End`. Это вполне обычный синтаксис в стиле VB и он не будет сюрпризом для любого VB-программиста.

```
Interface ISteering
  Sub TurnLeft()
  Sub TurnRight()
End Interface
```

Определив наш интерфейс, мы можем его реализовать. Наш класс Vehicle представляет собой абстрактный базовый класс, поэтому мы должны добавить при его определении ключевое слово `MustInherit`, явно сообщающее компилятору VB.NET, что экземпляр этого класса не может быть создан. В VB.NET ключевое слово `Class` позволяет определить класс, а ключевое слово `Implements` – реализовать интерфейс. Имейте также в виду, что ApplyBrakes() в этом классе не реализуется, и мы специально сигнализируем об этом компилятору VB.NET с помощью ключевого слова `MustOverride`.

```
MustInherit Class Vehicle
  Implements ISteering

  Public Sub TurnLeft() Implements ISteering.TurnLeft
    Console.WriteLine("Транспортное средство поворачивает налево.")
  End Sub

  Public Sub TurnRight() Implements ISteering.TurnRight
    Console.WriteLine("Транспортное средство поворачивает направо.")
  End Sub

  Public MustOverride Sub ApplyBrakes()

End Class
```

Различие между языками также выражается в том, что необходимо явно задать уровень доступа для каждого метода в отдельности (т. е. `public`, `private` и т. д.), в отличие от C++, где все члены используют ранее определенный тип доступа.

Теперь мы готовы к тому, чтобы перевести конкретный класс Car. В VB.NET можно создать производный класс от базового с помощью ключевого слова `Inherits`, как показано в следующем фрагменте кода. Как мы и говорили, метод ApplyBrakes должен быть перекрыт, и здесь мы предоставим его реализацию. Снова заметьте, что мы генерируем исключение.

```
Class Car
  Inherits Vehicle

  Public Overrides Sub ApplyBrakes()
```

```
        Console.WriteLine("Машина пытается остановиться.")
        throw new Exception("Поломка тормозов!")
      End Sub

    End Class

  End Namespace
```

Теперь у нас есть все части, и мы можем определить модуль с точкой входа Main(), который будет выполнен CLR. Обратите внимание, что в Main() мы обрабатываем исключения точно так же, как делали это в примере на Managed C++. Можно также заметить, что этот код демонстрирует применение полиморфизма, т. к. сначала мы создаем ссылку на Vehicle, которая во время выполнения ссылается на объект Car. Мы просим Vehicle исполнить метод ApplyBrakes(), однако поскольку Vehicle ссылается на Car, выполнять действия будет именно целевой объект Car.

```
Public Module Driver

  Sub Main()

    Try

      Dim v As Lang.Vehicle  ' идентификатор пространства имен
      v = New Lang.Car       ' v ссылается на объект Car
      v.TurnLeft()           ' использование интерфейса
      v.ApplyBrakes()        ' полиморфизм в действии

    Catch e As Exception

      Console.WriteLine(e.ToString())

    End Try

  End Sub

End Module
```

Эта простая программа демонстрирует, что применение языка VB.NET открывает нам доступ к преимуществам объектно-ориентированных средств .NET. Посмотрев на этот пример, вы поймете, что VB.NET – это объектно-ориентированный язык с возможностями, в точности соответствующими возможностям Managed C++ и других языков среды .NET.

Код на C#

Как вы видели, VB.NET гораздо легче, чем Managed C++, однако VB.NET не единственный простой язык в .NET – C# также поразительно прост. Разработанный с нуля, C# поддерживает все объектно-ориентированные средства среды .NET. Он настолько близок к языкам Java и C++, что если у вас есть опыт работы с одним из этих языков, вы можете взяться за C# и немедленно начать эффективно его использовать.

Корпорация Microsoft разработала многие средства с помощью C#; фактически большинство компонентов Visual Studio.NET были разработаны на C#. Microsoft широко применяет C#, и мы думаем, что C# будет языком следующего десятилетия.[1]

Сказав об этом, переведем нашу предыдущую программу на C# и проиллюстрируем все возможности, которые хотим увидеть. Снова начнем с определения пространства имен. Как видите, синтаксис C# действительно близок к синтаксису Managed C++.

```
using System;

namespace Lang
{
```

Далее идет спецификация интерфейса ISteering на C#. Поскольку C# был разработан с нуля, мы не должны добавлять всякие непонятные ключевые слова вроде `__gc` и `__interface`, как мы делали в версии этой программы на Managed C++.

```
interface ISteering
{
  void TurnLeft();
  void TurnRight();
}
```

Определив наш интерфейс, мы можем теперь реализовать его в абстрактном классе Vehicle. В отличие от Managed C++, но аналогично VB.NET, C# требует, чтобы вы явно сообщили компилятору C# о том, что класс Vehicle – это абстрактный базовый класс, посредством ключевого слова `abstract`. Так как ApplyBrakes() – это абстрактный метод, т. е. данный класс не предоставляет его реализации, вы должны создать именно абстрактный класс, иначе компилятор C# будет «ругаться». Другими словами, необходимо явно сообщать компилятору C# о требуемых средствах, в частности с помощью ключевых слов `abstract`, `public`, `private` и т. д., каждый раз, когда вы определяете класс, метод, свойство и т. п.

```
abstract class Vehicle : ISteering
{
  public void TurnLeft()
  {
    Console.WriteLine("Транспортное средство поворачивает налево.");
  }
```

[1] Для того чтобы больше узнать о C#, обратитесь к книгам издательства O'Reilly «C# Essentials» Альбахари, Дрэйтона и Меррила (Albahari, Drayton и Merrill, 2-е издание), «C# in a Nutshell» Дрэйтона, Альбахари и Ньюорда (Drayton, Albahari и Neward) и «Programming C#» Либерти (Liberty, 2-е издание). Последняя книга, «Программирование на C#», в 2002 г. выйдет в издательстве «Символ-Плюс».

```
  public void TurnRight()
  {
    Console.WriteLine("Транспортное средство поворачивает направо.");
  }

  public abstract void ApplyBrakes();
}
```

Затем мы определим класс Car, производный от Vehicle, и перекрывающий метод ApplyBrakes(), объявленный в Vehicle. Заметьте, мы явно сообщаем компилятору C#, что действительно перекрываем метод, ранее определенный в цепочке наследования. Необходимо добавить модификатор override, иначе ApplyBrakes() скроет метод родительского класса. В нашем случае мы, как и раньше, генерируем исключение:

```
class Car : Vehicle
{
  public override void ApplyBrakes()
  {
    Console.WriteLine("Машина пытается остановиться.");
    throw new Exception("Поломка тормозов!");
  }
}

} // Эта скобка обозначает окончание определения
  // пространства имен Lang.
```

И наконец, приведем класс, содержащий точку входа, вызываемую из CLR. Если внимательно посмотреть на код, то можно увидеть, что он точно соответствует как коду на Managed C++, так и коду на VB.NET.

```
class Drive
{
  public static void Main()
  {
    try
    {
      Lang.Vehicle v = null;  // идентификатор пространства имен
      v = new Lang.Car();     // v ссылается на объект Car
      v.TurnLeft();           // использование интерфейса
      v.ApplyBrakes();        // полиморфизм в действии
    }
    catch(Exception e)
    {
      Console.WriteLine(e.ToString());
    }
  }
}
```

Отметим еще два интересных момента в отношении C#. Во-первых, в отличие от C++ и по аналогии с Java, в C# нет заголовочных файлов.[1] Во-вторых, компилятор C# самостоятельно генерирует XML-докумен-

тацию, если в коде есть XML-комментарии. Чтобы воспользоваться этой возможностью, начинайте XML-комментарии с трех косых черт, как в следующих примерах:

```
/// <summary>Класс Vehicle</summary>
/// <remarks>
///   Этот класс является абстрактным классом,
///   который должен быть перекрыт в производных классах.
/// </remarks>
abstract class Vehicle : ISteering
{
  /// <summary>Добавим топливо в вашу машину.</summary>
  /// <param name="gallons">
  ///   Количество добавленных галлонов.
  /// </param>
  /// <return>Полон ли бензобак.</return>
  public bool FillUp(int gallons)
  {
    return true;
  }
}
```

Это простейшие примеры использования предопределенных тегов C#. Также можно включать в XML-комментарии собственные теги XML, учитывая, что полученный в результате XML-документ должен быть корректным. При наличии файла исходного кода с XML-комментариями можно автоматически сгенерировать справочный документ в формате XML с помощью параметра `/doc` компилятора C#:

```
csc /doc:doc.xml mylangdoc.cs
```

Хотя мы и не указали типы наших параметров в приведенных выше XML-комментариях, компилятор C# определит правильные типы и добавит полностью квалифицированные типы в сгенерированный XML-документ. Например, следующий сгенерированный XML-листинг соответствует XML-комментариям для метода FillUp(). Заметьте, что компилятор C# добавил System.Int32 в сгенерированный XML-документ.

```
<member name="M:Lang.Vehicle.FillUp(System.Int32)">
  <summary> Добавим топливо в вашу машину.</summary>
  <param name="gallons">
    Количество добавленных галлонов.
  </param>
  <return>Не полон ли бензобак.</return>
</member>
```

[1] Для тех, кто не знаком с C++, сообщаем, что заголовочный файл не является обязательным и обычно содержит объявления классов и типов. Реализации этих классов обычно хранятся в исходных файлах.

Итак, у нас есть сгенерированный XML-документ, и мы можем написать свой собственный XSL-документ для преобразования XML в любое желаемое визуальное представление.

Код на Common Intermediate Language

Поскольку все языки компилируются в IL-код, давайте рассмотрим IL-код программы, которую мы только что изучили. Как объяснялось в главе 2, IL – это набор стековых инструкций, поддерживающих исчерпывающий список популярных объектно-ориентированных возможностей, в том числе тех, что мы уже рассмотрели в этой главе. Это промежуточный этап, обеспечивающий склейку .NET-приложений с CLR.

Начнем с объявления пространства имен. Заметьте, что объявление IL `.namespace` позволяет создать наше пространство имен Lang. Как и в C#, в IL используются открывающие и закрывающие фигурные скобки:

```
.namespace Lang
{
```

Теперь займемся интерфейсом ISteering. В IL любой тип, который должен управляться CLR, следует объявлять с помощью команды IL `.class`. Так как CLR должна управлять ссылками на интерфейс, при определении интерфейса в IL необходимо использовать объявление IL `.class`, как показано в следующем листинге:

```
.class interface private abstract auto ansi ISteering
{
  .method public hidebysig newslot virtual abstract
          instance void TurnLeft() cil managed
  {
  } // конец определения метода ISteering::TurnLeft

  .method public hidebysig newslot virtual abstract
          instance void TurnRight() cil managed
  {
  } // конец определения метода ISteering::TurnRight

} // конец определения класса ISteering
```

Кроме того, надо вставить два специальных атрибута IL:

`interface`

Сообщает, что текущее определение типа является спецификацией интерфейса.

`abstract`

Сообщает, что в этом определении типа не будет реализаций методов и все методы, определенные в интерфейсе, должны быть реализованы в классе, реализующем этот интерфейс.

Другие указанные в этом определении атрибуты, не обязательные при определении интерфейса, включают следующее:

`private`

Мы не указали область видимости вашего интерфейса в C#, поэтому сгенерированный IL-код, приведенный здесь, добавляет к определению этого интерфейса IL-атрибут `private`. Это значит, что данный интерфейс будет виден только в пределах текущей сборки и никакие внешние сборки не смогут его увидеть.

`auto`

Сообщает, что CLR должна осуществлять автоматическую компоновку этого типа во время выполнения.

`ansi`

Сообщает CLR о необходимости использования строковых буферов в формате ANSI для маршалинга данных между управляемым и неуправляемым кодом.

Теперь вы знаете, как определить интерфейс в IL. Перед тем как пойти дальше, мы кратко рассмотрим атрибуты объявления `.method` – по крайней мере те атрибуты, которые мы еще не рассматривали. Вот они:

`newslot`

Уведомляет JIT-компилятор о необходимости зарезервировать новую ячейку в таблице `vtbl` данного типа, которая будет использоваться CLR во время выполнения для разрешения вызовов виртуального метода.

`instance`

Уведомляет CLR о том, что этот метод представляет собой экземпляр или метод уровня объекта, в отличие от статического метода или метода уровня класса.

Определив на IL интерфейс ISteering, давайте реализуем его в нашем классе Vehicle. Как видите, в следующем фрагменте кода особых сюрпризов нет. Мы расширили класс System.Object (на это указывает ключевое слово `extends`) и реализовали интерфейс Lang.ISteering (на это указывает ключевое слово `implements`):

```
.class private abstract auto ansi beforefieldinit Vehicle
       extends [mscorlib]System.Object
       implements Lang.ISteering
{
  .method public hidebysig newslot final virtual
          instance void TurnLeft() cil managed
  {
    // IL-код для ясности опущен
  } // конец определения метода Vehicle::TurnLeft
```

```
  .method public hidebysig newslot final virtual
          instance void TurnRight() cil managed
  {
    // IL-код для ясности опущен
  } // конец определения метода Vehicle::TurnRight

  .method public hidebysig newslot virtual abstract
          instance void ApplyBrakes() cil managed
  {
  } // конец определения метода Vehicle::ApplyBrakes

  // Метод .ctor для ясности опущен

} // конец определения класса Vehicle
```

Заметьте также, что этот класс является абстрактным классом и что метод ApplyBrakes() – абстрактный метод, аналогично тому, что мы видели в предыдущих примерах. Также обратите внимание на IL-атрибут `final`, который присутствует в объявлениях `.method` для методов TurnLeft() и TurnRight(). Этот атрибут IL указывает, что данные методы больше не могут быть перекрыты подклассами класса Vehicle. Видя все эти атрибуты, вы должны понимать, что все в IL объявляется явно, чтобы все компоненты CLR могли использовать преимущества наличия этой информации для управления вашими типами во время выполнения.

Теперь посмотрим на класс Car, являющийся производным от класса Vehicle. Как видите, в реализации метода ApplyBrakes IL-инструкция `newobj instance` создает новый экземпляр класса Exception. Затем IL-инструкция `throw` немедленно вызывает только что созданный объект исключения.

```
.class private auto ansi beforefieldinit Car
       extends Lang.Vehicle
  {
    .method public hidebysig virtual instance void
            ApplyBrakes() cil managed
    {
      // IL-код для ясности опущен
      newobj instance void
         [mscorlib]System.Exception::.ctor(class System.String)
      throw
    } // конец определения метода Car::ApplyBrakes

    // Метод .ctor для ясности опущен

  } // конец определения класса

} // конец определения пространства имен Lang
```

И наконец, давайте посмотрим на нашу функцию Main(), представляющую собой часть класса Drive. Мы удалили из этой функции большую часть IL-кода (который вы уже изучили), чтобы следующий фраг-

мент кода был проще для понимания, однако сохранили важные элементы, в которых надо разобраться. Во-первых, директива `.locals` идентифицирует все локальные переменные функции Main(). Во-вторых, вы видите, что IL также поддерживает обработку исключений через инструкцию `.try`. Заметьте, что как в блоке `.try`, так и в блоке `catch` имеется инструкция `leave.s`, заставляющая передать управление на инструкцию, расположенную в строке `IL_0024`, что приведет к выходу из блоков `.try` и `catch`.

```
.class private auto ansi beforefieldinit Drive
       extends [mscorlib]System.Object
{
  .method public hidebysig static void Main() cil managed
  {
    .entrypoint
    // Размер кода       37 (0x25)
    .maxstack  1
    .locals (class Lang.Vehicle V_0,
             class [mscorlib]System.Exception V_1)
    .try
    {
      // IL-код для ясности опущен
      leave.s    IL_0024
    }  // конец блока .try
    catch [mscorlib]System.Exception
    {
      // IL-код для ясности опущен
      leave.s    IL_0024
    }  // конец обработчика
    IL_0024:  ret
  } // конец метода Drive::Main

  // Метод .ctor для ясности опущен

} // конец определения класса Drive
```

Как видите, все рассмотренные нами основные концепции применимы и к IL. Так как вы уже видели поддерживающий данные концепции код на Managed C++, VB.NET, C# и IL, мы не будем пытаться дальше убеждать вас в том, что все эти средства работают и в других языках .NET.

Языковая интеграция

В предыдущем разделе мы видели, что преимущества объектно-ориентированных концепций .NET доступны в любом языке .NET. Здесь мы покажем, как воспользоваться *языковой интеграцией* (*language integration*) – возможностью создавать класс, производный от базового, определенного в совершенно ином языке, перехватывать исключения, вызванные кодом, написанным на другом языке, или использовать преимущества полиморфизма, выходя за рамки одного языка и т. п.

Прежде чем обсуждать примеры этого раздела, давайте разберемся, чего мы хотим добиться (рис. 3.1). Сначала мы используем Managed C++ для разработки класса Vehicle, являющегося абстрактным базовым классом. Класс Vehicle предоставляет три полиморфных метода: TurnLeft(), TurnRight() и ApplyBrakes(). Затем с помощью VB.NET разработаем класс Car, производный от Vehicle и перекрывающий эти три виртуальных метода. Кроме того, применим C# для разработки класса Plane, производного от Vehicle и также перекрывающего эти три виртуальных метода.

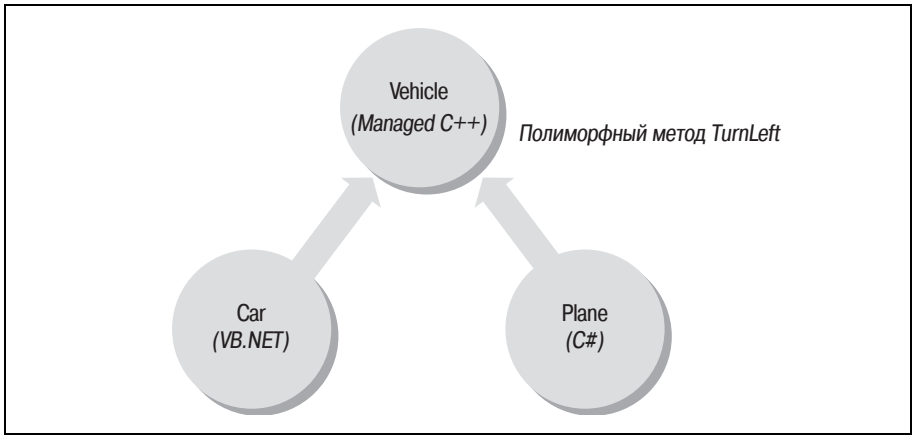


Рис. 3.1. *Полиморфизм между языками*

В последующих примерах кода мы можем попросить класс Vehicle выполнить метод TurnLeft() или TurnRight(), однако кто будет выполнять поворот, зависит от целевого объекта Car или Plane. В отличие от примеров предыдущего раздела, эти примеры иллюстрируют наследование классов и вызов виртуальных функций из классов, определенных в другом языке. Кроме того, в нашей программе мы продемонстрируем, что обработка исключений работает между разными языками.

Класс Vehicle в Managed C++

Обратимся к Managed C++ для разработки класса Vehicle, представляющего собой абстрактный класс, т. к. ApplyBrakes() – это чисто виртуальная функция. Vehicle реализует интерфейс ISteering для поддержки методов TurnLeft() и TurnRight(). Поскольку функция ApplyBrakes() является чисто виртуальной, любой производный от Vehicle класс должен реализовать этот метод:

```
#using <mscorlib.dll>
using namespace System;

public __gc __interface ISteering
{
  void TurnLeft();
```

```
  void TurnRight();
};

public __gc class Vehicle : public ISteering
{
  public:

    virtual void TurnLeft()
    {
      Console::WriteLine("Транспортное средство поворачивает налево.");
    }

    virtual void TurnRight()
    {
      Console::WriteLine("Транспортное средство поворачивает направо.");
    }

    virtual void ApplyBrakes() = 0;
};
```

Имея этот абстрактный базовый класс, мы можем создать DLL, содержащую это определение. Первая команда показывает, как использовать компилятор Managed C++ для компиляции (что указывается параметром /c) файла *vehicle.cpp*, содержащего приведенный выше код. Во второй команде показано, как при помощи редактора связей C++ создавать DLL с метаданными и IL-кодом:

```
cl /CLR /c vehicle.cpp
link -dll /out:vehicle.dll -noentry vehicle.obj
```

Имея лишь несколько строк кода на Managed C++, мы можем создать DLL, которая будет использоваться другим компонентом. Заметьте, что нет необходимости предоставлять код для функций IUnknown, DllGetClassObject(), DllCanUnloadNow(), DllRegisterServer(), DllUnregisterServer() и т. п. Раньше программист должен был предоставлять код этих функций и интерфейсов для традиционных COM DLL.

Класс Car в VB.NET

Имея абстрактный класс Vehicle, мы можем создать производный от него класс Car и предоставить реализацию трех виртуальных методов, определенных в Vehicle. Заметьте, что в следующем фрагменте кода мы перекрываем и предоставляем реализацию для методов TurnLeft(), TurnRight() и ApplyBrakes(). Особенность метода ApplyBrakes() состоит в том, что он генерирует исключение, которое должно быть перехвачено кодом, написанным на C#, что мы увидим позднее.

```
Imports System

Public Class Car
  Inherits Vehicle

  Overrides Public Sub TurnLeft()
    Console.WriteLine("Машина поворачивает налево.")
```

```
  End Sub

  Overrides Public Sub TurnRight()
    Console.WriteLine("Машина поворачивает направо.")
  End Sub

  Overrides Public Sub ApplyBrakes()
    Console.WriteLine("Машина пытается затормозить.")
    throw new Exception("Поломка тормозов!")
  End Sub

End Class
```

Имея этот код, мы можем собрать DLL-библиотеку с помощью компилятора VB.NET с интерфейсом командной строки следующим образом:

```
vbc /r:vehicle.dll /t:library /out:car.dll car.vb
```

Мы хотим, чтобы компилятор VB.NET создал DLL, поэтому должны сообщить ему об этом с помощью параметра `/t:library`. Кроме того, т. к. класс Car является производным от Vehicle, компилятор VB.NET должен разрешить ссылки на класс Vehicle. Мы можем сообщить компилятору VB.NET о месте нахождения внешних ссылок с помощью параметра `/r`. Важно отметить, что для повторного использования кода класса Vehicle не требуется наличия его исходного кода, поскольку вся информация о типах может быть получена из сборки .NET. Кроме того, этим примером мы доказываем, что можно создать производный VB.NET-класс от класса, написанного на Managed C++.

Класс Plane в C#

Теперь разработаем на C# класс Plane, производный от класса Vehicle, написанного на Managed C++. Как и класс Car, класс Plane реализует три виртуальные функции из класса Vehicle. В отличие от класса Car, метод ApplyBrakes() в этом классе не генерирует исключение:

```
using System;

public class Plane : Vehicle
{
  override public void TurnLeft()
  {
    Console.WriteLine("Самолет поворачивает налево.");
  }

  override public void TurnRight()
  {
    Console.WriteLine("Самолет поворачивает направо.");
  }

  override public void ApplyBrakes()
  {
    Console.WriteLine("Были использованы аэродинамические тормоза.");
  }
}
```

DLL из этого кода создается следующей командой:

```
csc /r:vehicle.dll /t:library /out:plane.dll plane.cs
```

Заметьте, что мы использовали параметр `/r:`, чтобы сообщить компилятору C#, что класс Vehicle определен в *vehicle.dll*.

Класс TestDrive на C#

Создав *vehicle.dll*, *car.dll* и *plane.dll*, мы готовы продемонстрировать, что полиморфизм и обработка исключений работают между разными языками. Следующий код, написанный на C#, содержит метод Main() со ссылкой на класс Vehicle и обработчик исключений.

Внутри блока `try` мы сначала создаем экземпляр класса Plane и ссылаемся на этот экземпляр с помощью локальной ссылки типа Vehicle. Вместо вызова метода TurnLeft() или ApplyBrakes() класса Plane мы используем ссылку типа Vehicle. Аналогично мы создаем экземпляр класса Car и ссылаемся на него с помощью локальной ссылки типа Vehicle. Как и раньше, вместо вызова метода TurnLeft() или ApplyBrakes() класса Car мы используем ссылку типа Vehicle. В обоих случаях мы просим транспортное средство повернуть налево или затормозить, однако реальным транспортным средством (Vehicle), которое будет осуществлять поворот налево или тормозить, будет в первом случае экземпляр класса Plane, а во втором – экземпляр класса Car; это и есть полиморфизм, и он работает между разными языками.

Обратите внимание, что второй вызов метода ApplyBrakes() вызовет исключение, т. к. мы генерируем его в методе ApplyBrakes() класса Car. Хотя этот метод был написан на VB.NET, мы все равно можем перехватить генерируемое исключение в C#, доказывая тем самым, что обработка исключений работает между разными языками.

```
using System;

class TestDrive
{
  public static void Main()
  {
    Vehicle v;  // Ссылка на Vehicle

    try
    {
      Plane p = new Plane();
      v = p;
      v.TurnLeft();
      v.ApplyBrakes();

      Car c = new Car();
      v = c;
      v.TurnLeft();
      v.ApplyBrakes();  // Исключение
```

```
      }
      catch(Exception e)
      {
        Console.WriteLine(e.ToString());
      }
    }
  }
```

Для того чтобы проверить работу этих средств, можно создать EXE-файл, выполнив следующую команду:

```
csc /r:vehicle.dll;car.dll;plane.dll /t:exe /out:drive.exe drive.cs
```

Поскольку в этом коде мы использовали классы Vehicle, Car и Plane, то должны включить в командную строку ссылки на *vehicle.dll*, *car.dll* и *plane.dll*. А раз мы создаем EXE-файл, то необходимо уведомить об этом компилятор C# с помощью параметра `/t:exe`. Создав и исполнив этот EXE-файл, мы получим следующий результат:

```
Самолет поворачивает налево.
Были использованы аэродинамические тормоза.
Машина поворачивает налево.
Машина пытается остановиться.
System.Exception: Поломка тормозов!
   at Car.ApplyBrakes()
   at TestDrive.Main()
```

Как и ожидалось, самолет сначала поворачивает налево, а затем включает аэродинамические тормоза. Затем машина поворачивает налево, пытается остановиться, однако не может этого сделать и генерирует исключение, перехваченное в методе Main().

В этом простом примере мы показали, что теперь в .NET Framework можно использовать преимущества наследования, полиморфизма и обработки исключений, выходя за границы одного языка.

Заключение

Мы начали эту главу с рассказа о том, что .NET предоставляет общую модель программирования, сокращающую время на обучение и увеличивающую производительность программиста. Узнав о том, как что-либо делать с помощью классов в .NET Framework, можно перенести это знание на любой язык среды .NET. Затем мы продемонстрировали, что можем написать один и тот же код, поддерживающий основные возможности .NET, на любом имеющемся языке. И наконец, мы доказали, что .NET действительно поддерживает языковую интеграцию, которая никогда не была возможна с использованием платформ и средств разработки Microsoft до появления .NET.

4

Работа с .NET-компонентами

Те, кто видел примеры языковой интеграции из предыдущей главы, знают, что все сборки .NET в основном являются двоичными компонентами.[1] Каждую сборку .NET можно рассматривать как компонент, который вы можете подключить к другому компоненту или приложению, при этом наличие исходного текста не обязательно, т. к. все метаданные компонента хранятся внутри .NET-сборки. Если в COM для построения компонента необходимы тонны кода «заплаток», то создание компонента в .NET не требует дополнительной работы, поскольку все сборки .NET по своей природе являются компонентами.

В этой главе мы рассмотрим более сложные темы, в том числе развертывание компонентов, распределенные компоненты и корпоративные службы, такие как управление транзакциями, организация пула объектов, безопасность на основе ролей и очереди сообщений.

Варианты развертывания

Для такой простой программы, как *hello.exe*, созданной нами в главе 2, развертывание выполняется просто: скопируйте сборку в каталог и она готова к запуску. Для того чтобы удалить ее, удалите файл из каталога. Однако когда вам требуется использовать компоненты совместно с другими приложениями, надо проделать некоторую работу.

---

[1] Помните, что, как мы объясняли ранее в главе 1, мы употребляем термин «компонент» в отношении двоичного файла, единицы развертывания, а не в отношении COM-класса.

В COM для обеспечения взаимодействия компонентов приходится хранить в реестре информацию об активации и маршалинге[1]; вот почему любой COM-разработчик долго может обсуждать боль и страдания, причиняемые ему COM и системным реестром. В .NET для интеграции компонентов системный реестр более не требуется.

В среде .NET компоненты могут быть *закрытыми* (*private*), т. е. не опубликованными и используемыми только известными клиентами, или *совместно используемыми* (*shared*), т. е. опубликованными и используемыми всеми клиентами. В этом разделе обсуждается несколько возможностей развертывания закрытых и совместно используемых компонентов.

Закрытые компоненты

Если у вас есть закрытые компоненты, используемые только определенными клиентами, то имеется два варианта развертывания. Можно хранить все закрытые компоненты и клиентские приложения, использующие эти компоненты, в одном каталоге или хранить компоненты в определенном каталоге, к которому клиентское приложение имеет доступ. Поскольку клиенты используют конкретные закрытые компоненты, на которые они ссылались при построении, CLR не поддерживает проверку версий и не навязывает никакой стратегии в отношении версий для закрытых компонентов.

Чтобы установить свое приложение, в обоих случаях просто выполните команду `xcopy`, чтобы скопировать файлы приложения из исходного инсталляционного каталога в целевой каталог. Когда потребуется удалить приложение, удалите эти каталоги. Вам не придется писать код для хранения информации в реестре, поэтому не придется и беспокоиться о том, что вы пропустили вставку какого-то необходимого для корректного выполнения приложения параметра в реестр. Кроме того, поскольку в реестре ничего не хранится, не приходится беспокоиться о данных, остающихся в нем после удаления приложения.

Развертывание в одном каталоге

Для того чтобы указать на расположение компонента в том же каталоге, что и клиентское приложение, воспользуйтесь следующим синтаксисом (как мы делали в примере главы 3):

```
csc /r:vehicle.dll;car.dll;plane.dll /t:exe /out:drive.exe drive.cs
```

[1] Распределенное приложение требует наличия коммуникационного уровня для сборки и разборки сетевого потока и потока данных. Этот уровень в терминологии Microsoft официально называется *маршалером*. Сборка и разборка содержимого сетевого буфера, используемого для протокола на прикладном уровне официально называются *маршалингом* и *демаршалингом*, соответственно.

Ссылка на *plane.dll* не включает имя каталога; следовательно, компилятор C# сохраняет эту ссылку в манифесте сборки клиентского приложения как есть, чтобы CLR могла разрешить ее во время выполнения (т. е. найти и загрузить *plane.dll* и активизировать класс Plane). Если переместить какие-либо DLL в другой каталог, при запуске *drive.exe возникнет исключение*.

Развертывание в нескольких каталогах

Можно не сохранять все компоненты в одном каталоге с клиентским приложением, а распределить их по нескольким отдельным каталогам, чтобы было легче находить компоненты и управлять ими. Например, мы можем разместить компоненты vehicle, car и plane в их собственные каталоги, как это показано на рис. 4.1. Приложение *drive.exe* мы оставим в каталоге верхнего уровня, *MultiDirectories*.

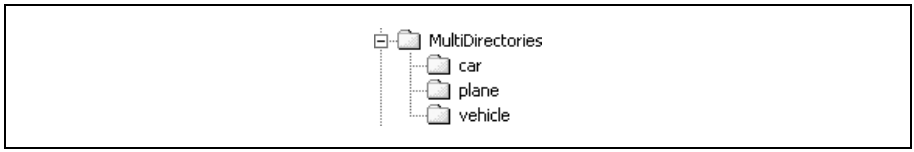


Рис. 4.1. *Дерево компонентов из нескольких каталогов*

При компиляции компонента vehicle не требуется делать ничего особенного, т. к. он не ссылается и не использует никаких сторонних компонентов. Однако при компиляции компонента car или plane необходимо сослаться на корректный компонент vehicle (т. е. на тот, что находится в каталоге *vehicle*). Например, для успешной компиляции компонента plane надо явно сослаться на *vehicle.dll*, указав абсолютный или относительный путь, как это показано в следующей команде (сначала перейдите в каталог *plane*):

```
csc /r:..\vehicle\vehicle.dll /t:library /out:plane.dll plane.cs
```

Компонент car компилируется таким же образом, как вы скомпилировали компонент plane. Для компиляции клиентского приложения вы также должны указать зависимости, задав правильные пути (перейдите в основной каталог *MultiDirectories* и введите эту команду в одной строке):

```
csc /r:vehicle\vehicle.dll;car\car.dll;plane\plane.dll
    /t:exe /out:drive.exe drive.cs
```

В результате выполнения этой команды компилятор C# запишет указанные пути в манифест сборки вашего приложения. При запуске *drive.exe* CLR обратится к манифесту для поиска и загрузки необходимых компонентов.

Совместно используемые компоненты

В отличие от закрытых сборок приложения, *совместно используемые сборки* (*shared assemblies*) – те, которые могут использоваться любым клиентским приложением, – должны быть опубликованы или зарегистрированы в глобальном кэше сборок (Global Assembly Cache, GAC) системы. Сборки, зарегистрированные в GAC, действуют как системные компоненты, такие как системные DLL, которые может использовать любой процесс в системе. Необходимым условием для регистрации в GAC является наличие в компоненте информации о создателе и версии. В дополнение к другим метаданным эти две позиции позволяют зарегистрировать и запустить несколько версий одного и того же компонента на одной машине. И снова, в отличие от COM, не требуется сохранять какую-либо информацию в системном реестре для того, чтобы клиентские приложения могли применять эти совместно используемые сборки.

Регистрация совместно используемых сборок в GAC состоит из трех основных этапов:

1. Воспользоваться утилитой *sn.exe* для получения пары открытого/закрытого ключа. Эта утилита генерирует случайную пару ключей и сохраняет информацию в выходном файле, например в файле *originator.key*.
2. Скомпилировать сборку с указанием номера версии и информации о ключах из файла *originator.key*.
3. Воспользоваться утилитой глобального кэша сборок .NET (*gacutil.exe*) для регистрации сборки в GAC. Теперь эта сборка является совместно используемой и может вызываться любым клиентским приложением.

Команды, обсуждаемые в этом разделе, ссылаются на относительные пути, поэтому если вы двигаетесь вместе с нами дальше, убедитесь, что создали структуру каталогов, показанную на рис. 4.2. Каталоги *vehicle*, *plane* и *car* содержат соответствующие сборки, а каталог *key* содержит пару из открытого и закрытого ключей, которые мы сейчас сгенерируем. Каталог *car-build* содержит сборку car с модифицированным номером построения, а каталог *car-revision* – эту же сборку с модифицированным номером ревизии.

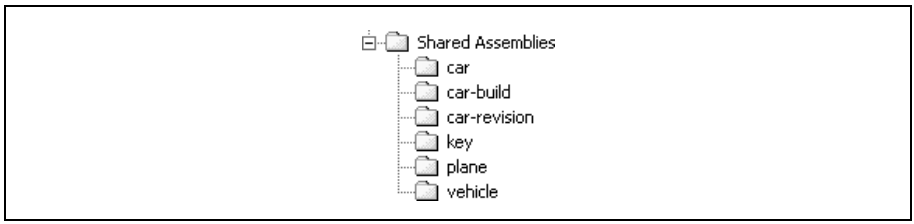


Рис. 4.2. *Структура каталогов для примеров этого раздела*

Генерирование случайной пары ключей

Мы выполним первый этап один раз и будем использовать пару ключей для всех совместно используемых библиотек, которые создадим в этом разделе. Мы делаем так для краткости, но вы можете применять различные наборы ключей для каждой сборки или даже для каждой компилируемой версии. Вот как генерируется случайная пара ключей (убедитесь, что команда выполняется в каталоге *key*):

```
sn -k originator.key
```

Параметр `-k` генерирует случайную пару ключей и сохраняет информацию о ключах в файле *originator.key*. Этот файл будет выступать как входной при создании наших совместно используемых сборок. Давайте теперь рассмотрим этапы 2 и 3 регистрации нашей совместно используемой сборки в GAC.

Создание совместно используемой сборки из компонента vehicle

Чтобы добавить информацию о версии и ключах в компонент vehicle (созданный на Managed C++), надо внести небольшие изменения в файл *vehicle.cpp*, приведенные ниже:

```
#using<mscorlib.dll>
using namespace System;

using namespace System::Reflection;
[assembly:AssemblyVersion("1.0.0.0")];
[assembly:AssemblyKeyFile("..\\key\\originator.key")];

public __gc __interface ISteering
{
  void TurnLeft();
  void TurnRight();
};

public __gc class Vehicle : public ISteering
{
  public:

    virtual void TurnLeft()
    {
      Console::WriteLine("Транспортное средство поворачивает налево.");
    }

    virtual void TurnRight()
    {
      Console::WriteLine("Транспортное средство поворачивает направо.");
    }

    virtual void ApplyBrakes() = 0;
};
```

Первая выделенная строка указывает, что мы работаем в пределах пространства имен Reflection, в котором определяются атрибуты, перехватываемые компилятором для добавления корректной информации в манифест нашей сборки. (Обсуждение атрибутов см. в разделе «Программирование на основе атрибутов» далее в этой главе.) Атрибут `AssemblyVersion` служит для указания версии сборки, а атрибут `AssemblyKeyFile` – для указания имени файла, содержащего информацию о ключах, которую компилятор должен использовать при получении значения метки открытого ключа (public-key-token).

Сделав это, вы можете построить сборку при помощи следующих команд, которые мы уже рассматривали:

```
cl /CLR /c vehicle.cpp
link -dll /out:vehicle.dll -noentry vehicle.obj
```

После создания сборки можно обратиться к утилите глобального кэша сборок для регистрации этой сборки в GAC:

```
gacutil.exe /i vehicle.dll
```

Успешная регистрация в кэше превращает этот компонент в совместно используемую сборку. Версия этого компонента копируется в GAC, поэтому если даже вы удалите файл локально, то все еще сможете запустить свою клиентскую программу.[1]

Создание совместно используемой сборки из компонента car

Чтобы добавить информацию о версии и ключах в компонент car, нам потребуется внести небольшие изменения в файл *car.vb*, приведенные ниже:

```
Imports System

Imports System.Reflection
<Assembly:AssemblyVersion("1.0.0.0")>
<assembly:AssemblyKeyFile("..\\key\\originator.key")>

Public Class Car
  Inherits Vehicle

  Overrides Public Sub TurnLeft()
    Console.WriteLine("Машина поворачивает налево.")
  End Sub

  Overrides Public Sub TurnRight()
    Console.WriteLine("Машина поворачивает направо.")
  End Sub

  Overrides Public Sub ApplyBrakes()
    Console.WriteLine("Машина пытается остановиться.")
```

[1] Однако не удаляйте файл сейчас, т. к. он нам понадобится для построения сборок car и plane.

```
    Console.WriteLine("ИСХОДНАЯ ВЕРСИЯ - 1.0.0.0.")
    throw new Exception("Поломка тормозов!")
  End Sub

End Class
```

Сделав это, вы можете построить компонент следующей командой:

```
vbc /r:..\vehicle\vehicle.dll /t:library /out:car.dll car.vb
```

Заметьте, что компонент car использует конкретный компонент vehicle, `..\vehicle\vehicle.dll`. Во время выполнения, если CLR не может найти этот конкретный файл здесь или в GAC, будет сгенерировано исключение. Скомпилировав этот компонент, его можно зарегистрировать в GAC:

```
gacutil /i car.dll
```

Теперь можно удалить *car.dll* из локального каталога, т. к. сборка уже зарегистрирована в GAC.

Создание разделяемой сборки из компонента plane

Чтобы добавить информацию о версии и ключах в компонент plane, надо немного изменить файл *plane.cs*:

```
using System;

using System.Reflection;
[assembly:AssemblyVersion("1.0.0.0")]
[assembly:AssemblyKeyFile("..\\key\\originator.key")]

public class Plane : Vehicle
{
  override public void TurnLeft()
  {
    Console.WriteLine("Самолет поворачивает налево.");
  }

  override public void TurnRight()
  {
    Console.WriteLine("Самолет поворачивает направо.");
  }

  override public void ApplyBrakes()
  {
    Console.WriteLine("Были использованы аэродинамические тормоза.");
  }
}
```

Сделав это, можно построить сборку при помощи следующих команд:

```
csc /r:..\vehicle\vehicle.dll /t:library /out:plane.dll plane.cs
gacutil /i plane.dll
```

Последняя строка данного фрагмента просто регистрирует компонент в GAC.

Просмотр GAC

Теперь, когда мы зарегистрировали все наши компоненты в GAC, давайте посмотрим, как он выглядит. Microsoft выпустила расширение оболочки, названное Shell Cache Viewer, облегчающее просмотр содержимого GAC. На наших компьютерах Shell Cache Viewer появляется, когда мы просматриваем папку *C:\WINNT\Assembly*, как показано на рис. 4.3.[1]

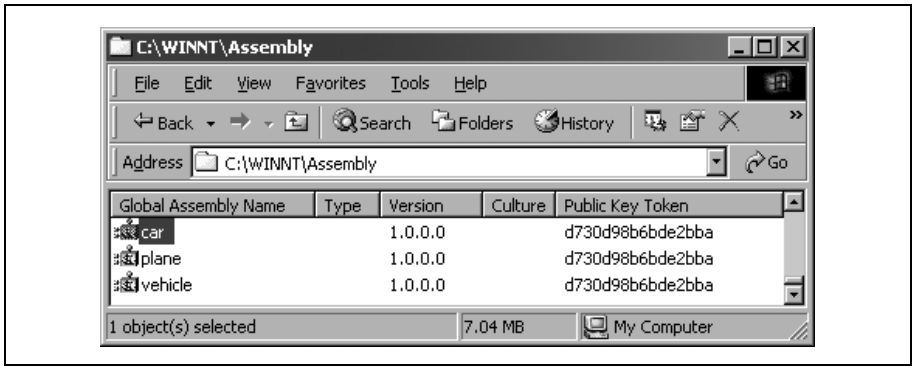


Рис. 4.3. *Наши разделяемые сборки в GAC*

Как видите, Shell Cache Viewer показывает, что все наши компоненты имеют одинаковый номер версии, т. к. при их создании мы указали в качестве номера версии 1.0.0.0. Кроме того, здесь показано, что все наши компоненты имеют одинаковое значение метки открытого ключа, т. к. мы использовали один и тот же файл ключей *originator.key*.

Компиляция и тестирование drive.exe

Приведенный ранее исходный текст *drive.cs* следует скопировать в каталог *Shared Assemblies*, корневой каталог структуры (показанной на рис. 4.2), с которой мы будем работать в этом разделе. Сделав это, можно будет скомпилировать данный компонент следующей командой (помните, что все это должно быть набрано в одну строку):

```
csc /r:vehicle\vehicle.dll;car\car.dll;plane\plane.dll
    /t:exe /out:drive.exe drive.cs
```

После этого можно запустить компонент *drive.exe*, который будет использовать сборки *vehicle.dll*, *car.dll* и *plane.dll*, зарегистрированные в GAC. Как часть результата его работы вы должны увидеть следующее:

```
ИСХОДНАЯ ВЕРСИЯ - 1.0.0.0.
```

[1] Этот путь полностью зависит от значения параметра `%windir%` на вашем компьютере.

Чтобы удалить эти совместно используемые компоненты, выберите соответствующие сборки и нажмите клавишу <Delete> (но если вы сделаете это сейчас, то должны будете заново зарегистрировать эти сборки, т. к. они потребуются нам в последующих примерах). Таким образом вы уберете все остатки ваших компонентов из GAC. Останется лишь удалить любые файлы, скопированные с установочной дискеты. Обычно все, что реально требуется сделать – это удалить каталог приложения со всеми вложенными подкаталогами.

Добавление новых версий

В отличие от закрытых сборок, разделяемые сборки могут использовать богатые возможности политики контроля версий, поддерживаемые CLR. В отличие от более ранних инфраструктур на уровне ОС, CLR реализует политику контроля версий во время загрузки всех совместно используемых сборок. По умолчанию CLR загружает сборку, с которой приложение было скомпилировано, но с помощью конфигурационного файла приложения можно поручить CLR загрузить определенную версию сборки, необходимую этому приложению. В конфигурационном файле можно указать правила или политики, которые должна использовать CLR при загрузке совместно используемых сборок, от которых зависит приложение.

Внесем некоторые изменения в наш компонент car, чтобы продемонстрировать, как поддержка контроля версий работает по умолчанию. Помните, что версия 1.0.0.0 метода ApplyBrakes() нашего компонента car генерирует исключение:

```
Overrides Public Sub ApplyBrakes()
  Console.WriteLine("Машина пытается остановиться.")
  Console.WriteLine("ИСХОДНАЯ ВЕРСИЯ - 1.0.0.0.")
  throw new Exception("Поломка тормозов!")
End Sub
```

Давайте создадим новое *построение* (*build*), убирающее это исключение. Для этого внесем следующие изменения в метод ApplyBrakes() (и сохраним этот исходный текст в каталоге *car-build*):

```
Overrides Public Sub ApplyBrakes()
  Console.WriteLine("Машина пытается остановиться.")
  Console.WriteLine("Изменен НОМЕР ПОСТРОЕНИЯ - 1.0.1.0.")
End Sub
```

Кроме того, необходимо следующим образом изменить номер построения в коде:

```
<Assembly:AssemblyVersion("1.0.1.0")>
```

Теперь построим этот компонент и зарегистрируем его следующими командами:

```
vbc /r:..\vehicle\vehicle.dll
```

```
    /t:library /out:car.dll car.vb
gacutil /i car.dll
```

Заметьте, что мы обозначили эту версию как 1.0.1.0, а это значит, что она совместима с версией 1.0.0.0. Зарегистрировав эту сборку в GAC, выполните свое приложение *drive.exe*, и как часть результата его работы вы увидите следующую строку:

```
ORIGINAL VERSION - 1.0.0.0.
```

Это поведение по умолчанию – CLR загрузит версию сборки, с которой приложение было построено. И чтобы доказать еще раз это утверждение, предположим, что вы предоставили версию 1.0.1.1, внеся следующие изменения (сохраните эту версию в каталоге *car-revision*):

```
Overrides Public Sub ApplyBrakes()
  Console.WriteLine("Машина пытается остановиться.")
  Console.WriteLine("Изменен НОМЕР РЕВИЗИИ - 1.0.1.1.")
End Sub

<Assembly:AssemblyVersion("1.0.1.1")>
```

В этот раз вместо изменения номера построения вы меняете номер ревизии, при этом сохраняя совместимость с предыдущими двумя версиями. Если скомпилировать эту сборку, зарегистрировать ее в GAC и снова выполнить *drive.exe*, то можно увидеть следующую строку как часть результата работы программы:

```
ORIGINAL VERSION - 1.0.0.0.
```

Опять CLR выбрала ту версию, с которой выполнялось построение вашего приложения.

Как показано на рис. 4.4, убедиться, что все три версии присутствуют в системе одновременно, можно при помощи Shell Cache Viewer. Это значит, что имеется поддержка одновременного исполнения версий, что устраняет DLL Hell в .NET.

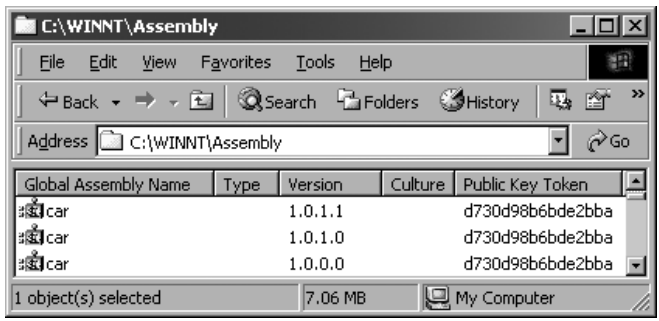


Рис. 4.4. *Несколько версий одной и той же разделяемой сборки*

Если вы хотите, чтобы ваша программа использовала другую совместимую версию сборки car, то должны предоставить файл конфигурации приложения. Имя конфигурационного файла составляется из полного имени исполняемого файла и добавленного к нему расширения «.config». Например, поскольку наша клиентская программа называется *drive.exe*, имя ее конфигурационного файла должно выглядеть так: *drive.exe.config*.

Ниже приведен файл *drive.exe.config*, позволяющий сообщить CLR о необходимости загрузки версии 1.0.1.0 сборки car (вместо загрузки по умолчанию версии 1.0.0.0). Два выделенных атрибута говорят, что, хотя клиентское приложение было построено с использованием версии 1.0.0.0 (`oldVersion`) сборки car, при запуске *drive.exe* необходимо загрузить версию 1.0.1.0 (`newVersion`).

```
<?xml version ="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

      <dependentAssembly>
        <assemblyIdentity name="car"
           publicKeyToken="D730D98B6BDE2BBA"
           culture="" />

        <bindingRedirect oldVersion="1.0.0.0"
                         newVersion="1.0.1.0" />

      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

В этом конфигурационном файле атрибут `name` тега `assemblyIdentity` указывает на пользовательское имя совместно используемой сборки, хранящейся в GAC. Если значение `name` может быть любым, то для того, чтобы можно было запустить `drive.exe`, значение `publicKeyToken` необходимо заменить соответствующим значением. В атрибуте `publicKeyToken` содержится значение метки открытого ключа, являющейся 8-байтовым хеш-кодом открытого ключа, использованного при построении этого компонента. Существует несколько способов получить хеш-код: его можно скопировать из Shell Cache Viewer, из дампа IL-кода вашего компонента или использовать для этого утилиту разделяемых имен:

```
sn -T car.dll
```

Если создать показанный выше конфигурационный файл (хранящийся в том же каталоге, что и исполняемый файл *drive.exe*) и запустить *drive.exe*, то как часть результата его выполнения будет выведено следующее:

```
Изменен НОМЕР ПОСТРОЕНИЯ - 1.0.1.0.
```

Если изменить конфигурационный файл так, чтобы в нем присутствовал атрибут `newVersion=1.0.1.1`, и снова выполнить *drive.exe*, то результат будет таким:

```
Изменен НОМЕР РЕВИЗИИ - 1.0.1.1.
```

После рассмотрения этих примеров вы должны понять, что имеете полный контроль над тем, какие версии зависимых сборок будет загружать CLR для вашего приложения. Независимо от того, какие версии использовались при построении приложения, во время выполнения вы можете выбирать различные версии путем простой смены нескольких атрибутов в конфигурационном файле приложения.

Распределенные компоненты

Компонентная технология должна поддерживать распределенные вычисления, позволяя активизировать и вызывать удаленные службы, так же как и сервисы, относящиеся к другому домену приложения.[1] Distributed COM, или DCOM, – это протокол связи, предоставляющий поддержку распределенных вычислений с помощью COM. DCOM хорошо подходит для распределенных вычислений, однако он не годится для глобального киберпространства, т. к. плохо работает с брандмауэрами и программным обеспечением NAT. Другими недостатками DCOM являются большие затраты на управление временем жизни, инициализацию протоколов и использование двоичных форматов.

Чтобы исключить или хотя бы смягчить эти недостатки, .NET предоставляет множество различных средств поддержки распределенных вычислений. Remoting API в .NET делает возможным применение разных каналов связи для распределенных вычислений, таких как TCP и HTTP (использующий SOAP). Он даже позволяет подключать в случае необходимости свои специальные каналы связи. Более того, поскольку среда является полностью объектно-ориентированной, распределенные вычисления в .NET становятся как никогда простыми. Чтобы показать, насколько просто написать в .NET распределенное приложение, рассмотрим пример использования сокетов, также известных в .NET как *каналы TCP*.

Распределенный сервер Hello

В этом примере мы напишем распределенное приложение Hello, выдающее строку текста на консоль, как только клиент вызывает предо-

[1] Любой процесс Windows требует собственного адресного пространства в памяти, что существенно увеличивает затраты при запуске нескольких процессов. Домен приложения (application domain) – это облегченный или виртуальный процесс. Все домены приложения данного процесса Windows могут использовать общее адресное пространство.

ставленный приложением метод SayHello(). Поскольку мы используем канал TCP, то сообщим компилятору, что нам требуются определения из пространств имен System.Runtime.Remoting и System.Runtime.Remoting.Channels.Tcp.

Заметьте, что класс CoHello является производным от MarshalByRefObject.[1] Это ключевой момент для распределенных вычислений в .NET, т. к. обеспечивает распределенную идентификацию, позволяя ссылаться на объект из других доменов приложения и даже из других процессов и с других компьютеров. Объект, использующий маршалинг по ссылке, требует наличия посредника на стороне клиента и заглушки на стороне сервера, но т. к. и то и другое обеспечивается инфраструктурой .NET, вам не нужно проделывать никакой дополнительной работы. Ваша работа состоит в создании производного от MarshalByRefObject класса для получения полной поддержки распределенных вычислений.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class CoHello : MarshalByRefObject
{
  public static void Main()
  {
    TcpChannel channel = new TcpChannel(4000);
    ChannelServices.RegisterChannel(channel);

    RemotingConfiguration.RegisterWellKnownServiceType (
      typeof(CoHello),              // Наименование типа
      "HelloDotNet",                // URI
      WellKnownObjectMode.Singleton // SingleCall или Singleton
    );

    System.Console.WriteLine("Нажмите <enter> для выхода...");
    System.Console.ReadLine();
  }

  public void SayHello()
  {
    Console.WriteLine("Здравствуй, вселенная .NET");
  }
}
```

Метод SayHello() определен как public, т. е. любой внешний клиент может вызвать его. Как видите, этот метод очень прост, но интересно, что

[1] Если вы этого не сделаете, ваш объект не будет иметь распределенной идентификации, т. к. по умолчанию используется маршалинг по значению (marshal-by-value), т. е. на клиентской стороне создается копия удаленного объекта.

его может вызвать удаленное клиентское приложение (которые мы скоро будем разрабатывать), потому что функция Main() использует класс TcpChannel. Посмотрите внимательно на функцию Main() и вы увидите, что она создает экземпляр класса TcpChannel, передавая ему номер порта, с которого сервер будет ожидать входящих запросов.[1]

Создав объект channel, мы затем регистрируем канал в службе ChannelServices, которая поддерживает регистрацию каналов и разрешение имен объектов. После этого вы должны зарегистрировать свой объект с помощью метода RegisterWellKnownServiceType() класса RemotingConfiguration, чтобы его можно было активизировать. Когда вы вызываете этот метод, вы должны передать ему имя класса, универсальный идентификатор ресурса (URI) и режим активизации объектов. URI – это ключевой элемент, который будет использоваться клиентским приложением для ссылки именно на этот зарегистрированный объект. Возможен либо режим активизации Singleton, т. е. один объект будет обслуживать много вызовов, либо SingleCall, т. е. один объект обслуживает не более одного вызова.[2] Вот как компилируется это удаленное приложение:

```
csc server.cs
```

После этого можно запустить серверную программу, которая будет ждать, пока вы не нажмете клавишу <Enter>. Сервер теперь готов к обслуживанию клиентских запросов.

Удаленный клиент Hello

Итак, у нас есть ожидающий запросов сервер, и мы можем приступить к разработке клиентского приложения, которое будет вызывать удаленный метод SayHello(). Вместо регистрации объекта нам необходимо будет активизировать удаленный объект. Поэтому теперь перейдем к коду, чтобы увидеть, как это работает. Изучая следующую программу, обратите внимание на несколько моментов:

- Мы работаем с типами из пространств имен System.Runtime.Remoting и System.Runtime.Remoting.Channels.Tcp, т. к. хотим использовать канал TCP.
- Наш класс Client не обязан быть производным от какого-либо определенного класса, т. к. это не серверный объект, которому необходима распределенная идентификация.

---

[1] Хотите верьте, хотите нет, но единственное, что нужно сделать, чтобы воспользоваться преимуществами протоколов HTTP и SOAP, – это заменить TcpChannel на HttpChannel.

[2] В этом примере показан код для создания канала, его регистрации в службе каналов и регистрации объекта, но писать весь этот код не надо, если вы предоставляете конфигурационный файл приложения, содержащий всю эту информацию.

- Мы разрабатываем клиентское приложение, поэтому не обязаны указывать клиентский порт при создании экземпляра TcpChannel.

Кроме этого, следует отметить такой ключевой момент, как активизация объекта, показанная во второй выделенной строке в следующем фрагменте кода. Для вызова удаленных методов сначала надо активизировать удаленный объект и получить связанный с ним посредник на стороне клиента. Чтобы активизировать объект и получить ссылку на посредник, надо вызвать метод GetObject() класса Activator. При этом необходимо передать имя удаленного класса и его полностью квалифицированное местоположение, включающее полный URI. Если эти действия были выполнены успешно, то можно вызывать удаленные методы.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class Client
{
  public static void Main()
  {
    try
    {
      TcpChannel channel = new TcpChannel();
      ChannelServices.RegisterChannel(channel);

      CoHello h = (CoHello) Activator.GetObject(
        typeof(CoHello),                    // Удаленный тип
        "tcp://127.0.0.1:4000/HelloDotNet"  // Местоположение
      );

      h.SayHello();
    }
    catch(Exception e)
    {
      Console.WriteLine(e.ToString());
    }
  }
}
```

Для того чтобы скомпилировать это клиентское приложение, надо включить ссылку на сборку *server.exe*:

```
csc /r:Server.exe Client.cs
```

Тем, кто знаком с DCOM, будет приятно узнать, что в .NET разрабатывать распределенные приложения относительно просто.[1]

[1] Если у вас есть под рукой экземпляр книги «Learning DCOM» (Изучаем DCOM) издательства O'Reilly, сравните эти две программы с соответствующими примерами на DCOM в приложении D и вы поймете, что мы имеем в виду.

Распределенный сборщик мусора

По той причине, что новый распределенный сборщик мусора столь замечателен, мы должны кратко рассказать об этом средстве. Вместо использования разностных тестовых опросов DCOM, требующих меньше сетевых пакетов, чем нормальные тестовые опросы (но все еще слишком много для распределенного протокола), распределенные вычисления в .NET используют аренду для управления временем жизни объектов. Если вы когда-либо продлевали срок аренды IP-адреса в сети с протоколом Dynamic Host Configuration Protocol (DHCP), то в значительной степени понимаете этот механизм, т. к. он основан на сходных концепциях.

В .NET распределенные объекты выдают разрешения на аренду вместо того, чтобы управлять временем жизни объектов, полагаясь на счетчики ссылок (как в COM). Домен приложения, в котором располагаются удаленные объекты, содержит специальный объект, называемый *диспетчером аренд* (*lease manager*) и управляющий всеми арендами, связанными с этими удаленными объектами. Когда срок аренды истекает, диспетчер аренд связывается со спонсором, сообщая ему об этом. *Спонсор* (*sponsor*) – это просто клиент, который ранее зарегистрировался в диспетчере аренд во время запроса на активизацию объекта, указав диспетчеру аренд, что хочет знать о том, когда истечет срок аренды. Если диспетчер аренд сможет связаться со спонсором, спонсор имеет право продлить срок аренды. Если спонсор отказывается продлевать аренду или если диспетчер аренд не может связаться со спонсором в течение настраиваемого периода ожидания, диспетчер аренд аннулирует аренду и удаляет объект. Есть еще два способа продления аренды: неявно, при каждом вызове удаленного объекта, или явно, путем вызова метода Renew() класса Lease.

Службы COM+ в .NET

COM-программирование требует при построении крупномасштабных, корпоративных приложений написания большого количества вспомогательного и инфраструктурного кода. Для облегчения разработки и развертывания транзакционных и масштабируемых COM-приложений корпорация Microsoft выпустила Microsoft Transaction Server (MTS). MTS позволяет разделять ресурсы, тем самым делая приложение более масштабируемым. Службы COM+ являются естественной эволюцией MTS. Если MTS – это просто еще одна библиотека поверх COM, то службы COM+ внедрены в библиотеку COM, объединяя COM и MTS в единую среду времени выполнения.

Службы COM+ оказались очень ценными для команд разработчиков, применяющих модель COM для построения приложений, использующих преимущества транзакций, организации пула объектов, безопасности на основе ролей и т. д. Тем, кто разрабатывает корпоративные .NET-приложения, службы COM+ в .NET необходимы.

В следующих примерах вместо того, чтобы продолжать пичкать вас принципиальными рассуждениями, мы покажем, как использовать основные службы COM+ в .NET, в том числе примеры транзакционного программирования, организации пула объектов и поддержки безопасности на основе ролей. Однако прежде чем вы увидите эти примеры, давайте поговорим о ключевых элементах – атрибутах, – дающих возможность применять эти службы в .NET.

Программирование с применением атрибутов

Атрибуты (*attributes*) – это ключевые элементы, помогающие писать меньше кода и позволяющие инфраструктуре автоматически выдавать необходимый вам код во время выполнения. Если вы применяли IDL (Interface Definition Language), то видели атрибуты `in` и `out`, как в приведенном ниже коде:

```
HRESULT SetAge([in] short age);
HRESULT GetAge([out] short *age);
```

IDL позволяет добавлять эти атрибуты так, чтобы маршалер знал, как оптимизировать загрузку сети. Здесь атрибут `in` говорит маршалеру, что содержимое посылается от клиента серверу, а атрибут `out` – что содержимое посылается от сервера клиенту. В методе SetAge() передача параметра `age` от сервера клиенту будет зря расходовать пропускную способность сети. Аналогично, нет необходимости посылать параметр `age` от клиента к серверу в методе GetAge().

Создание специальных атрибутов

Отметим, что `in` и `out` – это встроенные атрибуты, поддерживаемые компилятором MIDL, а .NET позволяет создавать собственные специальные атрибуты, производные от класса System.Attribute. Вот пример специального атрибута:

```
using System;

public enum Skill {  Guru, Senior, Junior }

[AttributeUsage(AttributeTargets.Class      |
                AttributeTargets.Field      |
                AttributeTargets.Method     |
                AttributeTargets.Property   |
                AttributeTargets.Constructor|
                AttributeTargets.Event)]
public class AuthorAttribute : System.Attribute
{
  public AuthorAttribute(Skill s)
  {
    level = s;
  }
  public Skill level;
}
```

Атрибут AttributeUsage, который мы применили к нашему классу AuthorAttribute, определяет правила использования AuthorAttribute[1]. В частности, он говорит о том, что AuthorAttribute может предшествовать классу или описывать класс или любой член класса.

Применение специальных атрибутов

Имея этот атрибут, мы можем написать простой класс, использующий его. Чтобы применить наш атрибут к классу или члену, обратимся к доступным для атрибута конструкторам. В данном случае у нас есть только один конструктор – AuthorAttribute(), принимающий в качестве параметра уровень квалификации автора. Можно создать атрибут при помощи метода AuthorAttribute(), но .NET позволяет опустить для удобства суффикс Attribute, что показано в следующем листинге:

```
[Author(Skill.Guru)]
public class Customer
{
  [Author(Skill.Senior)]
  public void Add(string strName)
  {
  }
  [Author(Skill.Junior)]
  public void Delete(string strName)
  {
  }
}
```

Заметьте, что мы применили атрибут Author к классу Customer, сообщая миру, что это определение класса было написано гуру. Этот код также показывает, что старший программист написал метод Add(), а младший – метод Delete().

Проверка атрибутов

Вы не увидите всех преимуществ атрибутов, пока не напишете простую программу-перехватчик, проверяющую определенные атрибуты и предоставляющую дополнительные услуги в соответствии с этими атрибутами. В числе реальных перехватчиков, использующих атрибуты, можно назвать маршалинг, транзакции, безопасность, организацию пулов и другие службы MTS и COM+.

Ниже приведена простая программа-перехватчик, использующая Reflection API для просмотра AuthorAttribute и предоставления дополнительных услуг. Как видно из ее кода, мы можем запросить тип, в на-

[1] Не обязательно писать слово Attribute в конце имени класса, но это стандартное соглашение о наименованиях, используемое Microsoft. C# позволяет разработчику называть класс атрибута так, как ему хочется, например, Author – это допустимое имя класса для вашего атрибута.

шем случае Customer, обо всех его специальных атрибутах. В нашем коде перед тем, как вывести соответствующее сообщение на консоль, мы удостоверяемся, что класс Customer имеет атрибуты и что первый атрибут – AuthorAttribute. Кроме того, мы просматриваем все члены, относящиеся к классу Customer, и проверяем, есть ли у них специальные атрибуты. Если они есть, то перед тем как вывести на консоль соответствующее сообщение, мы проверяем, является ли первый атрибут атрибутом AuthorAttribute.

```
using System;
using System.Reflection;

public class interceptor
{
  public static void Main()
  {
    Object[] attrs = typeof(Customer).GetCustomAttributes(false);
    if ((attrs.Length > 0) && (attrs[0] is AuthorAttribute))
    {
      Console.WriteLine("Класс [{0}], написан программистом уровня {1}.",
          typeof(Customer).Name, ((AuthorAttribute)attrs[0]).level);
    }

    MethodInfo[] mInfo = typeof(Customer).GetMethods();
    for ( int i=0; i < mInfo.Length; i++ )
    {
      attrs = mInfo[i].GetCustomAttributes(false);
      if ((attrs.Length > 0) && (attrs[0] is AuthorAttribute))
      {
        AuthorAttribute a = (AuthorAttribute)attrs[0];
        Console.WriteLine("Метод [{0}], написан программистом уровня {1}.",
            mInfo[i].Name, (a.level));
        if (a.level == Skill.Junior)
        {
          Console.WriteLine("***Выполняем автоматический " +
            " анализ кода {0}***", a.level);
        }
      }
    }
  }
}
```

Важно отметить, что когда эта программа видит фрагмент кода, написанный программистом с уровнем Junior (начинающий), она автоматически выполняет скрупулезный анализ кода. Скомпилировав и запустив эту программу, вы получите на консоли следующее:

```
Класс [Customer], написан программистом уровня Guru.
Метод [Add], написан программистом уровня Senior.
Метод [Delete], написан программистом уровня Junior.
***Выполняем автоматический анализ кода Junior ***
```

Хотя наша программа-перехватчик не перехватывает создания объектов и вызовы методов, она показывает, каким образом реальный перехватчик может проверять атрибуты во время выполнения и предоставлять необходимые сервисы, определяемые атрибутами. И снова ключевой здесь является выделенная строка, представляющая собой специальный сервис, предоставляемый перехватчиком как результат проверки атрибутов.

Транзакции

Написать .NET-класс, использующий преимущества, предоставляемые поддержкой транзакций в службах COM+, очень просто. Все, что требуется – это задать несколько атрибутов при разработке, и ваши .NET-компоненты будут автоматически зарегистрированы в каталоге COM+ при первом обращении к ним. Другими словами, вы получаете не только упрощение программирования, но и оперативную и автоматическую регистрацию своего COM+-приложения.[1] Для создания класса .NET, поддерживающего транзакции, должны выполняться следующие условия:

1. Чтобы использовать службы COM+, ваш класс должен быть производным от класса ServicedComponent.
2. При описании класса необходимо использовать правильный атрибут Transaction – `Transaction(TransactionOption.Required)`, указывающий, что экземпляры вашего класса должны исполняться внутри транзакции.

Кроме того, можно использовать класс ContextUtil (являющийся частью пространства имен System.EnterpriseServices) для получения информации о контексте объекта COM+. Этот класс предоставляет основную функциональность, имеющуюся в COM+, включая такие методы, как SetComplete(), SetAbort() и IsCallerInRole(), и такие свойства, как IsInTransaction и MyTransactionVote.

Скажем еще, что хотя и не обязательно указывать какие-либо параметры установки COM+-приложения, сделать это следует, т. к. вы получаете возможность указать, что именно хотите, в том числе имя своего COM+-приложения, параметры активизации, его версию и т. д. Например, в следующем фрагменте кода, если не указать атрибут ApplicationName, то среда .NET будет использовать имя модуля в качестве имени COM+-приложения, отображаемого в Component Services Explorer (или в COM+ Explorer). Так, если имя модуля *crm.dll*, то именем COM+-приложения будет *crm*. Кроме этого атрибута мы также воспользуемся атрибутом ApplicationActivation для указания, что этот компонент будет установлен как библиотечное приложение, т. е. что компонент будет активизирован в создавшем его процессе.

---

[1] Автоматическая регистрация хороша при разработке, однако не используйте ее в реальной среде, т. к. не все клиенты имеют административные привилегии для установки COM+-приложений.

```
using System;
using System.Reflection;
using System.EnterpriseServices;

[assembly: ApplicationName(".NET Essentials CRM")]
[assembly: ApplicationActivation(ActivationOption.Library)]
[assembly: AssemblyKeyFile("originator.key")]
[assembly: AssemblyVersion("1.0.0.0")]
```

Все остальное должно выглядеть для вас очень знакомым. В методе Add() мы просто вызываем метод SetComplete(), когда новый клиент был успешно добавлен в базу данных. Если в процессе выполнения что-то произошло не так, мы сообщим о необходимости прервать транзакцию, вызвав метод SetAbort().

```
[Transaction(TransactionOption.Required)]
public class Customer : ServicedComponent
{
  public void Add(string strName)
  {
    try
    {
      Console.WriteLine("Новый клиент: {0}", strName);

      // Добавляем нового клиента в систему и выполняем
      // соответствующие обновления в нескольких базах данных.

      ContextUtil.SetComplete();
    }
    catch(Exception e)
    {
      Console.WriteLine(e.ToString());
      ContextUtil.SetAbort();
    }
  }
}
```

Вместо того чтобы самому вызывать методы SetComplete() и SetAbort(), вы можете использовать атрибут AutoComplete, как в следующем фрагменте кода, по сути, эквивалентном показанному выше методу Add():

```
[AutoComplete]
public void Add(string strName)
{
      Console.WriteLine("Новый клиент: {0}", strName);

      // Добавляем нового клиента в систему и выполняем
      // соответствующие обновления в нескольких базах данных.
}
```

Вот как компилируется эта сборка:

```
csc /t:library /out:crm.dll crm.cs
```

Поскольку это совместно используемая сборка, не забудьте зарегистрировать ее в GAC с помощью утилиты GAC:

```
gacutil /i crm.dll
```

В данный момент сборка не зарегистрирована как COM+-приложение, но мы не должны регистрировать ее вручную. Вместо этого .NET автоматически регистрирует за нас этот компонент и включает его в COM+-приложение во время первого обращения к компоненту. Итак, теперь напишем простую клиентскую программу, использующую этот компонент. Как вы увидите в следующем коде, мы создадим экземпляр объекта Customer и добавим нового клиента.

```
using System;

public class Client
{
  public static void Main()
  {
    try
    {
      Customer c = new Customer();
      c.Add("John Osborn");
    }
    catch(Exception e)
    {
      Console.WriteLine(e.ToString());
    }
  }
}
```

Эта программа компилируется следующим образом:

```
csc /r:crm.dll /t:exe /out:client.exe client.cs
```

Когда мы запустим это приложение, службы COM+ автоматически создадут COM+-приложение под названием `.NET Essentials CRM`, содержащее .NET-сборку *crm.dll* (рис. 4.5). Кроме добавления нашего ком-

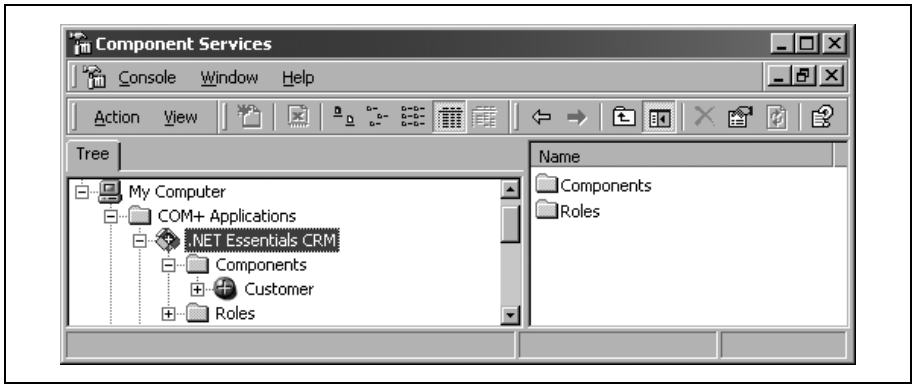


***Рис. 4.5.** Component Services Explorer*

понента в созданное COM+-приложение, .NET также проверит предоставленные нашими метаданными атрибуты и настроит соответствующие службы в каталоге COM+.

Организация пула объектов

Пул (*pool*) – это технический термин, обозначающий группу ресурсов, таких как соединения, потоки или объекты. Размещение нескольких объектов в пуле позволяет сотням клиентов совместно использовать небольшое количество объектов (это утверждение в одинаковой степени относится к потокам, соединениям и другим объектам). Следовательно, организация пула – это технология, минимизирующая расходование системных ресурсов, улучшающая производительность и облегчающая масштабируемость системы.

Отсутствующая в MTS *организация пула объектов* (*object pooling*) – это хорошее средство COM+, позволяющее поместить в пул объекты, поддерживающие транзакции и требующие больших затрат на свое создание. Аналогично поддержке транзакций, если требуется поддержка пула объектов в классе .NET, надо создать класс, производный от ServicedComponent, переопределить любой из методов Activate(), Deactivate() или CanBePooled() и указать требования к организации пула в атрибуте ObjectPooling, как показано в следующем примере:[1]

```
using System;
using System.Reflection;
using System.EnterpriseServices;

[assembly: ApplicationName(".NET Essentials CRM")]
[assembly: ApplicationActivation(ActivationOption.Library)]
[assembly: AssemblyKeyFile("originator.key")]
[assembly: AssemblyVersion("1.0.0.0")]

[Transaction(TransactionOption.Required)]
[ObjectPooling(MinPoolSize=1, MaxPoolSize=5)]
public class Customer : ServicedComponent
{
  public Customer()
  {
    Console.WriteLine("Некоторые ресурсоемкие операции "+
                      " по созданию объекта.");
  }

  [AutoComplete]
  public void Add(string strName)
  {
```

[1] Смешивать транзакции и организацию пула объектов следует с осторожностью. См. книгу издательства O'Reilly «COM and .NET Component Services», Джувала Лоуи (Juval Lowy).

```
    Console.WriteLine("Добавляем клиента: {0}", strName);
    // Добавляем нового клиента в систему и выполняем
    // соответствующие обновления в нескольких базах данных.
  }

  override protected void Activate()
  {
    Console.WriteLine("Activate");
    // Активизируется объект из пула.
    // Выполняем необходимую инициализацию.
  }

  override protected void Deactivate()
  {
    Console.WriteLine("Deactivate");
    // Объект будет возвращен в пул.
    // Выполняем необходимую зачистку.
  }

  override protected bool CanBePooled()
  {
    Console.WriteLine("CanBePooled");
    return true; // Возвращаем объект в пул.
  }
}
```

Методы Activate() и Deactivate() применяются для выполнения необходимой инициализации и очистки данных. Метод CanBePooled() позволяет сообщить COM+, может ли ваш объект быть помещен в пул в момент вызова этого метода. В конструктор следует поместить ресурсоемкую функциональность по созданию объекта, как показано в конструкторе этого класса.

Имея этот класс Customer, поддерживающий и транзакции и организацию пула, можно написать соответствующий клиентский код для проверки пула объектов. Для краткости мы создадим только два объекта, однако вы можете изменить это количество на любое желаемое, так чтобы можно было видеть эффект организации пула объектов. Чтобы гарантировать правильность конфигурации, удалите текущую версию COM+-приложения `.NET Essentials CRM` из Component Services Explorer перед запуском следующего кода:

```
for (int i=0; i<2; i++)
{
  Customer c = new Customer();
  c.Add(i.ToString());
}
```

Запуск этого кода даст следующий результат:

```
Некоторые ресурсоемкие операции по созданию объекта.
Activate
Добавляем клиента: 0
```

```
Deactivate
CanBePooled
Activate
Добавляем клиента: 1
Deactivate
CanBePooled
```

Мы создали два объекта, но поскольку мы использовали организацию пула объектов, только один объект реально необходим для поддержки наших вызовов, и поэтому вы видите только одну строку вывода: `Некоторые ресурсоемкие операции по созданию объекта`. В этом случае COM+ создает только один объект Customer, но активизирует и деактивизирует его дважды, чтобы обработать два произведенных вызова. После каждого вызова объект возвращается обратно в пул объектов. Когда приходит новый вызов, для обслуживания запроса из пула выбирается тот же самый объект.

Безопасность на основе ролей

Безопасность на основе ролей в MTS и COM+ радикально упрощает разработку и настройку систем безопасности бизнес-приложений. Достигается это тем, что она абстрагируется от сложных деталей работы со списком управления доступом (ACL) и идентификатором безопасности (SID). Все компоненты .NET, являющиеся частью COM+-приложения, могут использовать преимущества безопасности на основе ролей. Вы можете осуществить полную настройку безопасности на основе ролей с помощью Component Services Explorer, а можете управлять безопасностью в своем коде, чтобы обеспечить возможность точной настройки безопасности, отсутствующую в Component Services Explorer.

Настройка безопасности на основе ролей

Чтобы продемонстрировать поддержку безопасности на основе ролей, добавим в наше приложение `.NET Essentials CRM` две роли. Первая роль представляет собой агентов (`Agents`), которые могут использовать класс Customer любым способом, но не могут удалять записи о клиентах. Необходимо создать эту роль и добавить ее в локальную группу `Users` (Пользователи), как это показано на рис. 4.6. Вторая роль представляет собой менеджеров (`Managers`), которые могут использовать класс Customer любым способом, в том числе и удалять записи о клиентах. Создайте эту роль и добавьте ее в локальную группу `Administrators` (Администраторы).

Создав эти роли, необходимо включить проверку прав доступа для COM+-приложения `.NET Essentials CRM`. Откройте окно Properties (Свойства) COM+-приложения (выбрав `.NET Essentials CRM` и нажав клавиши <Alt>+<Enter>) и выберите вкладку Security (Безопасность). Включите проверку безопасности для вашего COM+-приложения, установив параметры, как показано на рис. 4.7.

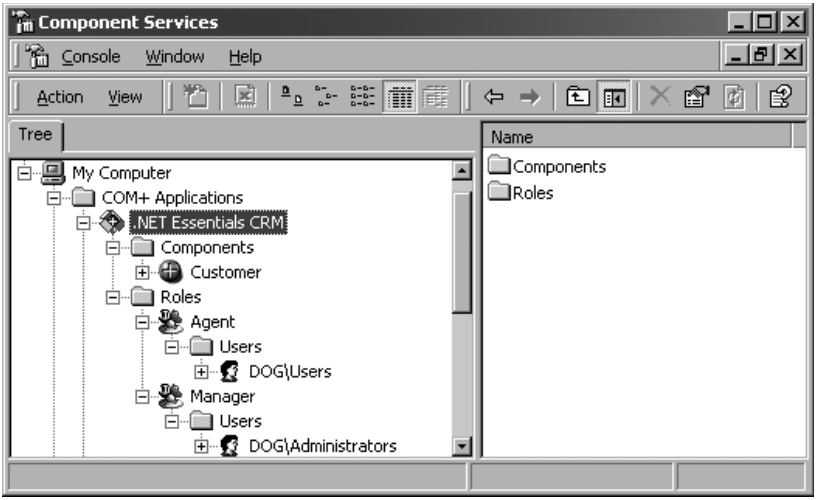


Рис. 4.6. *Создание ролей и добавление пользователей к ролям*

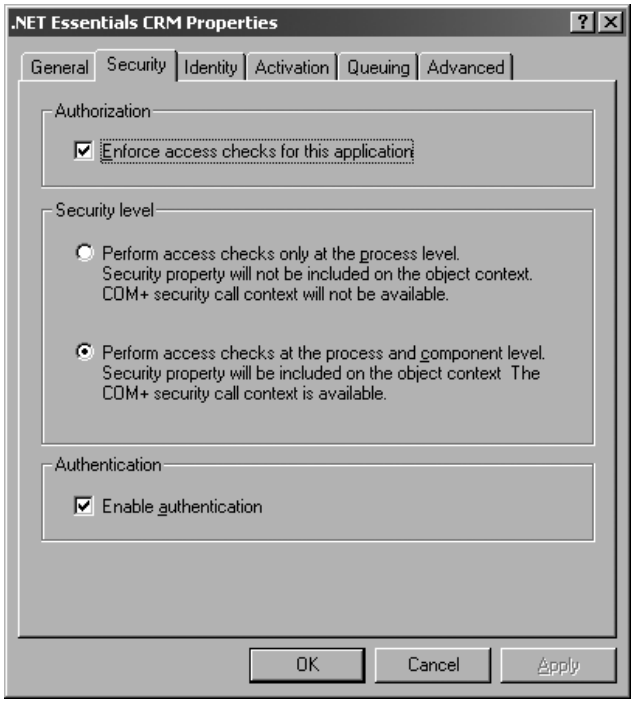


Рис. 4.7. *Включение авторизации для COM+-приложения*

Включив проверку прав доступа на уровне приложения, необходимо осуществить проверку прав и на уровне класса. Чтобы это сделать, откройте окно Properties (Свойства) класса `Customer` и выберите вкладку Security (Безопасность). Включите проверку прав доступа для этого .NET-класса, установив параметры, как показано на рис. 4.8. Этим мы сообщаем о том, что никто не может обращаться к классу Customer, кроме тех, кто принадлежит к ролям `Manager` или `Agent`.

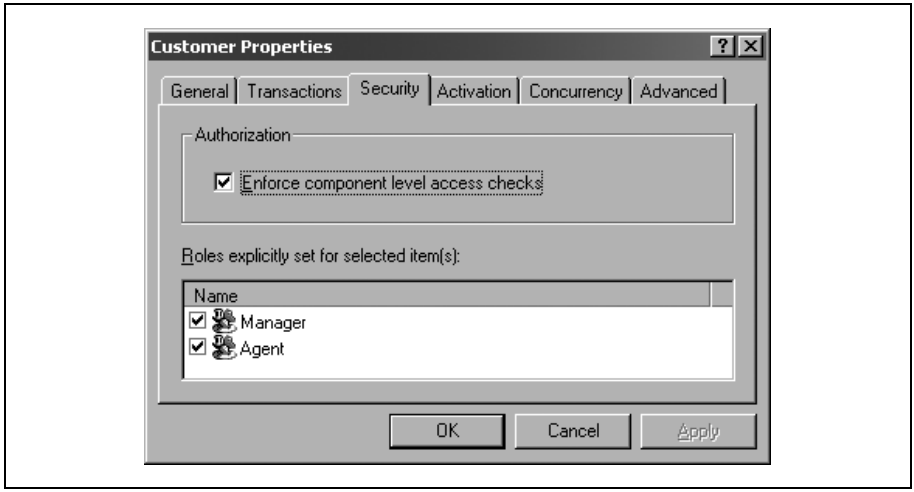


***Рис. 4.8.** Включение проверок прав доступа на уровне класса*

Теперь, если вы запустите клиентское приложение, разработанное в последнем разделе, то все будет работать, т. к. вы являетесь пользователем на своей машине. Но если сбросить флажки у ролей `Manager`[1] и `Agent` (рис. 4.8) и запустить клиентское приложение снова, то будет получено следующее сообщение:

```
System.UnauthorizedAccessException: Access is denied.
```

Вы получили исключение, т. к. удалили себя из ролей, имеющих доступ к классу Customer. Убедившись в этом, верните конфигурацию обратно к показанной на рис. 4.8, чтобы подготовить среду для следующего теста, который мы собираемся показать.

Программирование безопасности на основе ролей

Мы разрешили всем, кто относится к ролям `Agent` и `Manager`, обращаться к нашему классу, а теперь давайте создадим правило, позволяющее лишь пользователям, относящимся к роли `Manager`, удалять клиентов из

[1] Так как вы разработчик, то наверняка являетесь администратором на своей машине, поэтому вам следует отключить также и роль `Manager`, чтобы в приведенном нами тесте увидеть сообщение о нарушении прав доступа.

системы (в отсутствие лучшего примера). Поэтому давайте добавим в класс Customer новый метод – мы назовем его Delete(), – как показано в следующем коде. Все, кто принадлежит к ролям Agent или Manager, могут вызвать этот метод, поэтому сначала мы выведем на консоль учетную запись пользователя, под которой был вызван этот метод. После этого проверим, относится ли пользователь к роли Manager. Если это так, то позволим продолжить обработку вызова, а в противном случае сгенерируем исключение, указывающее, что только менеджеры могут выполнять удаление. Хотите, верьте, хотите, нет, но это и есть основная предпосылка для программирования безопасности на основе ролей.

```
[AutoComplete]
public void Delete(string strName)
{
  try
  {
    SecurityCallContext sec;
    sec = SecurityCallContext.CurrentCall;
    string strCaller = sec.DirectCaller.AccountName;

    Console.WriteLine("Пользователь: {0}", strCaller);

    bool bInRole = sec.IsCallerInRole("Manager");
    if (!bInRole)
    {
      throw new Exception ("Только менеджеры могут удалять клиентов.");
    }

    Console.WriteLine("Удаление клиента: {0}", strName);

    // Удаляем клиента из системы и выполняем
    // соответствующие обновления нескольких баз данных.
  }
  catch(Exception e)
  {
    Console.WriteLine(e.ToString());
  }
}
```

Ниже приведен клиентский код, включающий вызов метода Delete():

```
using System;

public class Client
{
  public static void Main()
  {
    try
    {
      Customer c = new Customer();
      c.Add("John Osborn");
      // Успешность выполнения зависит от роли,
      // под которой был вызван этот метод.
```

```
        c.Delete("Jane Smith");
      }
      catch(Exception e)
      {
        Console.WriteLine(e.ToString());
      }
    }
  }
```

Выполнив построение этой программы, вы можете проверить ее с помощью учетной записи, относящейся к локальной группе Users, т. к. мы ранее добавили эту группу в роль Agent. В Windows 2000 можно выполнить следующую команду для запуска командного окна под определенной учетной записью:

```
runas /user:dog\thuant cmd
```

Конечно, вам следует заменить dog и thuant соответственно на имя вашего компьютера и имя учетной записи пользователя. Запустив эту команду, надо будет ввести правильный пароль, и тогда появится новое командное окно. Выполните клиентское приложение под этой учетной записью, и вы увидите следующий результат:

```
Добавляем клиента: John Osborn
Пользователь: DOG\thuant
System.Exception: Только менеджеры могут удалять клиентов.
   at Customer.Delete(String strName)
```

Заметьте, что оператор Add() был выполнен успешно, однако операция удаления не прошла, т. к. мы выполняли клиентское приложение под учетной записью пользователя, не относящегося к роли Manager.

Чтобы это исправить, нужна пользовательская учетная запись, относящаяся к роли Manager, – для этого подойдет любой пользователь, принадлежащий к группе Administrators. Поэтому давайте запустим еще одно командное окно с помощью следующей команды:

```
runas /user:dog\administrator cmd
```

Снова выполним клиентское приложение и получим следующий результат:

```
Добавить клиента: John Osborn
Пользователь: DOG\Administrator
Удаление клиента: Jane Smith
```

Как видите, операция Delete() прошла без проблем потому, что мы выполнили клиентское приложение под учетной записью пользователя, относящегося к роли Manager.

Очереди сообщений

Кроме предоставления поддержки служб COM+, .NET также поддерживает организацию очередей сообщений. Те, кто работал со службами Microsoft Message Queuing (MSMQ), заметят, что базовая модель программирования сохранилась, однако классы пространства имен System.Messaging значительно облегчают разработку приложений с очередями сообщений. Пространство имен System.Messaging предоставляет поддержку базовой функциональности, такой как присоединение к очереди, открытие очереди, отправка сообщения в очередь, получение сообщений из очереди и извлечение сообщений из очереди. Чтобы продемонстрировать, насколько просты в использовании классы пространства имен System.Messaging, построим два простых приложения: одно для отправки сообщений в частную очередь на локальном компьютере, а второе – для получения этих сообщений из той же очереди.[1]

Отправка сообщений в очередь

Вот простая программа, ставящая объект Customer в частную очередь на локальном компьютере. Сначала заметьте, что мы должны включить пространство имен System.Messaging, т. к. оно содержит применяемые нами классы:

```
using System;
using System.Messaging;
```

Хотя следующая структура Customer очень проста, она может быть настолько сложной, насколько вы захотите, т. к. по умолчанию перед помещением в очередь она будет передана в XML-форматированный буфер.

```
public struct Customer
{
  public string Last;
  public string First;
}
```

Сначала наша программа проверяет, существует ли на локальном компьютере частная очередь. Если такая очередь отсутствует, программа создаст ее. Затем мы создадим экземпляр класса MessageQueue, передав ему имя целевой очереди. Имея этот объект MessageQueue, мы вызовем его метод Send(), передав ему объект Customer, как это показано в следующем фрагменте кода. Таким образом мы поместим наш объект в частную очередь.

---

[1] Чтобы выполнить эти программы, в вашей системе должен быть установлен компонент `MessageQueuing`. Чтобы убедиться в этом, запустите программу `Computer Management`, как показано на рис. 4.9.

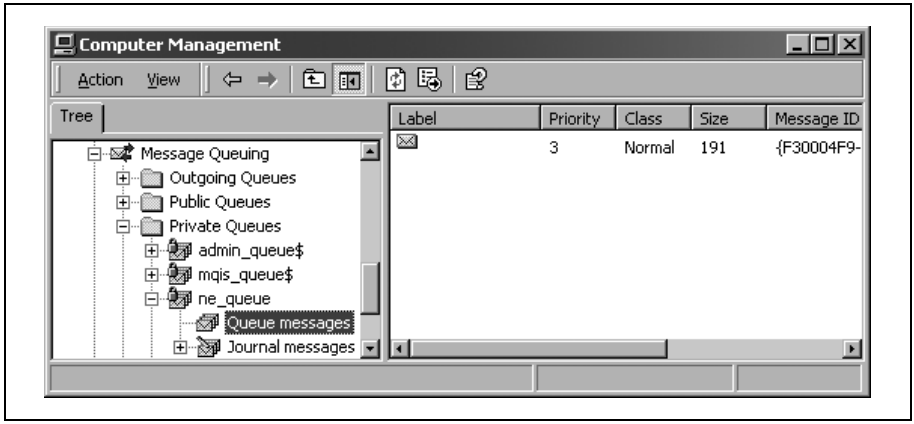


Рис. 4.9. *Наша частная очередь ne_queue с помещенным в нее сообщением*

```
public class Enqueue
{
  public static void Main()
  {
    try
    {
      string path = ".\\PRIVATE$\\NE_queue";
      if(!MessageQueue.Exists(path))
      {
        // Создаем нашу частную очередь.
        MessageQueue.Create(path);
      }

      // Инициализируем очередь.
      MessageQueue q = new MessageQueue(path);

      // Создаем новый объект.
      Customer c = new Customer();
      c.Last = "Osborn";
      c.First = "John";

      // Отправляем его в очередь.
      q.Send(c);
    }
    catch(Exception e)
    {
      Console.WriteLine(e.ToString());
    }
  }
}
```

Скомпилируйте эту программу при помощи следующей команды:

```
csc /t:exe /out:enqueue.exe enqueue.cs
```

Выполните эту программу, запустите программу Computer Management (Управление компьютером), и вы увидите ваше сообщение в частной очереди с именем ne_queue, как показано на рис. 4.9.

Получение сообщений из очереди

Теперь, когда в нашей частной очереди есть сообщение, напишем программу для получения и просмотра сообщений из очереди. Проверив, что желаемая частная очередь существует, мы инициализируем ее путем создания экземпляра класса MessageQueue, передав в качестве параметра путь к нашей частной очереди. Затем мы сообщаем объекту MessageQueue, что хотим получить из очереди объект типа Customer. Для фактического получения объекта необходимо вызвать метод Receive, передав ему время ожидания в терминах объекта TimeSpan, параметрами которого являются соответственно часы, минуты и секунды. И наконец, мы преобразуем тело полученного объекта Message в объект Customer и выводим его содержимое.

```
using System;
using System.Messaging;
using System.Runtime.Serialization;

public struct Customer
{
  public string Last;
  public string First;
}

public class Dequeue
{
  public static void Main()
  {
    try
    {
      string strQueuePath = ".\\PRIVATE$\\NE_queue";

      // Проверка, что очередь существует
      if (!MessageQueue.Exists(strQueuePath))
      {
        throw new Exception(strQueuePath + " не существует!");
      }

      // Инициализация очереди
      MessageQueue q = new MessageQueue(strQueuePath);

      // Указываем типы, которые мы хотим получить
      string[] types = {"Customer, dequeue"};
      ((XmlMessageFormatter)q.Formatter).TargetTypeNames = types;

      // Получаем сообщение (время ожидания 5 секунд)
      Message m = q.Receive(new TimeSpan(0,0,5));

      // Преобразуем тело сообщения в желаемый тип
```

```
            Customer c = (Customer) m.Body;
            Console.WriteLine("Клиент: {0}, {1}", c.Last, c.First);
          }
          catch(Exception e)
          {
            Console.WriteLine(e.ToString());
          }
        }
      }
```

Скомпилируйте и выполните программу, посмотрите на окно Computer Management, нажмите клавишу <F5> для обновления экрана, и вы увидите, что предыдущего сообщения там больше нет.

Заключение

В этой главе мы рассмотрели многие аспекты компонентного представления .NET-приложений, включая стратегии развертывания, распределенные вычисления и корпоративные службы, такие как управление транзакциями, организация пула объектов, безопасность на основе ролей и очереди сообщений. Надо отдать должное Microsoft за облегчение формирования компонентного представления в .NET Framework. Кстати, без .NET было бы невозможно показать полный код всех этих программ в одной главе книги.

5

Данные и XML

Почти все, что мы делаем в программной индустрии, так или иначе имеет отношение к данным. На каком-то этапе все разработчики программного обеспечения должны работать с данными, возможно, с применением базы данных, текстового файла, электронной таблицы или другого метода хранения данных. Существует много различных технологий для работы с данными и управления ими, и постоянно появляются все новые методы, позволяющие улучшить эти технологии. Данные методы начинаются с функциональных API и заканчиваются объектно-ориентированными средами и фирменными библиотеками.

Несколько лет назад для простого настольного приложения на Visual Basic было обычным обращение к базе данных в формате Microsoft Access, хранящейся на локальном жестком диске, но сейчас такая ситуация уже не типична. Современные приложения используют преимущества технологий распределенных компонентов для достижения масштабируемости и межплатформенного взаимодействия, тем самым расширяя сферу применимости приложения до уровня предприятия. Если ActiveX Data Objects (ADO) хорошо служили VB-приложению несколько лет назад, скоро они не смогут соответствовать возрастающим требованиям к масштабируемости, производительности и возможности взаимодействия между несколькими платформами.

Тут на помощь приходит ADO.NET, предоставляющая огромные преимущества, которые позволяют создавать еще лучшие корпоративные приложения. В этой главе рассказывается о преимуществах ADO.NET, ее архитектуре, основных классах ADO.NET, о том, как эти классы работают, и об интеграции ADO.NET и XML.

Архитектура ADO.NET

Объектная модель Microsoft ADO.NET включает в себя две отдельных группы классов: *компоненты сущностей* (*content components*) и *компоненты управляемых поставщиков* (*managed-provider components*). Компоненты сущностей включают класс DataSet и другие вспомогательные классы, такие как DataTable, DataRow, DataColumn и DataRelation. В этих классах находится фактическое содержимое пересылаемых данных. Компоненты управляемых поставщиков помогают в извлечении и обновлении данных. Разработчики могут использовать объекты Connection, Command и DataReader для непосредственных манипуляций с данными. Чаще всего разработчики отводят классу DataAdapter роль канала передачи данных между хранилищем данных и компонентами сущностей. Данные могут быть либо реальными строками из базы данных, либо иметь другую форму, такую как XML-файл или таблица Excel.

На рис. 5.1 показана высокоуровневая архитектура ADO.NET. Разработчики, знакомые с ADO, без труда поймут назначение объектов Connection и Command. Сейчас мы предлагаем вашему вниманию краткий обзор, а детали будут рассмотрены позже.

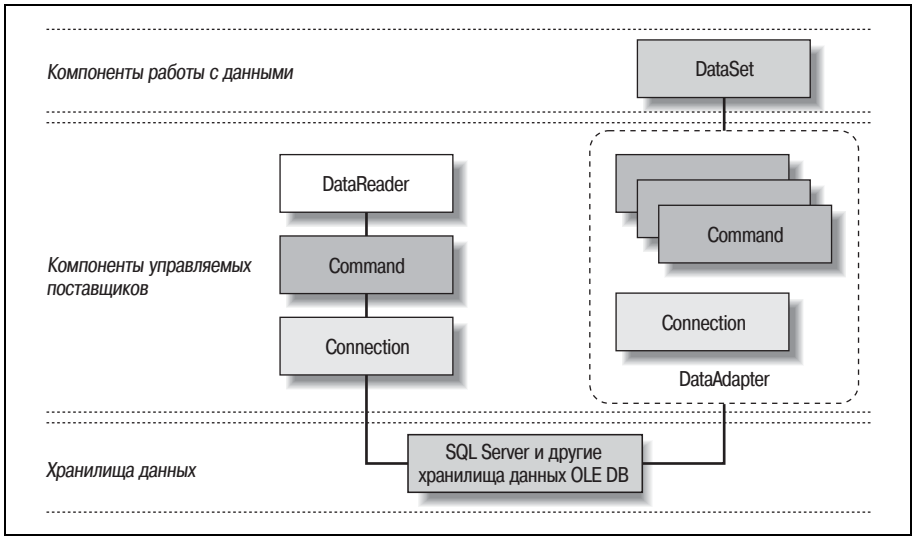


Рис. 5.1. *Высокоуровневая архитектура ADO.NET*

DataReader – это новый объект, предоставляющий быстрый, однонаправленный, только для чтения, доступ к данным. Эта структура схожа с ADO Recordset с серверным, однонаправленным, доступным только для чтения курсором.

Класс DataSet аналогичен облегченному кэшу отдельной базы данных из хранилища данных. Он позволяет читать и писать данные и схему в

XML и, как вы увидите позднее, тесно интегрирован с классом XmlDataDocument.

Класс DataAdapter представляет собой высокоуровневую абстракцию классов Connection и Command. Он позволяет загружать содержимое хранилища данных в объект DataSet и согласовывать изменения в объекте DataSet с хранилищем данных.

Достоинства ADO.NET

ADO.NET обладает рядом достоинств, которые можно разделить на следующие категории:

Межплатформенное взаимодействие

Возможность коммуникации между разнородными средами.

Масштабируемость

Возможность обслуживания большого количества клиентов без сокращения производительности системы.

Продуктивность

Возможность быстрой разработки надежных приложений для доступа к данным с помощью богатой и расширяемой компонентной объектной модели ADO.NET.

Производительность

Улучшение производительности по сравнению с предыдущими версиями ADO, обусловленное использованием модели отсоединенных данных.

Межплатформенное взаимодействие

Любые коммуникации связаны с обменом данными, будь это коммуникация между распределенными компонентами по методике «запрос-отклик» или технология, базирующаяся на сообщениях. Современные распределенные системы предполагают, что компоненты, участвующие в коммуникации, используют один и тот же протокол и формат данных. Это предположение является слишком ограничивающим для клиентской базы в масштабах предприятия или нескольких компаний. Уровни доступа к данным не должны накладывать таких ограничений.

В современной архитектуре распределенных сетевых приложений Microsoft Windows (Distributed interNet Applications, DNA) компоненты приложений передают и получают данные в виде отсоединенных наборов записей ADO. Компоненты, предоставляющие данные, как и компоненты, получающие данные, обязаны использовать технологию COM (Component Object Model). *Полезная нагрузка* (*payload*), т. е. фак-

тически передаваемое содержимое, упаковывается в формат данных под названием Network Data Representation (NDR). Из этих NDR-пакетов формируются потоки между компонентами.

Современным системам Windows DNA присущи две проблемы. Первая – необходимость наличия библиотеки COM на обеих сторонах коммуникационного канала. Вторая – сложность в установке и в организации этого взаимодействия через брандмауэры. Если ваши компоненты промежуточного уровня базируются на COM/DCOM и используются во внутренней сети, все будет хорошо. Другими словами: если все ваши компоненты основаны на технологии Microsoft, все нормально. Однако электронная коммерция (e-commerce) требует от корпоративных приложений умения взаимодействовать не только с системами, разработанными Microsoft. Необходимо улучшение библиотеки ADO, имеющее целью преодоление ограничений COM/DCOM и устранение затруднений, возникающих при совместном использовании данных многоплатформенными компонентами.

ADO.NET снимает ограничения на обмен данными, используя XML в качестве формата полезной нагрузки. Поскольку этот формат текстовый и легко обрабатывается, он представляет собой хороший выбор в качестве общего, платформо-независимого и переносимого формата данных. Более того, XML есть ни что иное, как структурированный текст, и поэтому его применение в качестве формата данных поверх сетевого протокола HTTP минимизирует проблемы, связанные с брандмауэрами. При использовании ADO и его XML-формата клиентам не надо знать COM для воссоздания упакованных данных. Необходим лишь XML-анализатор, доступный во многих вариантах на многих различных платформах. Производители и потребители данных должны только придерживаться XML-схемы для обмена данными между собой.

Масштабируемость

В модели «клиент-сервер» клиент обычно устанавливает соединение с сервером и удерживает его, пока не будут выполнены все запросы. Такое решение нормально работает в малых и средних приложениях, но оно не масштабируется на крупное предприятие. Как только количество клиентов достигает определенного предела, сервер становится узким местом, поскольку соединения с базами данных съедают ресурсы сети и процессора. ADO.NET отходит от модели «клиент-сервер», внедряя использование отсоединенных наборов данных. Когда клиент запрашивает какие-либо данные, они извлекаются, передаются клиенту, и (как только это становится возможным) соединение разрывается. Соединение между клиентом и источником данных является кратковременным, поэтому данный прием позволяет большему количеству клиентов запрашивать информацию от сервера, решая тем самым проблему ограниченного количества соединений.

Может показаться, что установка и разрыв соединений – это неудачная идея, т. к. затраты на установление соединений обычно высоки. Но это имеет значение лишь в отсутствие пула соединений. ADO.NET автоматически сохраняет соединения с источником данных в пуле, поэтому, когда приложение думает, что оно разрывает соединение, фактически оно возвращает соединение в пул ресурсов. Это позволяет повторно использовать соединения, избегая затрат на восстановление соединения с нуля.

Такая работа с данными не новость для ADO-программистов. Отсоединенные наборы записей появились в более ранних версиях ADO. Однако в ADO разработчик сам отвечал за реализацию этой возможности, тогда как в ADO.NET данные являются отсоединенными по своей природе.

Библиотека ADO.NET улучшена по сравнению с предшественником, вырастая из модели «клиент-сервер» в модель распределенных компонентов DCOM. Отсоединенные наборы данных как парадигма обмена данными делают ADO.NET значительно лучше масштабируемой по сравнению со своими предшественниками.

Продуктивность

Богатые возможности базовых классов ADO.NET позволяют разработчикам увеличить производительность труда. У тех, кто работает с текущей версией ADO, не должно быть трудностей с быстрым переходом к новой объектной модели, поскольку ADO.NET является естественным развитием ADO. Базовая функциональность осталась прежней. У нас все еще есть объект Connection, представляющий собой канал связи, через который выполняются команды.[1] В ADO.NET функциональность разбивается и распределяется между всеми объектами модели значительно лучше, чем в предыдущих версиях ADO. Например, объект Connection теперь отвечает только за присоединение и отсоединение от источника данных. В ADO.NET мы больше не можем выполнить запрос непосредственно через объект Connection. Не исключено, что некоторым разработчикам будет не хватать данной возможности, но это шаг в правильном направлении, т. к. он ведет к единству компонентной разработки.

ADO.NET также увеличивает производительность труда разработчиков за счет расширяемости. Базовые классы ADO.NET являются управляемым кодом, поэтому разработчики могут наследовать и расширять эти классы для своих специальных нужд. Те, кто предпочитает не выполнять эту низкоуровневую работу, могут использовать среду работы с данными в Visual Studio.NET для генерации этих классов.

---

[1] Вместе со знакомыми объектами Connection и Command ADO.NET предоставляет несколько новых объектов, таких как DataSet и DataAdapter. Все эти объекты обсуждались в разделе «Архитектура ADO.NET» ранее в этой главе.

Visual Studio.NET представляет собой прекрасное средство быстрой разработки приложений (RAD), базирующееся на ADO.NET. Для генерации типизированных наборов данных ADO.NET может применяться Component Designer. Типизированные наборы данных – это расширенные типы, моделирующие ваши данные. Код, полученный при помощи Component Designer, читается значительно легче, чем тот, который получался на выходе предыдущих генераторов кода от Microsoft. Кроме того, эти сгенерированные классы безопасны в отношении типов, сокращая таким образом возможность ошибок и позволяя компиляторам и CLR проверять правильность использования типов.

Короче говоря, ADO.NET улучшает продуктивность разработчиков за счет богатых и расширяемых базовых классов. Эти возможности дополняются богатым набором средств Visual Studio.NET для ADO.NET, обеспечивающих быструю разработку приложений.

Производительность

В ADO.NET в основном применяются отсоединенные наборы данных, поэтому система получает преимущества за счет улучшения производительности и масштабируемости. Сервер баз данных больше не является узким местом при росте количества запросов на соединение. Управляемые поставщики в ADO.NET также обеспечивают неявную организацию пула соединений, что сокращает время, требуемое для открытия соединения.

Ранее для обеспечения совместимости типов данных с COM маршалинг наборов записей требовал преобразования типов. Поскольку отсоединенный набор данных использует XML-формат, больше нет необходимости в таком преобразовании во время передачи данных, в отличие от работы с данными в формате NDR (Network Data Representation).

Компоненты сущностей

Компоненты сущностей служат для инкапсуляции данных. В предыдущих версиях ADO таким компонентом являлся объект Recordset. Данные, содержащиеся в компоненте Recordset, представлены в форме таблицы, состоящей из столбцов и строк. В ADO.NET данные, инкапсулированные в компоненте DataSet, имеют форму реляционной базы данных, состоящей из таблиц и отношений между ними. Это и есть основное усовершенствование в технологии доступа к данным. В этом разделе мы приведем общий обзор базовых классов, представляющих собой компоненты сущностей, в том числе DataSet, DataTable, DataColumn, DataRow, DataView и DataRelation.[1]

[1] Полный список всех классов можно найти в Microsoft .NET SDK.

DataSet

Если вы знакомы с ADO, то знаете, что обычно данные между компонентами передаются в виде *наборов записей* (*recordsets*). Набор записей содержит данные в табличной форме. Независимо от того, включает ли набор записей информацию из одной или нескольких таблиц базы данных, данные всегда возвращаются в виде строк и столбцов, словно они взяты из одной таблицы. ADO.NET позволяет компонентам приложения совместно использовать нечто большее, чем просто набор записей. Вот одна из наиболее важных черт ADO.NET: вместо набора записей передается набор данных (DataSet).

Объект DataSet можно рассматривать как размещенное в памяти представление базы данных. Он может содержать несколько объектов DataTable и DataRelation. В предыдущих версиях ADO наиболее близкую к этому функциональность можно было получить, обмениваясь данными в виде цепочки объектов Recordset. Клиентское приложение, получив эту цепочку, могло добраться до любого набора записей с помощью метода NextRecordset(); однако не было способа описать отношения между наборами записей в цепочке. В ADO.NET разработчики могут перемещаться и манипулировать коллекцией таблиц и отношений между ними.

Как мы упомянули ранее, ADO.NET применяет отсоединенные наборы данных, т. к. библиотека нацелена на распределенную архитектуру. Отсоединенный набор данных должен предоставлять способ отслеживать внесенные изменения. Объект DataSet предоставляет несколько методов, позволяющих позднее легко согласовать все выполненные с набором данных манипуляции с реальной базой данных (или другим источником данных). Эти методы включают: HasChanges(), HasErrors, GetChanges(), AcceptChanges() и RejectChanges(). Они могут применяться для проверки изменений, произошедших в объекте DataSet, получить изменения в форме измененного DataSet, проверить изменения на ошибки и затем принять или отменить изменения. Для того чтобы передать изменения в хранилище данных (наиболее обычная ситуация), просто попросите DataSet выполнить обновление.

DataSet полезен для корпоративных веб-приложений, по своей природе являющихся отсоединенными. Об изменениях данных на сервере мы не узнаем, пока не обновим отредактированные записи или не выполним любые другие действия, требующие согласования с базой данных.

Как показано на рис. 5.2, DataSet содержит две важных коллекции. Первая – это Tables (имеющая тип DataTableCollection), содержащая коллекцию таблиц, относящихся к набору данных. Вторая коллекция содержит все отношения между таблицами и соответственно имеет название Relations (и тип DataRelationCollection).

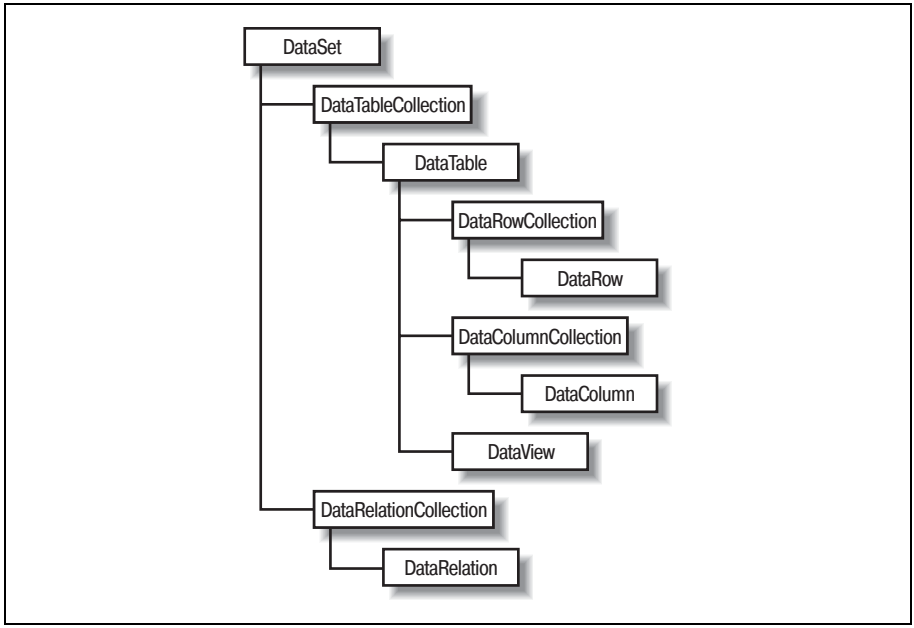


***Рис. 5.2.** Важнейшие объекты данных в ADO.NET, включенные в DataSet*

Создание набора данных: пример на C#

Все таблицы и отношения внутри набора данных доступны через свойства Tables и Relations соответственно. Обычно таблицы берутся из какого-либо источника данных, такого как SQL Server, или из другой базы данных; однако сначала мы хотим показать здесь основные элементы набора данных. Следующий блок кода на C# демонстрирует, как динамически создать набор данных, состоящий из двух таблиц, Orders и OrderDetails, и отношение между этими двумя таблицами.

```
using System;
using System.Data;

// Для краткости объявления класса и метода опущены...

// Создаем объект DataSet
DataSet m_ds = new DataSet("DynamicDS");

// Добавляем новую таблицу "Order" к коллекции таблиц объекта m_ds
m_ds.Tables.Add ("Order");

// Добавляем новые столбцы к таблице "Order"
m_ds.Tables["Order"].Columns.Add("OrderID",
                                 Type.GetType("System.Int32"));
m_ds.Tables["Order"].Columns.Add("CustomerFirstName",
                                 Type.GetType("System.String"));
```

```
m_ds.Tables["Order"].Columns.Add("CustomerLastName",
                                 Type.GetType("System.String"));
m_ds.Tables["Order"].Columns.Add("Date",
                                 Type.GetType("System.DateTime"));

// Регистрируем столбец "OrderID" как первичный ключ таблицы "Order"
DataColumn[] keys = new DataColumn[1];
keys[0] = m_ds.Tables["Order"].Columns["OrderID"];
m_ds.Tables["Order"].PrimaryKey = keys;

// Добавляем новую таблицу "OrderDetail" в коллекцию таблиц объекта m_ds
m_ds.Tables.Add ("OrderDetail");

// Добавляем новые столбцы к таблице "OrderDetail"
m_ds.Tables["OrderDetail"].Columns.Add("fk_OrderID",
                                Type.GetType("System.Int32"));
m_ds.Tables["OrderDetail"].Columns.Add("ProductCode",
                                Type.GetType("System.String"));
m_ds.Tables["OrderDetail"].Columns.Add("Quantity",
                                Type.GetType("System.Int32"));
m_ds.Tables["OrderDetail"].Columns.Add("Price",
                                Type.GetType("System.Currency"));

// Получаем объекты DataColumn из двух объектов DataTable в DataSet.
DataColumn parentCol = m_ds.Tables["Order"].Columns["OrderID"];
DataColumn childCol = m_ds.Tables["OrderDetail"].Columns["fk_OrderID"];

// Создаем и добавляем в DataSet отношение между таблицами.
m_ds.Relations.Add(new DataRelation("Order_OrderDetail",
                                    parentCol,
                                    childCol));
m_ds.Relations["Order_OrderDetail"].Nested = true;
```

Давайте отметим некоторые важные моменты в этом блоке кода. Создав с помощью оператора new экземпляр объекта DataSet, мы добавляем несколько таблиц с помощью метода Add объекта Tables. Аналогичные действия выполняются при добавлении столбцов в каждую коллекцию Columns объектов Tables. К любым из добавленных таблиц или столбцов можно позднее обратиться по имени. Чтобы назначить первичный ключ для таблицы Order, мы создаем массив DataColumn, содержащий одно или более полей, являющихся простым или составным ключом. В нашем случае имеется только одно ключевое поле, `OrderID`. Свойству таблицы PrimaryKey мы присваиваем массив ключевых столбцов. Для создания отношения между двумя таблицами мы сначала создаем объект DataRelation с именем `Order_OrderDetail`, содержащий два связанных столбца двух таблиц, а затем добавляем его в коллекцию Relations объекта DataSet. Последнее предложение указывает на наше желание представить отношение между таблицами Order и OrderDetail в виде вложенной структуры, что упрощает работу с этими сущностями в XML.

Следующий блок кода C# показывает, как вставить данные в каждую из двух таблиц:

```
DataRow newRow;
newRow = m_ds.Tables["Order"].NewRow();
newRow["OrderID"] = 101;
newRow["CustomerFirstName"] = "John";
newRow["CustomerLastName"] = "Doe";
newRow["Date"] = new DateTime(2001, 5, 1);
m_ds.Tables["Order"].Rows.Add(newRow);
newRow = m_ds.Tables["Order"].NewRow();
newRow["OrderID"] = 102;
newRow["CustomerFirstName"] = "Jane";
newRow["CustomerLastName"] = "Doe";
newRow["Date"] = new DateTime(2001, 4, 29);
m_ds.Tables["Order"].Rows.Add(newRow);

newRow = m_ds.Tables["OrderDetail"].NewRow();
newRow["fk_OrderID"] = 101;
newRow["ProductCode"] = "Item-100";
newRow["Quantity"] = 7;
newRow["Price"] = "59.95";
m_ds.Tables["OrderDetail"].Rows.Add(newRow);

newRow = m_ds.Tables["OrderDetail"].NewRow();
newRow["fk_OrderID"] = 101;
newRow["ProductCode"] = "Item-200";
newRow["Quantity"] = 1;
newRow["Price"] = "9.25";
m_ds.Tables["OrderDetail"].Rows.Add(newRow);

newRow = m_ds.Tables["OrderDetail"].NewRow();
newRow["fk_OrderID"] = 102;
newRow["ProductCode"] = "Item-200";
newRow["Quantity"] = 3;
newRow["Price"] = "9.25";
m_ds.Tables["OrderDetail"].Rows.Add(newRow);
```

Tables и Relations – это важные свойства объекта DataSet. Они не только описывают структуру базы данных в памяти, но и хранят содержимое DataSet в объектах DataTables внутри коллекции.

XML и наборы таблиц

Итак, у нас есть объект DataSet, заполненный таблицами и отношениями, давайте посмотрим, как он поможет в организации взаимодействия между компонентами. Ответом будет XML. DataSet имеет несколько методов, тесно интегрирующих DataSet с XML, делая его, таким образом, подходящим для универсального межплатформенного взаимодействия. Эти методы включают WriteXml(), WriteXmlSchema(), ReadXml() и ReadXmlSchema().

Метод WriteXmlSchema() выводит только схему таблиц, включая все таблицы и отношения между ними. WriteXml() может выводить как схему, так и табличные данные в виде строки в формате XML. И WriteXmlSchema() и WriteXml() принимают в качестве аргумента объект Stream, TextWriter, XmlWriter или строку, обозначающую имя файла. WriteXml() в качестве второго аргумента принимает объект XmlWriteMode, что позволяет при необходимости вместе с данными записать и схему.

По умолчанию WriteXml() пишет только данные. Чтобы также записать и схему, необходимо передать в качестве второго аргумента XmlWriteMode.WriteSchema. Можно также преобразовать в XML только данные, если явно использовать свойство XmlWriteMode.IgnoreSchema. Кроме того, можно задать режим XmlWriteMode.DiffGram. В этом режиме из DataSet будут выведены как исходные, так и измененные данные. Мы вернемся к этой теме, когда будем говорить о методе GetChanges().

Объект DataSet также предоставляет методы по восстановлению себя из XML-документа. Для чтения XML-документов с данными предназначен метод ReadXmlData(), для чтения XML-документов со схемами – ReadXmlSchema().

Следующий код создает XML-документ из ранее созданного набора данных:

```
// Выводим ранее показанный DataSet на консоль (а также в XML-файл)
m_ds.WriteXml(Console.Out, XmlWriteMode.WriteSchema);
m_ds.WriteXml("DS_Orders.xml", XmlWriteMode.WriteSchema);

// Создаем новый объект DataSet
DataSet ds2 = new DataSet("RestoredDS");
ds2.ReadXml("DS_Orders.xml");
```

Теперь посмотрим на получившийся XML-файл и на то, как в нем представлены наши данные:

```
<?xml version="1.0" standalone="yes"?>
<DynamicDS>
  <xs:schema id="DynamicDS"
             xmlns=""
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="DynamicDS" msdata:IsDataSet="true">

      <xs:complexType>
        <xs:choice maxOccurs="unbounded">

          <xs:element name="Order">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="OrderID"
                            type="xs:int" />
```

```
                <xs:element name="CustomerFirstName"
                            type="xs:string" minOccurs="0" />
                <xs:element name="CustomerLastName"
                            type="xs:string" minOccurs="0" />
                <xs:element name="Date"
                            type="xs:dateTime" minOccurs="0" />

                <xs:element name="OrderDetail"
                            minOccurs="0" maxOccurs="unbounded">

                  <xs:complexType>
                    <xs:sequence>
                      <xs:element name="fk_OrderID"
                                  type="xs:int" minOccurs="0" />
                      <xs:element name="ProductCode"
                                  type="xs:string" minOccurs="0" />
                      <xs:element name="Quantity"
                                  type="xs:int" minOccurs="0" />
                      <xs:element name="Price"
                                  msdata:DataType="System.Currency,
                                  mscorlib, Version=1.0.3300.0,
                                  Culture=neutral,
                                  PublicKeyToken=b77a5c561934e089"
                                  type="xs:string" minOccurs="0" />
                    </xs:sequence>
                  </xs:complexType>

                </xs:element>

              </xs:sequence>
            </xs:complexType>
          </xs:element>

        </xs:choice>
      </xs:complexType>

      <xs:unique name="Constraint1"
                 msdata:PrimaryKey="true">
        <xs:selector xpath=".//Order" />
        <xs:field xpath="OrderID" />
      </xs:unique>

      <xs:keyref name="Order_OrderDetail"
                 refer="Constraint1"
                 msdata:IsNested="true">
        <xs:selector xpath=".//OrderDetail" />
        <xs:field xpath="fk_OrderID" />
      </xs:keyref>

    </xs:element>
  </xs:schema>

  <... Раздел данных ...>

</DynamicDS>
```

Имя корневого элемента – DynamicDS, потому что таким было имя ранее созданного нами набора данных. Тег xsd:schema содержит определения всех таблиц и отношений набора данных DynamicDS. Поскольку мы указали, что отношение должно быть вложенным, схема показывает xsd:element для таблицы OrderDetail вложенным в xsd:element таблицы Order. Все столбцы также представлены в виде элементов xsd:element.

После определений таблиц в документе находятся определения ключей различных типов. Элемент xsd:unique используется вместе с msdata:PrimaryKey для ключей, как показано в элементе xsd:unique с именем Constraint1. Атрибут msdata:PrimaryKey обозначает первичный ключ, имеющий побочный эффект обеспечения уникальности (каждое значение OrderID в таблице должно быть уникальным).

Элемент xsd:keyref используется для внешних ключей, как показано в ключе Order_OrderDetail, ссылающемся на ключ Constraint1. Так объединяются строки таблиц OrderDetail и Order, для которых OrderDetail.fk_OrderID = Order.OrderID.

Теперь давайте посмотрим на часть XML-файла, содержащую данные:

```
<Order>
  <OrderID>101</OrderID>
  <CustomerFirstName>John</CustomerFirstName>
  <CustomerLastName>Doe</CustomerLastName>
  <Date>2001-05-01T00:00:00.0000000-04:00</Date>
  <OrderDetail>
    <fk_OrderID>101</fk_OrderID>
    <ProductCode>Item-100</ProductCode>
    <Quantity>7</Quantity>
    <Price>59.95</Price>
  </OrderDetail>
  <OrderDetail>
    <fk_OrderID>101</fk_OrderID>
    <ProductCode>Item-200</ProductCode>
    <Quantity>1</Quantity>
    <Price>9.25</Price>
  </OrderDetail>
</Order>
<Order>
  <OrderID>102</OrderID>
  <CustomerFirstName>Jane</CustomerFirstName>
  <CustomerLastName>Doe</CustomerLastName>
  <Date>2001-04-29T00:00:00.0000000-04:00</Date>
  <OrderDetail>
    <fk_OrderID>102</fk_OrderID>
    <ProductCode>Item-200</ProductCode>
    <Quantity>3</Quantity>
    <Price>9.25</Price>
  </OrderDetail>
</Order>
```

Эта часть XML-документа практически не требует пояснений. Для каждой строки данных таблицы Order мы получаем одну запись типа Order. То же самое для таблицы OrderDetail. Строка OrderDetail, связанная с определенной записью Order, вкладывается внутрь элемента Order.

Так как набор данных, по существу, отсоединен от своего источника, изменения в наборе данных должны отслеживаться им самим. Это делается с помощью методов HasChanges(), GetChanges() и Merge(). Приложение может внести изменения в набор данных и затем попросить объект DataAdapter синхронизировать изменения с источником данных посредством метода Update.

Следующий блок кода демонстрирует, как отслеживать и управлять изменениями в объекте DataSet:

```
m_ds.AcceptChanges();
/* Вносим изменения в набор данных */
m_ds.Tables["OrderDetail"].Rows[0]["Quantity"] = 12;

if(m_ds.HasChanges()){

  /* Получаем копию набора данных, содержащих внесенные  изменения. */
  DataSet changeDS = m_ds.GetChanges();

  /* Выводим измененные строки. */
  changeDS.WriteXml("ChangedDS.xml" , XmlWriteMode.DiffGram);

  /* Фиксируем все изменения. */
  m_ds.AcceptChanges();

}
```

Так как мы создаем этот набор данных динамически, первый вызов метода AcceptChange() сообщает объекту DataSet о сохранении сделанных к этому моменту изменений. Зная, что DataSet должен снова начать отслеживать изменения, мы затем изменяем значение ячейки Quantity в одной из строк таблицы OrderDetail. Затем мы запрашиваем у набора данных все выполненные изменения и переносим их в другой набор данных с именем changeDS. Этот набор данных приведен в следующей распечатке XML-кода, полученной в режиме DiffGram. Заметьте, что поскольку таблица OrderDetail является дочерней по отношению к таблице Order, то изменения также включают строку из родительской таблицы.

```
<?xml version="1.0" standalone="yes"?>
<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
                 xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
  <DynamicDS>
    <Order diffgr:id="Order1" msdata:rowOrder="0">
      <OrderID>101</OrderID>
      <CustomerFirstName>John</CustomerFirstName>
      <CustomerLastName>Doe</CustomerLastName>
      <Date>2001-05-01T00:00:00.0000000-04:00</Date>
```

```
      <OrderDetail diffgr:id="OrderDetail1"
                   msdata:rowOrder="0" diffgr:hasChanges="modified">
        <fk_OrderID>101</fk_OrderID>
        <ProductCode>Item-100</ProductCode>
        <Quantity>12</Quantity>
        <Price>59.95</Price>
      </OrderDetail>
    </Order>
  </DynamicDS>

  <diffgr:before>
    <OrderDetail diffgr:id="OrderDetail1" msdata:rowOrder="0">
      <fk_OrderID>101</fk_OrderID>
      <ProductCode>Item-100</ProductCode>
      <Quantity>7</Quantity>
      <Price>59.95</Price>
    </OrderDetail>
  </diffgr:before>

</diffgr:diffgram>
```

Мы бы хотели особо отметить, что объект DataSet – это наиболее важная конструкция в ADO.NET. Он не привязан к исходному представлению, такому как SQL Server или Microsoft Access, и поэтому характеризуется исключительной переносимостью. Его формат данных описывается в его собственной схеме, а данные – в виде чистого XML. Набор данных самодостаточен независимо от того, как он был создан – чтением данных из SQL Server, из Microsoft Access, из внешнего XML-файла или даже был динамически сгенерирован, как мы видели в приведенном ранее примере. Это переносимая сущность, основанная на XML, без сомнения должна стать новым стандартом обмена данными.

О DataSet мы сказали достаточно. Давайте углубимся и перейдем от DataSet к DataTable.

DataTable

Объект DataTable представляет собой таблицу данных и потому содержит коллекцию объектов DataColumn в свойстве Columns и коллекцию объектов DataRow в свойстве Rows. Свойство Columns определяет структуру таблицы, а свойство Rows предоставляет доступ к реальным данным строк. Поля таблицы представлены в виде объектов DataColumn, а записи таблицы – в виде объектов DataRow. Вот простой пример, выводящий имена всех столбцов в виде строки заголовков, а затем все строки данных:

```
/* Обходим содержимое DataTable и выводим заголовки всех
 * столбцов, а также все строки данных.
 */
DataTable myTable = m_ds.Tables["OrderDetail"];
```

```
/* Выводим названия всех столбцов. */
foreach(DataColumn c in myTable.Columns) {
  Console.Write(c.ColumnName + "\t");
}
Console.WriteLine(""); // новая строка

/* Обрабатываем каждую строку. */
foreach(DataRow r in myTable.Rows) {

  /* Выводим каждый столбец. */
  foreach(DataColumn c in myTable.Columns) {
    Console.Write(r[c] + "\t");
  }
  Console.WriteLine(""); // новая строка

}
```

Вот результат работы этого кода:

```
fk_OrderID    ProductCode     Quantity   Price
101           Item-100        12         59.95
101           Item-200        1          9.25
102           Item-200        3          9.25
```

Обычно таблица имеет одно или несколько полей, выступающих в качестве первичного ключа. Эта функциональность предоставляется свойством PrimaryKey. Так как первичный ключ может содержать более одного поля, это свойство является массивом объектов DataColumn. Мы вернемся к этому фрагменту кода, чтобы рассмотреть его в другом контексте. Заметьте, что в данном примере первичный ключ состоит только из одного поля, поэтому массив имеет размер 1.

```
// Регистрируем столбец "OrderID" как первичный ключ для таблицы "Order"
DataColumn[] keys = new DataColumn[1];
keys[0] = m_ds.Tables["Order"].Columns["OrderID"];
m_ds.Tables["Order"].PrimaryKey = keys;
```

Отношения и ограничения

Отношения (*relations*) определяют, каким образом таблицы базы данных соотносятся друг с другом. Объект DataSet целиком сохраняет коллекцию отношений между таблицами в свойстве Relations; однако каждая таблица, которая участвует в отношении, также должна знать о нем. За это отвечают два свойства объекта DataTable – ChildRelations и ParentRelations. В ChildRelations перечислены все отношения, в которых эта таблица участвует как основная таблица. С другой стороны, в ParentRelations перечислены отношения, в которых таблица играет роль подчиненной. Мы предоставим дополнительную информацию на тему отношений, когда в одном из следующих разделов главы рассмотрим объект DataRelation.

Обсуждая тему таблиц и отношений, важно понять, как устанавливаются ограничения. Существует два типа ограничений, которые можно устанавливать и накладывать на данные – UniqueConstraint и Fore-

ignKeyConstraint. UniqueConstraint требует, чтобы значение поля таблицы было уникальным. ForeignKeyConstraint определяет правила отношений между таблицами. Для ForeignKeyConstraint мы можем установить правила UpdateRule и DeleteRule, определяющие, каким образом приложение должно себя вести при обновлении или удалении строки данных в родительской таблице.

В табл. 5.1 показаны значения ограничений и соответствующие им действия для правил ForeignKeyConstraint.

Таблица 5.1. Типы ограничений и их назначение

Значение	Действия
None	Ничего.
Cascade	Зависимые строки (определяемые по внешнему ключу) удаляются/обновляются при удалении/обновлении родительской строки.
SetDefault	При удалении родительской строки внешние ключи в зависимых строках устанавливаются в значение по умолчанию.
SetNull	При удалении родительской строки внешние ключи в зависимых строках устанавливаются в null.

Ограничения активизируются, только когда свойство EnforceConstraint объекта DataSet установлено в `true`.

Следующий блок кода показывает, каким образом мы изменили ограничение внешнего ключа для таблиц Order и OrderDetail, чтобы разрешить каскадное удаление:

```
m_ds.Relations["Order_OrderDetail"].ChildKeyConstraint.DeleteRule =
Rule.Cascade;

m_ds.WriteXml("DS_BeforeCascadeDelete.xml");
m_ds.Tables["Order"].Rows[0].Delete();
m_ds.WriteXml("DS_AfterCascadeDelete.xml");
```

В результате работы этого кода в наборе данных остается лишь один заказ (заказ 102), содержащий одну позицию.

DataView

Объект DataView аналогичен *представлению* (*view*) в обычном программировании баз данных. Мы можем создать различные специализированные представления для объекта DataTable, каждое из которых имеет различные порядки сортировки и критерии фильтрации. Эти представления можно использовать для обхода, поиска и редактирования отдельных записей. Эта концепция ADO.NET ближе всего к старому набору записей (Recordset) из ADO. В ADO.NET DataView служит другой полезной задаче – привязке данных к формам Windows и вебформам. Мы покажем пример использования DataView, когда будем обсуждать привязку данных в главах 7 и 8.

DataRelation

Объект DataSet, являющийся лишь коллекцией объектов DataTable, не столь полезен. Коллекция объектов, возвращаемая серверным компонентом, дает мало преимуществ по сравнению с цепочкой наборов записей в предыдущих версиях ADO. Чтобы ваше клиентское приложение могло наиболее полно использовать возвращаемые таблицы, также должны быть возвращены отношения между ними. Здесь на помощь приходит объект DataRelation.

С помощью объекта DataRelation можно определить отношения между объектами DataTable. Клиентские компоненты могут либо просматривать отдельную таблицу, либо перемещаться по иерархии таблиц с помощью этих отношений. Например, можно найти определенную строку в родительской таблице и затем обойти все зависимые строки в дочерней таблице.

DataRelation содержит имя родительской таблицы, имя дочерней таблицы, столбец родительской таблицы (первичный ключ) и столбец дочерней таблицы (внешний ключ).

В DataSet может входить произвольное число объектов DataTable и DataRelation, поэтому ADO.NET позволяет создавать значительно более гибкую среду, в которой потребители информации могут использовать данные в том виде, в котором им хочется.

Одним из примеров может быть задача отображения всей информации об определенной родительской таблице и всех зависимых строк дочерней таблицы. В родительской таблице имеется десять строк. Каждая из строк родительской таблицы имеет десять зависимых строк в дочерней таблице. Рассмотрим два подхода, позволяющих предоставить эти данные потребителю. В первом подходе мы просто используем объединение (join) в строке запроса:

```
Select
  Order.CustomerFirstName, Order.CustomerLastName, Order.OrderDate,
  OrderDetail.ProductCode, OrderDetail.Quantity, OrderDetail.Price
from
  Order, OrderDetail
    where Order.OrderID = OrderDetail.fk_OrderID
```

Результирующий набор содержит 100 строк, где в каждой группе из десяти строк дублируется информация о родительской строке.

Второй подход состоит в том, что сначала из родительской таблицы извлекается список из десяти строк:

```
Select
  Order.OrderID,
  Order.CustomerFirstName, Order.CustomerLastName, Order.OrderDate
from
  Order
```

Затем для каждой из десяти строк родительской таблицы извлекаются зависимые строки из дочерней таблицы:

```
Select
  OrderDetail.ProductCode, OrderDetail.Quantity, OrderDetail.Price
from
  OrderDetail where fk_OrderID = thisOrderID
```

Второй подход из-за отсутствия избыточных данных расходует меньше ресурсов, однако вам придется совершить 11 обходов (один для родительской таблицы и десять для каждого родителя в дочерней таблице).

Лучше всего получить родительскую таблицу, дочернюю таблицу и отношения между ними с помощью одного обхода, вообще без избыточных данных. Это одно из самых больших преимуществ объекта DataSet. Следующий блок кода демонстрирует мощь, обусловленную наличием в DataSet и таблиц и отношений:

```
/*
 * Показываем один заказ по заданному номеру.
 */
public static void DisplaySingleOrder(DataSet m_ds, int iOrderID) {

  Decimal runningTotal = 0;
  Decimal lineTotal = 0;
  Decimal dPrice = 0;
  int iQty = 0;

  DataTable oTable = m_ds.Tables["Order"];

  // Ищем заказ в таблице Order.
  DataRow oRow = oTable.Rows.Find(iOrderID);

  /* Переходим к таблице OrderDetail
   * с помощью отношения Order_Details.
   */
  DataRow[] arrRows = oRow.GetChildRows("Order_OrderDetail");

  /* Выводим информацию по заказу. */
  Console.WriteLine ("Заказ: {0}", iOrderID);
  Console.WriteLine ("Имя: {0} {1}",
                     oRow["CustomerFirstName"].ToString(),
                     oRow["CustomerLastName"].ToString());
  Console.WriteLine ("Дата: {0}", oRow["Date"].ToString());
  Console.WriteLine("---------------------------");


  /*
   * Выводим и рассчитываем итог по строке для каждой позиции
   */
  for(int i = 0; i < arrRows.Length; i++)  {

    foreach(DataColumn myColumn in m_ds.Tables["OrderDetail"].Columns)
    {
      Console.Write(arrRows[i][myColumn] + " ");
    }
```

```
        iQty = System.Int32.Parse(arrRows[i]["Quantity"].ToString());
        dPrice = System.Decimal.Parse(arrRows[i]["Price"].ToString());

        lineTotal = iQty * dPrice;
        Console.WriteLine("{0}", lineTotal);

        /* Сохраняем промежуточный итог. */
        runningTotal += lineTotal;
      }

      /* Выводим общий итог по заказу. */
      Console.WriteLine("Total: {0}", runningTotal);
    }
```

Функция DisplaySingleOrder ищет единственную строку в таблице Order по заданному номеру заказа (OrderID). Когда эта строка найдена, мы запрашиваем у нее массив зависимых строк из таблицы OrderDetail в соответствии с отношением `Order_OrderDetail`. Получив массив объектов DataRow, мы переходим к отображению всех полей строки. Мы также подсчитываем значение `lineTotal` на основе заказанного количества и цены каждой позиции заказа, а также сохраняем значение в `runningTotal` для расчета итога по всему заказу. Далее показан результат работы функции DisplaySingleOrder.

```
Заказ: 101
Имя: John Doe
Дата: 5/1/2001 12:00:00 AM
---------------------------
101 Item-100 12 59.95 719.4
101 Item-200 1 9.25 9.25
Total: 728.65
```

Управляемые поставщики

Управляемый поставщик (*managed provider*) – это термин, обозначающий группу .NET-компонентов, реализующих фиксированный набор функциональности, определенный в архитектуре ADO.NET. Таким образом обеспечивается общий интерфейс для доступа к данным. Чтобы создать собственный управляемый поставщик, мы должны предоставить собственную реализацию объектов System.Data.Common.DbDataAdapter и реализовать такие интерфейсы, как IDbCommand, IDbConnection и IDataReader. Здесь мы не создаем собственный управляемый поставщик; однако в этом разделе мы все-таки углубимся в каждый из этих классов и интерфейсов.

В большинстве случаев разработчикам нет необходимости знать, как реализованы управляемые поставщики, хотя это может увеличить их продуктивность в отношении применения ADO.NET. Для разработки своих корпоративных приложений достаточно только знать, как использовать имеющиеся в наличии управляемые поставщики. В теку-

щей версии ADO.NET компания Microsoft предоставляет два управляемых поставщика: OLE DB и SQL. Управляемый поставщик OLE DB включает объекты OleDbConnection, OleDbCommand, OleDbParameter и OleDbDataReader. К управляемому поставщику SQL Server прилагается аналогичный набор объектов, имена которых начинаются с SqlClient вместо OleDb, что иллюстрирует рис. 5.3. Реализация этого базового набора функций для управляемых поставщиков содержится в пространстве имен System.Data. Реализация управляемого поставщика OLE DB находится в пространстве имен System.Data.OleDb, управляемого поставщика SQL – в System.Data.SqlClient.

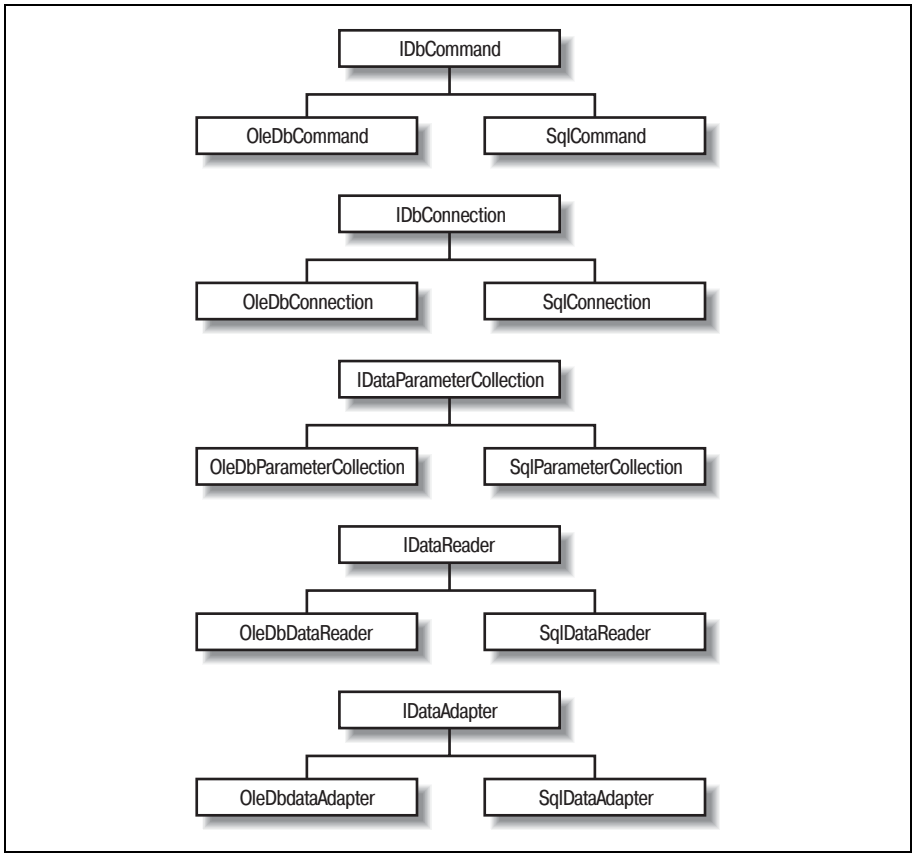


Рис. 5.3. *Иерархия классов среды управляемых поставщиков*

Оба управляемых поставщика реализуют набор интерфейсов для обращения к соответствующему хранилищу данных. Поставщик OLE DB полагается на OLE DB как на уровень доступа к широкому разнообразию источников данных, включая Microsoft SQL Server. По соображениям производительности поставщик SQL использует специальный протокол для связи с SQL Server напрямую. Независимо от способа по-

лучения данных результирующий набор остается неизменным. Это четкое разделение между управляемыми поставщиками и набором данных на основе XML помогает ADO.NET добиться переносимости данных.

На рис. 5.3 показаны базовые классы и две реализации управляемых поставщиков: OLE DB и SQL. Поскольку все управляемые поставщики, в том числе OLE DB и SQL, придерживаются фиксированного, общего набора интерфейсов (IDbCommand, IDbConnection, IDataParameterCollection, IDataReader и IDataAdapter), нетрудно адаптировать приложение к смене поставщика, если в этом возникает необходимость.

Объект Connection

И OleDbConnection и SqlConnection реализуют интерфейс System.Data.IDbConnection и, следовательно, наследуют такие свойства, как строка соединения и состояние соединения. Они реализуют базовый набор методов, определенных в IDbConnection, в том числе Open и Close.

В отличие от объекта Connection, в ADO поддержка транзакций в ADO.NET была перемещена в один из объектов Transaction (такой как OleDbTransaction и SqlTransaction). Причина состоит в том, что предположение о совпадении области действия транзакции с областью действия соединения не всегда справедливо. Например, могут существовать транзакции, охватывающие несколько соединений. Чтобы создать новую транзакцию, выполните метод BeginTransaction() объекта OleDbConnection или SqlConnection. Метод возвращает реализацию интерфейса IDbTransaction, поддерживающую такие функции транзакций, как Commit и Rollback. SqlTransaction также поддерживает сохранение контрольных точек, поэтому мы можем осуществить откат к определенной контрольной точке, а не отменять всю транзакцию. В табл. 5.2 перечислены поставщики, поддерживаемые в управляемом поставщике OLE DB.

Таблица 5.2. Управляемые поставщики ADO

Драйвер	Поставщик
SQLOLEDB	Microsoft OLE DB Provider для SQL Server
MSDAORA	Microsoft OLE DB Provider для Oracle
Microsoft.Jet.OLEDB.4.0	OLE DB Provider для Jet

И снова, если посмотрть на список методов, поддерживаемых OleDbConnection и SqlConnection, можно обнаружить, что их функциональность и фнкциональность старого объекта Connection в ADO во многом одинаковы. Однако ни OleDbConnection, ни SqlConnection больше не позволяют напрямую исполнять команды SQL или специфические для поставщика текстовые команды. Другими словами, метод Execute() больше в объекте соединения не поддерживается. Это более удачное

распределение функциональности между классами. Все выполнение осуществляется через объект Command, который обсуждается в следующем разделе вместе с описанием способа установки соединения.

Объекты Command и DataReader

К счастью для ADO-разработчиков, объекты SqlCommand и OleDbCommand в ADO.NET действуют так же, как объект Command в ADO; однако объекты Command в ADO.NET – единственный способ, с помощью которого мы можем выполнять запросы на вставку, обновление и удаление данных в ADO.NET. Это облегчает изучение объектной модели. Разработчики не столкнутся с таким разнообразием способов выполнения одного и того же действия, как в случае ADO, где можно выполнить запрос как через объект Connection, так и через Command или даже через Recordset.

Выполнение команды

Все команды связываются с объектом Connection через свойство Connection объекта SqlCommand или OleDbCommand. Объект Connection следует рассматривать как канал связи между компонентом, читающим данные, и сервером базы данных. Для выполнения команды должно быть открыто активное соединение. Объект Command также принимает параметры для исполнения хранимой процедуры на сервере. В левом верхнем углу рис. 5.5 показана связь между объектами Command, Connection и Parameters.

Существует два типа команд. Первый тип – это команда-запрос, возвращающая реализацию интерфейса IDataReader. Она реализуется с помощью метода ExecuteReader(). Команды второго типа обычно осуществляют обновление, вставку или удаление строк таблицы базы данных. Этот тип реализуется методом ExecuteNonQuery().

Одно из основных различий между объектом Command в ADO.NET и в ADO заключается в возвращаемых им данных. В ADO результатом выполнения команды-запроса является набор записей, содержащий данные в табличной форме. Однако в ADO.NET наборы записей больше не поддерживаются. Результатом выполнения команды-запроса теперь является объект DataReader (см. следующий раздел). Этот объект может быть либо OleDbDataReader для OLE DB, либо SqlDataReader для SQL Reader, либо любым классом, реализующим интерфейс DataReader для специальных случаев чтения данных. Получив подходящий объект DataReader, можно выполнить операцию Read и получить данные.

Применение объектов Command, Connection и DataReader – это низкоуровневый, прямой способ работы с управляемым поставщиком. Как вы узнаете немного позже, DataAdapter инкапсулирует все эти низкоуровневые операции как более прямой способ извлечения информации из источника данных в отсоединенный набор данных.

Объект DataReader

DataReader – это совсем новая, но вполне понятная для ADO-разработчиков концепция. DataReader аналогичен объекту-потоку в объектно-ориентированном программировании (ООП). Если вам требуется доступ к записям в однонаправленном, последовательном порядке, используйте DataReader, потому что он очень эффективен. Поскольку это серверный курсор, то соединение с сервером открыто в течение всего процесса чтения данных. Из-за постоянно открытого соединения мы советуем применять эту возможность осмотрительно и не заставлять DataReader задерживаться дольше, чем это требуется. В противном случае это может повлиять на масштабируемость приложения.

Следующий код демонстрирует основные приемы использования OleDbConnection, OleDbCommand и OleDbDataReader. Хотя здесь применяется управляемый поставщик OLE DB, строка соединения очень похожа на ту, которую мы раньше использовали в ADO.[1]

```
using System;
using System.Data;
using System.Data.OleDb;

public class pubsdemo {

  public static void Main() {

    /* Строка соединения OLE DB */
    String sConn =
      "provider=sqloledb;server=(local);database=pubs; Integrated
Security=SSPI";

    /* SQL-оператор */
    String sSQL = "select au_fname, au_lname, phone from authors";

    /* Создаем и открываем новое соединение */
    OleDbConnection oConn = new OleDbConnection(sConn);
    oConn.Open();

    /* Создаем новую команду и выполняем SQL-оператор */
    OleDbCommand oCmd = new OleDbCommand(sSQL, oConn);
    OleDbDataReader oReader = oCmd.ExecuteReader();

    /* Ищем индексы интересующих нас столбцов. */
    int idxFirstName = oReader.GetOrdinal("au_fname");
    int idxLastName = oReader.GetOrdinal("au_lname");
    int idxPhone = oReader.GetOrdinal("phone");

   /* Извлекаем и показываем каждый столбец в соответствии с его  индексом. */
    while(oReader.Read()) {
      Console.WriteLine("{0} {1} {2}",
                        oReader.GetValue(idxFirstName),
```

[1] Кроме того, вы можете создать объект Command для текущего соединения следующим образом: `oCmd = oConn.CreateCommand();`.

```
                        oReader.GetValue(idxLastName),
                        oReader.GetValue(idxPhone));
    }

  }
}
```

В этом коде открывается соединение с локальным SQL-сервером (используем интегрированную безопасность[1]) и выполняется запрос имени, фамилии и номера телефона из таблицы authors базы данных pubs. Если в системе не установлена база данных pubs, загрузите файл *instpubs.sql* в Query Analyzer и запустите его (этот файл находится в каталоге *MSSQL\Install* на вашем компьютере). Тем, кто установил примеры VS.NET Quickstart, необходимо изменить параметры сервера в строке соединения на `server=(local)\\NetSDK`, так как при установке примеров устанавливается и экземпляр NetSDK SQL Server, включающий также пресловутую базу данных Pubs. В следующем примере для получения той же информации применяется SqlClient:

```
using System;
using System.Data;
using System.Data.SqlClient;

public class pubsdemo {

  public static void Main() {

    /* Строка соединения для SQL Server */
    String sConn = "server=(local);database=pubs;Integrated Security=SSPI";

    /* SQL-оператор */
    String sSQL = "select au_fname, au_lname, phone from authors";

    /* Создаем и открываем новое соединение */
    SqlConnection oConn = new SqlConnection(sConn);
    oConn.Open();

    /* Создаем новую команду и выполняем SQL-оператор */
    SqlCommand oCmd = new SqlCommand(sSQL, oConn);
    SqlDataReader oReader = oCmd.ExecuteReader();

    /* Ищем индексы интересующих нас столбцов */
    int idxFirstName = oReader.GetOrdinal("au_fname");
    int idxLastName = oReader.GetOrdinal("au_lname");
```

[1] Пожалуйста, имейте в виду, что организация пула соединений с базами данных основывается на уникальности строк соединения. Если вы используете интегрированную модель безопасности SQL Server, и ваш код доступа к данным работает в контекстах безопасности каждого из подключенных пользователей, эффективность пула соединений уменьшится. Эта проблема решается созданием небольшого набора учетных записей Windows; мы не обсуждаем вопросы безопасности в этой книге особенно подробно по причине ее небольшого размера.

```
        int idxPhone = oReader.GetOrdinal("phone");

        /* Извлекаем и показываем каждый столбец в соответствии с его индексом. */
        while(oReader.Read()) {
          Console.WriteLine("{0} {1} {2}",
                            oReader.GetValue(idxFirstName),
                            oReader.GetValue(idxLastName),
                            oReader.GetValue(idxPhone));
        }

      }
    }
```

Объект DataAdapter

Вместе с введением DataReader в ADO.NET также появился объект DataAdapter, выполняющий функцию моста между источником данных и отсоединенным набором данных. Он содержит соединение и несколько команд для извлечения информации из хранилища данных в одну таблицу набора данных и обновления информации в хранилище данных в соответствии с изменениями, кэшированными в наборе данных. Несмотря на то что каждому объекту DataAdapter соответствует только одна таблица в наборе данных, можно иметь несколько адаптеров для заполнения объекта DataSet с несколькими DataTable. Иерархия классов DataAdapter показана на рис. 5.4. И OleDbDataAdapter и SqlDataAdapter – производные от DbDataAdapter, который, в свою очередь, является производным от абстрактного класса DataAdapter. Этот абстрактный класс реализует интерфейс IDataAdapter, определяющий поддержку методов Fill и Update. Интерфейс IDataAdapter определяется в пространстве имен System.Data, как и сам объект DataSet.

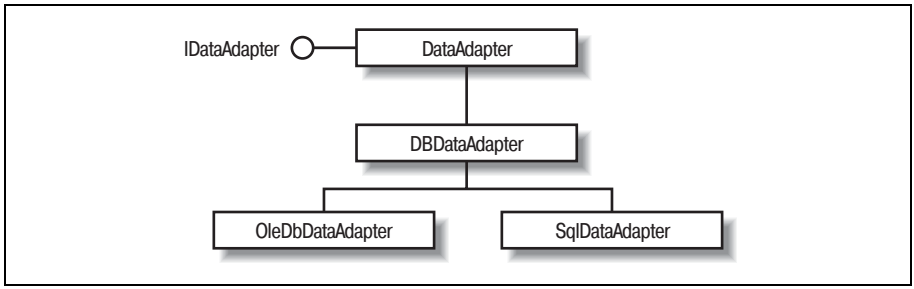


Рис. 5.4. *Иерархия класса DataSetCommand*

OleDbDataAdapter и SqlDataAdapter могут заполнять DataSet строками и обновлять источник данных при внесении изменений в набор данных. Например, OleDbAdapter может применяться для переноса данных от поставщика OLE DB в объект DataSet с помощью метода OleDbDataAdapter.Fill(). Затем можно изменить содержащиеся в объекте DataSet данные и зафиксировать внесенные изменения в базе дан-

ных с помощью метода OleDbDataAdapter.Update(). SqlDataAdapter поддерживает те же самые методы. Эти адаптеры действуют как посредники, передавая данные между сервером базы данных и отсоединенным набором данных.

DataAdapter извлекает данные при помощи SQL-команды SELECT (доступной через свойство SelectCommand). Эта команда применяется в реализации метода Fill интерфейса IDataAdapter. Обновление данных DataAdapter выполняет посредством SQL-команд UPDATE, INSERT и DELETE (доступных через свойства UpdateCommand, InsertCommand и DeleteCommand).

Наряду с методами Fill и Update класса DbDataAdapter, объекты OleDbDataAdapter и SqlDataAdapter также наследуют свойство TableMappings, коллекцию объектов TableMapping, позволяющих ставить в соответствие реальным именам столбцов базы данных пользовательские имена столбцов. Это еще больше изолирует объект DataSet от источника, из которого берутся данные. Даже именам таблиц и столбцов могут быть поставлены в соответствие более понятные имена, что облегчает использование набора данных. Разработчик приложения может больше заниматься тем, что он умеет делать лучше всего, а именно реализацией бизнес-логики, а не расшифровкой загадочных имен столбцов базы данных. На рис. 5.5 показана взаимосвязь между компонентами управляемых поставщиков.

Из четырех команд в объекте IDbDataAdapter только команда SELECT является обязательной. Остальные команды необязательны, т. к. могут генерироваться системой автоматически. Однако автоматическая генерация этих команд работает только тогда, когда соблюдаются определенные условия. Например, если объект DataAdapter заполняет набор данных из некоторого представления базы данных, включающего более одной таблицы, необходимо явно определить все четыре команды. Другой пример: когда адаптер не возвращает ключевых полей таблицы, система не сможет сгенерировать команды для вставки, обновления или удаления. Типичная схема использования DataAdapter включает следующие этапы:

- создание объекта DataAdapter (OleDbDataAdapter или SqlDataAdapter);
- задание строки запроса для внутреннего объекта SelectCommand;
- задание строки соединения для объекта Connection, связанного с объектом SelectCommand;
- задание строки запросов и соединений для InsertCommand, UpdateCommand или DeleteCommand (рекомендуется);
- вызов метода Fill() для заполнения заданного набора данных результатами выполнения запроса;
- внесение изменений и вызов метода Update() с измененным DataSet (не обязательно).

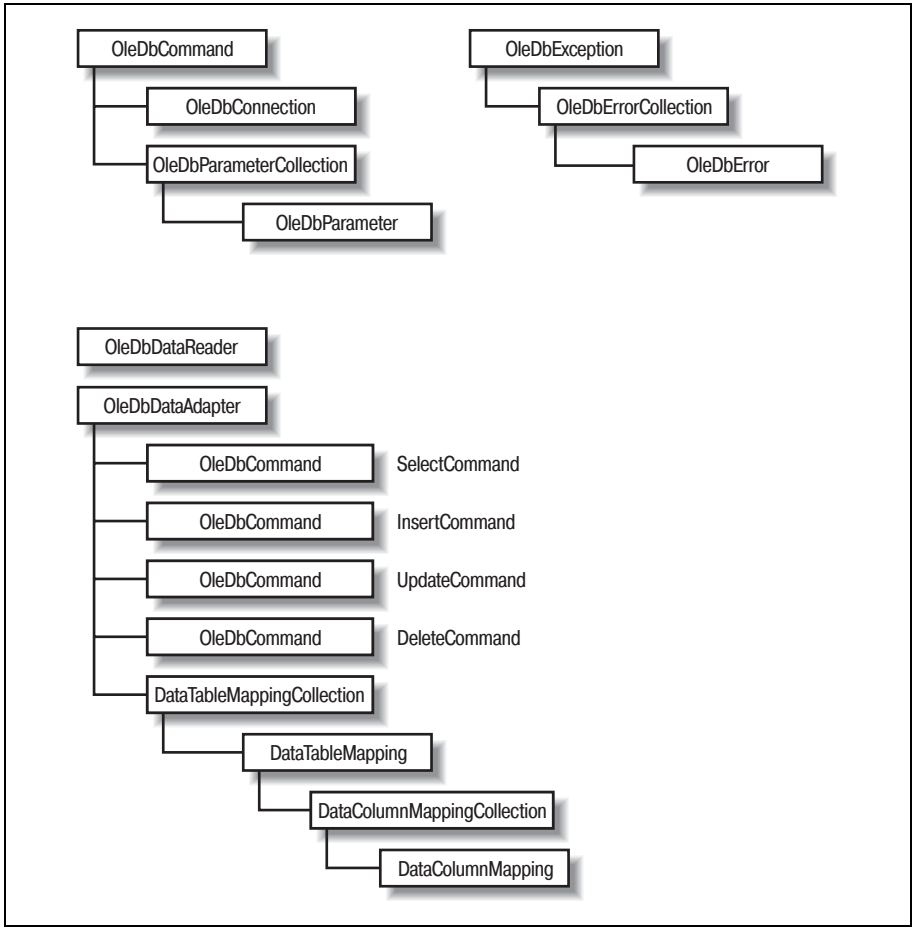


Рис. 5.5. *OleDbDataAdapter и вспомогательные классы*

Следующий фрагмент кода демонстрирует эти этапы:

```
static DataSet GenerateDS() {

  /* Создаем объект DataSet. */
  DataSet ds = new DataSet("DBDataSet");
  String sConn =
      "provider=SQLOLEDB;server=(local);database=pubs; Integrated
Security=SSPI ";

  /* Создаем адаптеры DataSet. */
  OleDbDataAdapter dsAdapter1 =
      new OleDbDataAdapter("select * from authors", sConn);

  OleDbDataAdapter dsAdapter2 =
      new OleDbDataAdapter("select * from titles", sConn);
```

```
    OleDbDataAdapter dsAdapter3 =
        new OleDbDataAdapter("select * from titleauthor", sConn);

    /* Заполняем набор данных из трех таблиц. */
    dsAdapter1.Fill(ds, "authors");
    dsAdapter2.Fill(ds, "titles");
    dsAdapter3.Fill(ds, "titleauthor");

    // Добавляем два отношения между тремя таблицами. */
    ds.Relations.Add("authors2titleauthor",
                     ds.Tables["authors"].Columns["au_id"],
                     ds.Tables["titleauthor"].Columns["au_id"]);

    ds.Relations.Add("titles2titleauthor",
                     ds.Tables["titles"].Columns["title_id"],
                     ds.Tables["titleauthor"].Columns["title_id"]);

    // Возвращаем DataSet
    return ds;

}
```

Здесь продемонстрировано создание набора данных из трех таблиц базы данных *pubs*. Набор данных также содержит два отношения, связывающие три таблицы между собой. Давайте посмотрим на набор данных в формате XML, генерируемый следующими строками кода:

```
DataSet ds = GenerateDS();
ds.WriteXml("DBDataSet.xml", XmlWriteMode.WriteSchema);
```

Содержимое файла *DBDataSet.xml* (с некоторыми пропусками для краткости) показано ниже:

```
<?xml version="1.0" standalone="yes"?>
<DBDataSet>
  <xsd:schema id="DBDataSet" targetNamespace="" xmlns=""
              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xsd:element name="DBDataSet" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">

          <xsd:element name="authors">
            <xsd:complexType>
              <xsd:sequence>
                <!-- для краткости определения столбцов упрощены -->
                <xsd:element name="au_id" type="xsd:string" />
                <xsd:element name="au_lname" type="xsd:string" />
                <xsd:element name="au_fname" type="xsd:string" />
                <xsd:element name="phone" type="xsd:string" />
                <xsd:element name="address" type="xsd:string" />
                <xsd:element name="city" type="xsd:string" />
                <xsd:element name="state" type="xsd:string" />
                <xsd:element name="zip" type="xsd:string" />
```

```
              <xsd:element name="contract" type="xsd:boolean" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>

        <!-- для краткости titles и titleauthor опущены -->

      </xsd:choice>
    </xsd:complexType>

    <xsd:unique name="Constraint1">
      <xsd:selector xpath=".//authors" />
      <xsd:field xpath="au_id" />
    </xsd:unique>

    <xsd:unique name="titles_Constraint1"
                msdata:ConstraintName="Constraint1">
      <xsd:selector xpath=".//titles" />
      <xsd:field xpath="title_id" />
    </xsd:unique>

    <xsd:keyref name="titles2titleauthor"
                refer="titles_Constraint1">
      <xsd:selector xpath=".//titleauthor" />
      <xsd:field xpath="title_id" />
    </xsd:keyref>

    <xsd:keyref name="authors2titleauthor"
                refer="Constraint1">
      <xsd:selector xpath=".//titleauthor" />
      <xsd:field xpath="au_id" />
    </xsd:keyref>

  </xsd:element>
</xsd:schema>

<!-- Некоторые строки для краткости удалены -->

<authors>
  <au_id>899-46-2035</au_id>
  <au_lname>Ringer</au_lname>
  <au_fname>Anne</au_fname>
  <phone>801 826-0752</phone>
  <address>67 Seventh Av.</address>
  <city>Salt Lake City</city>
  <state>UT</state>
  <zip>84152</zip>
  <contract>true</contract>
</authors>

<titles>
  <title_id>PS2091</title_id>
  <title>Is Anger the Enemy?</title>
  <type>psychology  </type>
  <pub_id>0736</pub_id>
```

```
    <price>10.95</price>
    <advance>2275</advance>
    <royalty>12</royalty>
    <ytd_sales>2045</ytd_sales>
    <notes>Carefully researched study of the effects of strong
    emotions on the body. Metabolic charts included.</notes>
    <pubdate>1991-06-15T00:00:00.0000</pubdate>
  </titles>
    <title_id>MC3021</title_id>
    <title>The Gourmet Microwave</title>
    <type>mod_cook</type>
    <pub_id>0877</pub_id>
    <price>2.99</price>
    <advance>15000</advance>
    <royalty>24</royalty>
    <ytd_sales>22246</ytd_sales>
    <notes>Traditional French gourmet recipes adapted for modern
    microwave cooking.</notes>
    <pubdate>1991-06-18T00:00:00.0000</pubdate>
  </titles>

  <titleauthor>
    <au_id>899-46-2035</au_id>
    <title_id>MC3021</title_id>
    <au_ord>2</au_ord>
    <royaltyper>25</royaltyper>
  </titleauthor>
  <titleauthor>
    <au_id>899-46-2035</au_id>
    <title_id>PS2091</title_id>
    <au_ord>2</au_ord>
    <royaltyper>50</royaltyper>
  </titleauthor>

</DBDataSet>
```

Таблицы представлены в виде пар тегов `<xsd:element name="имя таблицы" >...</xsd:element>`, содержащих определения столбцов. Кроме одного элемента xsd:element для каждой таблицы имеется по одному элементу xsd:unique для каждого ключа и по одному xsd:keyref на каждое отношение. Элемент xsd:unique определяет ключ в родительской таблице отношения. Тег xsd:keyref используется для дочерних таблиц отношения. Элемент xsd:keyref играет роль внешнего ключа и ссылается на ключ в родительской таблице.

Для экономии места мы сократили часть XML, содержащую данные, до одного автора, `Anne Ringer`, и двух написанных ею книг.

Может иметься несколько разных объектов DataAdapter для заполнения одного объекта DataSet. Каждый из этих адаптеров может обращаться к совершенно разным источникам данных или серверам. Другими словами, мы можем сконструировать объект DataSet, заполнен-

ный данными, получаемыми от нескольких серверов. В предыдущем примере у нас было три объекта DataAdapter; однако все они обращались к одному серверу.

Наборы данных и XML

XML быстро набирает популярность. Корпоративные приложения используют XML как основной формат для обмена данными.

ADO.NET отходит от набора записей на базе COM и в качестве своего формата транспортировки данных применяет XML. Поскольку XML является платформо-независимым, ADO.NET расширяет охватываемую область, включая в нее все, что может кодировать и декодировать XML. Это большое преимущество по сравнению с ADO, т. к. набор записей на базе COM зависит от платформы.

XML-анализаторы

Даже с учетом того, что XML представляет собой текстовый формат, пригодный для чтения пользователем, все равно нужен какой-нибудь способ программного чтения, просмотра и изменения XML. Этим и занимаются XML-анализаторы. Есть два типа XML-анализаторов: на основе деревьев и на основе потоков. В зависимости от предъявляемых требований оба типа анализаторов могут дополнять друг друга и хорошо работать на вас.

XML-анализаторы на основе деревьев читают XML-файл (или поток) целиком для формирования дерева XML-узлов. Считайте, что XML-узлы – это ваши XML-теги:

```
<car>
  <vin>VIO0000383148374</vin>
  <make>Acura</make>
  <model>Integra</model>
  <year>1995</year>
</car>
```

После преобразования в дерево эта информация будет иметь один корневой узел: `car`; под ним будут находиться четыре узла: `vin`, `make`, `model` и `year`. Как вы и могли подозревать, если XML-поток по своей природе очень велик, применение XML-анализатора на основе дерева будет не самой лучшей идеей. Дерево будет слишком большим и займет очень много памяти.

Потоковый XML-анализатор читает XML-поток по мере его получения. SAX (Simple API for XML, простой API для XML) – это спецификация для такого типа анализаторов. Анализатор в процессе чтения данных генерирует события, уведомляя приложение о теге или тексте, которые были им только что прочитаны. Он не пытается создать полное дерево всех XML-узлов, как это делает анализатор на основе дере-

ва. Следовательно, потребляемый им объем памяти минимален. Этот вид XML-анализаторов идеален для обработки больших XML-файлов с целью поиска малой части информации. В среде .NET появился еще один потоковый XML-анализатор: XmlReader. Если SAX при чтении данных проталкивает событие в приложение, XmlReader позволяет приложению получать данные из потока.

Microsoft реализует в своих анализаторах оба типа разбора XML. Поскольку XML настолько мощное средство, компания Microsoft, как и другие лидеры индустрии, внедряет XML почти во все свои разработки. Применение включает следующие области (но не ограничено ими):

- XML+HTTP в SOAP
- XML+SQL в SQL 2000
- XML в BizTalk
- XML+DataSet в ADO.NET
- XML в Web Services и Web Services Discovery (DISCO) (см. главу 6)

В этой главе мы обсудим XML+Dataset в ADO.NET, а XML в Web Services будет рассмотрен в следующей главе. Так как XML используется везде в архитектуре .NET, мы также приведем общий обзор XML-классов.

XML-классы

Чтобы разобраться в XML-анализаторе Microsoft на базе дерева, поддерживающем объектную модель документа (Document Object Model, DOM), нам необходимо знать лишь небольшое количество объектов:

- XmlNode и производные от него классы
- XmlNodeList, коллекция XmlNode
- XmlNamedNodeMap, коллекция XmlAttribute

Рассмотрим простой пример, чтобы увидеть, как XML-узлы преобразуются в эти объекты XML DOM.

XmlNode и его производные

XmlNode – это базовый класс, представляющий один узел XML-документа. В объектной модели почти все является производным от XmlNode, в том числе XmlAttribute, XmlDocument, XmlElement и XmlText, а также и другие типы XML-узлов.

Следующий фрагмент XML демонстрирует соответствие XML-тегов типам узлов дерева DOM:

```
<books>
<book category="How To">
<title>How to drive in DC metropolitan</title>
<author>Jack Daniel</author>
```

```
<price>19.95</price>
</book>
<book category="Fiction">
<title>Bring down the fence</title>
<author>Jack Smith</author>
<price>9.95</price>
</book>
</books>
```

После анализа этого XML-потока получится дерево, изображенное на рис. 5.6. Оно содержит один корневой узел, являющийся прямым производным от XmlNode. Этот корневой узел имеет тип XmlDocument. Под корневым узлом `books` располагаются два дочерних узла, оба производных от XmlNode. В этот раз они имеют тип XmlElement. Под каждым узлом элемента `book` имеется четыре дочерних элемента. Первый дочерний элемент – `category`. Этот узел имеет тип XmlAttribute, производный от XmlNode. Следующие три дочерних элемента имеют тип XmlElement: `title`, `author` и `price`. Каждый из этих элементов имеет один дочерний элемент типа XmlText.

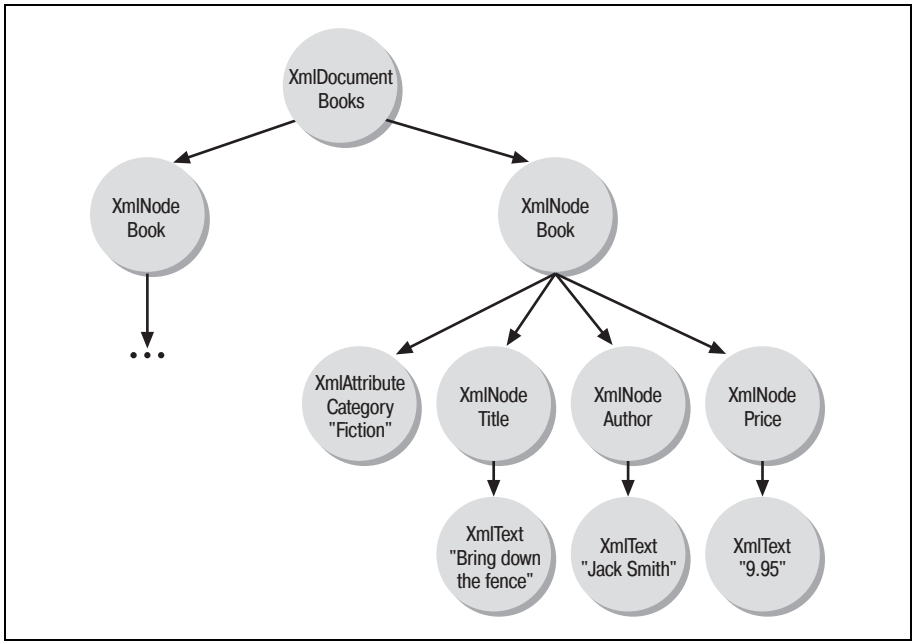


Рис. 5.6. *Древовидное представление XML-документа*

Как базовый класс XmlNode поддерживает несколько методов, помогающих в конструировании дерева XML-документа. Эти методы включают: AppendChild(), PrependChild(), InsertBefore(), InsertAfter() и Clone().

XmlNode также поддерживает группу свойств, помогающих в навигации по дереву XML-документа. Эти свойства включают: FirstChild, NextSibling, PrevousSibling, LastChild, ChildNodes и ParentNode. Свойство ChildNodes может применяться для перемещения вниз от корневого элемента. Для обхода в обратном направлении от любого из узлов предназначено свойство ParentNode.

XmlNodeList

Если XmlNode представляет один XML-элемент, то XmlNodeList представляет коллекцию из нуля или более объектов XmlNode. Свойство ChildNodes объекта XmlNode имеет тип XmlNodeList. Посмотрев на корневой узел `books`, мы увидим, что его свойство ChildNodes будет коллекцией из двух XmlNode. XmlNodeList поддерживает перечисление, и мы можем перебрать всю коллекцию и получить каждый из объектов XmlNode. Мы также можем обращаться к элементам коллекции по индексам, начиная с нуля.

Каждый объект XmlElement `book` будет иметь коллекцию ChildNodes, включающую объекты XmlElement `title`, `author` и `price`.

XmlNamedNodeMap

Аналогично XmlNodeList, XmlNamedNodeMap – это объект-коллекция. Точнее говоря, это коллекция объектов XmlAttribute, допускающих как перечисление, так и индексирование атрибутов по имени. Каждый объект XmlNode имеет свойство с именем Attributes. В случае элементов `book` эти коллекции содержат всего один атрибут – `category`.

XmlDocument

В дополнение ко всем методам и свойствам, поддерживаемым объектом XmlNode, этот производный от XmlNode класс добавляет или ограничивает методы и свойства. Здесь мы рассмотрим XmlDocument лишь как пример производного от XmlNode класса.

XmlDocument расширяет XmlNode и добавляет несколько вспомогательных функций. Эти функции используются для создания других типов XmlNode, таких как XmlAttribute, XmlComment, XmlElement и XmlText. Кроме обеспечения возможности создания XML-узлов других типов, XmlDocument также предоставляет механизм для загрузки и сохранения XML-содержимого.

Следующий код демонстрирует, как с помощью DOM программно сгенерировать XmlDocument:

```
using System;
using System.Xml;

public class XmlDemo {

  public static void Main() {
```

```
        // Код, демонстрирующий программную генерацию XmlDocument
        XmlDocument xmlDom = new XmlDocument();
        xmlDom.AppendChild(xmlDom.CreateElement("", "books", ""));
        XmlElement xmlRoot = xmlDom.DocumentElement;
        XmlElement xmlBook;
        XmlElement xmlTitle, xmlAuthor, xmlPrice;
        XmlText xmlText;

        xmlBook= xmlDom.CreateElement("", "book", "");
        xmlBook.SetAttribute("category", "", "How To");

        xmlTitle = xmlDom.CreateElement("", "title", "");
        xmlText = xmlDom.CreateTextNode("How to drive in DC metropolitan");
        xmlTitle.AppendChild(xmlText);
        xmlBook.AppendChild(xmlTitle);

        xmlAuthor = xmlDom.CreateElement("", "author", "");
        xmlText = xmlDom.CreateTextNode("Jack Daniel");
        xmlAuthor.AppendChild(xmlText);
        xmlBook.AppendChild(xmlAuthor);

        xmlPrice = xmlDom.CreateElement("", "price", "");
        xmlText = xmlDom.CreateTextNode("19.95");
        xmlPrice.AppendChild(xmlText);
        xmlBook.AppendChild(xmlPrice);

        xmlRoot.AppendChild(xmlBook);


        xmlBook= xmlDom.CreateElement("", "book", "");
        xmlBook.SetAttribute("category", "", "Fiction");

        xmlTitle = xmlDom.CreateElement("", "title", "");
        xmlText = xmlDom.CreateTextNode("Bring down the fence");
        xmlTitle.AppendChild(xmlText);
        xmlBook.AppendChild(xmlTitle);

        xmlAuthor = xmlDom.CreateElement("", "author", "");
        xmlText = xmlDom.CreateTextNode("Jack Smith");
        xmlAuthor.AppendChild(xmlText);
        xmlBook.AppendChild(xmlAuthor);

        xmlPrice = xmlDom.CreateElement("", "price", "");
        xmlText = xmlDom.CreateTextNode("9.95");
        xmlPrice.AppendChild(xmlText);
        xmlBook.AppendChild(xmlPrice);

        xmlRoot.AppendChild(xmlBook);

        Console.WriteLine(xmlDom.InnerXml);

    }

}
```

XmlDocument также поддерживает методы LoadXml и Load, формирующие полное XML-дерево из входного параметра. LoadXml принимает

в качестве аргумента строку в формате XML, а Load – имя файла, объект TextReader или XmlReader. Следующий пример продолжается с того места, на котором закончился предыдущий. Дерево XML сохраняется в файле с именем *books.xml*. Затем этот файл загружается обратно в другое XML-дерево. Это новое дерево выводится в тот же XML-поток, что и предыдущее.

```
...
xmlDom.Save("books.xml");
XmlDocument xmlDom2 = new XmlDocument();
xmlDom2.Load("books.xml");
Console.WriteLine(xmlDom2.InnerXml);
```

XmlReader

Объект XmlReader – это быстрый, некэширующий, однонаправленный способ доступа к потоковым XML-данным. Имеется два производных класса от XmlReader: XmlTextReader и XmlNodeReader. Оба этих объекта читают XML по одному тегу за раз. Единственное различие – это используемый ими источник данных. Как следует из названий, XmlTextReader читает поток чистого XML-текста, а XmlNodeReader читает поток узлов из XmlDocument. Поток может начинаться либо в начале XML-файла (если читается весь XML-документ), либо только с определенного узла (если документ читается частично).

Рассмотрим следующий фрагмент XML для обработки заказа. Если этот файл большой, неразумно читать его в XmlDocument и выполнять его обработку в объекте. Вместо этого следует читать только те узлы и атрибуты, которые нам нужны, и игнорировать остальные. Мы можем использовать для этого производные от XmlReader классы.

```
<Orders>
<Order id="ABC001" ...>
<Item code="101" qty="3" price="299.00" ...>Монитор 17"</Item>
<Item code="102" qty="1" price="15.99" ...>Клавиатура</Item>
<Item code="103" qty="2" price="395.95" ...>Системный блок</Item>
</Order>
<Order id="ABC002" ...>
<Item code="101b" qty="1" price="499.00" ...>Монитор 21"</Item>
<Item code="102" qty="1" price="15.99" ...>Клавиатура</Item>
</Order>
<...>
</Orders>
```

Следующий блок кода обходит и обрабатывает каждый заказ из большого входного файла `Order.xml`:

```
using System;
using System.IO;
using System.Xml;
```

```
class TestXMLReader
{

static void Main(string[] args)
{
    TestXMLReader tstObj = new TestXMLReader();
    StreamReader myStream = new StreamReader("Orders.xml");
    XmlTextReader xmlTxtRdr = new XmlTextReader(myStream);
    while(xmlTxtRdr.Read())
    {
        if(xmlTxtRdr.NodeType == XmlNodeType.Element
           && xmlTxtRdr.Name == "Order")
        {
            tstObj.ProcessOrder(xmlTxtRdr);
        }
    }
}

public void ProcessOrder(XmlTextReader reader)
{
    Console.WriteLine("Начинаем обработку заказа: " +
                      reader.GetAttribute("id"));
    while(!(reader.NodeType == XmlNodeType.EndElement
          && reader.Name == "Order")
          && reader.Read())
    {
        // Обрабатываем содержимое заказа
        if(reader.NodeType == XmlNodeType.Element
           && reader.Name == "Item")
        {
            Console.WriteLine("Код:" + reader.GetAttribute("code") +
                              ". Кол-во: " + reader.GetAttribute("qty"));
        }
    }
}

}
```

Посмотрим внимательнее, что происходит. После создания объекта XmlTextReader с потоком данных из строки нам останется лишь выполнять в цикле операцию Read() до тех пор, пока читать будет нечего. В процессе чтения мы начинаем обработку заказа, только если встречаем узел, имеющий тип XmlElement и имя `Order`. Внутри функции ProcessOrder мы читаем и обрабатываем все позиции в заказе, пока не встретим завершающий тег `Order`. Тогда мы возвращаемся из функции и переходим к поиску следующего тега `Order` для обработки следующего заказа.

XmlNodeReader аналогичен XmlTextReader, т. к. оба дают возможность последовательной обработки XML. Однако XmlNodeReader читает узлы XML из полного дерева XML или его фрагмента. Это значит, что XmlNodeReader не поможет вам в чтении больших XML-файлов.

XmlWriter

Объект XmlWriter – это быстрый, некэширующий способ записи потоковых XML-данных. Он также поддерживает пространства имен. Единственным производным от XmlWriter классом является XmlTextWriter.

XmlWriter поддерживает пространства имен путем предоставления нескольких перегруженных функций, которые используют пространство имен для связи с элементом. Если пространство имен уже определено и для него существует префикс, XmlWriter автоматически записывает имя элемента с этим префиксом. Почти все методы записи элементов перегруженные – для поддержки пространств имен.

Следующий код показывает, как использовать объект XmlTextWrite для создания действительного XML-файла:

```
XmlTextWriter writer =
  new XmlTextWriter("test.xml", new System.Text.ASCIIEncoding());
writer.Formatting = Formatting.Indented;
writer.Indentation = 4;
writer.WriteStartDocument();
writer.WriteComment("Comment");
writer.WriteStartElement("ElementName", "myns");
writer.WriteStartAttribute("prefix", "attrName", "myns");
writer.WriteEndAttribute();
writer.WriteElementString("ElementName", "myns", "value");
writer.WriteEndElement();
writer.WriteEndDocument();
writer.Flush();
writer.Close();
```

В результате в файле *text.xml* будет создан следующий XML-документ:

```
<?xml version="1.0" encoding="us-ascii"?>
<!--Comment-->
<ElementName prefix:attrName="" xmlns:prefix="myns" xmlns="myns">
    <prefix:ElementName>value</prefix:ElementName>
</ElementName>
```

XslTransform

XslTransform преобразует XML из одного формата в другой. Этот класс обычно применяется в программах преобразования данных или для преобразования XML в HTML для отображения XML-данных в броузере. Следующий код демонстрирует, каким образом происходит такое преобразование:

```
using System;
using System.Xml;           // XmlTextWriter
using System.Xml.Xsl;       // XslTransform
using System.Xml.XPath;     // XPathDocument
using System.IO;            // StreamReader
```

```
public class XSLDemo {
  public static void Main() {
    XslTransform xslt = new XslTransform();
    xslt.Load("XSLTemplate.xsl");
    XPathDocument xDoc = new XPathDocument("Books.xml");
    XmlTextWriter writer = new XmlTextWriter("Books.html", null);
    xslt.Transform(xDoc, null, writer);
    writer.Close();
    StreamReader stream = new StreamReader("Books.html");
    Console.Write(stream.ReadToEnd());
  }
}
```

По существу, в этом коде выполняется преобразование XML из файла *Books.xml*, который вы видели раньше, в HTML для отображения в броузере. Хотя в приведенном фрагменте вы и можете заменить XPathDocument на XmlDocument, предпочтительным в этом случае является класс XPathDocument, т. к. он оптимизирован для XSLT-преобразований.[1] На рис. 5.7 и 5.8 показаны исходный XML и HTML, полученный при просмотре в броузере.

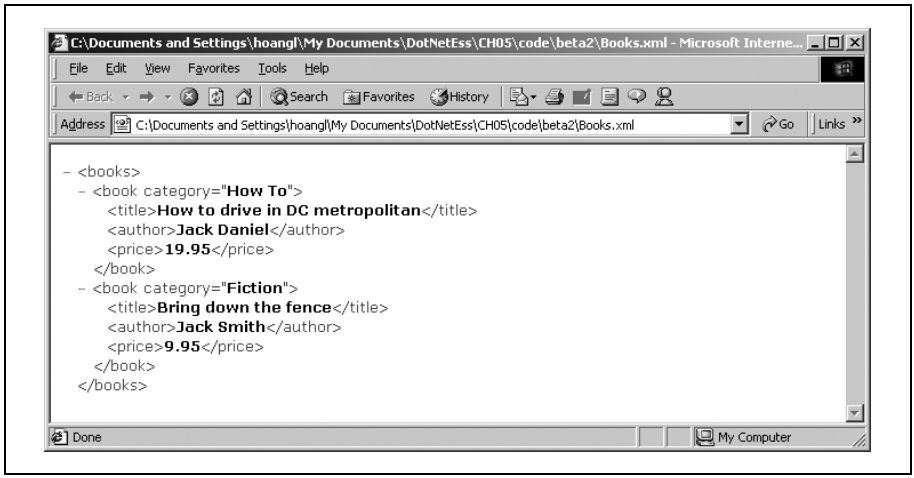


Рис. 5.7. *Books.xml, отображаемый в Internet Explorer*

Для преобразования XML был использован следующий XSL-шаблон:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match = "/" >
```

[1] XPathDocument загружает данные быстрее, чем XmlDocument, поскольку он не выполняет идентификацию узлов и проверку правил. Побочным эффектом этого является то, что содержимое доступно только для чтения.

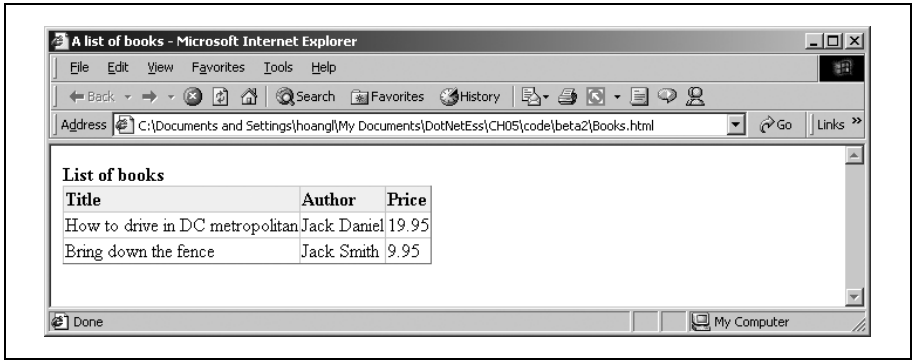


***Рис. 5.8.** Books.html, отображаемый в Internet Explorer*

```
<html>
<head><title>A list of books</title></head>
<style>
.hdr { background-color=#ffeedd; font-weight=bold; }
</style>
<body>
<B>List of books</B>
<table style="border-collapse:collapse" border="1">
<tr>
  <td class="hdr">Title</td>
  <td class="hdr">Author</td>
  <td class="hdr">Price</td>
</tr>
<xsl:for-each select="//books/book">
<tr>
  <td><xsl:value-of select="title"/></td>
  <td><xsl:value-of select="author"/></td>
  <td><xsl:value-of select="price"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>

</xsl:template>
</xsl:stylesheet>
```

XmlDataDocument

Одной из важнейших черт ADO.NET является тесная интеграция DataSet с XML. Объект DataSet может быть легко преобразован в XML и обратно, облегчая обмен данными с любыми другими компонентами в корпоративной системе. Схема набора данных может быть загружена и сохранена в виде определения XML-схемы (XML Schema Definition, XSD), как было описано выше.

XmlDataDocument может быть связан с объектом DataSet. Следующий фрагмент кода иллюстрирует, как установить эту связь:

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Xml;

class TestXMLDataDocument
{

static void Main(string[] args)
{
    TestXMLDataDocument tstObj = new TestXMLDataDocument();

    // создаем XmlDataDocument с DataSet
    XmlDataDocument doc = tstObj.GenerateXmlDataDocument();

    XmlNodeReader myXMLReader = new XmlNodeReader(doc);
    while (myXMLReader.Read())
    {
        if(myXMLReader.NodeType == XmlNodeType.Element
            && myXMLReader.Name == "Orders")
        {
            tstObj.ProcessOrder(myXMLReader);
        }
    }
}

public void ProcessOrder(XmlNodeReader reader)
{
    Console.Write("Начинаем обработку заказа: ");
    while(!(reader.NodeType == XmlNodeType.EndElement
        && reader.Name == "Orders")
        && reader.Read())
    {
        if(reader.NodeType == XmlNodeType.Element
            && reader.Name == "OrderID")
        {
            reader.Read();
            Console.WriteLine(reader.Value);
        }
        if(reader.NodeType == XmlNodeType.Element
            && reader.Name == "OrderDetails")
        {
            ProcessLine(reader);
        }
    }
}

public void ProcessLine(XmlNodeReader reader)
{
    while(!(reader.NodeType == XmlNodeType.EndElement
        && reader.Name == "OrderDetails")
```

```
        && reader.Read())
    {
        if(reader.NodeType == XmlNodeType.Element && reader.Name == "
ProductID")
        {
            reader.Read();
            Console.Write(".  Код: " + reader.Value);
        }
        if(reader.NodeType == XmlNodeType.Element && reader.Name == "
Quantity")
        {
            reader.Read();
            Console.WriteLine(".  Количество: " + reader.Value);
        }
    }
}
public XmlDataDocument GenerateXmlDataDocument()
{
    /* Создаем объект DataSet. */
    DataSet ds = new DataSet("DBDataSet");
    String sConn =
        "provider=SQLOLEDB;server=(local);database=NorthWind;Integrated
Security=SSPI";

    /* Создаем адаптеры для DataSet. */
    OleDbDataAdapter dsAdapter1 =
        new OleDbDataAdapter("select * from Orders", sConn);

    OleDbDataAdapter dsAdapter2 =
        new OleDbDataAdapter("select * from [Order Details]", sConn);

    /* Заполняем набор данных из трех таблиц. */
    dsAdapter1.Fill(ds, "Orders");
    dsAdapter2.Fill(ds, "OrderDetails");

    DataColumn[] keys = new DataColumn[1];
    keys[0] = ds.Tables["Orders"].Columns["OrderID"];
    ds.Tables["Orders"].PrimaryKey = keys;


    /* Добавляем два отношения между тремя таблицами. */
    ds.Relations.Add("Orders_OrderDetails",
        ds.Tables["Orders"].Columns["OrderID"],
        ds.Tables["OrderDetails"].Columns["OrderID"]);

    ds.Relations["Orders_OrderDetails"].Nested = true;
    //ds.WriteXml("NorthWindOrders.xml");

    return new XmlDataDocument(ds);

}

}
```

В предыдущем разделе в описании объекта DataSet было показано, что при наличии набора данных можно сохранить данные из него в XML-строке или в файле. В этот раз мы продемонстрировали, как преобразовать объект DataSet в XmlDataDocument, с которым можно выполнять манипуляции в памяти.

Заключение

В данной главе описано ядро ADO.NET. Ориентируясь на отсоединенные наборы данных, ADO.NET позволяет не только построить высокопроизводительные, масштабируемые решения для электронной коммерции, но и позволяет приложению охватывать другие платформы за счет использования XML. Эта глава представляет собой общий обзор классов, составляющих ADO.NET, и служит для того, чтобы познакомить вас с библиотекой System.Xml. В следующей главе мы углубимся в построение программного обеспечения в виде служб. Мы воспользуемся ADO.NET как механизмом доступа и обмена данными в наших программных службах.

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-054-5, название «Платформа .NET. Основы» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

6

Веб-службы

Веб-службы обеспечивают доступ к программным компонентам через стандартные интернет-протоколы, такие как HTTP и SMTP. С помощью Интернета и XML мы можем создавать программные компоненты, взаимодействующие друг с другом независимо от языка, платформы или культуры. До сегодняшнего дня разработчики программного обеспечения достигали этой цели, применяя защищенные авторскими правами технологии создания компонентного представления, например DCOM; однако поскольку каждый поставщик предоставляет свой собственный интерфейсный протокол, интеграция между компонентами различных разработчиков превращается в кошмар. За счет замены фирменных транспортных форматов на Интернет и применения стандартных протоколов, таких как SOAP, веб-службы помогают разработчикам ПО создавать программные строительные блоки, которые могут использоваться и интегрироваться независимо от своего местоположения.

В этой главе мы опишем архитектуру .NET Web Services и предоставим примеры поставщика веб-службы и нескольких его потребителей.

Веб-службы на практике

Возможно, вы уже слышали фразу «программное обеспечение как сервис» и интересовались ее значением. Термин *служба* (*service*) в обычной жизни относится к тому, что мы получаем от поставщика услуг. Например, относим свою грязную одежду в прачечную, чтобы воспользоваться услугами службы быта. С другой стороны, программное обеспечение обычно поставляется в виде приложения, которое либо покупается в магазине, либо специально разрабатывается програм-

мистской фирмой. Обычно вы покупаете программное обеспечение (в нашем случае создаете его). Оно, как правило, находится на каком-либо носителе, таком как дискета или CD, и продается в плотно завернутом пакете через торговые точки.

Каким образом программное обеспечение можно рассматривать как службу? Пример, который мы хотим привести, может показаться искусственным, однако он вполне реален для современной технологии. Представьте себе следующее. Вы все больше привязываетесь к Интернету и можете захотеть поменять свой домашний компьютер на некое устройство, специально созданное для работы с Интернетом. Назовем его iDev. С его помощью вы можете немедленно подключаться к Интернету. Если вам потребуется обработать текст, вы можете указать вашему iDev службу Microsoft Word, расположенную где-то в Редмонде, и набирать текст без необходимости установки программного обеспечения для редактирования текста. Завершив работу, вы сможете сохранить документ на сервере iStore, а впоследствии загрузить его оттуда. Заметьте, что для этого на сервере iStore должна находиться программная служба, позволяющая сохранять документы. Microsoft будет брать с вас плату в зависимости от количества времени, в течение которого был запущен текстовый процессор, и от того, какие его средства (например, проверка грамматики и правописания) вы использовали. Оплата iStore будет зависеть от размера документа и времени его хранения. Конечно, все эти счета будут приходить не по почте, а в службу условного депонирования, где эти деньги смогут сниматься с вашего банковского счета или кредитной карты и переводиться на них.

Такой тип служб избавляет от необходимости обновления копии приложения Microsoft Word. Те, кто устал от Microsoft Word, смогут выбрать другую программную службу от другой компании. Конечно, документ, который хранится на сервере iStore, будет иметь стандартный формат. Поскольку iStore использует программную службу iMaxSecure от компании под названием iNNSA (Internet Not National Security Agency, наднациональное агентство безопасности в Интернете), безопасность ваших файлов гарантируется. А так как вы прибегаете к службе хранения документов в iStore, то вы получите преимущества аутентификации и дешифрации своего документа при просмотре, а также его шифрования при сохранении.

Все это можно сделать уже сегодня с помощью Web Services.

На самом деле Microsoft развернула вариант парадигмы «программное обеспечение как служба» в виде службы аутентификации Passport. По существу, это централизованная служба аутентификации, которую вы можете включать в ваши веб-сайты. На тех сайтах, где работает служба Passport, больше не требуется помнить или отслеживать несколько пар «имя пользователя/пароль».

Недавно компания Microsoft также представила .NET My Services (проект, ранее известный под кодовым именем HailStorm), набор ориентированных на пользователя веб-служб, включающих идентификацию и аутентификацию, электронную почту, моментальный обмен сообщениями, автоматические уведомления, календарь, адресную книгу и хранение данных. Как видите, большинство этих услуг сегодня предоставляются отдельно. Идентификация и аутентификация – это задачи проекта Passport. Услуги электронной почты можгут соответствовать службе Hotmail или любому другому серверу электронной почты на базе веб-интерфейса. Моментальный обмен сообщениями должен быть знаком вам по MSN Messenger Service или AOL Instant Messenger. Календарь и адресная книга обычно объединяются со службой электронной почты. Объединение всех этих пользовательских служб и предоставление их в виде веб-службы обеспечит пользователю возможность публиковать свою информацию и управлять ею.

Клиент .NET My Services может также управлять правами доступа к данным, разрешая или запрещая доступ к информации. Эти службы также позволяют другим пользователям, организациям и «умным» устройствам взаимодействовать между собой и получать информацию о нас. Например, сколько раз вы были в дороге со своим мобильным телефоном и хотели получить свой список контактов из Outlook? Ваш мобильный телефон должен иметь возможность устанавливать связь с веб-службой вашей адресной книги для получения номера телефона какого-либо человека, не правда ли? Или, если ваша машина сломалась посреди дороги, у вас должна быть возможность найти ближайшую автомастерскую по мобильному телефону. Пользователь сам определяет, какая информация может быть опубликована и кому она может быть показана. Вы, видимо, захотите, чтобы только вы могли обращаться к своей адресной книге, в то время как веб-служба «Желтые страницы», публикующая местоположение ближайшей к вам автомастерской, будет доступна всем.

В настоящее время пользователи хранят важнейшие данные и персональную информацию во многих разных местах. В .NET My Services информация будет храниться централизованно. Например, ваш автомеханик сможет уведомить вас о времени следующего техосмотра. А при переезде и смене адреса вам не придется составлять список адресатов, которым надо отправить уведомление об изменении, – .NET My Services поможет вам опубликовать все изменения одним действием.

Потенциал пользовательских и business-to-business веб-служб, таких как .NET My Services, огромен, хотя есть серьезное и обоснованное беспокойство в отношении безопасности и конфиденциальности. Однако в той или иной форме веб-службы никуда не денутся, так что давайте углубимся в их изучение.

Структура Web Services

Web Services (веб-службы) сочетают в себе лучшее из технологий распределенных компонентов и Интернета, расширяя понятие распределенных вычислений на более широкий диапазон клиентских приложений. Лучше всего то, что это делается путем естественного соединения и улучшения существующих технологий.

Архитектура веб-служб

Веб-службы – это распределенные программные компоненты, доступные через стандартные интернет-протоколы. Первая часть определения также соответствует компонентам COM/DCOM. Однако вторая часть как раз и выделяет веб-службы среди других. Веб-службы позволяют программному обеспечению взаимодействовать с более широким диапазоном клиентов. Если COM-клиенты могут понимать только COM-компоненты, то веб-службы могут использоваться любым приложением, знающим, как разбирать XML-поток, переданный по каналам HTTP. XML – это ключевая технология, применяемая в веб-службах и в следующих компонентах структуры Microsoft .NET Web Services:

Форматы связи веб-служб

Технология, обеспечивающая универсальное понимание, как происходит обмен данными между поставщиком и потребителем услуг и формат данных для запроса и отклика.

Описание веб-службы на языке WSDL (Web Services Description Language)

Язык, описывающий, как можно использовать службу. Его можно рассматривать как инструкцию по обращению со стиральной машиной в прачечной самообслуживания, в которой говорится, куда складывать белье, какие нажимать кнопки и т. д.

Открытие веб-службы

Процесс рекламирования или публикации части программного обеспечения в виде службы и учет открытия таких служб.

На рис. 6.1 показана архитектура веб-приложений, использующих Windows DNA, а на рис. 6.2 – архитектура веб-приложений в среде .NET. Как видите, взаимодействие между компонентами веб-приложения не обязательно должно происходить в корпоративной сети. Более того, при взаимодействии между компонентами может также применяться HTTP/XML.

Форматы связи веб-служб

Возможно, вы слышали фразу «DCOM – это COM по проводам». Веб-службы схожи с DCOM, за исключением того, что для связи больше не

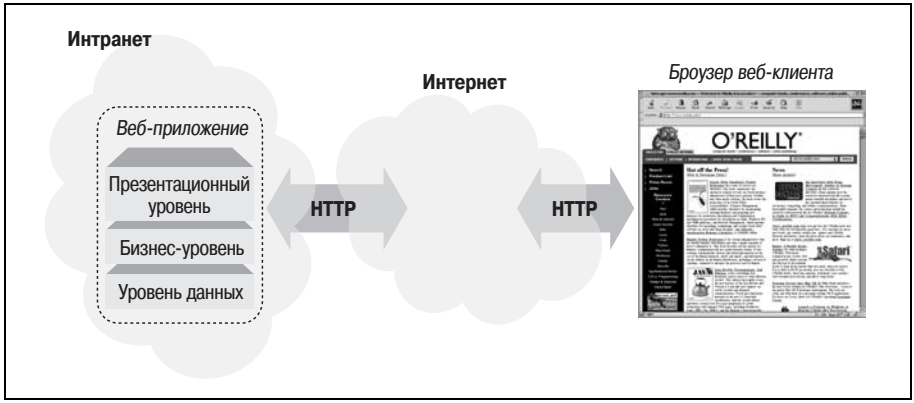


Рис. 6.1. *Архитектура Windows DNA*

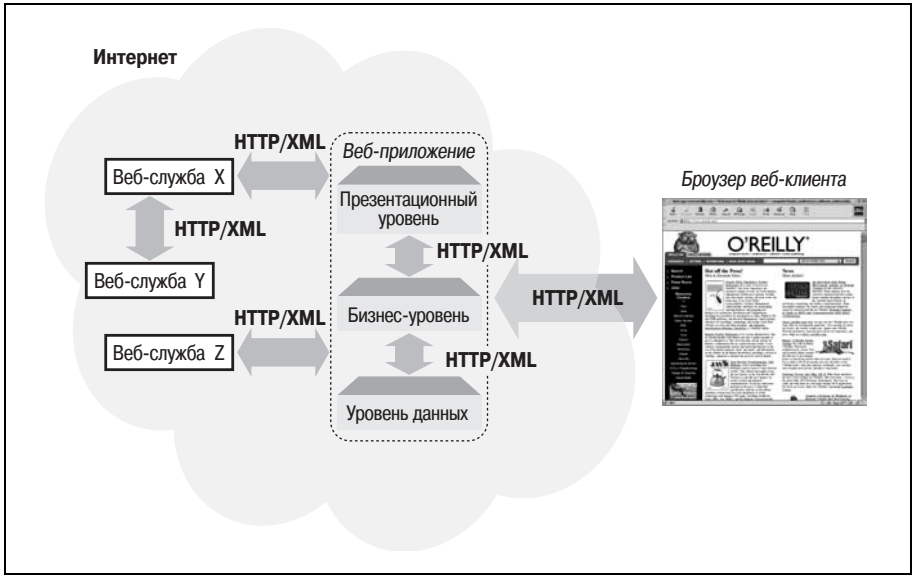


Рис. 6.2. *Среда веб-приложения .NET*

используется фирменный коммуникационный протокол. В случае применения веб-служб форматы связи основаны на более открытых интернет-протоколах, таких как HTTP или SMTP.

Веб-службу можно считать компонентом, работающим на веб-сервере и доступным миру через стандартные интернет-протоколы. Microsoft .NET Web Services на данный момент поддерживает три протокола: HTTP GET, HTTP POST и SOAP (Simple Object Access Protocol), описанные в следующих разделах. Это стандартные протоколы Интернета, и поэтому клиентским приложениям легко взаимодействовать со службами, предоставляемыми сервером.

HTTP GET и HTTP POST

Как следует из их названия, HTTP GET и HTTP POST используют в качестве своего протокола HTTP. Методы GET и POST протокола HTTP уже много лет широко применяются в ASP, CGI и других серверных архитектурах. Оба метода кодируют параметры запроса в виде пар «имя/значение» в HTTP-запросе. Метод GET создает строку запроса и добавляет ее к URL сценария, обрабатывающего запрос на сервере. В методе POST пары имя/значение передаются в теле HTTP-запроса.

SOAP

Так же как HTTP GET и HTTP POST, SOAP выступает в качестве механизма передачи сообщений между клиентами и серверами. В данном контексте клиентами являются потребители услуг, а серверами – веб-службы. Клиенты просто посылают сообщение-запрос в формате XML серверу для получения услуги. Сервер отвечает, отсылая обратно другое сообщение в формате XML. Спецификация SOAP описывает формат этих запросов и откликов. Она проста, но при этом расширяема, поскольку базируется на XML.

SOAP отличается от HTTP GET и HTTP POST, т. к. в нем полезная нагрузка имеет формат XML. Пересылаемые туда и обратно сообщения имеют лучшую структуру и могут нести более сложную информацию по сравнению с парами «имя/значение» в протоколах HTTP GET/POST. Другое отличие состоит в том, что SOAP может работать с другими транспортными протоколами, например SMTP в дополнение к HTTP.

Описание веб-службы (WSDL)

Для того чтобы клиенты веб-службы понимали, как взаимодействовать с ней, должно существовать описание вызовов методов или интерфейса, поддерживаемого веб-службой. Этот документ веб-службы задается в XML-схеме под названием *WSDL* (Web Services Description Language, язык описания веб-службы). Вспомните, что для описания COM-компонента применялись библиотеки типов и сценарии IDL. И IDL- и WSDL-файлы описывают вызовы методов интерфейса в виде списка входных и выходных параметров каждого из них. Единственное важное отличие двух этих языков состоит в том, что все описания в WSDL-файле имеют формат XML.

Теоретически любой WSDL-совместимый SOAP-клиент может получить описание вашей веб-службы из WSDL-файла. Он может использовать информацию, содержащуюся в этом файле, чтобы разобраться в интерфейсе и вызывать методы вашей веб-службы.

Структура WSDL

Корневым элементом файла описания веб-службы является элемент <definitions>. Внутри этого элемента следующие элементы содержат как абстрактное, так и конкретное описание службы:

Types

Контейнер для описаний типов.

Message

Абстрактное типизированное определение данных, которыми обмениваются поставщики и потребители веб-службы. Каждый веб-метод имеет два сообщения: входное и выходное. Входное сообщение описывает параметры веб-метода; выходное сообщение описывает возвращаемые веб-методом данные. Каждое сообщение содержит ноль или более параметров <part>. Каждый параметр связан с конкретным типом, определенным в элементе-контейнере <types>.

portType

Абстрактный набор операций, поддерживаемых одной или несколькими конечными точками.

Operation

Абстрактное описание действия, поддерживаемого службой. Для каждой операции задаются входные и выходные сообщения, определенные в виде элементов <message>.

Binding

Спецификация конкретного протокола и формата данных определенного типа порта. Аналогично элементу portType, элемент binding содержит операции, а также входные и выходные данные для каждой операции. Основное отличие состоит в том, что в элементе binding мы имеем дело с реальным типом транспорта и форматом входных и выходных данных.

Service

Коллекция сетевых конечных точек – портов. Каждый из описанных ранее форматов связи веб-служб определяет отдельный порт службы (порты HTTP GET, HTTP POST и SOAP).

Port

Единственная конечная точка, определяемая сочетанием привязки и сетевого адреса. Другими словами, здесь описан используемый протокол и спецификация формата данных, а также сетевой адрес, к которому клиенты веб-службы могут обращаться.

Ниже показана структура типичного WSDL-файла:

```
<definitions name="" targetNamespace="" xmlns:...>

  <types>...</types>
```

```
  <message name="">...</message>
  ...
  <portType name="">
    <operation name="">
      <input message="" />
      <output message="" />
    </operation>
    ...
  </portType>
  ...
  <binding name="">
    <protocol:binding ...>
    <operation name="">
      <protocol:operation ...>
      <input>...</input>
      <output>...</output>
    </operation>
    ...
  </binding>
  ...
  <service name="">
    <port name="" binding="">
      <protocol:address location="" />
    </port>
    ...
  </service>
</definitions>
```

В элементе <types> содержится описание физического типа, определенного в XML-схеме (XSD). На эти типы ссылаются элементы <message>.

Для каждого из веб-методов веб-службы для данного порта определено два сообщения: входное и выходное. Это значит, что если веб-служба поддерживает все три протокола: SOAP, HTTP GET и HTTP POST, то будет определено шесть элементов <message> – по одной паре на каждый порт. В автоматически генерируемом Microsoft .NET WSDL действует следующее соглашение об именовании:

```
ИмяМетода + Протокол + {In, Out}
```

Например, веб-метод GetBooks() имеет следующие сообщения:

```
<message name="GetBooksSoapIn">...</message>
<message name="GetBooksSoapOut">...</message>
<message name="GetBooksHttpGetIn">...</message>
<message name="GetBooksHttpGetOut">...</message>
<message name="GetBooksHttpPostIn">...</message>
<message name="GetBooksHttpPostOut">...</message>
```

Для каждого протокола, поддерживаемого веб-службой, определяется один элемент <portType>. В каждом элементе <portType> все операции

задаются в виде элементов <operation>. Для наименований типов портов принято следующее соглашение:

```
ИмяВебСлужбы + Протокол
```

Чтобы продолжить наш пример, приведем типы портов, связанных с веб-службой, которую мы создадим позднее в этой главе, – PubsWS:

```
<portType name="PubsWSSoap">
  <operation name="GetBooks">
    <input message="GetBooksSoapIn" />
    <output message="GetBooksSoapOut" />
  </operation>
</portType>

<portType name="PubsWSHttpGet">
  <operation name="GetBooks">
    <input message="GetBooksHttpGetIn" />
    <output message="GetBooksHttpGetOut" />
  </operation>
</portType>

<portType name="PubsWSHttpPost">
  <operation name="GetBooks">
    <input message="GetBooksHttpPostIn" />
    <output message="GetBooksHttpPostOut" />
  </operation>
</portType>
```

Для лучшей читаемости мы удалили из примера пространства имен.

Если типы портов – это абстрактные операции для каждого порта, то привязки предоставляют конкретную информацию о том, какой используется протокол, каким образом передаются данные и где расположена служба. Как и раньше, для каждого протокола, поддерживаемого веб-службой, имеется по одному элементу <binding>.

```
<binding name="PubsWSSoap" type="s0:PubsWSSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                style="document" />
  <operation name="GetBooks">
    <soap:operation soapAction="http://tempuri.org/GetBooks"
                    style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>

<binding name="PubsWSHttpGet" type="s0:PubsWSHttpGet">
  <http:binding verb="GET" />
```

```
    <operation name="GetBooks">
      <http:operation location="/GetBooks" />
      <input>
        <http:urlEncoded />
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
  </binding>

  <binding name="PubsWSHttpPost" type="s0:PubsWSHttpPost">
    <http:binding verb="POST" />
    <operation name="GetBooks">
      <http:operation location="/GetBooks" />
      <input>
        <mime:content type="application/x-www-form-urlencoded" />
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
  </binding>
```

Для протокола SOAP задана привязка `<soap:binding>`, а роль транспорта отведена SOAP-сообщениям, передаваемым поверх протокола HTTP. Элемент `<soap:operation>` определяет HTTP-заголовок `soapAction`, указывающий на веб-метод. Входными и выходными данными SOAP-вызова являются SOAP-сообщения.

Для протоколов HTTP GET и HTTP POST роль привязки играет `<http:binding>` и, соответственно, команда GET и POST. Так как команды GET и POST являются частью протокола HTTP, нет необходимости в задании расширенного HTTP-заголовка (такого как `soapAction` для протокола SOAP). Единственное, что нам необходимо – это URL, указывающий на веб-метод; в данном случае элемент `<http:operation>`, содержащий атрибут location, равный `/GetBooks`.

Единственное реальное различие между протоколами HTTP GET и POST заключается в способе передачи параметров веб-серверу. HTTP GET посылает параметры в строке запроса, а HTTP POST – в виде данных формы. Это отличие отражается в элементах `<input>` операции GetBooks для двух HTTP-протоколов. Для протокола HTTP GET входные данные определяются как `<http:urlEncoded />`, а для HTTP POST – как `<mime:content type="application/x-www-form-urlencoded" />`.

Вернувшись к шаблону WSDL-документа, мы увидим, что нам осталось обсудить только элемент `<service>`, определяющий порты, которые поддерживаются данной веб-службой. Для каждого поддерживаемого протокола имеется один элемент `<port>`:

```
  <service name="PubsWS">
```

```
    <port name="PubsWSSoap" binding="s0:PubsWSSoap">
      <soap:address
        location="http://.../PubsWs.asmx" />
    </port>
    <port name="PubsWSHttpGet" binding="s0:PubsWSHttpGet">
      <http:address
        location="http://.../PubsWs.asmx" />
    </port>
    <port name="PubsWSHttpPost" binding="s0:PubsWSHttpPost">
      <http:address
        location="http://.../PubsWs.asmx" />
    </port>
  </service>
```

Несмотря на то что различные порты выглядят похожими, их атрибуты `binding` связывают адрес службы с элементом `binding`, определенным ранее. Клиенты веб-службы теперь имеют достаточно информации о том, как обращаться к ней, через какой порт надо обращаться к методу и как формируются коммуникационные сообщения.

Хотя вы можете прочитать WSDL и вручную вести сеанс связи HTTP[1] с сервером для обращения к определенной веб-службе, есть утилиты, которые автоматически генерируют исходный текст клиентской программы-посредника, выполняющей то же самое. Мы покажем такую утилиту в разделе «Клиенты веб-служб» позднее в этой главе.

Открытие веб-служб

Несмотря на то что реклама веб-службы имеет большое значение, она не является обязательной. Веб-службы могут быть как закрытыми, так и открытыми. В зависимости от бизнес-модели некоторые службы business-to-business (B2B) не станут рекламироваться публично. Вместо этого владельцы веб-службы будут предоставлять определенные инструкции для доступа и использования своей службы только бизнес-партнеру.

Чтобы публично рекламировать веб-службы, авторы размещают в Интернете файлы открытий. Потенциальные клиенты веб-служб могут просматривать файлы в поисках информации о том, как использовать веб-службы. Рассматривайте эти файлы как «желтые страницы» для веб-служб. Все, что они делают – это указывают, где находятся реальные веб-службы и их описания.

Процесс поиска службы и соответствующего ей описания называется *открытием веб-службы* (*Web Service discovery*). Существует два способа рекламирования службы: статический и динамический. В обоих случаях о местоположении веб-служб сообщает XML.

---

[1] Текущая реализация Microsoft .NET SOAP работает поверх HTTP.

Статическое открытие

Статическое открытие (*static discovery*) проще в понимании, поскольку оно явное по своей природе. Рекламируя свою веб-службу, следует явно создать файл открытия *.disco* и указать в нем на WSDL.[1] Как показано в следующем примере, все disco-файлы содержат корневой элемент `discovery`. Заметьте, что `discovery` находится в пространстве имен `http://schemas.xmlsoap.org/disco`, которое в этом примере обозначено как `disco`.

```
<?xml version="1.0" ?>
<disco:discovery xmlns:disco="http://schemas.xmlsoap.org/disco/">
</disco:discovery>
```

Внутри элемента `discovery` может быть один или несколько элементов `contractRef` или `discoveryRef`. Оба этих элемента описаны в пространстве имен `http://schemas.xmlsoap.org/disco/scl/`. Тег `contractRef` применяется для ссылки на реальный URL веб-службы, который должен содержать WSDL или описание реального контракта веб-службы. С другой стороны, тег `discoveryRef` ссылается на другой disco-документ.

Этот XML-документ содержит ссылку на веб-службу и на другой disco-документ:

```
<?xml version="1.0" ?>
<disco:discovery
       xmlns:disco="http://schemas.xmlsoap.org/disco/"
       xmlns:scl="http://schemas.xmlsoap.org/disco/scl/">
<scl:contractRef ref="http://вашВебСервер/вашаВебСлужба.asmx?WSDL"/>
<scl:discoveryRef ref="http://сайтВашегоБрата/каталогЕгоВебСлужбы.disco"/>
</disco:discovery>
```

Disco-файл в этом примере определяет два различных пространства имен: `disco`, являющееся мнемоническим именем для пространства имен `http://schemas.xmlsoap.org/disco/`; и `scl` – сокращение для `http://schemas.xmlsoap.org/disco/scl/`. В элементе `contractRef` указан URL, откуда может быть получен WSDL для службы *вашаВебСлужба*. Сразу после этого имеется элемент `discoveryRef`, ссылающийся на файл открытия на сайте *сайтВашегоБрата*. Такие связи позволяют создавать структурированные сети связанных disco-документов.

Динамическое открытие

Вместо явного указания URL всех веб-служб, поддерживаемых сайтом, можно разрешить *динамическое открытие* (*dynamic discovery*), позволяющее автоматически формировать список всех веб-служб, расположенных по URL данного сайта. Для своего веб-сайта можно сгруппиро-

[1] Если для создания веб-сервиса применяется Visual Studio.NET, то disco-файл создается автоматически.

вать связанные веб-службы по разным каталогам и затем предоставить один динамический disco-файл в каждом каталоге. Корневым тегом динамического disco-файла является не *discovery*, а *dynamicDiscovery*.

```
<?xml version="1.0" encoding="utf-8"?>
<dynamicDiscovery xmlns="urn://schemas-dynamic:disco.2000-03-17" />
```

Также можно указать исключаемые пути, чтобы динамическому механизму не надо было искать веб-службы во всех подкаталогах каталога, где расположен файл динамического открытия. Исключаемые пути задаются в следующей форме:

```
<exclude path="pathname" />
```

Если в качестве веб-сервера используется IIS, то файл динамического открытия может быть похож на показанный в следующем примере:[1]

```
<?xml version="1.0" encoding="utf-8"?>
<dynamicDiscovery xmlns="urn://schemas-dynamic:disco.2000-03-17">
    <exclude path="_vti_cnf" />
    <exclude path="_vti_pvt" />
    <exclude path="_vti_log" />
    <exclude path="_vti_script" />
    <exclude path="_vti_txt" />
    <exclude path="Web References" />
</dynamicDiscovery>
```

Настройки открытия на практике

Комбинация динамического и статического открытия позволяет создавать очень гибкую конфигурацию. Например, можно предоставить статические disco-документы в каждом из каталогов, содержащих веб-службы. В корневой каталог веб-сервера поместите документ динамического открытия со ссылками на все документы статического открытия во всех подкаталогах. Чтобы исключить веб-службу из открытого просмотра, добавьте аргумент `exclude` к XML-узлам, чтобы исключить соответствующие каталоги из документа динамического открытия.

UDDI

Universal Description, Discovery, and Integration (UDDI) Business Registry (универсальный бизнес-реестр описаний, открытий и интеграции) – это что-то вроде «желтых страниц» веб-служб. Он позволяет компаниям публиковать свои службы и находить веб-службы, опубликованные партнерскими организациями, чтобы быстро, легко и динамично проводить транзакции со своим торговым партнером.

---

[1] В VS.NET для файлов динамического открытия используется расширение *vsdisco*.

Через UDDI API компании могут находить в Интернете службы, соответствующие их критериям (например, самой низкой цены), предлагающие запрашиваемые ими услуги (например, доставку по воскресеньям) и т. д. Поддерживаемый такими гигантами программной индустрии, как Microsoft, IBM и Ariba, UDDI имеет большое значение для веб-служб, т. к. обеспечивает доступ к бизнес-службам из единого места.

Пространство имен System.Web.Services

Итак, мы рассмотрели общую схему Microsoft .NET Web Services, а теперь давайте взглянем, что предоставляет нам .NET SDK в пространстве имен System.Web.Services.

В пространстве имен System.Web.Services совсем немного классов, и здесь мы рассмотрим самые важные для повседневной работы:

WebService

Базовый класс для всех веб-служб.

WebServiceAttribute

Атрибут, который может быть связан с производным от WebService классом.

WebMethodAttribute

Атрибут, который может быть связан с открытыми методами производного от WebService класса.

Два обязательных класса для создания веб-службы – это базовый класс WebService и WebMethodAttribute. Мы воспользуемся этими классами в следующем разделе, где реализуем поставщик веб-службы и несколько потребителей веб-службы. WebService – это базовый класс, от которого наследуются все веб-службы. Он предоставляет свойства, присущие традиционному ASP-программированию, такие как Application, Server, Session, и новое свойство – Context, теперь включающее в себя Request и Response.

Класс WebMethodAttribute позволяет применять атрибуты к каждому открытому методу вашей веб-службы. При помощи этого класса можно присваивать определенные значения следующим атрибутам: описание, флаг включения состояния сеанса, имя сообщения и режим транзакции. Пример установки атрибута в C# и VB приведен в следующем разделе.

Класс WebServiceAttribute применяется для предоставления дополнительных атрибутов самой веб-службы. Вы можете отобразить описание веб-службы, а также пространство имен, к которому эта веб-служба относится.

Поставщик веб-службы

В этом разделе мы расскажем, как создать веб-службу, сначала с точки зрения поставщика службы, а затем – с точки зрения потребителей. Поставщики веб-служб реализуют и рекламируют их таким образом, чтобы клиенты могли их открыть и воспользоваться услугами. Веб-службы работают поверх HTTP, поэтому на машине, содержащей веб-службы, должен работать какой-либо веб-сервер. Этим веб-сервером может быть Microsoft Internet Information Services (IIS), Apache или любая другая программа, которая поддерживает работу с протоколом HTTP. В нашем примере мы выбрали Microsoft IIS, поскольку это единственный веб-сервер, поддерживаемый средой .NET на данный момент.

Пример поставщика веб-службы

Мы создадим веб-службу под названием *PubsWS*, которая позволит потребителям получать информацию из базы данных *Pubs*. Весь доступ к данным будет осуществляться через ADO.NET, поэтому перед работой с примерами прочитайте главу 5.

Создание веб-службы включает три этапа.

1. Создайте новый файл с расширением *.asmx* для веб-службы. Он должен содержать директиву `<% webservice … %>`, а также класс, предоставляющий реализацию веб-службы. Для клиентов веб-службы этот файл будет точкой входа в вашу веб-службу. Его необходимо поместить в виртуальный каталог, для которого включены права на исполнение сценариев.
2. Создайте класс, наследуемый от класса WebService из пространства имен System.Web.Services. Это позволит производному классу обращаться ко всем обычным объектам ASP, предоставляемым базовым классом WebService. С этого момента вы можете работать с ASP-объектами так же, как делали это при разработке ASP-приложения.[1] Мы настоятельно рекомендуем вам указать пространство имен своей веб-службы, перед тем как публиковать ее, поскольку пространство имен по умолчанию, *http://tempuri.org/*, не обеспечит уникальной идентификации вашей веб-службы среди других. Чтобы это сделать, пометьте класс веб-службы атрибутом `Namespace`, определяющим ваше собственное пространство имен.
3. Пометьте открытые методы атрибутами WebMethod, чтобы сделать их *веб-методами* – открытыми методами распределенного компонента, доступными через Интернет. Не следует помечать методы как WebMethod, если не хотите, чтобы этот метод публиковался как веб-метод.

---

[1] К объектам Request и Response обращайтесь через свойство Context класса WebService.

Следующий код на C# демонстрирует простейшую веб-службу,[1] предоставляющую *Интернет*-клиентам четыре метода. Мы выделяем слово «Интернет», поскольку каждый, кто может обращаться к *asmx*-файлу на веб-сервере, имеет доступ к этим методам, в отличие от COM-компонента, к которому имеют доступ только COM-клиенты:

```
<%@ WebService Language="C#" Class="PubsWS.PubsWS" %>

namespace PubsWS
{
  using System;
  using System.Data;
  using System.Data.OleDb;
  using System.Web;
  using System.Web.Services;

  [WebService(Namespace="http://Oreilly/DotNetEssentials/")]
  public class PubsWS : WebService
  {
    private static string m_sConnStr =
"provider=sqloledb;server=(local);database=pubs; Integrated Security=SSPI";

    [WebMethod(Description="Возвращает DataSet, содержащий всех авторов.")]
    public DataSet GetAuthors()
    {
      OleDbDataAdapter oDBAdapter;
      DataSet oDS;

      oDBAdapter = new OleDbDataAdapter("select * from authors",
                                        m_sConnStr);
      oDS = new DataSet();
      oDBAdapter.Fill(oDS, "Authors");
      return oDS;
    }

    [WebMethod]
    public DataSet GetAuthor(string sSSN)
    {
      OleDbDataAdapter oDBAdapter;
```

[1] По соображениям безопасности текущая версия ASP.NET работает под учетной записью ASPNET. Если вы используете интегрированную безопасность для доступа к ресурсам базы данных, то должны предоставить учетной записи ASPNET права на доступ к данным. Также можно разрешить заимствование прав в файле *web.config* или *machine.config*:

```
<system.web>
      <identity impersonate="true" userName="" password=""/>
</system.web>
```

Если установить атрибут impersonate в `true`, но оставить пустыми `userName` и `password`, приложение будет работать от имени учетной записи *ИмяКомпьютера*\IUSR_*ИмяКомпьютера*, поэтому убедитесь, что этому пользователю (или любому другому, указанному в `userName`) предоставлен доступ к базе данных.

```
    DataSet oDS;

    oDBAdapter = new OleDbDataAdapter(
                 "select * from authors where au_id ='"
                 + sSSN + "'", m_sConnStr);
    oDS = new DataSet();
    oDBAdapter.Fill(oDS, "SelectedAuthor");
    return oDS;
  }

  [WebMethod(MessageName="GetBooksByAuthor",
             Description="Поиск книг по SSN автора.")]
  public DataSet GetBooks(string sAuthorSSN)
  {
    OleDbDataAdapter oDBAdapter;
    DataSet oDS;

    oDBAdapter = new OleDbDataAdapter(
                   "select * from titles inner join titleauthor on " +
                   "titles.title_id=titleauthor.title_id " +
                   "where au_id='" + sAuthorSSN + "'", m_sConnStr);
    oDS = new DataSet();
    oDBAdapter.Fill(oDS, "Books");
    oDBAdapter = new OleDbDataAdapter("select * from authors " +
                   "where au_id='" + sAuthorSSN + "'", m_sConnStr);
    oDBAdapter.Fill(oDS, "Author");

    return oDS;
  }

  [WebMethod]
  public DataSet GetBooks()
  {
    OleDbDataAdapter oDBAdapter;
    DataSet oDS;

    oDBAdapter = new OleDbDataAdapter("select * from titles" ,
                                      m_sConnStr);
    oDS = new DataSet();
    oDBAdapter.Fill(oDS, "Books");
    return oDS;
  }

 } // end PubsWS
}
```

Те, кто знаком с ASP, могут узнать символ @ перед ключевым словом `WebService`. Эта директива определяет язык веб-службы, чтобы ASP.NET мог компилировать его соответствующим компилятором. Эта директива также указывает класс, реализующий веб-службу, чтобы среда могла загрузить правильный класс и применить отражение для генерации WSDL для веб-службы.

Поскольку PubsWS также использует для работы с данными поставщик OLE DB ADO.NET, необходимо добавить ссылки на пространства

имен System.Data и System.Data.OleDb в дополнение к ссылкам на System, System.Web и System.Web.Services.

Класс PubsWS наследуется от WebService, при этом используется синтаксис с двоеточием, знакомый разработчикам на C++ или C#:

```
public class PubsWS : WebService
```

Четыре метода, помеченные атрибутами WebMethod, – это GetAuthors(), GetAuthor(), GetBooks(*строка*) и GetBooks(). В C# можно пометить открытые методы атрибутом WebMethod с применением скобок [ ]. В VB это должны быть символы <>. Например, в VB второй метод будет объявлен следующим образом:

```
<WebMethod()> Public Function GetAuthor(sSSN As String) As DataSet
```

Добавив атрибут [WebMethod] перед открытым методом, вы даете возможность вызывать этот метод любому интернет-клиенту. За кадром остается то, что ваш открытый метод связан с *атрибутом*, реализованным в виде класса WebMethodAttribute. Этот класс имеет шесть свойств:

BufferResponse (boolean)

Определяет наличие буферизации отклика метода.

CacheDuration

Определяет период времени в секундах, в течение которого отклик метода сохраняется в кэше; по умолчанию отклик метода в кэше не сохраняется (0 секунд).

Description

Предоставляет дополнительную информацию о конкретном веб-методе.

EnableSession (boolean)

Разрешает или запрещает сохранение состояния сеанса. Если вы не хотите использовать состояние сеанса для веб-метода, то можете выключить этот флаг, и тогда веб-серверу не придется генерировать и управлять идентификаторами сеанса для каждого пользователя, обращающегося к этому веб-методу. Это может улучшить производительность. По умолчанию этот флаг установлен в true.

MessageName

Применяется для установления различия методов с одинаковыми именами. Например, есть два различных метода, названных GetBooks (один метод извлекает все книги, а другой – только книги, написанные определенным автором), и требуется опубликовать оба этих метода как веб-методы. В этом случае в системе будут иметь место трудности с установлением различия между двумя методами, поскольку их имена дублируются. Вам придется использовать свойство MessageName, чтобы гарантировать, что все сигнатуры сервиса в

WSDL уникальны. Если применяется протокол SOAP, то свойству MessageName соответствует заголовок запроса `SOAPAction`, расположенный внутри элемента `soap:Body`. Для HTTP GET и HTTP POST этому свойству соответствует раздел PathInfo в URI (как, например, в *http://localhost//PubsWS /PubsWS.asmx/GetBooksByAuthor*).

TransactionOption

Содержит значение, соответствующее одному из пяти режимов: Disabled, NotSupported, Supported, Required и RequiresNew. Несмотря на то что имеется пять режимов, веб-методы могут участвовать в транзакции только как корневые объекты. Следовательно, значения Required и RequiresNew приводят к созданию новой транзакции для веб-метода. В случае выбора одного из значений Disabled, NotSupported и Supported транзакция для веб-метода не используется. По умолчанию свойство TransactionOption веб-метода равно Disabled.

Чтобы установить эти свойства, укажите имя свойства и его значение в виде пары *имя = значение*:

```
[WebMethod(Description="Возвращает DataSet, содержащий всех авторов.")]
 public DataSet GetAuthors()
```

Можно указать несколько свойств через запятую:

```
[WebMethod(MessageName="GetBooksByAuthor",
           Description="Поиск книг по SSN автора.")]
 public DataSet GetBooks(string sAuthorSSN)
```

Web.Config

Если веб-службы устанавливаются с нуля, потребуется также предоставить конфигурационный файл *web.config* в том же каталоге, что и ваш *asmx*-файл. Этот конфигурационный файл позволяет управлять различными настройками приложения, относящимися к виртуальному каталогу. Здесь мы установим режим аутентификации в `None`, чтобы немного облегчить разработку и тестирование нашей веб-службы. При публичном запуске нашей веб-службы следует изменить эту настройку на `Windows`, `Forms` или `Passport` вместо `None`.

```
<configuration>
  <system.web>
    <authentication mode="None" />
  </system.web>
</configuration>
```

В следующем списке перечислены различные режимы аутентификации:

Forms

При базовой аутентификации Forms запросы, не прошедшие аутентификацию, перенаправляются на форму регистрации.

Windows

Аутентификация выполняется средствами IIS одним из трех способов: базовая, краткая или встроенная аутентификация Windows.

Passport

Не прошедшие авторизацию запросы к ресурсу переправляются на централизованный сервис аутентификации Microsoft. После аутентификации возвращается маркер, который используется при последующих запросах.

Файлы открытия

После создания веб-службы вы можете предоставить вспомогательные файлы, облегчающие ее открытие. Статический disco-файл выглядит следующим образом:[1]

```
<?xml version="1.0" ?>
<disco:discovery xmlns:disco="http://schemas.xmlsoap.org/disco/"
                 xmlns:scl="http://schemas.xmlsoap.org/disco/scl/">
<scl:contractRef ref="http://localhost/PubsWS/PubsWS.asmx?WSDL"/>
</disco:discovery>
```

Клиенты веб-служб

Теперь, когда мы успешно создали веб-службу, давайте посмотрим, как эта веб-служба используется веб-клиентами. Клиенты взаимодействуют с веб-службами через стандартные веб-протоколы. Они обмениваются с веб-службой сообщениями в формате XML. Это значит, что любое приложение на любой платформе может обращаться к веб-службам, если только оно использует стандартные веб-протоколы и понимает сообщения в формате XML. Как упоминалось раньше, имеется три протокола, которые веб-клиенты могут применять для связи с серверами (веб-службами): HTTP GET, HTTP POST и SOAP. Далее мы продемонстрируем, как создавать клиентские приложения, использующие каждый из этих протоколов. Эти приложения будут написаны на VB6 и на языках среды .NET, таких как C# и VB.NET, чтобы продемонстрировать межъязыковые и межплатформенные достоинства веб-служб. Например, можно заменить пример на VB6 Perl-сценарием, работающим под Unix, и веб-службы все равно будут осуществлять обслуживание.

Клиент HTTP GET

Сначала посмотрим, как это сделать с помощью протокола HTTP GET, поскольку он самый простой. В последующих примерах *localhost* бу-

[1] В этом фрагменте кода предполагается, что виртуальный каталог, в котором вы устанавливаете веб-службу, – это */PubsWS* на вашем локальном веб-сервере.

дет выступать в качестве имени веб-сервера, на котором работает служба, и *PubsWS* – в качестве имени виртуального каталога. Если веб-служба развернута на удаленном сервере, то необходимо будет подставить соответствующие имена сервера и виртуального каталога.

Если указать броузеру URL веб-службы (*http://localhost /PubsWS/PubsWS.asmx*), то будет получен список поддерживаемых им методов. Чтобы узнать о методах больше, щелкните мышью по одному из них. Тем самым вы вызовете стандартный клиент веб-службы. Этот клиент, автоматически генерируемый с помощью отражения, прекрасно подходит для тестирования методов вашей веб-службы.[1] Он использует протокол HTTP GET для связи с веб-службой. Этот клиент представляет собой форму, позволяющую вам протестировать метод (рис. 6.3), а также описание работы с методом через SOAP, HTTP GET или HTTP POST.

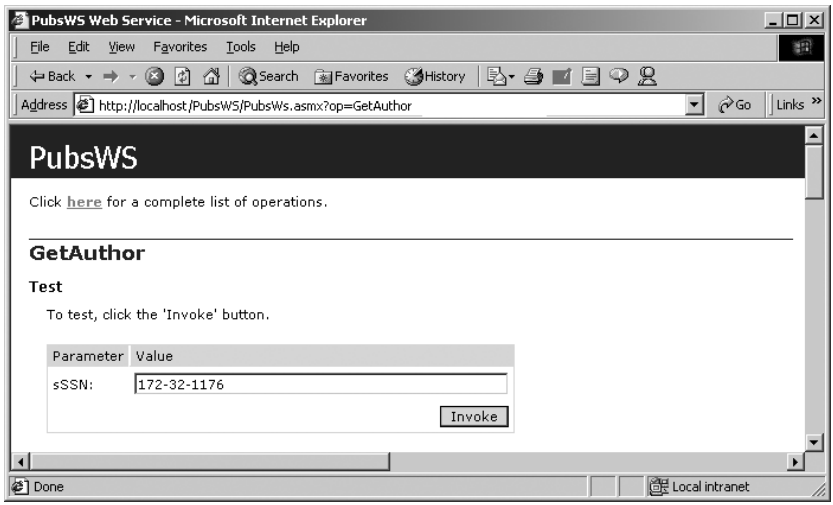


Рис. 6.3. *Автоматически сгенерированный клиент веб-службы*

Вот описание GET-запроса и отклика, предоставляемых стандартным клиентом:

```
Ниже приводится пример запроса и ответа HTTP GET. Выделенные жирным шрифтом
фрагменты должны быть заменены реальными значениями.

GET /PubsWS/PubsWS.asmx/GetAuthor?sSSN=string HTTP/1.1
Host: localhost

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
```

[1] Простой пример использования отражения (Reflection) приводился в разделе «Программирование с применением атрибутов» главы 4.

```
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<DataSet xmlns="http://Oreilly/DotNetEssentials/">
  <schema xmlns="http://www.w3.org/2001/XMLSchema">schema</schema>xml
</DataSet>
```

Для протокола HTTP GET полный URL для вызова веб-метода вместе с параметрами может быть следующим:

```
http://localhost/PubsWS/PubsWS.asmx/GetAuthor?sSSN=172-32-1176
```

Ниже приведен отклик, включающий заголовок HTTP-отклика и чистый XML (заметьте, что отклик включает схему и данные из объекта DataSet):

```
Cache-Control: private, max-age=0
Date: Tue, 08 May 2001 20:53:16 GMT
Server: Microsoft-IIS/5.0
Content-Length: 2450
Content-Type: text/xml; charset=utf-8
Client-Date: Tue, 08 May 2001 20:53:16 GMT
Client-Peer: 127.0.0.1:80

<?xml version="1.0" encoding="utf-8"?>
<DataSet xmlns="http://Oreilly/DotNetEssentials/">
  <xs:schema id="NewDataSet"
             xmlns=""
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet" msdata:IsDataSet="true">
      <xs:complexType>
        <xs:choice maxOccurs="unbounded">
          <xs:element name="SelectedAuthor">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="au_id" type="xs:string"
                            minOccurs="0" />
                <xs:element name="au_lname" type="xs:string"
                            minOccurs="0" />
                <xs:element name="au_fname" type="xs:string"
                            minOccurs="0" />
                <xs:element name="phone" type="xs:string"
                            minOccurs="0" />
                <xs:element name="address" type="xs:string"
                            minOccurs="0" />
                <xs:element name="city" type="xs:string"
                            minOccurs="0" />
                <xs:element name="state" type="xs:string"
                            minOccurs="0" />
                <xs:element name="zip" type="xs:string"
                            minOccurs="0" />
                <xs:element name="contract" type="xs:boolean"
                            minOccurs="0" />
```

```
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
<diffgr:diffgram
          xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
          xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
  <NewDataSet xmlns="">
    <SelectedAuthor diffgr:id="SelectedAuthor1" msdata:rowOrder="0">
      <au_id>172-32-1176</au_id>
      <au_lname>White</au_lname>
      <au_fname>Johnson</au_fname>
      <phone>408 496-7223</phone>
      <address>10932 Bigge Rd.</address>
      <city>Menlo Park</city>
      <state>CA</state>
      <zip>94025</zip>
      <contract>true</contract>
    </SelectedAuthor>
  </NewDataSet>
</diffgr:diffgram>
</DataSet>
```

Клиент HTTP POST

В разделе «Клиент HTTP GET» данной главы мы видели автоматическое создание клиента веб-службы при простом обращении к URL веб-службы, *http://localhost/PubsWS/PubsWS.asmx*. Теперь пора рассмотреть, как веб-клиент может использовать HTTP POST и SOAP для доступа к веб-службе. В этот раз мы собираемся написать клиентское приложение веб-службы на C#.

В Microsoft .NET SDK имеется богатый набор средств, упрощающих процесс создания и использования веб-служб. Мы собираемся применить одно из этих средств, утилиту `wsdl`, для генерации исходного кода посредников для реальных веб-служб:[1]

```
wsdl /l:CS /protocol:HttpPost http://localhost/PubsWS/PubsWS.asmx?WSDL
```

Эта командная строка создает посредник для веб-службы PubsWS на основе WSDL-документа, расположенного по URL *http://localhost/PubsWS/PubsWS.asmx?WSDL*. Посредник генерируется в виде исход-

[1] *wsdl.exe* генерирует исходный код посредника, аналогично тому, как компиляторы IDL генерируют исходные тексты посредников для DCOM. Единственное отличие состоит в том, что WSDL – это базирующийся на XML язык, описывающий интерфейс компонента программного обеспечения.

ного файла C# и использует протокол HTTP POST для взаимодействия с веб-службой. Утилита `wsdl` может также принимать в качестве входного параметра имя WSDL-файла вместо URL, указывающего на местоположение WSDL.

Исходный файл посредника на C# представляет собой класс-посредник для веб-службы PubsWS, который клиенты могут использовать при компиляции. Сгенерированный файл содержит класс-посредник PubsWS, производный от класса HttpPostClientProtocol. Если применяются параметры `/protocol:HttpGet` или `/protocol:SOAP`, PubsWS будет производным от класса HttpGetClientProtocol или SoapHttpClientProtocol.

После генерации исходного файла *PubsWS.cs* мы имеем две возможности использования этого посредника. Первый способ – включить исходный файл в проект клиентского приложения с помощью Visual Studio .NET. Если вы выбираете этот путь, проект должен быть написан на C#. Чтобы использовать посредник, необходимо также добавить к проекту все ссылки, от которых зависит посредник. В этом примере необходимыми ссылками являются System.Web.Services, System.Web.Services.Protocols, System.Xml.Serialization и System.Data.

Другой способ применения посредника является более гибким. Можно скомпилировать исходный файл C# в библиотеку DLL, а затем добавить ссылку на нее в любой создаваемый проект. Такой способ позволяет создать даже VB-проект, использующий эту DLL.

Ниже приведена командная строка компиляции посредника на C# в DLL. Обратите внимание, что три ссылки присоединяются к *PubsWS.cs*, чтобы результирующая библиотека *PubsWS.DLL* была самодостаточной (всю команду следует набирать в одну строку):

```
csc /t:library
    /r:system.web.services.dll
    /r:system.xml.dll
    /r:system.data.dll
    PubsWS.cs
```

Независимо от того, какой вариант применения посредника вы выберете, код клиентского приложения будет выглядеть одинаково. Рассмотрим следующие два примера кода на C# и VB. В обоих языках первые строки создают экземпляр посредника веб-службы PubsWS. Вторые строки вызывают веб-метод GetAuthors для получения DataSet в качестве результата. Оставшиеся строки привязывают стандартное представление таблицы `Authors` к элементу управления для табличного представления данных (data grid), добавляют его на форму и отображают форму. Заметьте, что в этих примерах используется API Windows Forms, который мы будем обсуждать в главе 8. Вот клиент веб-службы на C#, *TestProxy.cs*:

```
using System;
using System.Drawing;
```

```
using System.Windows.Forms;
using System.Data;

public class TestProxy
{

  public static void Main()
  {

    /* Создаем посредник. */
    PubsWS oProxy = new PubsWS();

    /* Вызываем GetBooks() по протоколу HTTP POST и получаем набор данных. */
    DataSet oDS = oProxy.GetAuthors();

    /* Создаем data grid и привязываем его к набору данных. */
    DataGrid dg = new DataGrid();
    dg.Size = new Size(490, 270);
    dg.DataSource = oDS.Tables["Authors"].DefaultView;

    /* Устанавливаем свойства формы и добавляем data grid. */
    Form myForm = new Form();
    myForm.Text = "DataGrid Sample";
    myForm.Size = new Size(500, 300);
    myForm.Controls.Add(dg);

    /* Отображаем форму. */
    System.Windows.Forms.Application.Run(myForm);

  }

}
```

Если вы создали DLL в соответствии с вышеприведенными указаниями, то можете скомпилировать этот код, выполнив следующую команду:

```
csc TestProxy.cs /r:PubsWS.dll
```

Так вы создадите исполняемый файл *TextProxy.exe*, получающий DataSet через вызов HTTP POST и отображающий элемент управления data grid с этим набором данных. На рис. 6.4 показан результат работы клиентского приложения на C# после получения данных от веб-службы PubsWS по протоколу HTTP POST.

Вот код клиента веб-службы, написанный на VB, *TestProxyVB.vb*:

```
imports System
imports System.Drawing
imports System.Windows.Forms
imports System.Data

Module TestProxyVB
  Sub Main()
    ' Создаем посредник.
    dim oProxy as PubsWS = new PubsWS()

    ' Вызываем GetBooks() через SOAP и получаем набор данных.
    dim oDS as DataSet = oProxy.GetAuthors()
```

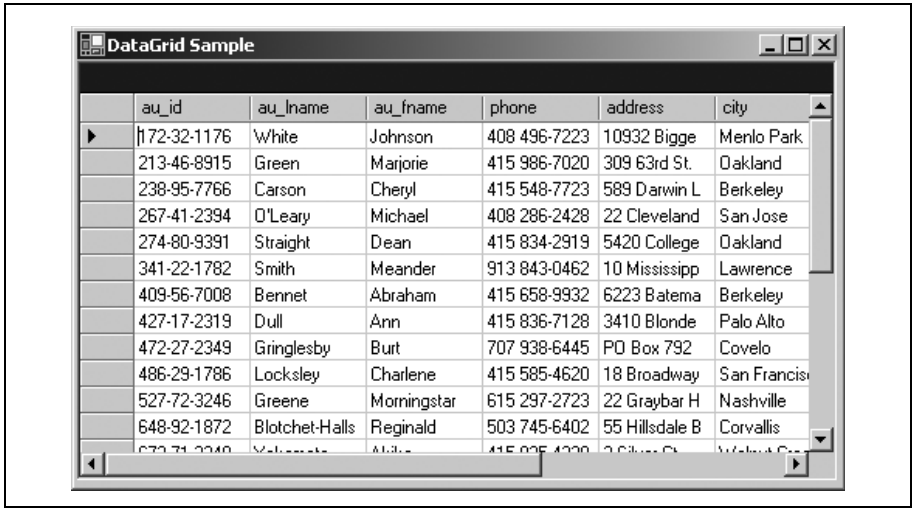


Рис. 6.4. *Клиент веб-службы на C# после вызова метода GetAuthors( )*

```
    ' Создаем data grid и привязываем его к набору данных.
    dim dg as DataGrid = new DataGrid()
    dg.Size = new Size(490, 270)
    dg.DataSource = oDS.Tables("Authors").DefaultView

    ' Устанавливаем свойства формы и добавляем data grid.
    dim myForm as Form = new Form()
    myForm.Text = "DataGrid Sample"
    myForm.Size = new Size(500, 300)
    myForm.Controls.Add(dg)

    ' Отображаем форму.
    System.Windows.Forms.Application.Run(myForm)
  End Sub
End Module
```

Клиент веб-службы на VB компилируется следующей командой (набирайте команду в одну строку):

```
vbc TestProxyVB.vb
    /r:System.Drawing.dll
    /r:System.Windows.Forms.dll
    /r:System.Data.dll
    /r:PubsWS.dll
    /r:System.Web.Services.dll
    /r:System.dll
    /r:System.xml.dll
```

Вместо применения `wsdl` для генерации посредника и включения его в свое приложение можно положиться на VS.NET в автоматизации всего процесса. В VS.NET можно просто добавить веб-ссылку (Web Reference) в приложение. Процесс добавления веб-ссылки включает от-

крытие веб-службы, получение WSDL, генерацию посредника и подключение его к приложению.[1]

Не .NET-клиенты

В этом разделе показано, как разрабатывать не .NET-клиенты веб-службы с применением протоколов HTTP GET, HTTP POST и SOAP. Так как мы не можем просто создать класс-посредник из WSDL и скомпилировать его непосредственно с клиентским кодом, мы должны посмотреть на WSDL-файл и понять, как конструировать и интерпретировать сообщения, пересылаемые между веб-службой и клиентами. Мы сократили WSDL-файл нашей веб-службы PubsWS, оставив только типы, сообщения, порты, операции и привязки, которые мы действительно будем использовать в нескольких следующих примерах клиентских приложений. В частности, наш VB6-клиент будет обращаться к следующим методам:

Веб-метод	Протокол
GetBooks()	HTTP GET
GetAuthor(ssn)	HTTP POST
GetBooksByAuthor(ssn)	SOAP

Для справки приведем упрощенный вариант WSDL-файла для экспериментов с клиентским приложением на VB6:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:...
    xmlns:s0="http://Oreilly/DotNetEssentials/"
    targetNamespace="http://Oreilly/DotNetEssentials/" >

  <types>
      <!-- Этот тип данных используется вызовом HTTP POST -->
      <s:element name="GetAuthor">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1"
                       name="sSSN" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <!-- Этот тип данных используется вызовом HTTP POST -->
      <s:element name="GetAuthorResponse">
        <s:complexType>
          <s:sequence>
```

[1] Исходный файл посредника можно найти в каталоге *Web References\ReferenceName* под именем *reference.cs* (если вы работаете с C#). Если имя ссылки не изменялось, то по умолчанию устанавливается имя *localhost*.

```
      <s:element minOccurs="1" maxOccurs="1"
                 name="GetAuthorResult"">
        <s:complexType>
          <s:sequence>
            <s:element ref="s:schema" />
            <s:any />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:sequence>
  </s:complexType>
</s:element>

<!-- Этот тип данных используется вызовом SOAP -->
<s:element name="GetBooksByAuthor">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1"
       name="sAuthorSSN" type="s:string" />
    </s:sequence>
  </s:complexType>
</s:element>
<!-- Этот тип данных используется вызовом SOAP -->
<s:element name="GetBooksByAuthorResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1"
                 name="GetBooksByAuthorResult"">
        <s:complexType>
          <s:sequence>
            <s:element ref="s:schema" />
            <s:any />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:sequence>
  </s:complexType>
</s:element>

<!-- Этот тип данных используется вызовом HTTP GET -->
<s:element name="GetBooks">
  <s:complexType />
</s:element>
<!-- Этот тип данных используется вызовом HTTP GET -->
<s:element name="GetBooksResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1"
                 name="GetBooksResult">
        <s:complexType>
          <s:sequence>
            <s:element ref="s:schema" />
```

```
                <s:any />
              </s:sequence>
            </s:complexType>
          </s:element>
        </s:sequence>
      </s:complexType>
    </s:element>

    <!-- Этот тип данных используется откликами HTTP GET/POST -->
    <s:element name="DataSet"
      <s:complexType>
        <s:sequence>
          <s:element ref="s:schema" />
          <s:any />
        </s:sequence>
      </s:complexType>
    </s:element>

</types>

<!-- Эти сообщения используются вызовом SOAP -->
<message name="GetBooksByAuthorSoapIn">
  <part name="parameters" element="s0:GetBooksByAuthor" />
</message>
<message name="GetBooksByAuthorSoapOut">
  <part name="parameters" element="s0:GetBooksByAuthorResponse" />
</message>

<!-- Эти сообщения используются вызовом HTTP GET -->
<message name="GetBooksHttpGetIn" />
<message name="GetBooksHttpGetOut">
  <part name="Body" element="s0:DataSet" />
</message>

<!-- Эти сообщения используются вызовом HTTP POST -->
<message name="GetAuthorHttpPostIn">
  <part name="sSSN" type="s:string" />
</message>
<message name="GetAuthorHttpPostOut">
  <part name="Body" element="s0:DataSet" />
</message>

<!-- Порт SOAP -->
<portType name="PubsWSSoap">
  <operation name="GetBooks">
    <documentation>Поиск книг по SSN автора.</documentation>
    <input name="GetBooksByAuthor"
           message="s0:GetBooksByAuthorSoapIn" />
    <output name="GetBooksByAuthor"
            message="s0:GetBooksByAuthorSoapOut" />
  </operation>
</portType>

<!-- Порт HTTP GET -->
<portType name="PubsWSHttpGet">
```

```
  <operation name="GetBooks">
    <input message="s0:GetBooksHttpGetIn" />
    <output message="s0:GetBooksHttpGetOut" />
  </operation>
</portType>

<!-- Порт HTTP POST -->
<portType name="PubsWSHttpPost">
  <operation name="GetAuthor">
    <input message="s0:GetAuthorHttpPostIn" />
    <output message="s0:GetAuthorHttpPostOut" />
  </operation>
</portType>

<!-- Привязка SOAP -->
<binding name="PubsWSSoap" type="s0:PubsWSSoap">
  <soap:binding
        transport="http://schemas.xmlsoap.org/soap/http"
        style="document" />
  <operation name="GetBooks">
    <soap:operation
          soapAction="http://Oreilly/DotNetEssentials/GetBooksByAuthor"
          style="document" />
    <input name="GetBooksByAuthor">
      <soap:body use="literal" />
    </input>
    <output name="GetBooksByAuthor">
      <soap:body use="literal" />
    </output>
  </operation>
</binding>

<!-- Привязка HTTP GET -->
<binding name="PubsWSHttpGet" type="s0:PubsWSHttpGet">
  <http:binding verb="GET" />
  <operation name="GetBooks">
    <http:operation location="/GetBooks" />
    <input>
      <http:urlEncoded />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>

<!-- Привязка HTTP POST -->
<binding name="PubsWSHttpPost" type="s0:PubsWSHttpPost">
  <http:binding verb="POST" />
  <operation name="GetAuthor">
    <http:operation location="/GetAuthor" />
    <input>
      <mime:content type="application/x-www-form-urlencoded" />
```

```
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
  </binding>

  <!-- Весь веб-сервис и адресные привязки -->
  <service name="PubsWS">

    <port name="PubsWSSoap" binding="s0:PubsWSSoap">
      <soap:address location="http://localhost/PubsWS/PubsWS.asmx" />
    </port>

    <port name="PubsWSHttpGet" binding="s0:PubsWSHttpGet">
      <http:address location="http://localhost/PubsWS/PubsWS.asmx" />
    </port>

    <port name="PubsWSHttpPost" binding="s0:PubsWSHttpPost">
      <http:address location="http://localhost/PubsWS/PubsWS.asmx" />
    </port>

  </service>

</definitions>
```

В протоколах HTTP GET и HTTP POST параметры передаются веб-службе в виде пар «имя/значение». В протоколе HTTP GET необходимо передать параметры в строке запроса, а в протоколе HTTP POST они упаковываются в тело пакета запроса. Чтобы продемонстрировать это, мы создадим простой VB-клиент, использующий оба протокола – HTTP GET и HTTP POST – для взаимодействия с веб-службой PubsWS.

Создадим сначала стандартное приложение VB6. Нам требуется добавить ссылку на Microsoft XML, версия 3.0 (*msxml3.dll*), поскольку мы будем использовать объект XMLHTTP, который поможет нам в коммуникации с веб-службами. Для демонстрационных целей мы также используем компонент Microsoft Internet Controls (*shdocvw.dll*) для отображения XML и HTML.

Сначала добавим две кнопки к стандартной форме, `form1`, и зададим заголовки `GET` и `POST`, а также имена `cmdGet` и `cmdPost` соответственно. После этого перенесем объект WebBrowser с панели инструментов на форму и назовем его `myWebBrowser`. Если указать в качестве стартовой страницы WebBrowser адрес `about:blank`, то получится что-то вроде изображенного на рис. 6.5.

Теперь нам только осталось написать некоторый код для обработки событий нажатия двух кнопок, похожий на приведенный ниже:

```
Private Sub cmdGet_Click()
  Dim oXMLHTTP As XMLHTTP
  Dim oDOM As DOMDocument
  Dim oXSL As DOMDocument
```

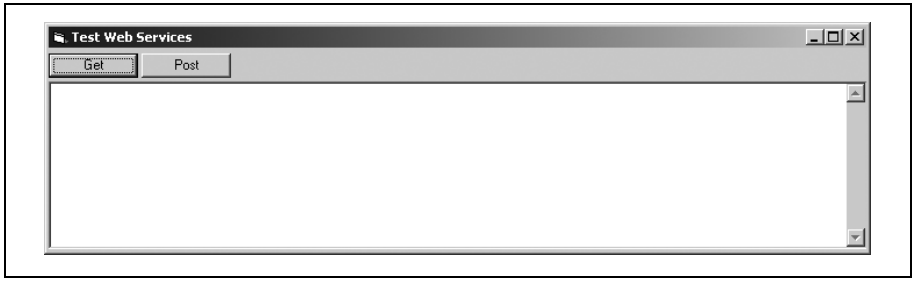


***Рис. 6.5.** Форма VB-клиента для тестирования веб-служб*

```
    ' Вызов веб-сервиса для получения XML-документа
    Set oXMLHTTP = New XMLHTTP
    oXMLHTTP.open "GET",_
                  "http://localhost/PubsWS/PubsWS.asmx/GetBooks", _
                  False
    oXMLHTTP.send
    Set oDOM = oXMLHTTP.responseXML

    ' Создаем XSL-документ для использования при трансформации
    Set oXSL = New DOMDocument
    oXSL.Load App.Path & "\templateTitle.xsl"

    ' Преобразуем XML-документ в HTML-документ и показываем результат
    myWebBrowser.Document.Write CStr(oDOM.transformNode(oXSL))
    myWebBrowser.Document.Close

    Set oXSL = Nothing
    Set oDOM = Nothing
    Set oXMLHTTP = Nothing
End Sub

Private Sub cmdPost_Click()
    Dim oXMLHTTP As XMLHTTP
    Dim oDOM As DOMDocument
    Dim oXSL As DOMDocument

    ' Вызываем веб-службу для получения XML-документа
    Set oXMLHTTP = New XMLHTTP
    oXMLHTTP.open "POST", _
                  "http://localhost/PubsWS/PubsWS.asmx/GetAuthor", _
                  False
    oXMLHTTP.setRequestHeader "Content-Type", _
                              "application/x-www-form-urlencoded"
    oXMLHTTP.send "sSSN=172-32-1176"
    Set oDOM = oXMLHTTP.responseXML

    ' Создаем XSL-документ для использования в трансформации
    Set oXSL = New DOMDocument
    oXSL.Load App.Path & "\templateAuthor.xsl"

    ' Преобразуем XML-документ в HTML-документ и показываем результат
    myWebBrowser.Document.Write oDOM.transformNode(oXSL)
```

```
    myWebBrowser.Document.Close

    Set oXSL = Nothing
    Set oDOM = Nothing
    Set oXMLHTTP = Nothing
End Sub
```

Две подпрограммы имеют аналогичную структуру, за исключением того, что в первой для обращения к веб-службе PubsWS применяется протокол HTTP GET, а во второй – протокол HTTP POST. Давайте внимательнее присмотримся к тому, что делают две подпрограммы.

Для протокола HTTP GET мы используем объект XMLHTTP для задания URL веб-метода, как указано в WSDL. Поскольку веб-метод GetBooks не требует никаких параметров, в этом случае строка запроса пуста. Метод вызывается синхронно, т. к. параметр async метода `open` объекта XMLHTTP равен `false`. После осуществления вызова метода мы преобразуем XML-результат с помощью файла *templateTitle.xsl* и показываем HTML в экземпляре `myWebBrowser` на форме. На рис. 6.6 показан экран нашего тестового приложения после вызова веб-метода GetBooks по URL *http://localhost/PubsWS/PubsWS.asmx/* через протокол HTTP GET.

В протоколе HTTP POST мы также используем XMLHTTP для задания URL веб-метода – в этом случае метода GetAuthor. Так как это POST-запрос, мы должны указать в HTTP-заголовке, что запрос передается в виде формы, установив переменную `Content-Type` в `application/x-www-from-urlencoded`. Если эта переменная не установлена, XMLHTTP по умолчанию передает данные серверу в формате XML.

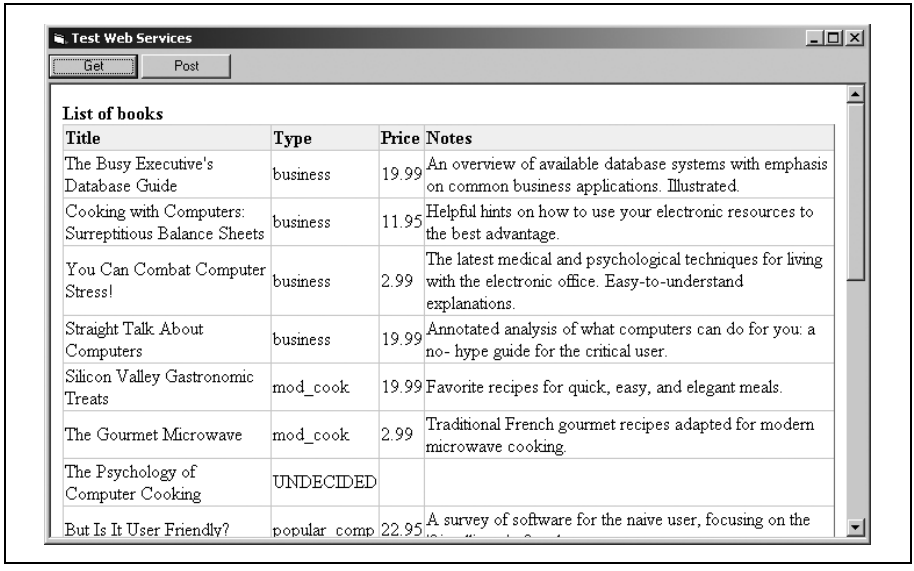


Рис. 6.6. *Форма VB-клиента после вызова метода GetBooks*

Другое отличие, которое стоит отметить, заключается в том, что метод GetAuthor требует одного параметра – номера SSN автора в виде строки. Поскольку это POST-запрос, мы будем отправлять пару «имя/значение» на сервер непосредственно в теле сообщения. А так как в заголовке значение `Content-Type` установлено в `application/x-www-form-urlencoded`, сервер знает, как получить параметры и выполнить запрошенное действие. На этот раз мы используем файл *templateAuthor.xsl* для преобразования XML-результата в HTML и его отображения. На рис. 6.7 показано наше приложение после вызова веб-метода GetAuthor веб-службы PubsWS по протоколу HTTP POST.

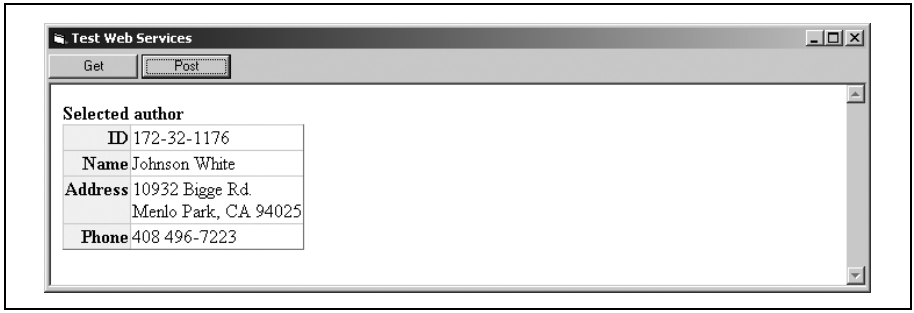


***Рис. 6.7.** Форма VB-клиента после вызова метода GetAuthor*

Ниже приведен файл XSL, используемый для преобразования XML-результата вызова веб-метода GetBooks в HTML для отображения в экземпляре веб-броузера на VB-форме:

```
<html version="1.0" xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<head><title>A list of books</title></head>
<style>
.hdr { background-color=#ffeedd; font-weight=bold; }
</style>
<body>
<B>List of books</B>
<table style="border-collapse:collapse" border="1">
<tr>
  <td class="hdr">Title</td>
  <td class="hdr">Type</td>
  <td class="hdr">Price</td>
  <td class="hdr">Notes</td>
</tr>
<xsl:for-each select="//Books">
<tr>
  <td><xsl:value-of select="title"/></td>
  <td><xsl:value-of select="type"/></td>
  <td><xsl:value-of select="price"/></td>
  <td><xsl:value-of select="notes"/></td>
</tr>
</xsl:for-each>
```

```
</table>
</body>
</html>
```

Далее показан файл XSL, используемый для преобразования XML-результата вызова веб-метода GetAuthor в HTML для отображения в экземпляре веб-броузера на VB-форме:

```
<html version="1.0" xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<head><title>Selected author</title></head>
<STYLE>
.hdr { background-color:'#ffeedd';
       text-align:'right'; vertical-align:'top';
       font-weight=bold; }
</STYLE>
<body>
<B>Selected author</B>
<xsl:for-each select="//SelectedAuthor">
<table style="border-collapse:'collapse'" border="1">
<tr><td class="hdr">ID</td>
    <td><xsl:value-of select="au_id"/></td></tr>
<tr><td class="hdr">Name</td>
    <td><xsl:value-of select="au_fname"/>
        <xsl:value-of select="au_lname"/></td></tr>
<tr><td class="hdr">Address</td>
    <td><xsl:value-of select="address"/><br>
        <xsl:value-of select="city"/>,
        <xsl:value-of select="state"/>
        <xsl:value-of select="zip"/></br></td></tr>
<tr><td class="hdr">Phone</td>
    <td><xsl:value-of select="phone"/></td></tr>
</table>
</xsl:for-each>
</body>
</html>
```

Мы также можем выбрать для доступа к веб-службе протокол SOAP. Веб-служба доступна через HTTP и XML, поэтому любой клиент на любой платформе может обращаться к ней, если только он придерживается спецификации службы. Такой спецификацией является WSDL-файл. Изучив WSDL-файл, – в частности его SOAP-секцию, – мы снова можем использовать XMLHTTP для осуществления SOAP-диалога. Посмотрим, как это можно сделать.

Вернемся к примеру клиента веб-службы, использующему VB6 и XMLHTTP. Добавим еще одну кнопку на форму и назовем ее `cmdSOAP` с заголовком `SOAP`. На этот раз мы попросим веб-службу выдать все книги, написанные определенным автором:

```
Private Sub cmdSOAP_Click()
  Dim oXMLHTTP As XMLHTTP
  Dim oDOM As DOMDocument
```

```
    Dim oXSL As DOMDocument

    ' Вызываем веб-сервис для получения XML-документа
    Set oXMLHTTP = New XMLHTTP
    oXMLHTTP.open "POST", "http://localhost/PubsWS/PubsWS.asmx", False

    Dim sBody As String

    sBody = "" & _
    "<soap:Envelope" & _
    " xmlns:xsi=""http://www.w3.org/2001/XMLSchema-instance""" & _
    " xmlns:xsd=""http://www.w3.org/2001/XMLSchema""" & _
    " xmlns:soap=""http://schemas.xmlsoap.org/soap/envelope/"">" & _
    "<soap:Body>" & _
    "<GetBooksByAuthor xmlns=""http://Oreilly/DotNetEssentials/"">" & _
    "<sAuthorSSN>213-46-8915</sAuthorSSN>" & _
    "</GetBooksByAuthor>" & _
    "</soap:Body>" & _
    "</soap:Envelope>"

    oXMLHTTP.setRequestHeader "Content-Type", "text/xml"
    oXMLHTTP.setRequestHeader "SOAPAction",
                          "http://Oreilly/DotNetEssentials/GetBooksByAuthor"

    oXMLHTTP.send sBody

    Set oDOM = oXMLHTTP.responseXML

    ' Создаем XSL-документ для использования в преобразованиях
    Set oXSL = New DOMDocument
    oXSL.Load App.Path & "\templateAuthorTitle.xsl"

    ' Преобразуем XML-документ в HTML-документ
    myWebBrowser.Document.Write oDOM.transformNode(oXSL)
    myWebBrowser.Document.Close

    Set oXSL = Nothing
    Set oDOM = Nothing
    Set oXMLHTTP = Nothing
End Sub
```

Структура этого метода аналогична использованной для HTTP GET и HTTP POST; однако он имеет несколько важных отличий. В SOAP необходимо установить параметр `Content-Type` равным `text/xml` вместо `application/x-www-form-urlencoded`, как для HTTP POST. Теперь вам должно быть ясно, что только для HTTP POST и SOAP параметр `Content-Type` имеет значение, т. к. в этих протоколах данные пересылаются в теле HTTP-запроса. Для протокола HTTP GET `Content-Type` фактически не имеет значения, поскольку все параметры упаковываются в строку запроса. Кроме отличия в формате данных, надо также обратиться к WSDL для установки переменной заголовка `SOAPAction` в соответствии с вашим вызовом. Снова обратившись к разделу SOAP в WSDL, вы поймете, что для вызова метода `GetBooks(sAuthorSSN)` веб-службы переменная `SOAPAction` должна быть равна `http://Oreilly/DotNetEssenti-`

als/GetBooksByAuthor. С другой стороны, если вы захотите вызвать GetBooks, переменную SOAPAction надо будет установить в http://Oreilly/DotNetEssentials/GetBooks. Пространство имен http://Oreilly/DotNetEssentials/ используется потому, что мы установили его в качестве атрибута класса веб-службы PubsWS.

После установки значений переменных заголовка передайте параметры серверу в теле сообщения. Если HTTP POST передает параметры в парах «имя/значение», то SOAP передает параметры в строго определенной XML-структуре:

```
<soap:Envelope ...пространство имен опущено...">
  <soap:Body>
    <GetBooksByAuthor xmlns="http://Oreilly/DotNetEssentials/">
      <sAuthorSSN>213-46-8915</sAuthorSSN>
    </GetBooksByAuthor>
  </soap:Body>
</soap:Envelope>
```

SOAP-сообщения запроса и отклика упаковываются в элемент Body, находящийся внутри элемента Envelope. Для приведенного выше запроса сообщение-отклик выглядит следующим образом:

```
<?xml version="1.0"?>
<soap:Envelope ...пространство имен опущено...>
  <soap:Body>
    <GetBooksByAuthorResult xmlns="http://Oreilly/DotNetEssentials/">
      <result>
        <xsd:schema id="NewDataSet" ...>

          <... часть содержимого опущена ...>

        </xsd:schema>
        <NewDataSet xmlns="">
          <Books>
            <title_id>BU1032</title_id>
            <title>The Busy Executive's Database Guide</title>
          <... еще ...>
          </Books>
          <Books>
            <title_id>BU2075</title_id>
            <title>You Can Combat Computer Stress!</title>
            <... еще...>
          </Books>
          <Author>
            <au_id>213-46-8915</au_id>
            <au_lname>Green</au_lname>
            <au_fname>Marjorie</au_fname>
            <phone>415 986-7020</phone>
            <address>309 63rd St. #411</address>
            <city>Oakland</city>
            <state>CA</state>
            <zip>94618</zip>
```

```
            <contract>True</contract>
          </Author>
        </NewDataSet>
      </result>
    </GetBooksByAuthorResult>
  </soap:Body>
</soap:Envelope>
```

На рис. 6.8 показана тестовая форма после вызова веб-метода GetBooksByAuthor с помощью протокола SOAP.

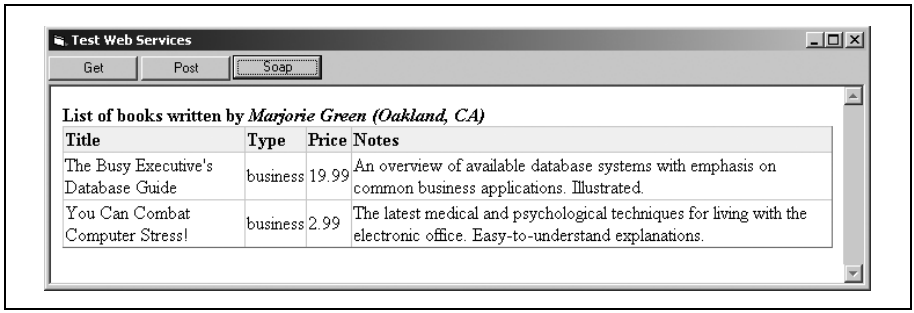


***Рис. 6.8.** Форма VB-клиента после вызова метода GetBooksByAuthor*

Для справки здесь приведена таблица стилей XSL, использованная для преобразования XML в HTML. Заметьте, что поскольку метод GetBooksByAuthor возвращает в наборе данных две таблицы, author и books, мы можем показать как информацию об авторе, так и книги, написанные этим автором.

```
<html version="1.0" xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<head><title>A list of books</title></head>
<style>
.hdr { background-color=#ffeedd; font-weight=bold; }
</style>
<body>
<B>List of books written by
  <I><xsl:value-of select="//Author/au_fname"/>
     <xsl:value-of select="//Author/au_lname"/>
     (<xsl:value-of select="//Author/city"/>,
     <xsl:value-of select="//Author/state"/>)
  </I>
</B>
<table style="border-collapse:collapse" border="1">
<tr>
  <td class="hdr">Title</td>
  <td class="hdr">Type</td>
  <td class="hdr">Price</td>
  <td class="hdr">Notes</td>
</tr>
<xsl:for-each select="//Books">
```

```
<tr>
  <td><xsl:value-of select="title"/></td>
  <td><xsl:value-of select="type"/></td>
  <td><xsl:value-of select="price"/></td>
  <td><xsl:value-of select="notes"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
```

Как видите, клиенты, обращающиеся к веб-службам .NET, могут быть любого типа. Клиенты веб-служб для взаимодействия с сервером должны лишь знать, как осуществлять связь по протоколу HTTP, и понимать язык WSDL. Аналогично, веб-службы могут быть разработаны на любом языке и на любой платформе, если только они соответствуют спецификации WSDL.

Веб–службы и безопасность

В этом разделе мы продемонстрируем, как внедрить в веб-службу поддержку безопасности. Мы сделаем это двумя способами: на уровне системы и на уровне приложения. Безопасность на уровне системы позволяет ограничивать доступ неавторизованных клиентов к веб-службам. Это делается в декларативной манере, тогда как безопасность на уровне приложения – это более гибкий подход. Системная безопасность чаще всего реализуется путем задания IP-адресов авторизованных клиентов, которым с помощью каких-либо утилит настройки вы разрешите доступ к вашей веб-службе. Реализуя безопасность на уровне приложения, вы внедряете аутентификацию в вашу веб-службу, тем самым обеспечивая более гибкую конфигурацию.

Безопасность на уровне системы

Взаимодействие веб-служб осуществляется через HTTP, поэтому можно применить к веб-службам безопасность на уровне системы так же, как это делается для остальных веб-страниц и ресурсов вашего веб-сайта.

Существует несколько способов защиты веб-служб. Для B2B-решений может применяться утилита администрирования IIS, позволяющая запрещать или предоставлять доступ набору IP-адресов с помощью Internet Protocol Security (IPSec), чтобы удостовериться, что IP-адрес в заголовке TCP/IP аутентифицирован. Если вы полагаетесь только на то, что клиент предоставит свой IP-адрес в заголовке TCP/IP, хакеры смогут позаимствовать IP-адреса других хостов для доступа к вашим веб-службам. IPSec подтверждает адреса хостов с помощью протокола аутентификации Kerberos. Для ограничения доступа к веб-службам только партнерскими компаниями могут применяться брандмауэры.

Для сценариев B2C (business-to-consumer, бизнес-клиент) можно обратиться к средствам аутентификации протокола HTTP.

Чтобы продемонстрировать применение средств аутентификации протокола HTTP для защиты веб-служб, вернемся к нашему примеру веб-службы PubsWS, рассматриваемому в этой главе. Все, что мы должны сделать для защиты веб-службы PubsWS – это запустить утилиту администрирования IIS и отредактировать свойства `File Security` для файла *PubsWS.asmx*. Измените стандартные настройки, предусматривающие доступ к этому файлу для всех анонимных пользователей, на поддержку только базовой аутентификации, т. е. сбросьте флажок Anonymous Access (Анонимный доступ) и установите единственный флажок Basic Authentication (Базовая аутентификация) в группе Authenticated Access. После этого изменения только пользователи, прошедшие аутентификацию, смогут работать с веб-службой.

Конечно, в реальной жизни мы не станем применять метод базовой аутентификации, т. к. он посылает имя пользователя и пароль в виде простого текста по каналу HTTP. Мы выберем другие методы, такие как Secure Sockets Layer (SSL), в сочетании с базовой аутентификацией, чтобы защитить данные, передаваемые в обоих направлениях. Доступные методы включают:

Basic Authentication (базовая аутентификация)

Посылает веб-серверу имя пользователя и пароль в текстовом виде. IIS подтверждает регистрационное имя в соответствии с базой данных пользователей домена.

Basic over SSL Authentication (базовая аутентификация поверх SSL)

Аналогична базовой аутентификации, за исключением того, что имя и пароль пересылаются с SSL-шифрованием.

Digest Authentication (краткая аутентификация)

Для безопасной отправки реквизитов клиента на сервер применяет хеш-технологию вместо SSL-шифрования.

Integrated Windows Authentication (интегрированная аутентификация Windows)

Подходит только для интранет-сценариев. Использует для аутентификации регистрационную информацию клиента.

Client Certificates Authentication (аутентификация с помощью клиентских сертификатов)

Требует от каждого клиента наличия сертификата, соответствующего учетной записи пользователя. Применение клиентских цифровых сертификатов в данный момент не очень распространено.

Безопасность на уровне приложений

Менее систематический способ обеспечения безопасности веб-служб требует взятия этого вопроса в собственные руки. Можно запрограм-

мировать свои веб-службы таким образом, чтобы все их методы требовали маркер доступа, который можно получить от веб-службы после отправки клиентом своего имени пользователя и пароля. Реквизиты клиента могут отправляться на сервер через SSL, исключая риск отправки текстовых паролей по сети. По каналу SSL сервер возвращает маркер доступа отправителю, который может использовать его для вызова всех остальных методов веб-службы. Конечно, все остальные публикуемые веб-методы должны иметь один параметр для маркера доступа. Простой пример псевдокода веб-службы, обслуживающей банковский счет, может выглядеть следующим образом:

```
Веб-служба Банковский счет
  Веб-методы:
    Login(user id, password) возвращает маркер доступа или ничего
    Deposit(маркер доступа, номер счета, объем, баланс) возвращает boolean
    Withdraw(маркер доступа, номер счета, объем, баланс) возвращает boolean
```

Единственный метод, который должен работать по SSL, – это метод Login. После получения маркера доступа его можно применять для других веб-методов. Конечно, необходимо убедиться, что последующие вызовы, использующие этот маркер, поступают с того же IP-адреса, что и вызов Login(). Вы также можете установить временные рамки действия этого маркера доступа, обеспечив его существование в течение лишь определенного периода, после чего потребуется обновление маркера доступа.

Выбравшим этот путь могут быть очень полезны службы Microsoft .NET Cryptographic Services. Они поддерживают алгоритмы шифрации/дешифрации DES, RC2, TripleDES и RSA, а также методы хеширования, такие как SHA и MD5. Эти реализации, присутствующие в библиотеке .NET, позволяют разработчикам избежать низкоуровневого программирования и сосредоточиться на логике приложения.

Заключение

В этой главе мы представили новую парадигму приложений – корпоративные приложения. Разработчики больше не ограничены однородными платформами в реализации своих решений. С применением Microsoft Web Services эти решения могут распределяться между различными платформами, поскольку связь между веб-службами осуществляется через стандартные протоколы Интернета, такие как HTTP и XML. Известные вам ранее распределенные компоненты Windows DNA теперь уступают место веб-службам. Применение веб-служб в качестве компонентов распределенной среды позволяет создавать неоднородные системы. Веб-службы в вашей системе могут не только разрабатываться на разных языках, но и работать на различных платформах. По причине лучшей возможности взаимодействия веб-службы особенно подходят для B2B (бизнес-бизнес)-интеграции.

7

Веб-формы

В данной главе представлена очередная технология создания динамических и интерактивных страниц. ASP.NET продвигает ASP на шаг вперед к упрощению процесса разработки веб-страниц и повышению масштабируемости веб-приложений. Сначала мы рассмотрим обычный способ разработки веб-приложений с помощью ASP, чтобы выявить некоторые ловушки, преодолеваемые в ASP.NET. Затем рассмотрим достоинства ASP.NET и предоставим общий обзор классов ASP.NET, таких как Control и Page, а также полный синтаксис ASP.NET Web Forms. В завершение главы мы обсудим многие аспекты разработки в ASP.NET: как ASP.NET поддерживает разработку веб-служб, как использовать нестандартные серверные элементы управления и как было улучшено управление сеансами для увеличения масштабируемости веб-решений. Эта обзорная глава предназначена для того, чтобы предоставить разработчикам прочный фундамент вводных знаний по ASP.NET и веб-формам; дополнительную подробную информацию можно найти в книгах *Programming ASP.NET (Программирование для ASP.Net)*, авторы Джесси Либерти и Дэн Гурвиц (Jesse Liberty, Dan Hurwitz) и *ASP.NET in a Nutshell (ASP.Net. Справочник)*, авторы Эндрю Дьюти и Мэттью Мак-Дональд (Andrew Duthie, Matthew MacDonald) (обе книги выпущены издательством O'Reilly).

ASP

Microsoft Active Server Pages (ASP) – это технология сценариев на стороне сервера, позволяющая создавать динамические веб-страницы. ASP-страница содержит HTML-разметку и серверные сценарии, динамически генерирующие HTML-содержимое. Серверные сценарии за-

пускаются, когда на веб-сервер поступает запрос на получение ASP-страницы. Входные данные для ASP-страницы поступают от клиентских броузеров с помощью методов HTTP POST и GET. ASP предоставляет объектную модель для упрощения задач разработчика. Кроме объектов из объектной модели ASP, таких как Application, Server, Request, Response и Session, разработчики могут использовать любые COM-компоненты, расположенные на сервере.

Те, кто уже разрабатывал веб-приложения с помощью ASP, скорее всего, согласятся с тем, что в результате разработки легко получались запутанные, возможно, конфликтующие HTML-разметка и серверные сценарии. Плохая модель инкапсуляции ASP-страниц усложняет их поддержку и повторное использование. Были сделаны попытки улучшить эту модель за счет применения включаемых на стороне сервера файлов и параметризованных функций в сценариях; однако эти попытки требовали таких уступок, как время разработки, ухудшение производительности при использовании вложенного включения файлов, а также управление идентификаторами объектов и областями действия переменных.

Разработчики, имевшие дело с веб-приложениями для разных броузеров, также сталкивались с проблемами генерации HTML, соответствующего возможностям броузера клиента. В большинстве случаев приходилось ограничиваться простейшими тегами HTML и клиентскими сценариями, понятными большинству броузеров, и отказываться от возможностей более развитых броузеров. Полученное веб-приложение могло быть хорошим лишь настолько, насколько позволял наихудший поддерживаемый им броузер. Иногда мы также пытались генерировать различную HTML-разметку для различных броузеров, чтобы использовать возможности определенных броузеров, добиваясь значительно лучшего впечатления от клиентской части; однако это требовало значительно большего времени и усилий на разработку.

Скрипты в ASP доступны только на языках позднего связывания, таких как VBScript и JavaScript, поэтому безопасность типов в них недоступна. Кроме того, серверные сценарии ASP-страниц интерпретируются при каждом обращении к странице, что не лучшим образом сказывается на производительности.

Поддержка состояния формы в ASP-приложении также требует больших усилий – разработчикам приходится все делать вручную, включая повторный ввод данных, использование скрытых полей и переменных сеанса. Время от времени возникает необходимость в настройке веб-приложений для работы в среде веб-ферм (web farms), когда клиенту доступен более чем один сервер. Поддержка состояния сеанса в таких ситуациях значительно затрудняется, поскольку не гарантируется, что следующий запрос клиента будет направлен тому же серверу. Обычно разработчикам приходится сохранять состояния вручную на сервере SQL или в другом внешнем хранилище данных.

Хотя ASP – замечательная технология построения динамических страниц, в ней есть место для совершенствования. ASP.NET представляет собой развитие ASP и преодолевает большинство, если не все ее недостатки.

ASP.NET

Разработчики, применяющие Visual Basic, имеют возможность ощутить легкость программирования форм и элементов управления. Создание приложения с формами VB заключается в простом перетаскивании некоторых элементов управления на форму и в написании функций обработки событий. Это одна из причин, по которым VB привлек множество пользователей, заинтересованных в скорости разработки. Компания Microsoft поступила разумно, встроив эту возможность в ASP.NET.

ASP.NET упрощает разработку веб-страниц за счет программирования на базе форм. В ASP.NET эти формы называются *веб-формами* (*Web Forms*), и аналогичны формам VB, заменяя собой ASP-страницы. Как и в VB, программирование веб-формы также базируется на событиях. Теперь нам не надо писать встраиваемые в страницу сценарии ASP и полагаться на линейную интерпретацию, как в ASP-программировании. Для соответствия богатому набору любимых VB-программистами элементов управления ActiveX из их инструментария ASP.NET оснащает ASP-программистов *серверными элементами управления* (*server controls*). Для дальнейшего увеличения производительности разработчиков веб-формы ASP.NET также позволяют разделять логику приложения и уровень представления.

ASP.NET развивает модель программирования ASP за счет следующих дополнительных преимуществ:

- четкого разделения логики приложения (серверного кода) и уровня представления (HTML-разметки) – отсутствие «спагетти»-кода;
- богатого набора серверных элементов управления, автоматически генерирующих HTML для любых клиентов, а также управляющих своим состоянием;
- расширенного управления состоянием сеанса;
- событийно-ориентированной модели программирования на сервере, более простой и интуитивно понятной;
- возможности написания логики приложения на любом языке Microsoft .NET (VB, C#, Managed C++ и т. д.), для лучшей производительности серверный код приложения компилируется;
- использования Visual Studio .NET в качестве средства быстрой разработки, упрощающего процесс разработки веб-форм.

Пространство имен System.Web.UI

Прежде чем заняться разработкой нашего любимого приложения *Hello, World* на ASP.NET, важно ознакомиться с базовой структурой ASP.NET. В этом разделе описаны некоторые из самых важных классов пространства имен System.Web.UI в среде ASP.NET.

В пространстве имен System.Web.UI определены классы и интерфейсы, используемые при создании и отображении элементов веб-формы. Самый важный класс System.Web.UI – это класс Control, определяющий свойства, методы и события, общие для всех серверных элементов управления среды Web Forms. Другой важный класс этого пространства имен – это Page, являющийся производным от класса Control. Все веб-страницы ASP.NET представляют собой экземпляры производных от Page классов. Чтобы сделать среду расширяемой, пространство имен System.Web.UI также включает класс UserControl, аналогичный классу Page, за исключением того, что он в основном выступает в качестве базового класса пользовательских элементов управления. Классы UserControl и Page еще встретятся нам в разделах «Разработка приложений ASP.NET» и «Нестандартные серверные элементы управления» далее в этой главе.

Класс Control

Класс Control – это корневой класс для всех элементов управления. Например, текстовое поле ввода – это элемент управления; кнопка или список – это также элементы управления. Класс Control, по существу, инкапсулирует общую функциональность и свойства всех элементов пользовательского интерфейса. Все, что вы увидите, углубляясь в разработку в ASP.NET, – это различные производные класса Control.

Свойства класса Control

В классе Control имеются следующие важные свойства: Controls, ID, Parent, EnableViewState, Visible, Context и ViewState. Мы кратко рассмотрим каждое из этих свойств, чтобы показать вам, из чего состоит класс Control и как наследованием от класса Control можно создать согласованную и простую в использовании модель. Свойство Controls представляет собой дочерние элементы данного экземпляра элемента управления; свойство Parent определяет родителя данного элемента управления. Эти свойства позволяют формировать иерархию элементов управления веб-страницы. Свойство ID позволяет программно обращаться к элементу управления с помощью нотации «ID и точка» для получения свойств и методов объекта, т. е. Id_Моего_Объекта.имя_свойства. Оно также позволяет нам писать обработчики событий, вызываемых этим элементом управления. Флаг EnableViewState указывает, что элемент управления будет сохранять свое визуальное состояние, а также визуальные состояния своих дочерних элементов.

Если этот флаг установлен в `true`, элемент управления будет помнить свое предыдущее визуальное состояние, когда данные возвращаются в страницу для повторного редактирования (*postback*[1]). Например, если флаг EnableViewState установлен в `true`, то предыдущий выбор пользователя или данные формы автоматически сохраняются, когда пользователь производит какие-либо действия, требующие возврата к странице. Когда страница посылается обратно в броузер, пользователь может просто продолжить заполнение формы, как будто он никогда ее не покидал. Так все производные от класса Control сохраняют свои состояния между запросами и освобождают разработчиков от «слесарных работ» со скрытыми полями форм. Свойство Context позволяет получать информацию о текущем HTTP-запросе, такую как объекты Application, Server, Session, Request и Response. ASP-разработчикам должны быть знакомы эти встроенные объекты. Вы, скорее всего, будете использовать свойство Context при обработке события OnLoad для веб-страницы, чтобы получить переменные уровня приложения или сеанса и параметры запроса для формирования страницы. Свойство Context позволяет получить и другую информацию, такую как кэшированные ресурсы, в том числе соединения с базой данных – для повышения производительности; состояние трассировки – для отладки; и данные пользователя – для обеспечения безопасности.

Свойство ViewState – это экземпляр класса StateBag, предназначенный для хранения информации в виде пар «имя/значение», которая может быть сделана доступной для нескольких запросов к одной веб-странице. Эти пары «имя/значение» представляют собой экземпляры класса StateItem.

Методы класса Control

Список методов класса Control значительно длиннее, чем приведенный в этом разделе; однако этот краткий список, пожалуй, включает все, что вам надо знать, чтобы начать работать с классом Control:

Метод DataBind

Привязка управляющего элемента к источнику данных. Этот метод используется в сочетании с синтаксисом выражения привязки к данным в веб-форме. Когда вызывается метод, все теги привязки к данным, `<%# %>`, вычисляются заново, так чтобы в местах расположения тегов оказались новые данные.

Метод CreateChildControls

Вызывается перед отображением любого составного нестандартного элемента управления. Составной нестандартный элемент управ-

[1] *Postback* – это условие, когда ASP-страница возвращает данные обратно себе для обработки. В обычном ASP-программировании состояние полей формы должно было сохраняться вручную. В ASP.NET мы можем делать это автоматически с помощью простого флага EnableViewState.

ления аналогичен элементу управления ActiveX: он составлен из других элементов управления. Этот метод не требуется, если вам надо просто воспользоваться элементом управления. При разработке нестандартных элементов управления этот метод может быть перекрыт, так что разработчики нестандартных элементов управления могут создать и разместить дочерние элементы управления перед генерацией составного элемента, как при первом запросе, так и при возврате назад.

Метод Render

Как и CreateChildControls, в основном применяется для разработки нестандартных элементов управления. Разработчики элементов управления могут перекрывать этот метод для формирования содержимого элемента управления через предусмотренный параметр HtmlTextWriter. Мы вернемся к методам Render и CreateChildControls, когда далее в этой главе в разделе «Нестандартные серверные элементы управления» будем обсуждать создание элементов управления.

Методы SaveViewState и LoadViewState

Сохраняют и загружают состояние элемента управления. Серверные элементы управления с помощью этих методов сохраняют свое состояние в промежутках между запросами.

Класс Page

Как мы упоминали ранее, класс Page[1] фактически является производным от класса Control. Это значит, что он наследует все свойства, методы и события, предоставляемые классом Control. Кроме унаследованных в классе Page определены свойства, методы и события, более специфичные для веб-страницы ASP.NET. Те, кто занимался разработкой ASP, уже знают, что Application, Request, Response, Server и Session – это внутренние объекты, к которым можно обращаться при программировании собственной ASP-страницы. В ASP.NET эти объекты фактически являются свойствами класса Page. Кроме знакомых вам объектов класс Page также предоставляет другие свойства, такие как Cache, ErrorPage, IsPostBack, IsValid, Trace и Validators.

Свойства и методы класса Page

Этот список неполон; однако он включает некоторые самые важные возможности, которые мы хотим представить:

Свойство Cache

Указывает на объект Cache контекста текущей страницы. Здесь сохраняются для повторного использования такие ресурсы, как под-

[1] Класс Page является производным от TemplateControl, в свою очередь, производного от класса Control.

ключения к базе данных. Их можно не создавать заново, если срок использования кэшированного ресурса еще не истек.

Свойство ErrorPage

Указывает на страницу, отображаемую при возникновении ошибки. Можно определить страницу для сообщений об ошибках и с помощью директивы @Page, как это показано в разделе «Синтаксис веб-формы» этой главы.

Свойство IsPostBack

Определяет, выполняется ли первичный запрос страницы или возврат к ней, когда при взаимодействии пользователя и серверного элемента управления потребовалось выполнить возврат на текущую страницу. Если свойство IsPostBack равно `true`, то не требуется выполнять заново инициализацию вашей страницы, благодаря чему вы можете улучшить производительность.

Свойство Validators

Группирует вместе серверные элементы управления, которые могут подтверждать свою правильность, в свойстве Validators объекта Page. (В ASP.NET веб-страница обычно состоит из нескольких серверных элементов управления.) Когда странице требуется проверить свою правильность, она может делегировать эту проверку всем этим элементам управления и затем установить соответствующее значение свойства IsValid.

Свойство Trace

Ссылается на объект TraceContext, позволяющий выдавать предупреждения и сообщения об ошибках. Трассировка может быть в любое время включена или выключена в настройках `web.config`. Файл *web.config* – это текстовый файл в формате XML, в котором хранится конфигурация времени выполнения приложения ASP.NET. Изменения в этом файле вступают в силу немедленно, в отличие от изменений файла *global.asa* при разработке в ASP. Главный конфигурационный файл находится в корневом каталоге вашего веб-приложения; однако такие файлы могут существовать для каждого подкаталога приложения. Ближайший конфигурационный файл перекрывает настройки более далеких конфигурационных файлов. Возможность отключать трассировку в файле конфигурации значительно лучше, чем ручное отключение трассировки в ASP-разработке, когда приходится просматривать все ASP-файлы для удаления всех экземпляров Response.Write, выводящих отладочные сообщения, когда приложение готово к развертыванию.

Метод LoadControl

Программно загружает серверные элементы управления на страницу. Вы также можете иметь статические серверные элементы управления, объявленные на странице с применением синтаксиса сер-

верных объектов, описанного в разделе «Синтаксис веб-форм» далее в этой главе.

Метод MapPath

Ставит физический путь в соответствие виртуальному пути для осуществления файлового ввода/вывода. Это должно быть знакомо ASP-разработчикам.

Метод Validate

Работает с проверяющими правильность серверными элементами управления для проверки правильности данных на странице. Если какой-либо серверный элемент управления не прошел проверку, метод возвращает `false`, и элемент показывает пользователю сообщение об ошибке.

Метод CreateHtmlTextWriter

Создает объект HtmlTextWriter для выдачи HTML в выходной поток. Это аналогично методу Response.Write в ASP; однако объект HtmlTextWriter значительно умнее, чем простой метод Write. Он помогает разработчику писать корректный HTML-код.

Методы SavePageStateToPersistenceMedium и LoadPageStateFromPersistenceMedium

По умолчанию сохраняют и загружают визуальное состояние всех элементов управления в виде скрытых полей страницы. Если вам это не требуется, можете перекрыть метод SavePageStateFromPersistenceMedium для сохранения визуального состояния не в скрытых полях, а другим способом. Придется также перекрыть метод LoadPageStateFromPersistenceMedium для загрузки сохраненных визуальных состояний на страницу перед ее отображением.

Класс UserControl

Класс UserControl схож с классом Page (см. предыдущий раздел) за исключением свойств и методов, относящихся к странице, таких как ErrorPage, IsValid, User, Validators, MapPath, Validate и CreateHtmlTextWriter.

Класс UserControl обычно применяется как базовый класс для нестандартных элементов управления. Мы также можем создавать нестандартные элементы управления, выполняя наследование непосредственно от класса Control, однако лучше начинать с UserControl, поскольку он не такой «сырой», как класс Control. Если окажется, что UserControl поддерживает свойства и методы, которые в вашем нестандартном элементе управления не нужны, вы можете выбрать наследование от класса Control. Мы покажем, как создавать нестандартные элементы управления, в разделе «Нестандартные серверные элементы управления» позднее в этой главе.

Пространство имен System.Web.UI.HtmlControls

Те, кто когда-нибудь создавал клиентские DHTML-скрипты, знают, каким образом всем HTML-тегам можно поставить в соответствие объекты, используемые в сценариях. ASP.NET переносит это соответствие на сторону сервера. Пока веб-страница не сгенерирована и не отослана клиенту, можно обращаться к любому объекту на странице и манипулировать им.

ASP.NET ставит в соответствие HTML-тегам объекты иерархии серверных классов, определенные в пространстве имен System.Web.UI.HtmlControls. Эти серверные объекты называются HtmlControls, т. к. непосредственно соответствуют стандартным элементам HTML.

Вот, например, простая HTML-страница, полагающаяся на клиентские сценарии для динамического изменения результирующей страницы. (Эта страница не будет работать в броузерах, не поддерживающих клиентские сценарии VBScript, или в броузерах, в которых клиентские сценарии отключены.)

```
<html>
<head>
  <script language=vbscript>
  sub cmd1_onclick()
    txtMessage.InnerHtml = _
        "(Client-side) Ваше имя: " & frm1.txtName.value
  end sub
  </script>
</head>
<body>
  <form id=frm1>
    Введите имя: <input id="txtName" type="text" size="40">
    <input type=button id="cmd1" value="Click Me">
    <span id="txtMessage"></span>
  </form>
</body>
</html>
```

Мы преобразуем эту страницу таким образом, чтобы она полагалась на серверные элементы управления, а не на объектную модель документа Internet Explorer. Вывод страницы осуществляется со стороны сервера, поэтому страница работает независимо от типа броузера. Единственный недостаток состоит в том, что все взаимодействие со страницей потребует отправки данных назад на сервер.

Чтобы воспользоваться преимуществами серверных элементов управления, единственное, что вы должны сделать, это добавить атрибуты `id` и `runat`, и тогда ваш серверный сценарий сможет обращаться к серверным элементам управления и манипулировать ими:

```
<html>
<head>
```

```
  <script id="scr1" language="c#" runat="server">
    void svr_cmd1_onclick(Object o, EventArgs e)
    {
      txtMessage.InnerHtml =
          "(Server-side) Ваше имя: " + txtName.Value;
    }
  </script>
</head>
<body>
  <form id="frm1" runat="server">
    Enter Name: <input id="txtName" type="text" size="40" runat="server">
    <input type="button" id="cmd1" value="Click Me"
        onserverclick="svr_cmd1_onclick" runat="server">
    <span id="txtMessage" runat="server"></span>
  </form>
</body>
</html>
```

Добавив атрибут `runat="server"` к HTML-форме и ее элементам управления, вы предоставите своему серверному сценарию объекты HtmlForm, HtmlInputText, HtmlInputButton и HtmlGenericControl (обозначающий тег span), что показано на рис. 7.1. Как можно видеть из предыдущего сценария, разработчик может манипулировать атрибутом txtMessage объекта HtmlGenericControl для установки его свойства InnerHtml.

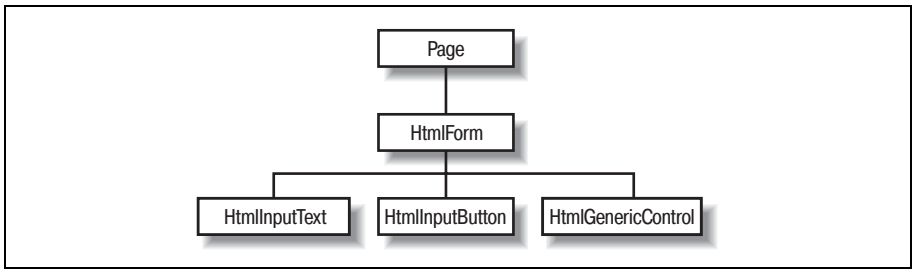


***Рис. 7.1.** Объекты для серверного сценария из примера*

Хотя результаты работы двух простых примеров кажутся одинаковыми, они радикально отличаются с технической точки зрения. Клиентский сценарий, как это следует из названия, работает в рамках процесса клиентского броузера. С другой стороны, когда элементы управления обозначены как работающие на сервере, мы можем из них получить доступ к другим ресурсам сервера.

Большинство классов пространства имен System.Web.UI.HtmlControls – производные от класса HtmlControl, который, в свою очередь, является производным от класса Control пространства имен System.Web.UI. На рис. 7.2 приведено графическое представление иерархии. Класс HtmlControl служит в качестве базового класса для классов HtmlControls, поскольку большинство HTML-элементов разделяют об-

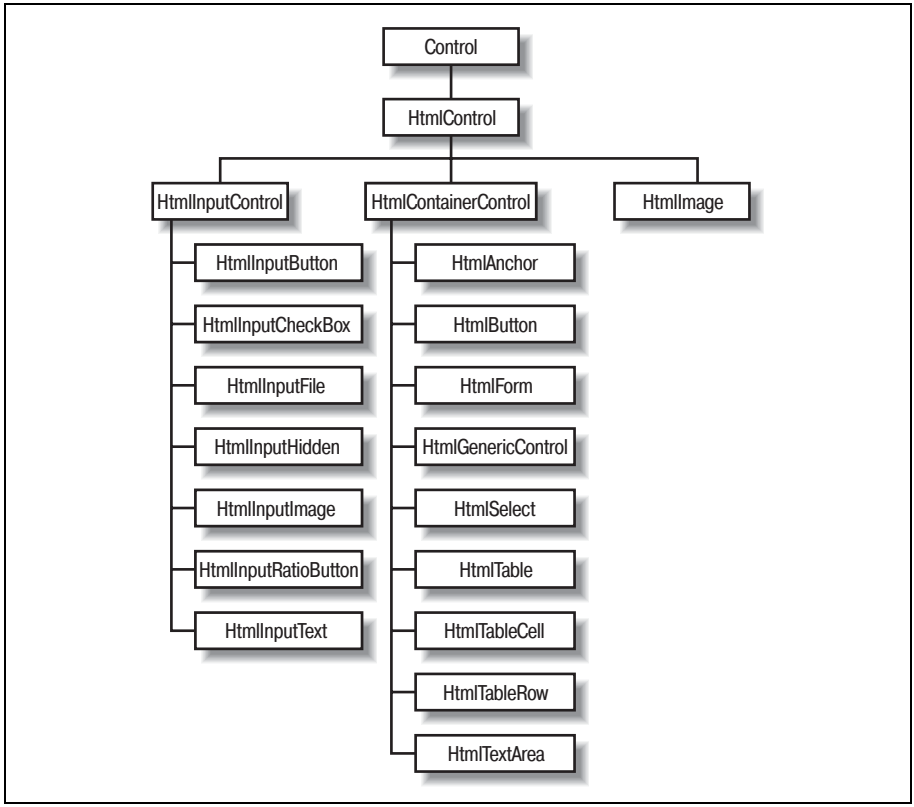


Рис. 7.2. *Иерархия объектов HtmlControls*

щие характеристики, определенные в базовом классе HtmlControl, в частности ID, Disabled, Value, Style и TagName. Классы HtmlControls в конечном счете являются производными класса Control, поэтому они также имеют методы и события, предоставляемые классом Control.

В табл. 7.1 приведено соответствие между HtmlControls и стандартными тегами HTML. Это значит, что если у вас есть тег HTML, помеченный, как работающий на стороне сервера (`runat="server"`), ASP.NET создает соответствующий объект HtmlControl, который вы можете использовать в сценариях.

Таблица 7.1. Соответствие между HtmlControls и тегами HTML

Объект HTMLControl	Описание	Тег HTML
HtmlImage	Графическое изображение	`<img>`
HtmlInputFile	Выбор файла	`<input type="file">`
HtmlInputHidden	Скрытые поля формы	`<input type="hidden">`
HtmlInputImage	Ввод изображений	`<input type="image">`

Объект HTMLControl	Описание	Тег HTML
HtmlInputRadioButton	Переключатель	`<input type="radio">`
HtmlInputText	Стандартное текстовое поле ввода	`<input type="text">`
HtmlInputButton	Стандартная кнопка HTML	`<input type="button">`
HtmlInputCheckBox	Стандартный флажок HTML	`<input type="checkbox">`
HtmlForm	Тег form	`<form>`
HtmlGenericControl	Различные группирующие теги HTML	`<span, div и т.д.>`
HtmlSelect	Поле с выпадающим списком	`<select>`
HtmlTable	Стандартная таблица HTML	`<table>`
HtmlTableCell	Ячейка в строке таблицы	`<td>`
HtmlTableRow	Строка таблицы	`<tr>`
HtmlTextArea	Многострочное текстовое поле ввода	`<textarea rows=n cols=n>`
HtmlAnchor	Стандартная гиперссылка	`<a href=...>` или `<a name=...>`
HtmlButton	Кнопка HTML	`<button>`

Пространство имен System.Web.UI.WebControls

Кроме HtmlControls, соответствующих стандартным HTML-элементам, ASP.NET также предоставляет другую группу элементов пользовательского интерфейса, класс WebControl (рис. 7.3). Кроме традиционных элементов управления, аналогичных HtmlControls, в WebControls также имеются более развитые элементы, такие как календари, таблицы и валидаторы.

Объекты WebControls – более богатые, более мощные и более гибкие по сравнению с HtmlControls. Они представляются естественным выбором для новых приложений ASP.NET; однако HtmlControls являются предпочтительными для переноса приложений ASP. Вот еще одно соображение, которое может склонить вас к выбору HtmlControls, – ваши клиентские сценарии смогут продолжать обращаться к объектам и манипулировать ими.

Большинство классов этого пространства имен базируются на классе WebControl, являющегося производным от класса Control. Класс WebControl предоставляет общие свойства и методы, наследуемые всеми его потомками, в том числе клавишу быстрого доступа, индекс табуляции, подсказку, цвет, шрифт и параметры рамки.

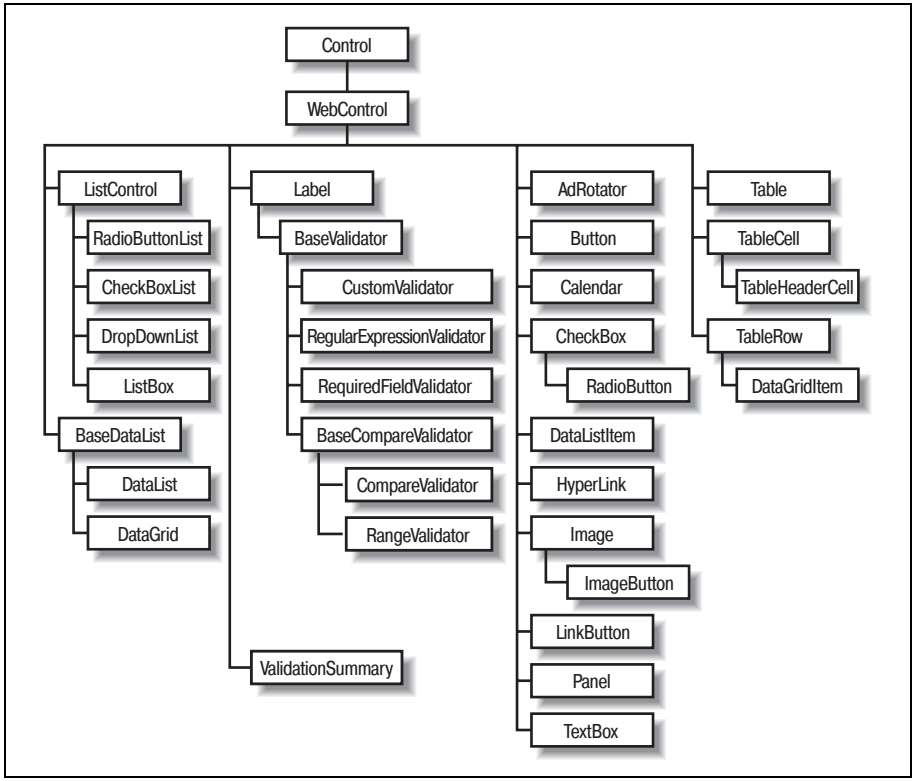


Рис. 7.3. *Иерархия объектов WebControls*

Синтаксис веб-форм

Как и Active Server Pages, веб-формы представляют собой текстовые файлы, состоящие из тегов HTML и управляющих тегов, таких как директивы и блоки сценариев. Стандартное расширение для веб-формы – *aspx*; однако можно воспользоваться утилитой администрирования IIS для явного установления соответствия других расширений модулю *aspnet_isapi.dll*, чтобы эти файлы анализировались и компилировались при обращении к ним, словно они являются ресурсами ASP.NET.

Существует десять различных синтаксических элементов в ASP.NET; т. к. большинство из них перенесены из ASP, здесь мы перечислим те, которые вам знакомы, и обсудим только те, которые в ASP.NET наиболее важны:

Директивы

Блоки объявления кода

Блоки генерации кода

Синтаксис элементов управления HTML

Синтаксис нестандартных элементов управления
Выражения для привязки данных
Теги серверных объектов
Директивы включения файлов на стороне сервера
Серверные комментарии
Текстовые литералы

Директивы

Раньше все ASP-директивы имели формат <%@ [*атрибут=значение*]+ %>, т. к. это был единственный тип директивы.[1]

ASP.NET добавляет несколько директив для файлов ASP.NET. С новым списком директив – Page, Control, Import, Implements, Register, Assembly, OutputCache и Reference – синтаксис директивы теперь определяется как <%@ directive [*атрибут=значение*]+ %>. Все старые ASP-директивы являются атрибутами директивы Page. Если вы придерживаетесь старого синтаксиса и опускаете имя директивы, то пары «атрибут/значение» будут применяться к директиве по умолчанию, то есть к директиве Page.

@Page

Кроме включения всех предыдущих ASP-директив (CodePage, EnableSessionState, Language, LCID и Transaction) директива Page в ASP.NET также поддерживает важные атрибуты ErrorPage, Inherits, Src и EnableViewState, которыми мы воспользуемся в этой главе. Полный список всех атрибутов директивы Page можно найти в *.NET Framework Developers' Guide*.

```
<@ Page Language="VB" ErrorPage="URL" EnableViewState="true">
```

@Control

Так же как директива Page используется для ASP.NET-страницы (*aspx*-файла), директива Control используется для элемента управления ASP.NET (*ascx*-файл). (В разработку управляющих элементов ASP.NET мы углубимся в разделе «Нестандартные серверные управляющие элементы» далее в этой главе.)

@Import

Директива Import применяется для добавления в текущую страницу ссылок на пространства имен. Ваш код сможет обращаться ко всем классами и интерфейсам импортированных пространств имен. Так, для того чтобы использовать ADO.NET, включите следующий код:

---

[1] Как было отмечено в предисловии, знак «плюс» здесь обозначает один и более экземпляров предыдущего терма, указанного в скобках.

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.OleDb" %>
```

Несколько пространств имен автоматически импортируются во все страницы ASP.NET для упрощения выполнения задач разработчика:

System
System.Collections
System.Collections.Specialized
System.Configuration
System.IO
System.Text
System.Text.RegularExpressions
System.Web
System.Web.Caching
System.Web.Security
System.Web.SessionState
System.Web.UI
System.Web.UI.HtmlControls
System.Web.UI.WebControls

@Implements

Поскольку *aspx*-файл, по существу, является производным от класса Page, он также может реализовать интерфейс. При помощи директивы @Implements мы объявляем, что *aspx*-файл реализует указанный интерфейс. Например, в следующей строке объявляется, что страница реализует интерфейс IPostBackEventHandler:

```
<%@ Implements Interface="System.Web.UI.IPostBackEventHandler" %>
```

@Register

Эта директива регистрирует нестандартные серверные элементы управления для применения в текущей странице путем указания псевдонимов для префиксов имен классов. Она используется в сочетании с серверными элементами управления для предоставления краткого способа указания их имен. Следующая строка кода регистрирует нестандартный элемент управления для применения на этой странице:

```
<%@ Register Tagprefix="Ch07"
             TagName="MyCustomControl
             Src="MyCustomControl.ascx" %>
```

Имя элемента управления – `MyCustomControl`, префикс, указываемый при объявлении элемента управления – `Ch07`, исходный текст элемента управления – `MyCustomControl.ascx`. (Мы продемонстрируем его, когда далее в этой главе будем разрабатывать свои элементы управления ASP.NET.)

@Assembly

Директива Assembly указывает сборку, к которой принадлежит текущая страница. Это делает все классы и интерфейсы, принадлежащие к сборке, доступными в текущей странице. Например, следующая строка кода указывает, что текущая страница относится к сборке Ch07:

```
<%@ Assembly Name="Ch07" %>
```

Это значит, что код этой страницы может обращаться ко всему, что находится в сборке Ch07.

@OutputCache

Директива OutputCache может применяться для управления временем выходного кэширования текущей страницы. Это аналогично установке времени истечения срока действия в объекте Response при программировании ASP. Атрибут Duration директивы OutputCache определяет время в секундах до истечения срока действия страницы.

@Reference

Директива @Reference используется для добавления к *aspx*-странице ссылки на страницу или элемент управления.

Блоки объявления кода

Как и в ASP, *блоки объявления кода* (*code declaration blocks*) определяют код страницы, который должен быть проанализирован и запущен. В этих блоках атрибут `runat` указывает, клиентским или серверным является код. Для серверного программирования установите этот атрибут в `server`. Если разработчик проигнорирует атрибут `runat`, IIS будет интерпретировать этот блок как клиентский код, который используется для Dynamic HTML (DHTML).

```
<script runat="server" [language="язык_программирования_кода"]>
  Код
</script>
```

Для блоков объявления как клиентского, так и серверного кода вы также можете указать атрибут src, указывающий на внешний исходный файл, содержащий этот код. Это помогает отделить код от HTML-содержимого страницы. Значением src может быть либо относительный путь, либо URL исходного файла. URL может указывать как на тот же, так и на другой веб-сервер.

```
<script runat="server"
  [language="язык_программирования_кода"]
  [src="имя_внешнего_файла"] />
```

Блоки генерации кода

В отношении этого синтаксиса по сравнению с ASP изменений нет. Встроенный код или встроенные выражения, указанные в этих блоках генерации кода, исполняются при генерации страницы. Все эти блоки заключены между тегами <% и %>. Язык, используемый в этих тегах, задается атрибутом language директивы Page.

Синтаксис элементов управления HTML

Элементы управления HTML очень схожи со стандартными HTML-элементами, за исключением наличия атрибутов id и runat. Те, кто разрабатывает веб-приложения с помощью DHTML, должны быть знакомы с атрибутом id HTML-элемента и с тем, как программно ссылаться на клиентский элемент управления, представляющий элемент HTML. Разница в этом случае состоит в том, что элемент управления находится не на стороне клиента, а на стороне сервера. Например, следующий код представляет собой серверный элемент управления HTML-кнопку:

```
<input id="cmd1" runat="server"
  type="button" value="Нажми меня" />
```

Все серверные элементы управления HTML должны находиться внутри элемента управления <form runat="server">, т. к. веб-формы используют метод POST для сохранения состояний элементов управления.

При обнаружении HTML-элемента, помеченного идентификатором id и атрибутом runat, равным server, ASP.NET создает соответствующий объект сценария HtmlControl. Например, предыдущий фрагмент HTML генерирует серверный объект HtmlControl типа HtmlInputButton с id равным cmd1.

С событием этого элемента управления можно связать обработчик, который будет обрабатывать уведомления от этого элемента, такие как события onclick. Существует два способа привязки обработчика к событию элемента управления – декларативный и программный. Декларативный способ реализуется внутри тега HTML-элемента в виде пары «атрибут/значение». Атрибут – это имя события, а значение – имя функции обработчика события. Например, для обработки события onclick добавьте к предыдущему HTML-тегу следующее:

```
onserverclick="handleServerClick"
```

Программный способ привязки события к обработчику требует написания строки кода, присваивающей делегат свойству события элемента управления. В C# код для привязки события ServerClick кнопки к обработчику handlerServerClick будет следующим:

```
cmd1.ServerClick += new System.EventHandler(handleServerClick);
```

Для тех, кто применял в своих веб-приложениях клиентский DHTML, привязка событий не должна представлять ничего нового, за исключением некоторых мелких различий. Первое различие очевидно: обработчик события исполняется на сервере перед тем, как страница отсылается обратно броузеру, вместо исполнения на стороне клиента. Другое различие заключается в том, что все обработчики событий должны иметь два параметра: Sender и Event. Параметр Sender имеет тип `object` и указывает на исходный элемент, приведший к возникновению события; параметр Event имеет тип `EventArgs` и представляет фактически произошедшее событие. В сценариях DHTML нам надо было тщательно исследовать объект window.event, чтобы выяснить, какой элемент является источником события, и другую информацию о событии.

Синтаксис нестандартных элементов управления

Аналогично серверным элементам управления HTML *нестандартные элементы управления* (*custom controls*) также имеют атрибуты `id` и `runat`; однако нестандартные элементы управления не являются стандартными HTML-элементами. Чтобы вставить на страницу нестандартный элемент управления, воспользуйтесь следующим синтаксисом:

```
<tagprefix:tagname id="controlID" runat="server" eventname=
  "eventHandler" />
```

Заметьте, что все теги нестандартных элементов управления имеют префикс, являющийся псевдонимом пространства имен, в котором определен этот элемент. Посмотрите описание директивы Register ранее в этой главе для получения сведений о регистрации псевдонимов пространств имен. Привязка событий к их обработчикам для нестандартных элементов управления выполняется так же, как и для элементов управления HTML. Несмотря на то что мы показали два способа привязки событий, следует предпочесть второй, т. к. это позволяет более четко отделять теги HTML от обслуживающего их кода.

Все элементы управления, определенные в пространстве имен Web-Controls, могут использоваться таким же образом (они имеют префикс `asp`). Например, в вашей *aspx*-странице могут присутствовать следующие теги:

```
<asp:TextBox id=txt1 runat=server></asp:TextBox>
<asp:Button id=cmd1 runat=server Text="Веб-кнопка"></asp:Button>
<asp:Label id=label1 runat=server></asp:Label>
```

Результатом обработки этих тегов будут три объекта, сгенерированные из трех классов: TextBox, Button и Label из пространства имен System.Web.UI.WebControls. В своем серверном сценарии вы можете обращаться к этим объектам и манипулировать ими для правильной генерации страницы.

Выражения привязки данных

Выражения привязки данных (*data-binding expressions*) связывают серверные элементы управления с какими-либо источниками данных. Синтаксис привязки данных выглядит следующим образом:

```
<%# выражение_привязки_данных %>
```

Изучим следующий блок кода, чтобы увидеть простую привязку данных:

```
<asp:Label text='<%# TestData %>' runat=server/>
```

Приведенное здесь выражение привязки данных указывает, что текстовое содержимое тега label привязано к определенному открытому свойству веб-формы TestData. Это значит, что при выполнении привязки данных к форме тег `<%# TestData %>` будет заменен содержимым свойства TestData. Определим это свойство веб-формы:

```
public string TestData = "Hello World";
```

Среда Web Forms не выполняет привязку данных автоматически. Разработчики должны явно вызывать метод DataBind() для активизации вычисления выражений привязки данных и выполнения подстановки. Мы можем вызвать метод DataBind при загрузке страницы или когда мы меняем значение свойства TestData и хотим отразить это на странице. В данном примере DataBind() вызывается при загрузке страницы для привязки текста метки к переменной TestData.

```
<html>
  <head><title>Пример привязки данных</title></head>
  <body>

    <script language="C#" runat=server>
      /* Объявляем переменную, к которой хотим выполнить привязку. */
      public string TestData;
      void Page_Load(Object oSender, EventArgs oEvent) {
        TestData = "Hello World!\n";
        Page.DataBind();
      }
    </script>

    <asp:Label text='<%# TestData %>' runat=server/>

  </body>
</html>
```

Попробуем сделать что-нибудь более сложное. В следующем блоке тегов у нас есть три метки, привязанные к трем различным свойствам объекта `currStudent`:

```
Name: <asp:Label text='<%# currStudent.FirstName %>' runat=server/>
```

```
<asp:Label text='<%# currStudent.LastName %>' runat=server/> <br/>
SSN: <asp:Label text='<%# currStudent.SSN %>' runat=server/>
```

Объект `currStudent` является открытым свойством текущей страницы.

```
<script language="C#" runat=server>
  public class CStudent {
    /* Объявляем переменную, к которой хотим выполнить привязку. */
    public string FirstName;
    public string LastName;
    public string SSN;
  }
  public CStudent currStudent;

  void Page_Load(Object oSender, EventArgs oEvent) {
    currStudent = new CStudent();
    currStudent.FirstName = "Jack";
    currStudent.LastName = "Daniel";
    currStudent.SSN = "123-45-6789";
    Page.DataBind();
  }

</script>
```

Объект `currStudent` можно заполнить данными из любого источника, а затем выполнить вызов метода DataBind и обновить страницу с использованием текущей информации о студенте. Конечно, здесь предполагается, что класс Student предоставляет ранее упомянутые свойства.

Теги серверных объектов

Теги серверных объектов статически объявляют и создают COM- и .NET-объекты. Синтаксис объявления серверных объектов в *global.asax* следующий:

```
<object id="id" runat="server" scope="диапазон" class="имя_класса .NET ">
<object id="id" runat="server" scope="диапазон" progid="COM_ProgID">
<object id="id" runat="server" scope="диапазон" classid="COM_classID">
```

Параметр `диапазон` может иметь значение pipeline, application или session, т. е. объект будет доступен на странице в качестве переменной приложения или, соответственно, переменной сеанса. Чтобы динамически добавить на страницу серверный объект, необходим метод Page.LoadControl().

Другие элементы

Включение файлов на стороне сервера, серверные комментарии и текстовые литералы аналогичны используемым в ASP. Поэтому здесь мы не будем их рассматривать.

Разработка приложения ASP.NET

В традиционном ASP-программировании разработчики обычно обращаются к объекту Request для получения параметров, необходимых для отображения страниц, и выводят содержимое страницы либо через объект Response, либо в блоках генерации кода. Мы также используем другие объекты ASP, такие как Application, Session и Server для управления переменными приложения, сеанса, настройками сервера и т. д.

Как мы говорили ранее, ASP.NET нацелена на отказ от этой путаницы путем введения значительно более ясного подхода к структуре серверных сценариев: веб-формам или программируемым страницам и серверным элементам управления.

В следующих разделах мы рассмотрим компоненты веб-формы, ее жизненный цикл, серверные элементы управления, содержащиеся в веб-форме, обработку событий для серверных элементов управления, а также создание собственных серверных элементов управления.

Компоненты веб-формы

Как и VB-формы, веб-формы состоят из двух компонентов: формы с элементами управления и кода, обрабатывающего события, связанные с расположенными на форме элементами управления. Файл веб-формы имеет расширение *aspx* и содержит HTML-элементы, а также серверные элементы управления. Код, обслуживающий форму, обычно находится в отдельном файле класса. Заметьте, что хотя возможно размещение и формы и кода в одном файле, лучше работать с разными файлами. Это разделение пользовательского интерфейса и кода приложения помогает устранить симптом «спагетти», от которого страдает большинство приложений ASP.

ASP.NET предоставляет класс Page в пространстве имен System.Web.UI. Этот класс инкапсулирует все общие свойства и методы, связанные с веб-страницами. Код, обслуживающий класс, является производным от класса Page и предоставляет расширения, специфические для реализуемой нами страницы. В *aspx*-файле задается схема страницы и объявления элементов управления. На рис. 7.4 показаны отношения между базовым классом Page, кодом веб-формы, обслуживающим класс, и пользовательским интерфейсом веб-формы.

Как разработчик веб-формы вы должны предоставить последние два компонента. Пользовательский интерфейс веб-формы – это объявление серверных элементов управления с соответствующими идентификаторами (ID). Код, обслуживающий класс, – это программа, обращающаяся к серверным элементам управления, объявленным в пользовательском интерфейсе веб-формы, а также обрабатывающая события от этих элементов управления. В следующем простом примере показана *aspx*-страница, исходный текст обслуживающего кода и их совмест-

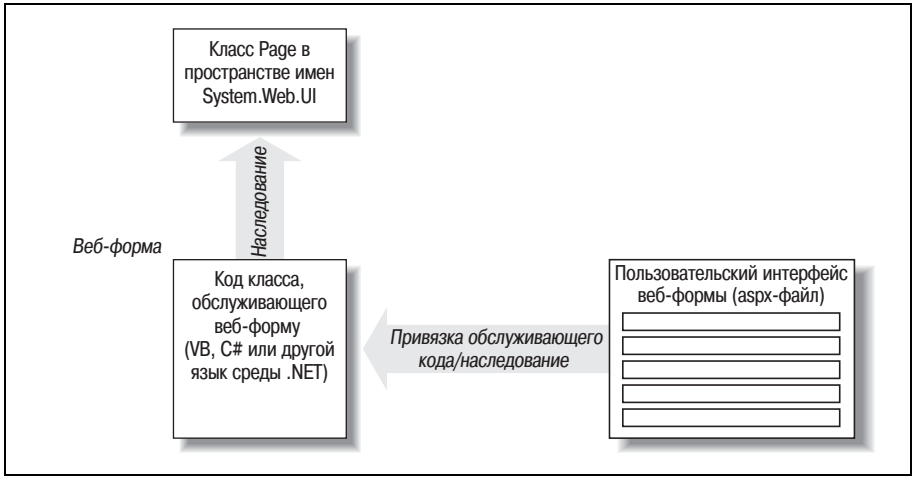


***Рис. 7.4.** Компоненты веб-формы*

ная работа. *Aspx*-файл (*TestEvent.aspx*) содержит только теги HTML и директиву, ссылающуюся на обслуживающий код:

```
<%@ Page language="c#" codebehind="TestEvents.cs" inherits="CTestEvents" %>
<html>
  <head><title>Тестирование обработки событий страницы с помощью
 обслуживающего кода</title></head>
  <body>
    <form runat=server>
      Init Time: <asp:Label id=labelInit runat=server/><br/>
      Load Time: <asp:Label id=labelLoad runat=server/><br/>
      <input type=submit />
    </form>
  </body>
</html>
```

Обслуживающий код, *TestEvents.cs*, содержит класс CTestEvents, на который ссылается *aspx*-страница:

```
using System;

public class CTestEvents : System.Web.UI.Page {
  protected System.Web.UI.WebControls.Label labelInit;
  protected System.Web.UI.WebControls.Label labelLoad;

  public CTestEvents() {
    labelInit = new System.Web.UI.WebControls.Label();
    labelLoad = new System.Web.UI.WebControls.Label();
  }

  public void Page_Init(Object oSender, EventArgs oEvent) {
    labelInit.Text = DateTime.Now.ToString();
  }
```

```
  public void Page_Load(Object oSender, EventArgs oEvent) {
    labelLoad.Text = DateTime.Now.ToString();
    if(IsPostBack) {
      labelLoad.Text += "(PostBack)";
    }
  }
}
```

Вы должны скомпилировать *TestEvents.cs* и поместить DLL в подкаталог */bin* виртуального каталога вашего приложения до того, как попытаетесь обратиться к *aspx*-странице.[1] Команда компиляции этого файла на C# выглядит следующим образом:

```
csc /t:library TestEvents.cs
```

ASP.NET анализирует файлы веб-формы для генерации дерева объектов сценария, где корневым является производный от класса Page объект, представляющий текущую веб-форму. Это аналогично тому, как броузер IE анализирует HTML-файл и генерирует дерево объектов сценария для использования в DHTML; однако дерево объектов для файлов веб-формы располагается на стороне сервера.

Как вы уже знаете из нашего обзора пространства имен System.Web.UI, класс Page фактически является производным от класса Control. В известном смысле веб-форма – это иерархия объектов, производных от Control. Эти объекты устанавливают отношение между родительскими и дочерними элементами через свойства Parent и Controls.

Кроме свойств Controls и Parent класс Page также предоставляет другие полезные свойства, знакомые ASP-разработчикам, – такие как Request, Response, Application, Session и Server.

Поскольку веб-форма – это ни что иное, как программируемый объект страницы, использование такой объектно-ориентированной модели значительно более наглядно и понятно, чем традиционная разработка ASP. В отличие от линейного исполнения серверных сценариев ASP-страницы, ASP.NET предоставляет событийную объектно-ориентированную модель программирования.

Возьмем для примера веб-страницу, содержащую форму с несколькими полями. Одно или более из этих полей выводят информацию из базы данных. Естественно, на нашей ASP-странице имеется код, заполняющий эти поля, чтобы сгенерированная страница уже содержала нужную информацию, когда пользователь ее запросит. Как только последняя строка данных отправлена броузеру, работа ASP-страницы заканчивается. Это значит, что если при передаче формы от пользовате-

[1] Каталог веб-приложения – это корневой виртуальный каталог, где располагается ваше веб-приложение. Чтобы его создать, воспользуйтесь утилитой администрирования IIS.

ля возникли ошибки, нам придется заново заполнить поля формы, связанные с базой данных, а также программно переустановить значения, выбранные пользователем до отправления формы. В ASP.NET нам не надо заново заполнять связанные с базой данных поля, если мы знаем, что они уже были заполнены. Кроме того, выбранные значения останутся выбранными без каких-либо ручных манипуляций. В двух следующих разделах мы рассмотрим эту концепцию более подробно.

События веб-формы

Класс Page предоставляет такие события, как Init, Load, PreRender и Unload. Задача программиста состоит в организации обработки этих событий и выполнении соответствующих действий на каждой из этих стадий. Такая модель значительно лучше, чем модель линейного выполнения в ASP-программировании, поскольку вам не приходится беспокоиться о местоположении ваших сценариев инициализации.

Первое событие, происходящее в жизни веб-формы, – это событие Init. Оно генерируется для того, чтобы мы могли иметь код инициализации страницы. Расположенные на странице элементы управления на данный момент еще не созданы. Это событие возникает один раз для каждого пользователя страницы.

Событие Load следует за событием Init. Впоследствии оно возникает каждый раз, когда запрашивается страница. Когда оно возникает, все дочерние элементы управления веб-формы загружены и доступны. Вы должны иметь возможность извлечь данные и заполнить элементы управления, чтобы они могли отобразить свое содержимое на странице при отсылке ее клиенту.

В следующем примере показано, как в ASP.NET могут быть обработаны события Init и Load. В этом примере мы для простоты приведем HTML и соответствующий ему код вместе в одном файле:

```
<html>
  <head><title>Тестирование событий страницы</title></head>
  <body>

    <script language="C#" runat="server">
      void Page_Init(Object oSender, EventArgs oEvent) {
        labelInit.Text = DateTime.Now.ToString();
      }

      void Page_Load(Object oSender, EventArgs oEvent) {
        labelLoad.Text = DateTime.Now.ToString();
        if(IsPostBack) {
          labelLoad.Text += "(PostBack)";
        }
      }
    </script>

    <form runat="server">
```

```
        Время события Init: <asp:Label id="labelInit" runat="server"/><br />
        Время события Load: <asp:Label id="labelLoad" runat="server"/><br />
        <input type="submit" />
      </form>
    </body>
  </html>
```

При первом обращении к этой странице происходит событие Init, за которым следует событие Load. Поскольку эти события происходят быстро, в строках «Время события Init» и «Время события Load», вероятно, будет показано одно и то же время. При нажатии кнопки submit для повторной загрузки страницы вы увидите, что значение в строке «Время события Init» осталось прежним, а значение в строке «Время события Load» изменяется при каждой загрузке страницы.

Событие PreRender происходит непосредственно перед генерацией страницы и отсылкой ее клиенту. Это событие обрабатывается не часто; однако все зависит от конкретной ситуации.

Последнее событие в жизни формы – это Unload. Оно происходит, когда страница выгружается из памяти. Здесь должна выполняться финальная очистка.

Кроме событий уровня страницы элементы управления могут вызывать такие события, как ServerClick и ServerChange для объектов HtmlControls, а также события Click, Command, CheckedChanged, SelectedIndexChanged и TextChanged для объектов WebControls. Именно обработка этих событий делает страницы ASP.NET по-настоящему динамическими и интерактивными.

Жизненный цикл веб-формы

В ASP веб-страница начинает свое существование, когда клиент запрашивает ее. IIS анализирует и запускает сценарии ASP-страницы для генерации HTML-содержимого. Как только веб-страница сгенерирована, жизнь страницы прекращается. При наличии форм, передающих данные обратно ASP-странице, ASP-страница запускает новый запрос, ничего не зная о своих предыдущих состояниях. Передача данных обратно для обработки исходной страницей называется также обратной передачей (postback).

В ASP.NET все несколько по-другому. Страница также начинается с клиентского запроса; однако она продолжает существовать, пока клиент взаимодействует с ней. Для простоты мы говорим, что страница продолжает существовать, но фактически между запросами к странице сохраняются только ее визуальные состояния. Эти визуальные состояния позволяют элементам управления на сервере вести себя так, будто они все еще существуют и обрабатывают серверные события. Мы можем отследить состояние обратной передачи данных через свойство IsPostBack объекта Page и отказаться от ресурсоемкой инициализа-

ции. Обработка событий во время обратной передачи данных – это именно то, что так сильно отличает ASP.NET от традиционной разработки ASP.

В следующем примере мы расширим код из предыдущего раздела за счет обработки обратной передачи данных. Когда событие Load обрабатывается первый раз, мы заполняем выпадающий список данными. Впоследствии мы лишь указываем время возникновения события, без повторной загрузки данных. В этом примере также продемонстрирован серверный обработчик событий `handleButtonClick`, привязанный к событию `ServerClick` кнопки.

```
<html>
  <head><title>Тестирование событий страницы</title></head>
  <body>

    <script language="C#" runat="server">
      void Page_Init(Object oSender, EventArgs oEvent) {
        labelInit.Text = DateTime.Now.ToString();
      }

      void Page_Load(Object oSender, EventArgs oEvent) {
        labelLoad.Text = DateTime.Now.ToString();
        if(!IsPostBack) {
          selectCtrl.Items.Add("Acura");
          selectCtrl.Items.Add("BMW");
          selectCtrl.Items.Add("Cadillac");
          selectCtrl.Items.Add("Mercedes");
          selectCtrl.Items.Add("Porche");
        } else {
          labelLoad.Text += " (Postback)";
        }
      }

      void handleButtonClick(Object oSender, EventArgs oEvent) {
        labelOutput.Text = "Вы выбрали: " + selectCtrl.Value;
        labelEvent.Text = DateTime.Now.ToString();
      }
    </script>

    <form runat="server">
      Время события Init: <asp:Label id="labelInit" runat="server"/><br/>
      Время события Load: <asp:Label id="labelLoad" runat="server"/><br/>
      Время события ServerClick:
                          <asp:Label id="labelEvent" runat="server"/><br/>
      Выбор: <select id="selectCtrl" runat="server"></select><br/>
      <asp:Label id="labelOutput" runat="server"/><br/>
      <input type=button value="update"
             OnServerClick="handleButtonClick" runat="server" />
    </form>

  </body>
</html>
```

Жизненный цикл веб-формы состоит из трех основных стадий: конфигурации, обработки событий и завершения. Как было упомянуто ранее, эти стадии распределены между несколькими запросами к одной странице, в отличие от политики «страница обслуживается сразу», принятой в ASP.

Конфигурирование

На стадии конфигурирования возникает событие Load объекта страницы. Ваша задача состоит в обработке этого события для настройки своей страницы. Так как событие Load возникает, когда все элементы управления уже готовы, теперь можно читать и изменять свойства элементов управления в процессе настройки страницы. В предыдущем фрагменте кода мы обрабатывали событие Load для заполнения выпадающего списка данными. Мы также изменяли свойство Text элемента управления labelLoad, чтобы на нем было показано время возникновения события Load. В своем приложении вы, вероятно, будете загружать информацию из базы данных и инициализировать поля формы значениями по умолчанию.

Свойство IsPostBack указывает, загружается ли страница первый раз или выполняется обратная передача данных. Например, если у вас есть элемент управления, содержащий список, вы будете заполнять его только при первой загрузке формы, проверив свойство IsPostBack страницы. Если IsPostBack равно `true`, то мы знаем, что в объект списка уже загружена информация и нет необходимости заново его заполнять. В предыдущем фрагменте кода мы пропускали заполнение списка и просто отображали строку `"(Postback)"`.

При первом и последующих посещениях этой страницы может потребоваться выполнить привязку данных и повторное вычисление выражений привязки.

Обработка событий

На этой стадии в результате некоторых событий, инициируемых на стороне клиента, вызываются серверные функции обработки событий страницы. Эти события исходят от элементов управления, которые вы поместили на веб-форму. Жизненный цикл события показан на рис. 7.5.

Завершение

На этой стадии генерация страницы завершена, и она готова к тому, чтобы быть уничтоженной. Вы отвечаете за освобождение файловых дескрипторов, соединений с базой данных и удаление объектов из памяти. Даже несмотря на то, что можно положиться на CLR в осуществлении сборки мусора, мы настоятельно советуем выполнять все действия по очистке самостоятельно, т. к. сборка мусора выполняется лишь периодически. В сильно загруженных системах, если цикл сбор-

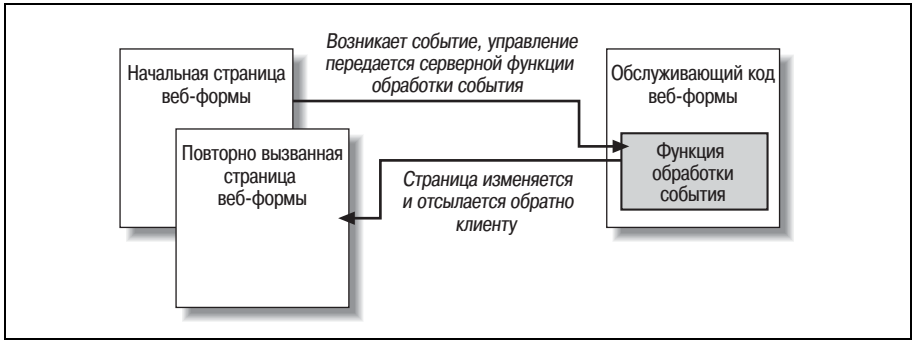


***Рис. 7.5.** Жизненный цикл события веб-формы*

ки мусора не оптимален, неосвобожденные ресурсы могут исчерпать память и привести систему к останову.[1]

Очистку для предыдущего примера можно выполнить с помощью приведенного здесь обработчика события Unload. В этом простом примере освобождать нечего, поэтому мы просто приведем шаблон функции.

```
void Page_Unload(Object oSender, EventArgs oEvent) {
   // здесь находится код очистки
 }
```

Серверные элементы управления

Как мы видели из описаний пространств имен System.Web.UI.HtmlControls и System.Web.UI.WebControls, серверные элементы управления – это программируемые объекты, запускаемые на сервере перед генерацией страницы средой ASP.NET. Они сохраняют свое состояние на сервере между запросами к той же странице путем вставки скрытого поля с данными о визуальном состоянии формы. Таким образом, исключается необходимость в повторном заполнении значений полей формы введенными данными перед отсылкой страницы обратно клиенту.

Серверные элементы управления также независимы от броузера. Поскольку они работают на стороне сервера, то могут использовать свойство Request.Browser для получения информации о возможностях клиентского броузера и генерации соответствующего кода HTML.

Серверные элементы управления – это просто экземпляры классов .NET, поэтому их программирование приводит к созданию кода, который легко поддерживать. В особенности при наличии нестандартных серверных элементов управления, инкапсулирующих другие элемен-

[1] Сборщик мусора не освобождает непреднамеренно занятую память, если у вас есть ссылки на неиспользуемые объекты, которые, однако, остаются в области видимости в течение всей работы приложения. Следовательно, не следует полагаться исключительно на автоматику.

ты, программирование веб-приложений представляет собой ни что иное, как склеивание этих блоков вместе.

Все элементы управления HTML и элементы управления для Интернета, упомянутые в разделах System.Web.UI.HtmlControls и System.Web.UI.WebControls, – это серверные элементы управления, поставляемые с ASP.NET.

Нестандартные серверные элементы управления

В процессе изучения среды ASP.NET и работы с серверными элементами управления в своих веб-формах вам, в конце концов, потребуется узнать, как самому их разрабатывать. В ASP.NET имеется два способа создания нестандартных серверных элементов управления: *пейджлеты* (*pagelets*) – простой, но довольно ограниченный по функциональности подход, и производные классы от базового класса Control (или UserControl) – подход более сложный, однако при этом более мощный и гибкий.

Пейджлеты

До сегодняшнего дня повторное использование кода в ASP было возможно только в форме включений на стороне сервера (server-side includes). Если у вас есть общие блоки пользовательского интерфейса или сценарии, вы можете перенести их во включаемый файл. Для включения общего файла в основную передаваемую в броузер страницу применяется синтаксис `<!--#include file="url" -->`. Это хороший подход, однако он имеет серьезные ограничения. Самое главное – убедиться в уникальности идентификаторов (ID) тегов HTML-сценариев и имен переменных. Это необходимо потому, что при разборе серверной директивы include IIS не делает ничего, кроме включения файла. Включаемый файл в результате оказывается в той же области видимости, что и файл-контейнер. Нельзя включить один и тот же файл дважды, т. к. возникнут конфликты идентификаторов тегов и сценариев.

В ASP.NET можно выделить общий HTML-код и сценарии в то, что сейчас называется «пейджлет», и повторно использовать его, не беспокоясь о конфликтах идентификаторов. Пейджлет – это веб-форма, не имеющая тегов BODY и FORM и связанных с ней сценариев. HTML-часть пейджлета отвечает за компоновку и пользовательский интерфейс, а сценарии обеспечивают программируемость пейджлета, предоставляя свойства и методы. Пейджлет рассматривается как пользовательский элемент управления, поэтому он предоставляет независимую область видимости. Можно совершенно спокойно поместить на страницу несколько экземпляров пользовательского элемента управления.

Веб-форма, являющаяся контейнером, должна зарегистрировать пейджлет как пользовательский элемент управления с помощью директивы `@Register` и включить его на страницу в соответствии с син-

таксисом <*prefix:tagname*>. Если на странице-контейнере используется более одной копии пейджлета, каждой из них должен быть присвоен уникальный идентификатор для того, чтобы скрипты страницы-контейнера работали корректно. Сценарий формы-контейнера может обращаться к пейджлету и манипулировать им так же, как и любым другим серверным элементом управления. В следующем примере показано, как форма ввода адреса применяется несколько раз в качестве пейджлета. С помощью этой формы можно организовать регистрацию пользователя в приложении или отображение адресов доставки и плательщика по завершении пользователем своей работы:

```
<table>
  <tr>
    <td><asp:Label id="labelName" runat="server">Имя</asp:Label></td>
    <td><asp:TextBox id="txtUserName" runat="server"
         Width="332" Height="24"></asp:TextBox></td>
    </tr>
  <tr>
    <td><asp:Label id="labelAddr1" runat="server">Адрес</asp:Label></td>
    <td><asp:TextBox id="txtAddr1" runat="server"
         Width="332" Height="24"></asp:TextBox></td>
    </tr>
  <tr>
    <td><asp:Label id="labelAddr2" runat="server"></asp:Label></td>
    <td><asp:TextBox id="txtAddr2" runat="server"
         Width="332" Height="24"></asp:TextBox></td>
    </tr>
  <tr>
    <td><asp:Label id="labelCity" runat="server">Город</asp:Label></td>
    <td>
    <asp:TextBox id="txtCity" runat="server"></asp:TextBox>
    <asp:Label id="labelState" runat="server">Штат</asp:Label>
    <asp:TextBox id="txtState" runat="server" Width="34" Height="24">
      </asp:TextBox>
    <asp:Label id="labelZIP" runat="server">ZIP</asp:Label>
    <asp:TextBox id="txtZIP" runat="server" Width="60" Height="24">
      </asp:TextBox>
    </td>
    </tr>
  <tr>
    <td><asp:Label id="labelEmail" runat="server">Email</asp:Label></td>
    <td><asp:TextBox id="txtEmail" runat="server"
         Width="332" Height="24"></asp:TextBox></td>
    </tr>
</table>

<script language="C#" runat="server" ID="Script1">
  public String UserName {
    get { return txtUserName.Text; }
    set { txtUserName.Text = value; }
  }
```

```
  public String Address1 {
    get { return txtAddr1.Text; }
    set { txtAddr1.Text = value; }
  }
  public String Address2 {
    get { return txtAddr2.Text; }
    set { txtAddr2.Text = value; }
  }
  public String City {
    get { return txtCity.Text; }
    set { txtCity.Text = value; }
  }
  public String State {
    get { return txtState.Text; }
    set { txtState.Text = value; }
  }
  public String ZIP {
    get { return txtZIP.Text; }
    set { txtZIP.Text = value; }
  }
</script>
```

Чтобы воспользоваться своим пейджлетом, зарегистрируйте его как серверный элемент управления с помощью директивы `@Register`**, как это показано в следующем блоке кода. После регистрации включите тег пейджлета, словно это обычный серверный элемент управления. Укажите префикс, имя тега, идентификатор серверного элемента управления и установите свойство** `runat` **равным** `server`**.**

```
<%@ Register TagPrefix="Acme" TagName="Address" Src="Address.ascx" %>
<%@ Page language="c#"%>
<html>
<head>
  <script language="C#" runat="server">
    void Page_Load(Object oSender, EventArgs evt) {
      addr.UserName = "Jack Daniel";
    }
  </script>
</head>
<body>
    Добро пожаловать в E-Shop.
    Регистрация в E-Shop позволит вам получать ежемесячные новости о
    выгодных покупках.
  <form method="post" runat="server">
    <p><Acme:Address id="addr" runat="server"></Acme:Address></p>
    <p><asp:Button id="cmdClear" runat="server" Text="Clear"></asp:Button>
       <asp:Button id="cmdSubmit" runat="server" Text="Submit">
       </asp:Button></p>
  </form>
</body>
</html>
```

Вы должны иметь возможность обращаться из программы к свойствам пейджлета, используя идентификатор серверного элемента управления (в нашем случае addr). В предыдущем примере мы обращаемся к свойству UserName пейджлета Address через его идентификатор:

```
addr.UserName = "Jack Daniel";
```

Для страницы завершения работы с электронным магазином необходимы два экземпляра <Acme:Address> на одной странице: один для адреса плательщика и другой для адреса доставки. Ваш сценарий должен обращаться к этим экземплярам пейджлета через идентификаторы, которые вы присвоили каждому элементу управления.

Можно также программно создавать экземпляры пейджлета при помощи метода LoadControl объекта Page. Сначала необходимо объявить в своем сценарии переменную типа Control, которая будет содержать пейджлет. Это делается потому, что Control является корневым для всех объектов, в том числе для вашего пейджлета. Затем установите значение переменной с помощью вызова метода LoadControl, передав ему имя файла страницы с элементом управления. Чтобы сделать элемент управления видимым на странице, добавьте его в коллекцию элементов управления страницы. Так как в данный момент у вас имеется объект класса Control, вы не можете вызвать свойства и методы пейджлета, пока не преобразуете тип переменной из Control в тип вашего пейджлета. Это аналогично использованию переменной типа Object для COM-компонента в Visual Basic. Чтобы обратиться к методам и свойствам COM-компонента, надо преобразовать переменную типа Object в тип компонента. При загрузке пейджлета ему автоматически присваивается тип имястраницы_расширение. Например, если пейджлет назван *myControl.ascx*, то для него будет генерироваться тип myControl_ascx. В выделенной полужирным начертанием строке в следующем примере показано, как addr1 преобразуется из типа Control в тип Address_ascx, чтобы можно было обращаться к свойству UserName пейджлета:

```
<%@ Register TagPrefix="Acme" TagName="Address" Src="Address.ascx" %>
<%@ Page language="C#" %>
<html>
<head>
  <script language="C#" runat="server">
    void Page_Load(Object oSender, EventArgs evt) {
      addr.UserName = "Jack Daniel";
      Control addr1;
      addr1 = LoadControl("Address.ascx");
      ((Address_ascx)addr1).UserName = addr.UserName;
      this.frm.Controls.AddAt(3, addr1);
    }
  </script>
</head>
```

```
<body>
  <form id="frm" method="post" runat="server">
    Адрес плательщика:<br/>
    <Acme:Address id="addr" runat="server"></Acme:Address>
    Адрес доставки:<br/>
    <p><asp:Button id="cmdClear" runat="server" Text="Clear">
       </asp:Button>
       <asp:Button id="cmdSubmit" runat="server" Text="Submit">
       </asp:Button>
    </p>
  </form>
</body>
</html>
```

Этот пример со страницей выписки счета показывает, как статически объявить пейджлет на странице с помощью тега <Acme:Address> и как динамически создать нестандартный элемент управления Address с помощью метода LoadControl() объекта Page. Динамически создавая элемент управления, необходимо прежде, чем с ним работать, преобразовать объект к типу данного элемента управления.

Метод AddAt() применяется для вставки пейджлета Address в определенное место страницы выписки счета. Вместо объявления динамического пейджлета как объекта типа Control можно также указать в объявлении его собственный тип – Address_ascx. В этом случае надо преобразовать его один раз при загрузке динамического элемента управления:

```
Address_ascx addr2 = (Address_ascx)LoadControl("Address.ascx");
addr2.UserName = "ABC";
```

Производные элементы управления

Хотя пейджлеты позволяют легко создавать нестандартные элементы управления, этот прием недостаточно гибок для создания мощных элементов, например, генерирующих события или обеспечивающих иерархию элементов управления. В ASP.NET можно создавать нестандартные элементы управления, наследуя от базового класса Control и перекрыв некоторые из его методов.

Следующий пример иллюстрирует создание простейшего нестандартного элемента управления как производного от Control.

```
namespace MyWebControls
{
  using System;
  using System.Web.UI;
  using System.Web.UI.WebControls;
  using System.ComponentModel;

  public class MyWebControl : System.Web.UI.WebControls.WebControl
  {
```

```
        //protected override void Render(HtmlTextWriter output)
        //{
        //    output.Write("тестирование нестандартного элемента управления"+
        //    "через метод Render()");
        //}

        protected override void CreateChildControls()
        {
          Table tbl = new Table();
          TableRow row = new TableRow();
          TableCell cell = new TableCell();
          HyperLink a = new HyperLink();
          a.NavigateUrl = "http://msdn.microsoft.com";
          a.ImageUrl = "image url";
          cell.Controls.Add (a);
          row.Cells.Add(cell);
          tbl.Rows.Add(row);

          row = new TableRow();
          cell = new TableCell();
          cell.Controls.Add (new LiteralControl("тестирование нестандартного"+
                             " элемента управления "));
          row.Cells.Add(cell);
          tbl.Rows.Add(row);

          tbl.BorderWidth = 1;
          tbl.BorderStyle = BorderStyle.Ridge;

          Controls.Add(tbl);
        }
      }
    }
```

Как видите, объект MyWebControl является производным от класса WebControl. Мы знаем, что WebControl в конечном счете – производный от базового класса Control. Все, что мы здесь делаем, это переопределяем либо метод Render, либо метод CreateChildControl для формирования нестандартного элемента управления. Тому, кто предпочитает перекрыть метод Render, придется генерировать HTML-код для своего нестандартного элемента управления через объект HtmlTextWriter, `output`. Можно применять такие методы, как Write, WriteBeginTag, WriteAttribute и WriteEndTag.

В нашем примере мы перекрываем метод CreateChildControls. Не беспокоясь о реальных HTML-тегах и именах атрибутов, мы создаем объекты ASP.NET непосредственно по именам их классов, таким как Table, TableRow, TableCell, HyperLink и Literal, конструируя иерархию объектов внутри таблицы. Мы также можем манипулировать атрибутами объектов через их свойства. В конце метода мы добавляем таблицу к коллекции элементов управления нашего нестандартного элемента управления.

Для генерации DLL-сборки надо скомпилировать код элемента управления (например, так: `csc /t:library MyWebControls.cs`). Чтобы использовать элемент управления, установите сборку, скопировав ее в каталог */bin* своего приложения. Тогда у вас будет возможность зарегистрировать элемент управления с помощью директивы `@Register` и использовать его как обычный серверный элемент управления ASP.NET. Если вы работаете с Visual Studio.NET, то можете добавить ссылку на файлы сборки элемента управления или его проект в тестовый проект, использующий этот элемент управления.

Ваша страница для тестирования нестандартного элемента управления теперь выглядит следующим образом:

```
<%@ Page language="c#"%>
<%@ Register TagPrefix="WC" Namespace="MyWebControls"
             Assembly="MyWebControls"%>
<html>
<head>
  <script language="C#" runat=server>
    void Page_Load(object sender, EventArgs e) {
      MyWebControls.MyWebControl myCtrl;
      myCtrl = new MyWebControls.MyWebControl();
      this.Controls.Add(myCtrl);
    }
  </script>
</head>
<body>
  <form method="post" runat="server">
    Это главная страница
    <WC:MyWebControl id="myControl1" runat="server" />
  </form>
</body>
</html>
```

Как видите, мы зарегистрировали нестандартный элемент управления с помощью директивы `@Register` и присвоили пространству имен MyWebControls префикс `WC`. Мы можем добавить нестандартный элемент управления в тело веб-формы с помощью тега `<WC:MyWebControl>`.

Кроме вставки нестандартного элемента управления на страницу декларативным образом, как показано выше, мы также можем программно создать нестандартный элемент управления во время выполнения. Код обработчика события Page_Load это демонстрирует:

```
MyWebControls.MyWebControl myCtrl;
myCtrl = new MyWebControls.MyWebControl();
this.Controls.Add(myCtrl);
```

Результирующая страница показана на рис. 7.6.

Рис. 7.6. *Результат работы теста нестандартного элемента управления, созданного статически и динамически*

Событийное программирование

Существует два способа привязки обработчиков событий (функций, обрабатывающих события) к элементам пользовательского интерфейса.

Обратитесь к одному из предыдущих разделов этой главы, «Синтаксис веб-форм», в частности к нашему описанию синтаксиса серверных элементов управления. Все, что требуется для привязки события элемента управления к обработчику – это использовать пару «атрибут/значение» следующего вида: *имя_события=имя_обработчика*. Например, для того чтобы обработать событие `onclick` для поля ввода HTML, необходимо сделать следующее. Заметьте, что для элементов управления HTML серверное событие нажатия кнопки мыши называется `onserverclick`, в отличие от клиентского события `onclick`, которое все еще может использоваться для сценариев DHTML:

```
<input id="cmd1" runat="server"
  onserverclick="OnClickHandler"
  type="button" value="Щелкни меня">
```

Для веб-элементов управления в ASP.NET предусмотрен тот же синтаксис:

```
<asp:Button id="cmd2" runat="server"
  onclick="OnclickHandler2"
  Text="Меня тоже щелкни"></asp:Button>
```

После привязки события к имени функции-обработчика мы должны предоставить сам обработчик события:

```
void OnClickHandler(object sender, EventArgs e)
{
```

```
    // код для извлечения и обработки введенных данных
}
```

Второй способ привязки событий – это делегирование. Вам не надо иметь какое-либо представление о коде в *aspx*-файле, даже об имени функции обработки события. Все, что необходимо – это зарегистрировать обработчик события в свойстве события элемента управления. Для веб-элементов управления свойство-событие нажатия кнопки мыши – это `Click`. Для элементов управления HTML – `ServerClick`:

```
IDУпрЭлемента.Click += new System.EventHandler (this.ИмяОбработчикаСобытия);

IDУпрЭлемента.ServerClick +=
                    new System.EventHandler (this. ИмяОбработчикаСобытия);
```

ASP.NET и веб-службы

Среда ASP.NET упрощает разработку веб-служб. Вся низкоуровневая работа, такая как упаковка и распаковка данных в формат XML и использование протокола HTTP для передачи веб-сообщений между распределенными компонентами, выполняется на уровне среды. Это позволяет разработчикам сосредоточиться на логике приложения.

Среда .NET использует расширение *asmx* в качестве стандартного расширения для файлов веб-служб, в отличие от *aspx* для веб-форм и *ascx* для элементов управления веб.

Директива WebService

Все *asmx*-файлы начинаются с директивы `@WebService`, указывающей ASP.NET на то, как компилировать код, а также задающей имя основного класса. Директива WebService имеет следующие атрибуты:

Language

Определяет язык, на котором написан код. Этот атрибут указывает среде ASP.NET, с помощью какого компилятора следует создавать вашу веб-службу. Для Visual Basic применяется `VB`, для C# – `C#`, а для JScript .NET – `JS`. По мере появления других языков, очевидно, можно будет указать и их.

Class

Определяет класс, предоставляющий веб-методы. Среда ASP.NET создает экземпляр этого класса, чтобы клиенты могли вызывать его методы.

Codebehind

Указывает файл исходного текста вашего кода, обеспечивая полное разделение кода и ASP-страницы.

Нетрудно создать простую веб-службу, подобную приведенному ниже файлу *asmx*:

```
<%@ WebService Language="VB" class="MathClass" %>
imports System.Web.Services
Public Class MathClass
  <WebMethod> _
  public function Add(a as integer, b as integer) as integer
    return(a + b)
  end function
end class
```

Обратите внимание на символ продолжения сразу после <WebMethod>. Если вы предпочитаете полностью отделять ваш код от любых элементов ASP.NET, то можете сохранить код вашей веб-службы в отдельный файл и задать атрибут Codebehind директивы @WebService, указывающий на файл кода:

```
<%@ WebService Language="VB" Codebehind="MathClass.vb" Class="MathClass" %>
```

Исходный код *MathClass.vb* выглядит точно так же, как и файл *asmx*, приведенный выше, только без первой строки. Скомпилировать *MathClass.dll* можно при помощи следующей команды:

```
vbc /t:library /r:System.Web.Services.dll MathClass.vb
```

Как и любой обслуживающий код, двоичный файл должен быть помещен в подкаталог */bin* каталога приложения.

Атрибут WebMethod

Открытые методы любых классов могут быть помечены атрибутом WebMethod, чтобы сделать их доступными через Интернет. Синтаксис атрибутов методов различен в разных языках .NET. Например, в C# он имеет следующую форму:

```
[WebMethod(атрибут="значение" атрибут="значение" ...)]
    public типВозвращаемогоЗначения ИмяФункции(списокПараметров)
```

В VB вместо квадратных скобок применяются угловые, а для присваивания – := вместо =. Заметьте также, что объявление веб-метода располагается в одной строке. Если вам для улучшения его читаемости надо его разделить, используйте символ продолжения строки — _:

```
<WebMethod(атрибут:="значение" атрибут:="значение" ...)> Public Function
  ИмяФункции(списокПараметров) as типВозвращаемогоЗначения

<WebMethod(атрибут:="значение" атрибут:="значение" ...)> Public Sub
  ИмяПодпрограммы(списокПараметров)
```

Использование веб-служб

Если вы работаете с Visual Studio.NET, то можете выбрать в меню Project пункт Add Web Reference и затем ввести URL веб-службы. Для наших целей мы укажем веб-службу, созданную в предыдущей главе, PubsWS. URL этой веб-службы на нашем сервере – *http://localhost/PubsWS/PubsWS.asmx*. Добавив эту ссылку, можно обращаться к объекту-посреднику веб-службы, вызывая его с помощью выражения вида имясервера.имяОбъектаПосредника. В нашем случае оно выглядит так: localhost.PubsWS.[1]

Следующий фрагмент кода демонстрирует, как использовать веб-службу через посредника. Мы создадим экземпляр объекта-посредника и затем попросим его передать сообщение реальной веб-службе для получения списка авторов. Результат будет передан обратно в формате XML, из которого будет реконструирован объект DataSet. Затем мы привязываем элемент управления DataGrid1, представляющий собой просто объект класса DataGrid, расположенный на веб-форме, к стандартному представлению первой таблицы объекта DataSet. И наконец, мы посылаем запрос на осуществление реальной привязки. Полученная в результате страница представляет собой элемент управления data grid, заполненный строками из таблицы Authors базы данных Pubs.

```
localhost.PubsWS ws = new localhost.PubsWS();
DataSet ds = ws.GetAuthors();
DataGrid1.DataSource = ds.Tables[0].DefaultView;
DataGrid1.DataBind();
```

Вместо поиска и автоматической генерации класса-посредника средствами Visual Studio.NET, можно также на основании информации из предыдущей главы сгенерировать исходный текст класса самостоятельно. А затем включить этот исходный текст в проект или скомпилировать его в DLL и добавить DLL к проекту в виде ссылки. В любом случае конечный результат будет одним и тем же. Вот пример, связывающийся с посредником, созданным нами в предыдущей главе, и заполняющий элемент управления data grid данными:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>

<!-- Связываемся с посредником, сгенерированным утилитой wsdl.exe -->
<%@ Assembly Src="PubsWS.cs" %>

<html>
  <head>
    <title>Клиент SOAP</title>
```

[1] Можно переименовать веб-ссылку после добавления ее в свой проект. В этом случае веб-служба будет иметь имя *<вашеимявебслужбы>*.имяОбъектаПосредника вместо *имясервера*.имяОбъектаПосредника.

```
  </head>
  <body>

  <!-- Выполняем SOAP-вызов и заполняем data grid. -->
  <%
    PubsWS ws = new PubsWS();
    DataSet ds = ws.GetAuthors();
    dg.DataSource = ds.Tables[0].DefaultView;
    dg.DataBind();
  %>

  <!-- Создаем data grid. -->
  <asp:DataGrid id="dg" runat="server"/>

  </body>
</html>
```

Привязка данных и применение шаблонов

Хотя все элементы управления могут иметь привязку к данным, только DataGrid, DataList и Repeater используют шаблон для отображения данных. В этом разделе мы покажем, как выполнять простую привязку данных с некоторыми распространенными элементами управления и как применять шаблон HTML для создания полностью настраиваемых элементов управления, привязанных к данным.

В своей простейшей форме привязка данных – это установление связи между элементом управления и источником данных. Раньше привязка данных требовала, чтобы источником данных был набор записей ADO, что не очень гибко. Например, нет возможности привязать к элементу управления массив или коллекцию объектов. В ASP.NET вся система привязки данных была переделана. Единственное требование, которому должен отвечать источник данных, чтобы его можно было использовать для привязки к элементам управления, – это реализация им интерфейса System.Collections.ICollection. Другими словами, подходящий для привязки источник данных – это коллекция однородных объектов, из которых элементы управления могут получать данные.

Хотя можно написать собственные классы работы с данными, реализующие интерфейс Icollection, и привязать их к элементам управления, существует множество классов, делающих это за вас, таких, как Array, ArrayList, DataView, HashTable, Queue, SortedList и Stack. Требуется лишь поместить свои данные в объекты этих классов, и можно будет привязать их к веб-элементам управления.

Вот простейшая форма привязки данных. В файле формы у нас есть лишь два списка с идентификаторами `list0` и `list1`:

```
<asp:listbox id="list0" runat="server"></asp:listbox>
<asp:listbox id="list1" runat="server"></asp:listbox>
```

В обработчике Page_Load, расположенном в файле обслуживающего кода, мы создаем источники данных типа Array, которые реализуют интерфейс ICollection, упомянутый ранее, а затем выполняем привязку элементов управления – списков к этим источникам данных:

```
int[] myArray0 = new int[7] { 1, 2, 3, 5, 7, 11, 13 };
string[] myArray1 = new string[7] {
                                  "Monday",
                                  "Tuesday",
                                  "Wednesday",
                                  "Thursday",
                                  "Friday",
                                  "Saturday",
                                  "Sunday"
                                     };
list0.DataSource = myArray0;
list0.DataBind();
list1.DataSource = myArray1;
list1.DataBind();
```

На рис. 7.7 показан результат работы этой страницы.

Рис. 7.7. *Привязка к данным из массивов*

В большинстве случаев мы склоняемся к привязке к данным из источников, относящихся к базам данных. Следующий пример извлекает данные из таблицы Categories знакомой нам базы данных NorthWind. Мы все еще можем использовать элемент управления `asp:listbox`, но на сей раз мы задаем свойство элемента привязки, указывающее, откуда берется отображаемый текст и свойство, указывающее, откуда берутся значения элементов списка. В предыдущем случае нам не обязательно было это делать, поскольку список привязывался к массиву простых типов (целых и строк). Если бы нам требовался массив объектов, мы бы указали свойство, которое хотим привязать к `datavaluefield` и `datatextfield`, так же, как мы это делаем в следующем примере:

```
<asp:listbox id=ListBox1 runat="server"
    datavaluefield="CategoryID"
    datatextfield="CategoryName">
</asp:listbox>
```

И снова в исходном тексте обслуживающего кода нам придется запрограммировать создание источника данных и его привязку к элементу управления – списку. Заметьте, что поскольку мы используем ADO.NET для получения информации из базы данных, нам необходи-

мы ссылки на пространства имен System.Data и System.Data.OleDb. Свойство DefaultView класса Table имеет тип DataView, в котором реализован интерфейс IColleciton.

```
System.Data.DataSet m_ds = new System.Data.DataSet();
String sConn =
    "provider=SQLOLEDB;server=(local);database=NorthWind;Integrated
Security=SSPI";
String sSQL =
    "select * from Categories";

System.Data.OleDb.OleDbDataAdapter da =
    new System.Data.OleDb.OleDbDataAdapter(sSQL, sConn);
da.Fill(m_ds, "Categories");

ListBox1.DataSource = m_ds.Tables["Categories"].DefaultView;
ListBox1.DataBind();
```

На рис. 7.8 показан результат работы этого примера.

***Рис. 7.8.** Привязка к информации из базы данных*

DataGrid

Управляющий элемент DataGrid делает еще один шаг вперед в отношении привязки данных, допуская отображение более одного свойства элемента привязки. Пример этого раздела показывает, как управлять привязкой столбцов данных к элементу управления DataGrid и как настраивать внешний вид элемента управления DataGrid с помощью стилей.

По умолчанию элемент управления DataGrid автоматически привязывает все столбцы источника данных в том порядке, в котором они получены из базы данных. Иногда требуется не это. Чтобы полностью управлять тем, какие столбцы и в каком порядке будут привязываться, отключите атрибут `autogeneratecolumns` объекта DataGrid и предоставьте свойство columns, как это показано в следующем примере:

```
<asp:DataGrid id=DataGrid1 runat="server"
      ForeColor="Black"
      autogeneratecolumns=false>

  <columns>
    <asp:boundcolumn datafield=CategoryID
                     headertext="ID" readonly=True/>
```

```
    <asp:boundcolumn datafield=CategoryName
                     headertext="Category" />
    <asp:boundcolumn datafield=Description
                     headertext="Description" />
  </columns>

  <SelectedItemStyle backcolor="#ffcc99" font-bold=True/>

  <AlternatingItemStyle BackColor="Gainsboro"/>

  <FooterStyle BackColor="Silver" ForeColor="White"/>

  <ItemStyle BackColor="White"/>

  <HeaderStyle BackColor="Navy" Font-Bold="True" ForeColor="White"/>

</asp:DataGrid>
```

На рис. 7.9 показан результат работы этого примера.

ID	Category	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
3	Confections	Desserts, candies, and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads, crackers, pasta, and cereal
6	Meat/Poultry	Prepared meats
7	Produce	Dried fruit and bean curd
8	Seafood	Seaweed and fish

Рис. 7.9. *Привязка данных к DataGrid*

Кроме тега `asp:boundcolumn`, применяемого для привязки столбца элемента управления DataGrid к столбцу источника данных, может также применяться тег `asp:buttoncolumn` для вставки столбца с кнопками, генерирующими сообщения. Эти сообщения можно обрабатывать, выполняя предопределенные действия, такие как выбор строки, ее удаление и добавление в корзину для покупок. У вас также может быть тег `asp:hyperlinkcolumn` для вставки в столбец ссылок на другие страницы, `asp:editcommandcolumn` для управления редактированием выбранных строк данных или `asp:templatecolumn` для настройки отображения столбцов данных.

Существует несколько стилей, применяемых для управления визуальным форматированием элемента управления DataGrid. HeaderStyle и FooterStyle, как следует из их названий, управляют стилем верхнего и нижнего колонтитула DataGrid. Элементы управления ItemStyle, AlternatingItemStyle, SelectedItemStyle и EditItemStyle используются для каждого типа строк списка. PagerStyle управляет внешним видом и расположением интерфейса переключения страниц.

Файл с исходным текстом обслуживающего кода для привязки данных к элементу управления DataGrid аналогичен предыдущему примеру. По существу, мы привязываем набор свойств DataSource объекта DataGrid к стандартному представлению (свойство DefaultView) таблицы Categories и выполняем привязку с помощью метода DataBind:

```
DataGrid1.DataSource = m_ds.Tables["Categories"].DefaultView;
DataGrid1.DataBind();
```

DataList

В отличие от элемента управления DataGrid, где привязываемые данные все равно помещаются в табличную форму, управляющий элемент DataList позволяет расположить данные в списке любым способом с помощью HTML-шаблонов.

В теге DataList можно настроить несколько шаблонов. Настраиваемые шаблоны включают:

AlternatingItemTemplate
EditItemTemplate
FooterTemplate
HeaderTemplate
ItemTemplate
SelectedItemTemplate
SeparatorTemplate

Для задания стиля каждого типа элементов, которые мы хотим отобразить, предназначены особые теги. Аналогично приведенному выше списку, имеются также теги ItemStyle, SelectedItemStyle и т. д.

В следующем примере мы только покажем вам один шаблон (ItemTemplate), который будет применяться ко всем элементам списка. В этом шаблоне мы используем синтаксис привязки веб-форм для привязки к двум свойствам элемента данных – полям CategoryID и CategoryName. В этом простом шаблоне данные из поля CategoryID всегда будут отображаться шрифтом Verdana высотой 10 пунктов.

Можно также управлять потоком данных в DataList, установив такие атрибуты, как `repeatcolumns`, `repeatdirection` (vertical, horizontal) или `repeatlayout` (flow, table):

```
<asp:DataList id=DataList1 runat="server"
    repeatcolumns=3
    repeatdirection=Horizontal>

  <AlternatingItemStyle BackColor="Gainsboro"/>
  <ItemTemplate>
    <font face=Verdana size=10>
      <%# DataBinder.Eval(Container.DataItem, "CategoryID") %>
```

```
      </font>
      <%# DataBinder.Eval(Container.DataItem, "CategoryName") %>
    </ItemTemplate>

</asp:DataList>
```

Обслуживающий привязку данных код приведен ниже:

```
DataList1.DataSource = m_ds.Tables["Categories"].DefaultView;
DataList1.DataBind();
```

На рис. 7.10 показан результат работы этого примера привязки данных к элементу управления DataList.

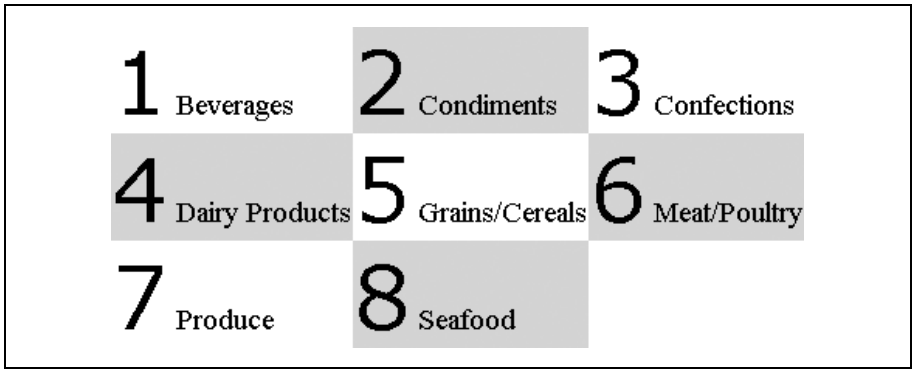


Рис. 7.10. *Привязка данных к элементу управления DataList с шаблоном*

Repeater

Элемент управления Repeater в ASP.NET полностью управляется HTML-шаблоном и по порядку выводит каждый из элементов данных, привязанных к нему. При генерации страницы элемент управления Repeater выполняет цикл по всем записям источника данных и генерирует HTML для отображения записи в соответствии с HTML-шаблоном. Он имеет свободную форму, насколько это позволяет привязка к данным. У вас могут быть шаблоны, генерирующие маркированные списки, нумерованные списки, списки, разделенные запятыми, и табуляции.

В элементе управления Repeater имеется всего пять шаблонов:

AlternatingItemTemplate
FooterTemplate
HeaderTemplate
ItemTemplate
SeparatorTemplate

Мы используем два из них для управления отображением элемента данных и разделителя.

Мы снова привязываем к шаблону два поля каждого элемента данных. Конечным результатом является разделенный запятыми список URL, ссылающихся на другую веб-форму, которая показывает более подробную информацию по выбранной категории. Как видите, для выполнения привязки мы также используем теги привязки данных веб-формы, <%# и %>. CategoryID служит в качестве параметра cat в строке запроса к фиктивной веб-форме *DisplayCategory.aspx*, а CategoryName выводится в виде текста тега <A>.

Также можно заменить тег <A> и запятую на графические изображения, чтобы сделать страницу визуально более привлекательной:

```
<asp:Repeater id=Repeater1 runat="server">

  <ItemTemplate>
    <A HREF="http://YourURL/DisplayCategory.aspx?cat=
       <%# DataBinder.Eval(Container.DataItem, "CategoryID") %>"
      ><%# DataBinder.Eval(Container.DataItem, "CategoryName") %>
    </A>
  </ItemTemplate>
  <SeparatorTemplate>, </SeparatorTemplate>

</asp:Repeater>
```

На рис. 7.11 показан результат применения элемента управления Repeater для привязки данных.

Beverages , Condiments , Confections , Dairy Products , Grains/Cereals , Meat/Poultry , Produce , Seafood

***Рис. 7.11.** Привязка данных с помощью элемента управления Repeater и шаблона*

Как и другие управляющие элементы, Repeater должен быть привязан к источнику данных:

```
Repeater1.DataSource = m_ds.Tables["Categories"].DefaultView;
Repeater1.DataBind();
```

Как видите, использование шаблонов для привязки данных к этим списочным элементам управления может быть очень простым и, тем не менее, мощным средством. Однако надо знать, как будет выглядеть сгенерированный HTML. Следует избегать сложных, излишне раздутых шаблонов, приводящих к созданию непривлекательных и больших файлов. В разработке веб-приложений размер страницы прямо пропорционален времени отклика.

Управление состоянием и масштабируемость

ASP.NET преодолевает все основные ограничения ASP, касающиеся управления состояниями сеансов. Как вы знаете из ASP-разработки,

состояние сеанса – это именованная переменная, кэшируемая на сервере на время сеанса веб-пользователя. Пока пользователь перемещается по веб-приложению, состояние сеанса сохраняет свое значение до истечения срока сеанса.

Управление состоянием сеанса в ASP происходит следующим образом:

- Сеанс начинается, и веб-приложение присваивает пользователю уникальный ключ.
- Этот ключ сохраняется в файлах cookie. При последующих вызовах клиентский броузер посылает уникальный ключ обратно на сервер.
- Сервер ищет состояния, сохраненные для данного ключа, и соответствующим образом обрабатывает запрос.

Данный механизм прекрасно работал все эти годы, но мы обнаружили, что имеется несколько ограничений, с которыми надо либо смириться, либо обходить их. Самое большое ограничение – это зависимость состояния сеанса от процесса, что делает невозможным реализацию его в среде «веб-фермы» без специального управления сеансом.

Управление состояниями сеансов в ASP.NET

ASP.NET улучшает управление состояниями сеансов по сравнению с ASP, перейдя к внепроцессной (out-of-process) модели. Все веб-серверы фермы указывают на общий сервер, на котором работает внепроцессный диспетчер состояний, и веб-клиент сможет перенаправляться в пределах веб-фермы, не теряя своих состояний сеанса.

Применяя внепроцессную модель, мы больше не сталкиваемся с проблемой потери состояний сеанса при зацикливании процесса IIS. Это значит, что если в приложении веб-сервера по какой-то причине возник сбой и он был перезапущен во время действия сеанса, все состояния сеансов веб-клиентов все равно останутся нетронутыми. Конечно, если произошел сбой во внепроцессном диспетчере состояний, это совсем другое дело. Из этого следует еще одно улучшение ASP.NET – возможность сохранять состояние сеанса в базе данных.

Идея сохранения состояния сеанса в базе данных не нова. Многие из нас реализовывали ее в качестве обходного пути при работе с веб-фермами. Однако ASP.NET облегчает эту задачу.

Как и все остальные настройки ASP.NET, управление сеансами осуществляется с помощью файлов *web.config*. Существует два уровня конфигурации: уровень машины и уровень приложения. Конфигурация на уровне машины связана с файлом *machine.config*, хранящимся в *WinNT\Microsoft.NET\Framework\<номер версии>\ CONFIG\machine.config*, а конфигурация на уровне приложения задана в файле *web.config* в корневом каталоге приложения. Конфигурация уровня приложения перекрывает конфигурацию на уровне машины.

Следующий код представляет собой часть файла *web.config*, относящуюся к управлению состоянием сеанса:[1]

```
<configuration>
  <system.web>
    <sessionState
      mode="InProc"
      cookieless="false"
      timeout="20" />
  </system.web>
</configuration>
```

В табл. 7.2 перечислены свойства класса SessionState.

Таблица 7.2. Свойства класса SessionState

Свойство	Описание
mode	`Off` указывает, что отслеживание состояния сеанса отключено; `InProc` – данные сеанса сохраняются локально; `StateServer` – данные сеанса хранятся на удаленном сервере; и `SQLServer` – данные сеанса сохраняются в базе данных SQL Server.
Cookieless	Указывает, полагаться ли на принятие cookie клиентом. Если это свойство установлено в `true`, ASP.NET вставляет уникальный ключ в URL для навигации между страницами приложения вместо задания его в cookie клиента.
Timeout	Указывает время ожидания в минутах для сеанса. Это плавающее временное окно, отсчет начинается заново после каждого запроса. По умолчанию равно 20 минутам.
stateConnectionString	Указывает сервер и порт удаленного сервера состояний сеанса (не SQL-сервера). Используется следующий формат: `tcpip=HOST:PORT`, например `tcpip=192.168.254.1:42424`. Используется, только если `mode=StateServer`.
sqlConnectionString	Задает строку соединения с SQL-сервером, например `user id=sa;password=;database=ASPState;server=(local)`. Это требуется в том случае, если `mode=SQLServer`.

Внепроцессное управление состоянием сеанса

Настраивая режим состояний сеанса на работу с удаленным сервером (`mode=StateServer`), необходимо подготовить удаленный сервер для автоматического запуска службы управления состояниями.

ASP.NET SDK включает службу NT под названием *ASP.NET State Service*, которую можно применять для внепроцессного управления со-

[1] Содержимое файла чувствительно к регистру.

стояниями сеансов. Перед настройкой своих файлов *web.config* для использования внепроцессного режима вы должны запустить службу ASP State в консоли NT Services Management Console. Можно изменить параметр *startup type*, сделав его равным `automatic`, чтобы эта служба запускалась автоматически при последующих перезагрузках.

Управление состоянием сеанса с помощью SQL Server

Для работы в этом режиме машина с SQL Server должна быть подготовлена. В ASP.NET SDK включен сценарий SQL для создания базы данных ASPState, в которой хранятся все состояния сеансов. Этот сценарий SQL (*InstallSqlState.sql*) можно найти в каталоге *%SystemRoot%\Microsoft.NET\Framework\НОМЕРПОСТРОЕНИЯ*. Чтобы применить сценарий к своему SQL-серверу, используйте утилиту с интерфейсом командной строки *osql.exe* или SQL Query Analyzer. Мы применяем последнюю утилиту, т. к. она позволяет изучить сценарий для лучшего понимания реализации этого режима управления сеансами. Вам потребуется остановить и перезапустить SQL Server, поскольку этот скрипт меняет базу данных `master` для запуска вспомогательной процедуры `ASPState_Startup` при запуске SQL Server.

Управление состоянием сеанса без cookie

При разработке ASP обычно вводится требование, предусматривающее настройку клиентских веб-броузеров на принятие cookie, так чтобы состояние сеансов можно было использовать так, как это предполагалось. Однако когда это требование не может быть выполнено, в частности в приложениях типа «бизнес-клиент» (B2C), разработчику приходится упаковывать идентификатор сеанса в URL в виде переменной строки запроса или в виде поля формы и управлять состояниями сеанса вручную.

В ASP.NET, как видно в разделе `sessionstate` файла конфигурации, требуется лишь установить значение параметра `cookieless` равным `true`, и все будет сделано автоматически. Состояние сеанса сможет быть использовано так, как будто ничего не изменилось.

Чтобы установить такую конфигурацию состояния сеанса и поэкспериментировать с ней, мы создали две фиктивных страницы ASP.NET: `login.aspx` и `main.aspx`. Главная страница перенаправляет пользователя на страницу регистрации, если он не был зарегистрирован. Страница регистрации перенаправляет пользователя на главную страницу, если он аутентифицирован. После регистрации пользователя заполняется переменная `UserName`.

Далее приведен исходный текст упрощенной страницы регистрации:

```
<HTML>

<script language="VB" runat="server">
Sub cmdLogin_Click(ByVal sender As System.Object, _
```

```
                    ByVal e As System.EventArgs)
  ' здесь дополнительная обработка
  Session("UserName") = txtUID.Text
  Response.Redirect("Main.aspx")
End Sub
</script>

<body>
<form id="Form1" method="post" runat="server">
<table>
  <tr>
    <td>User ID</td>
    <td><asp:TextBox id="txtUID"
                     runat="server"></asp:TextBox></td>
    </tr>
  <tr>
    <td>Пароль</td>
    <td><asp:TextBox id="txtPWD"
                     textmode="password"
                     runat="server">
                     </asp:TextBox></td>
    </tr>
  <tr>
    <td></td>
    <td><asp:Button id="cmdLogin"
                    runat="server"
                    Text="Login"
                    onclick="cmdLogin_Click">
                    </asp:Button></td>
    </tr>
</table>
</form>
</body>
</HTML>
```

Шаблон главной страницы выглядит следующим образом:

```
<HTML>

<script language="VB" runat="server">
Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
  If (Session("UserName") <> "") Then
    labelData.Text = "Добро пожаловать, " + Session("UserName")
  Else
    Response.Redirect("Login.aspx")
  End If
End Sub
</script>

<body>
<form id="Form1" method="post" runat="server">
  <asp:Label id="labelData" runat="server"></asp:Label>
</form>
```

```
</body>
</HTML>
```

В первом сценарии мы использовали режим состояния сеансов `InProc`. Состояние сеанса обрабатывается процессом IIS, поэтому если мы перезапустим веб-сервер с помощью команды `iisreset` и попытаемся обновить главную страницу, то будем перенаправлены на страницу регистрации.

Во втором сценарии мы меняем режим состояния сеанса на `StateServer` и запускаем службу ASP.NET Session State (например, командной строкой `net start aspnet_state`). Заметьте, что здесь мы запускаем службу Session State Service на той же машине, что и веб-сервер, даже если для большей надежности у нас есть такая служба, работающая на другом компьютере. В этот раз состояние сеанса сохраняется при перезапуске веб-сервера. Конечно, если мы перезапустим саму службу ASP.NET Session State, главная страница все равно перенаправит нас на страницу регистрации.

Теперь, рассмотрев внутрипроцессное и внепроцессное управление состоянием сеанса, в последнем сценарии, который мы опробуем, мы сохраним состояние сеанса в базе данных. Это достигается простой установкой атрибутов `mode` и `sqlConnectionString` узла `sessionState` в файле *web.config*. Конечно, мы запустили *InstallSqlState.sql* на SQL-сервере для генерации схемы и вспомогательных хранимых процедур, требуемых ASP.NET для сохранения состояния в базе данных. Однако результат будет аналогичен предыдущим попыткам. Поскольку данные сеанса хранятся в *tempdb*, они очищаются при перезапуске SQL Server. Кстати, не забудьте установить автоматический запуск SQL Server Agent, чтобы задание по очистке состояний сеансов могло быть выполнено корректно.

Производительность или масштабируемость и надежность?

Как мы уже говорили, в ASP.NET вводится внепроцессная модель управления состоянием сеанса, позволяющая создавать более масштабируемые решения, но за это приходится платить. Внепроцессные коммуникации имеют значительно худшую производительность по сравнению с внутрипроцессными, не говоря о сохранении состояний сеанса в базе данных. Необходимо взвесить преимущества каждого из различных режимов управления состоянием и найти наиболее подходящий для вашего приложения. В табл. 7.3 приводится общее описание различных режимов и их недостатков.

Таблица 7.3. Режимы коммуникации для управления состояниями сеанса

Режим	Описание
Внутрипроцессный	Этот режим дает наилучшую производительность. Он ненадежен, т. к. базируется в памяти. Он не масштабируем, поскольку базируется на процессе. Если вы устанавливаете веб-ферму, вам придется обеспечить, чтобы при последующих запросах клиент обращался к тому же серверу.
Внепроцессный	Фактор надежности еще под вопросом, т. к. этот режим все равно использует хранение в памяти. Однако поскольку состоянием сеанса управляет отдельный процесс, этот режим более надежен, чем внутрипроцессный режим. Из-за накладных коммуникационных расходов внепроцессный режим значительно медленнее, чем внутрипроцессный. Он масштабируем для применения в веб-фермах.
SQL Server	Этот режим обеспечивает наивысший уровень надежности за счет производительности. Он масштабируем для применения в веб-фермах.

Заключение

На протяжении всей главы мы представляли вам возможности ASP.NET и преимущества, которые дает эта среда при разработке веб-приложений. Эти преимущества включают новый и расширенный жизненный цикл веб-страницы, поддерживающий события, запускаемые со стороны клиентских броузеров, серверные элементы управления, сохраняющие свое состояние, разделение пользовательского интерфейса и обслуживающего кода, замену языков сценариев позднего связывания на строго типизированные компилируемые языки и новое, усовершенствованное управление состоянием сеанса, улучшающее масштабируемость.

Если вы пытаетесь охватить веб-парадигму, используя ASP.NET для веб-приложения вместе с веб-службами для интеграции между сайтами, вы на правильном пути. Однако не все приложения подходят для Интернета. Существует огромный рынок стандартных приложений, где разработка ведется на традиционных языках VB, C, C++, Java и т. д. и нет необходимости в ориентации на веб-технологии. В следующей главе мы рассматриваем компоненты Windows Forms, предназначенные для замены традиционных Windows-приложений.

8

Windows Forms

Если задача Microsoft .NET состоит в охвате Всемирной паутины, то что же произойдет с традиционными Windows-приложениями? Оказывается, среда .NET дает преимущества не только в разработке веб-приложений, но и улучшает способ построения стандартных приложений Windows. В этой главе мы дадим вам возможность понять, что такое Windows Forms, как использовать классы .NET для создания приложений на основе Windows Forms и как все-таки «охватить Всемирную паутину», создавая Windows-приложения.

Введение в Windows Forms

Те, кто разрабатывает Windows-приложения с начала 1990-х, возможно, использовали чистые функции Windows API, такие как RegisterClass, CreateWindow, ShowWindow, GetMessage, TranslateMessage и DispatchMessage. И в приложении наверняка была точка входа WinMain. Внутри этой функции выполнялась регистрация приложения в Windows, создавалось и отображалось окно и обрабатывались системные сообщения. Любое Windows-приложение должно иметь цикл обработки сообщений, в котором сообщения Windows собираются и передаются в функцию-обработчик сообщений, зарегистрированную с помощью функции RegisterClass. Значительную часть усилий программист затрачивал на обработку сообщений Windows, таких как WM_CREATE, WM_SIZE или WM_CLOSE, которые он создавал и передавал в систему с помощью PostMessage или SendMessage.

Классическая схема создания Windows-приложений утомительна и приводит к ошибкам. В результате поверх всех этих Windows API в качестве уровня абстракции строятся прикладные среды. Такие среды,

как Microsoft Foundation Class Library (MFC) и Active Template Library (ATL), были созданы для того, чтобы помочь разработчикам Windows-приложений больше сконцентрироваться на решении бизнес-задач, а не на том, как обрабатывать определенные сообщения Windows. Эти среды предоставляют каркас или шаблон для Windows-приложения. На разработчика же возлагается ответственность за реализацию бизнес-логики.

Хотя эти среды значительно облегчают разработку Windows-приложений, все равно иногда возникает необходимость опускаться до уровня Windows API, когда среда не предоставляет необходимых элементов управления. Эта ситуация приводит к противоречивости кода. Более того, существуют многочисленные среды, похожие на MFC и ATL, такие как Object Windows Library (OWL) от Borland, zApp от Rogue Wave, Windows-дополнения для Python, такие как Win32 Extensions или PythonWin GUI Extensions, Visual Basic и другие доморощенные среды, приносящие много огорчений разработчикам при переключении от одной среды к другой.

Windows Forms предоставляет унифицированную программную модель разработки стандартных Windows-приложений. В том, что касается уровня абстракции, они аналогичны базовому Windows API, но намного богаче и мощнее. Вместо того чтобы основываться на функциях подобно родному Windows API, Windows Forms предоставляет иерархию классов. Вместо вызова CreateWindows для любого типа элементов пользовательского интерфейса с помощью соответствующего класса создается определенный тип элемента управления. Вы можете думать, что MFC и другие среды уже предоставляют иерархию классов. Какие же иные преимущества, выделяющие эту среду среди других, дает Windows Forms? Ответ заключается в независимости новой среды от языка. Любой язык среды .NET может использовать коллекцию классов, составляющих объектную модель Windows Forms.

Те, кто разрабатывал Windows-приложения на C++ и Visual Basic, могут подумать, что было бы хорошо совместить мощь C++ с работой в интегрированной среде разработки, такой как VB. Это стало возможным с появлением Visual Studio.NET и Windows Forms. Windows Forms приносит в C#, Managed C++ и другие языки интегрированную среду разработки в стиле VB.

В текущей модели разработки Windows-приложений, если вы используете компоненты COM, DCOM или ActiveX, развертывание приложения требует обширной настройки. Вероятно вы, как минимум, воспользуетесь утилитой *regsvr32* для регистрации или отмены регистрации компонентов в реестре клиентской машины. Все эти задачи развертывания, связанные с установкой, в Microsoft .NET исключаются – в частности, за счет Windows Forms. Теперь для установки приложения надо лишь скопировать исполняемый файл на клиентскую машину.

Среда Windows Forms представляет собой часть глобальной схемы Microsoft .NET, она полностью поддерживает веб-службы, ADO.NET и классы .NET и интегрируется с ними. Windows Forms может выступать в качестве пользовательского интерфейса веб-приложения, если используются такие классы .NET, как HttpWebRequest и HttpWebResponse. Эти классы позволяют вашему приложению на основе Windows Forms взаимодействовать с веб-серверами. Помните, что приложения Windows Forms – это не всегда изолированные приложения.

Пространство имен System.Windows.Forms

В этом разделе мы опишем архитектуру Windows Forms и представим классы, составляющие пространство имен Windows Forms.

Архитектура Windows Forms довольно проста. Она описывается в терминах *элементов управления* (*controls*) и *контейнеров* (*containers*). Это аналогично модели JFC в Java, где имеются контейнерные типы классов – Panel, Window, JComponent и т. п. и классы элементов управления – Button, Checkbox, Label и т. п. Большинство классов пользовательского интерфейса в пространстве имен Windows.Forms являются производными от класса Control. По существу, все, что мы видим в приложении Windows Forms, – это элементы управления. Если элемент управления содержит другие элементы, то он представляет собой контейнер. Пользовательский интерфейс приложения состоит из объекта формы, выступающего в качестве главного контейнера, а также из элементов управления и других контейнеров, расположенных на форме.

Аналогично основным функциями базового Windows API, пространство имен System.Windows.Forms предоставляет общий набор классов, которые можно использовать и порождать для построения приложения Windows Forms. Классы и интерфейсы позволяют конструировать и отображать элементы пользовательского интерфейса на формах Windows.

Как мы видели в предыдущей главе, пространство имен System.Web.UI предоставляет классы для построения веб-приложений. Таким же образом пространство имен System.Windows.Forms предоставляет классы для построения стандартных приложений. Пространство имен System.Windows.Forms аналогично пространству имен System.Web.UI, описанному в предыдущей главе.

Аналогично классам Control и Page пространства имен System.Web.UI, классы Control и Form пространства имен System.Windows.Forms – это два самых важных класса.

Класс Control

Control – это базовый класс всех элементов пользовательского интерфейса в приложении Windows Forms. Он предоставляет общие для всех

элементов управления свойства, а также общее поведение элементов пользовательского интерфейса, например, прием пользовательского ввода с клавиатуры или мыши и генерация соответствующих событий.

В табл. 8.1 приведен список некоторых характерных свойств, методов и событий, с которыми вы, скорее всего, столкнетесь. Полный список можно найти в Microsoft .NET SDK.

Таблица 8.1. Основные свойства, методы и события класса Control

Свойство	**Описание**
Controls	Эти свойства позволяют конструировать иерархию управляющих элементов. Свойство Controls содержит список всех дочерних элементов управления, а свойство Parent указывает на родителя данного элемента управления.
Parent	
Enabled	Эти свойства контролируют визуальное состояние элементов управления.
Focused	
Visible	
Left	Эти свойства управляют расположением и размером элемента управления.
Top	
Right	
Bottom	
Width	
Height	
Size	
Метод	**Описание**
Show	Эти методы предназначены для контроля визуального состояния элемента управления.
Hide	
Focus	
Select	
Refresh	Эти методы определяют, когда и какая часть экрана требует перерисовки. Метод Refresh немедленно заставляет элемент управления перерисовать самого себя и все свои дочерние элементы. Методы Invalidate и Update осуществляют избирательное управление частью экрана, которая должна быть перерисована.
Invalidate	

Таблица 8.1 (продолжение)

Метод	Описание
Update	
ProcessCmdKey	Разрабатывая собственные элементы управления, переопределите эти методы для перехвата сообщений Windows. Это схоже с тем, как Windows-разработчики обрабатывали сообщения Windows, когда разрабатывали Win32-приложения с помощью базового Win32 API.
WndProc	
Событие	**Описание**
Click	Для обработки стандартных событий элемента управления вы скорее всего перекроете защищенные виртуальные методы, предоставленные классом Control. Эти виртуальные методы обработки событий мыши могут быть переопределены для осуществления специальной обработки.
MouseDown	
MouseUp	
MouseMove	
MouseWheel	
KeyDown	Аналогично событиям мыши, эти виртуальные методы событий клавиатуры также могут быть перекрыты.
KeyUp	
KeyPress	

Класс Control также предоставляет такие возможности, как привязка данных, контекстное меню, drag-and-drop, прикрепление (anchoring) и стыковка (docking), а также такие свойства, как шрифт, цвет, фон, курсор и т. д.

Класс Form

Форма (*form*) в Windows Forms по концепции аналогична *странице* (*page*) в Web Forms. Это контейнерный элемент управления, содержащий другие элементы пользовательского интерфейса. Свойства объекта Form позволяют управлять внешним видом, размером и цветом отображаемой формы. Windows Form – это, по существу, представление любого окна, отображаемого в вашем приложении.

Стандартная форма содержит заголовок, в котором имеются значок, текст заголовка и кнопки управления Minimize (Свернуть), Maximize (Развернуть) и Close (Закрыть) (рис. 8.1). В большинстве случаев форма также содержит меню, расположенное непосредственно под заголовком. Рабочая область формы – это место, где отображаются дочер-

ние управляющие элементы. Рамка вокруг всей формы показывает ее границы и позволяет менять размеры формы. Иногда форма также содержит полосы прокрутки, чтобы показать больше управляющих элементов или элементы большего размера, чем размер рабочей области формы.

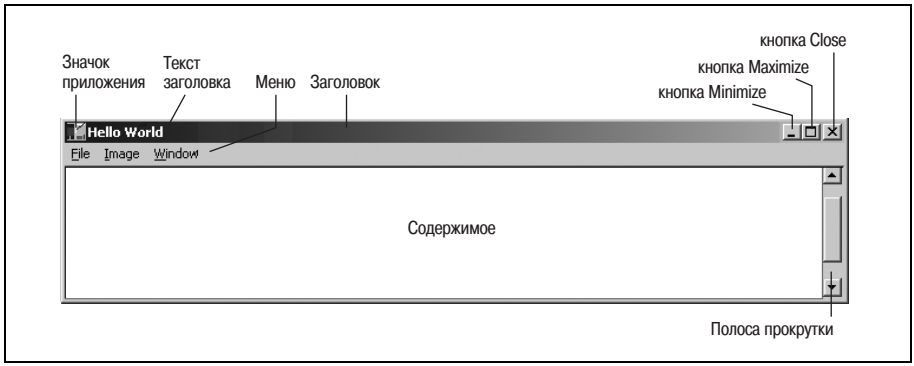


Рис. 8.1. *Пустое окно приложения*

Управление стандартными визуальными элементами формы обеспечивается такими свойствами, как Icon, Text, ControlBox, MinimizeBox, MaximizeBox и FormBorderStyle. Например, если вы хотите, чтобы в тексте заголовка было написано `Hello World`, включите оператор присваивания `formName.Text = "Hello World";`. Чтобы на форме не было кнопок в верхнем левом углу, вы можете установить свойство ControlBox в `false`. Если вы хотите выборочно отключить кнопки Maximize или Minimize, установите в `false` свойство MaximizeBox или MinimizeBox.

Можно назначить форме меню, задав свойство Menu формы с применением экземпляра класса MainMenu. Мы покажем, как это сделать, в разделе «Разработка Windows Forms» далее в этой главе.

Аналогично кнопкам Submit и Reset на веб-форме, стандартные формы часто включают кнопки OK и Cancel для подтверждения или отмены введенных в форму данных. В Windows Forms можно назначить любую кнопку свойству формы AcceptButton, сделав ее кнопкой по умолчанию, срабатывающей при нажатии пользователем клавиши <Enter>. Аналогично можно установить свойство CancelButton для обработки клавиши <Esc>.

Класс Form поддерживает несколько собственных методов в дополнение к методам, унаследованным от базового класса. Active, Show, Hide, ShowDialog и Close – вот несколько обязательных методов, используемых в любой форме для реализации функциональности управления окном. Далее в этой главе в разделе «Разработка Windows Forms» эти методы показаны в действии.

Расширение существующих управляющих элементов

Windows Forms API – это объектно-ориентированный API, поэтому расширение функциональности элементов управления осуществляется простым созданием производного класса от того элемента управления, который мы хотим расширить, добавив к нему методы, свойства и события или переопределив стандартное поведение элемента управления:

```
class MyCustomTextBox : TextBox
{
  // Здесь осуществляется настройка
}
```

Создание составных управляющих элементов

Составные элементы управления (*composite controls*) – это элементы, содержащие другие элементы. По определению они обязаны быть производными от класса ContainerControl; однако объектная модель Windows Forms предоставляет класс UserControl, который больше подходит в качестве отправной точки создания специального составного элемента управления (UserControl на самом деле является производным от ContainerControl):

```
class MyCustomComposite : UserControl
{
  // Здесь определяются составные управляющие элементы
}
```

Хотя создание специальных составных элементов управления, производных от класса UserControl, – несложная задача, Microsoft Visual Studio.NET прекрасно подойдет для еще большего облегчения этой задачи. Действительно непросто поднять планку для средств быстрой разработки. Подобные средства значительно повышают продуктивность разработчика.

Класс Application

Класс Application предоставляет статические методы для запуска, остановки или фильтрации сообщений Windows в приложении. Все приложения Windows Forms содержат ссылку на этот класс. Точнее, все приложения Windows Forms начинаются примерно со следующего:

```
System.Windows.Forms.Application.Run(new MyForm());
```

Хотя данный класс предоставляет и другие методы и свойства кроме метода Run, этот метод действительно единственный важный среди них. Остальные методы (перечисленные далее в этом разделе) – низкоуровневые и применяются не часто.

Метод Run запускает цикл обработки сообщений приложения. Этот метод имеет две сигнатуры. Первая не имеет параметров и обычно используется для приложений без графического интерфейса.

```
System.Windows.Forms.Application.Run();
```

Вторая сигнатура принимает в качестве параметра форму, что можно видеть в первом примере. Форма MyForm представляет собой точку входа приложения Windows Forms с графическим интерфейсом пользователя.

В табл. 8.2 приводится общее описание класса Application.

Таблица 8.2. Основные свойства и методы класса Application

Свойство	**Описание**
CommonAppDataRegistry	Это общий ключ реестра для приложения, в котором хранятся общие данные, разделяемые между всеми пользователями.
StartupPath	Это свойство обозначает маршрут к тому месту, откуда запускается приложение.
UserAppDataRegistry	Это ключ реестра, содержащий данные отдельных пользователей.
Метод	**Описание**
Run	Этот метод запускает приложение, независимо от того, имеет ли оно графический интерфейс.
Exit	Этот метод останавливает приложение путем отправки сообщения во все циклы обработки сообщений всех потоков приложения.
ExitThread	Аналогично предыдущему, этот метод останавливает текущий цикл обработки сообщений в текущем потоке.
AddMessageFilter	Можно также добавить фильтр сообщений в приложение для перехвата и фильтрации сообщений Windows.[a]
RemoveMessageFilter	Вы также можете удалить фильтр сообщений.
DoEvents	Этот метод обрабатывает все сообщения Windows, находящиеся в данный момент в очереди сообщений.

[a] Для этого метода надо указать лишь один параметр – объект, реализующий интерфейс IMessageFilter. В данный момент единственный метод этого интерфейса, который необходимо переопределить для перехвата и фильтрации любых сообщений, – PreFilterMessage. Если ваш метод PreFilterMessage возвращает `true`, сообщение Windows поглощается и не передается в свой пункт назначения. Сообщение можно пропустить, если этот метод возвращает `false`.

На рис. 8.2 показана иерархия элементов управления Windows в пространстве имен System.Windows.Forms. Эти элементы управления помещаются на форму для создания приложения Windows Forms и на контейнер UserControl для создания элементов интерфейса пользователя (аналогично современным элементам управления ActiveX). Эта схема не включает класс Application.

Разработка Windows Forms

Класс Form в пространстве имен System.Windows.Forms представляет стандартное окно, содержащее элементы управления Windows. В этом разделе мы пройдем по циклу разработки приложения Windows Forms и рассмотрим богатый набор элементов управления Windows, применение которых возможно в Windows Form.

Приложение Windows Forms

Все приложения Windows Forms начинаются с класса, производного от класса Windows.Forms.Form. Простейшее приложение Windows Forms выглядит следующим образом:

```
public class MyForm : System.Windows.Forms.Form
{
  public MyForm()
  {
    Text = "Hello World";
  }
  public static void Main()
  {
    System.Windows.Forms.Application.Run(new MyForm());
  }
}
```

По существу, здесь определяется класс MyForm, производный от класса System.Windows.Forms.Form. В конструкторе класса MyForm свойство Text формы устанавливается равным `Hello World`. Больше в нем ничего нет. Статическая функция Main – это точка входа всех приложений. В функции, являющейся точкой входа, вызывается статический метод Application.Run, запускающий цикл обработки сообщений для вашего приложения. Поскольку мы тоже передаем в метод Run, производный от Form, объект MyForm, у нас получается приложение Windows Forms.

Можно также включить ссылки на пространства имен, чтобы избежать ввода полных имен классов, таких как System.Windows.Forms.Form или System.Windows.Forms.Application. Для этого добавьте следующую строку в исходный текст и опустите префикс System.Windows.Forms в именах классов:

```
using System.Windows.Forms;
```

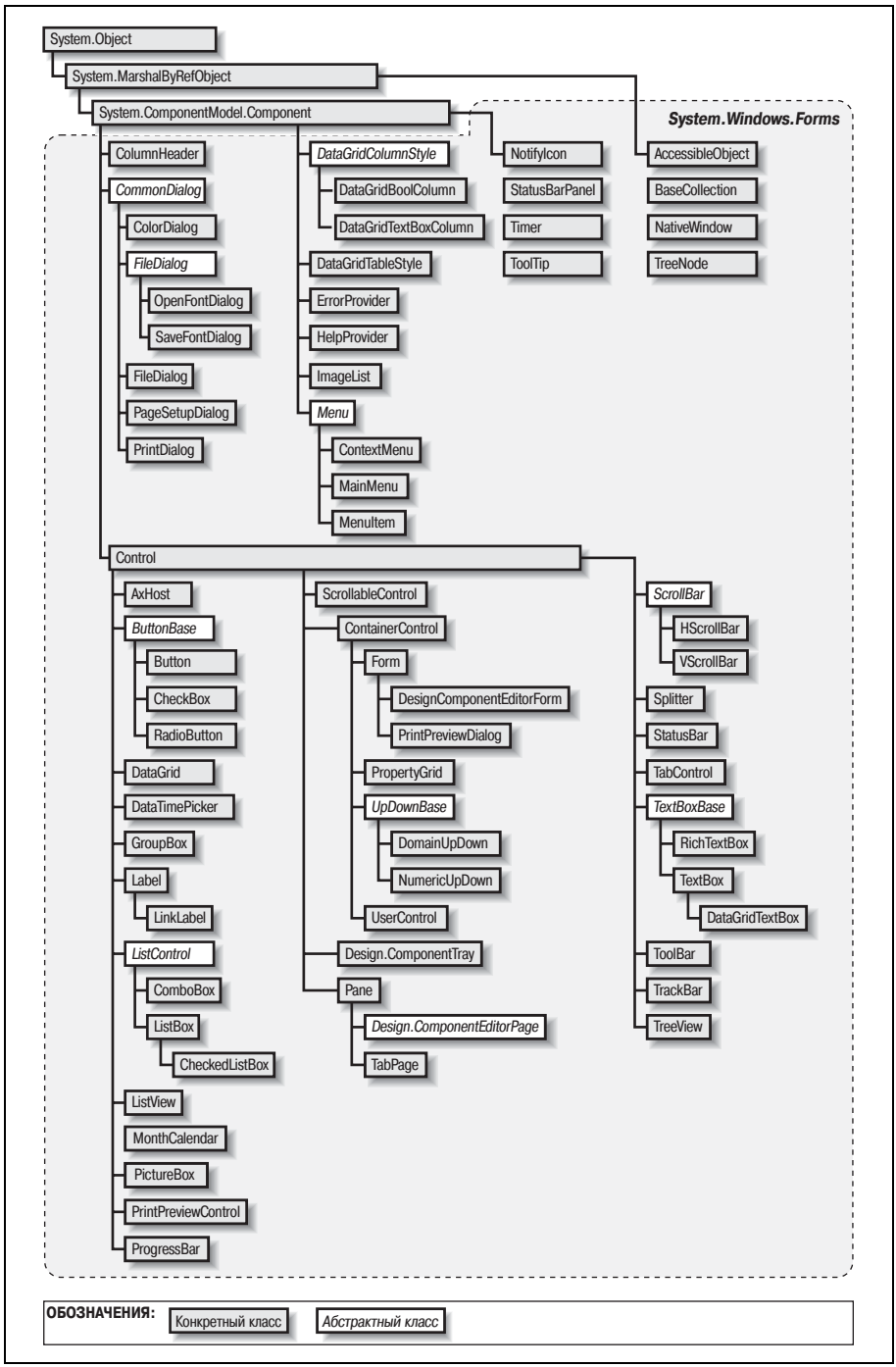


Рис. 8.2. *Иерархия управляющих элементов в System.Windows.Forms*

Для компиляции приведенного выше приложения мы применяем компилятор C# с интерфейсом командной строки. Заметьте, что создается исполняемый файл, а не DLL, как было в случае компиляции нашей веб-службы PubsWS (набирайте эту команду в одну строку):[1]

```
csc /t:winexe
    /r:system.dll
    /r:System.Windows.Forms.dll
    MyForm.cs
```

Стандартный объект Form, который будет отображен на экране, умеет не много; однако он демонстрирует простоту создания приложения Windows Forms. Из приложения можно выйти, нажав кнопку Close в заголовке формы. При этом в цикл обработки сообщений вводится сообщение о выходе, по умолчанию оно обрабатывается, и экземпляр приложения будет остановлен.

Элементы управления Windows

Приложение Windows Forms может включать в себя намного больше, чем приложение, показанное выше; однако базовые концепции остаются прежними. В данном разделе мы рассмотрим обширный набор элементов управления Windows, которые можно использовать в форме, а также привязку данных к некоторым из этих элементов управления. Кроме того, мы покажем, как работает обработка событий в приложениях Windows Forms.

Добавление элементов управления на форму

В первую очередь мы создадим элемент управления и добавим его в коллекцию Controls формы:

```
Button btn1 = new Button();
btn1.Text = "Нажми меня";
this.Controls.Add(btn1);
```

Добавление элементов управления других типов осуществляется в соответствии с тем же соглашением и состоит из трех базовых этапов:

1. Создание элемента управления.
2. Задание свойств элемента управления.
3. Добавление элемента управления в коллекцию Controls объекта Form.

[1] Файл примера можно также скомпилировать с помощью команды `csc MyForm.cs`, но лучше знать, как указывается тип создаваемого файла и ссылки, от которых зависит исходный текст.

Привязка обработчика событий

Это превосходно, но что же делает приложение, когда мы нажимаем кнопку? Ничего. Мы пока не присоединили обработчик к событию кнопки. Чтобы это сделать, сначала необходимо создать обработчик события. Обработчик – это обычная функция, но она всегда имеет два параметра: объект и EventArgs. Параметру объекта присваивается инициатор события. Например, если нажать кнопку формы, что приводит к возникновению события onclick, то параметр объекта в обработчике события будет указывать на объект кнопки, которую вы нажали. Объект EventArgs представляет собой само событие. В данном примере параметром EventArgs будет событие onclick с такими аргументами, как координаты указателя мыши, тип нажатой и т. д. В следующем фрагменте кода показан обработчик события нажатия кнопки:

```
void btn1_onclick(Object sender, EventArgs e)
{
  Text = "Отправитель: " + sender.ToString() + " - Событие: " + e.ToString();
}
```

Этот обработчик изменяет заголовок формы при каждом нажатии кнопки. Теперь, создав обработчик события, мы можем присоединить его к событию нажатия кнопки:

```
btn1.Click += new EventHandler(btn1_onclick);
```

Эта строка кода создает объект EventHandler на основе переданного в качестве аргумента события и передает созданный объект в событие Click кнопки. По существу, мы регистрируем функцию обратного вызова (callback), вызываемую при нажатии кнопки. (Возможно, кто-то захочет вернуться к главе 2, где мы обсуждали делегаты.) Ниже приводится пример целиком:

```
using System;
using System.Windows.Forms;

public class MyForm : Form
{

  void btn1_onclick(object sender, EventArgs e)
  {
    Text = "Отправитель: " + sender.ToString() +
           " - Событие: " + e.ToString();
  }

  public MyForm()
  {
    Text = "Hello World";

    Button btn1 = new Button();
    btn1.Text = "НажмиО меня";
```

```
    this.Controls.Add(btn1);

    btn1.Click += new EventHandler(btn1_onclick);
  }

  public static void Main()
  {
    Application.Run(new MyForm());
  }

}
```

Когда пользователь нажимает кнопку, вызывается наш обработчик события, т. к. мы уже зарегистрировали событие Click. К одному событию можно добавить несколько обработчиков, повторив строку присваивания для других обработчиков события. Все обработчики, зарегистрированные для обработки события, выполняются в том порядке, в каком они были зарегистрированы.

Можно также удалить обработчик. Замените += на -=:

```
btn1.Click -= new EventHandler(btn1_onclick);
```

Привязка обработчика к событию во время выполнения предоставляет разработчику неограниченную гибкость. Можно программно присоединять различные обработчики событий к элементу управления в зависимости от состояния приложения. Например, нажатие кнопки может быть привязано к *функции обновления*, если строка данных существует, или к *функции вставки*, когда создается новая строка.

Как видите, процесс привязки обработчиков к событиям в Windows Forms происходит так же, как и в Web Forms. Эта согласованность модели программирования, вероятно, обусловлена тем, что в обоих случаях в основе лежит CLR.

Привязка данных

Имеется два типа привязки данных к Windows Forms. Первый относится к простым элементам управления Windows, таким как Label, TextBox и Button. Эти элементы управления могут быть привязаны только к одиночному значению. Второй тип относится к элементам управления Windows, которые могут работать со списками данных, таким как ListBox, ComboBox и DataGrid. Эти списковые элементы могут быть привязаны к списку значений.

Давайте посмотрим на первый тип привязки. В следующем примере мы привязываем текстовые поля к полям таблицы из базы данных Pubs. Мы расширим простой пример приложения Windows Form *Hello, World* за счет включения доступа к данным и привязки данных.

Начнем с получения информации из базы данных. (Сейчас время обратиться к описанию ADO.NET в главе 5, если вы читаете эту книгу не по порядку.) Рассмотрим пример 8.1.

Пример 8.1. Исходный текст на C#

```
using System;
using System.Windows.Forms;
using System.Data;
using System.Data.OleDb;

public class MyForm : Form
{
  public static void Main()
  {
    Application.Run(new MyForm());
  }

  private TextBox m_txtFirstName, m_txtLastName, m_txtPhone;
  private Button m_btnPrev, m_btnNext;
  private CurrencyManager m_lm;
  private DataSet m_ds;

  public MyForm()
  {
    Text = "Simple Controls Data Binding";

    // Создание поля ввода имени
    m_txtFirstName = new TextBox();
    m_txtFirstName.Dock = DockStyle.Top;

    // Создание поля ввода фамилии
    m_txtLastName = new TextBox();
    m_txtLastName.Dock = DockStyle.Top;

    // Создание поля ввода телефона
    m_txtPhone = new TextBox();
    m_txtPhone.Dock = DockStyle.Top;

    // Добавляем поля имени и фамилии к панели panel1
    Panel panel1 = new Panel();
    panel1.Dock = DockStyle.Left;
    panel1.Controls.Add(m_txtFirstName);
    panel1.Controls.Add(m_txtLastName);
    panel1.Controls.Add(m_txtPhone);
    // Добавляем panel1 на левой стороне формы
    this.Controls.Add(panel1);

    // Создаем кнопку 'Up' и привязываем к ней обработчик события
    m_btnPrev = new Button();
    m_btnPrev.Text = "Up";
    m_btnPrev.Dock = DockStyle.Top;
    m_btnPrev.Click += new EventHandler(btnPrev_onclick);

    // Создаем кнопку 'Down' и привязываем обработчик события к ее нажатию
    m_btnNext = new Button();
    m_btnNext.Text = "Down";
    m_btnNext.Dock = DockStyle.Top;
    m_btnNext.Click += new EventHandler(btnNext_onclick);
```

```
    // Добавляем обе кнопки на panel2
    Panel panel2 = new Panel();
    panel2.Dock = DockStyle.Right;
    panel2.Width = 50;
    panel2.Controls.Add(m_btnNext);
    panel2.Controls.Add(m_btnPrev);
    // Добавляем panel2 на правую сторону формы
    this.Controls.Add(panel2);

    // Заполняем набор данных из таблицы authors базы данных Pubs
    m_ds = new DataSet();
    string oSQL = "select au_fname, au_lname, phone from authors";
    string oConnStr =
      "provider=sqloledb;server=(local);database=pubs;Integrated
Security=SSPI";
    OleDbDataAdapter oDA = new OleDbDataAdapter(oSQL, oConnStr);
    oDA.Fill(m_ds, "tbl");

    // Привязываем свойство Text поля ввода фамилии к полю au_lname
    // базы данных
    m_txtLastName.DataBindings.Add("Text",
                                   m_ds.Tables["tbl"],
                                   "au_lname");

   // Привязываем свойство Text поля ввода имени к полю au_fname базы данных
    m_txtFirstName.DataBindings.Add("Text",
                                    m_ds.Tables["tbl"],
                                    "au_fname");

   // Привязываем свойство Text поля ввода телефона к полю phone базы данных
    m_txtPhone.DataBindings.Add("Text",
                                m_ds.Tables["tbl"],
                                "phone");

    // Получаем диспетчер списка из контекста привязки

    m_lm = (CurrencyManager)this.BindingContext[m_ds.Tables["tbl"]];

  }

  protected void btnNext_onclick(object sender, EventArgs e)
  {
    // Перемещаем позицию в диспетчере списка
    m_lm.Position += 1;
  }
  protected void btnPrev_onclick(object sender, EventArgs e)
  {
    // Перемещаем позицию в диспетчере списка
    m_lm.Position -= 1;
  }
}
```

Элементы пользовательского интерфейса являются производными от класса Control и наследуют свойство DataBindings (имеющего тип

ControlsBindingCollection). Это свойство DataBindings содержит набор объектов Binding, применяемых для привязки любого свойства элемента управления к полю источника данных списка.

Для привязки простого элемента управления к записи из источника данных можно добавить объект Binding к коллекции DataBindings элемента управления с помощью следующего синтаксиса:

```
controlName.DataBindings.Add("Свойство", источник_данных, "имя_столбца");
```

где controlName – это имя простого элемента управления, к которому мы хотим осуществить привязку данных. Параметр Свойство определяет свойство простого элемента управления, которое мы хотим привязать к данным столбца, определяемого параметром имя_столбца.

Пример 8.1 показывает, как привязать свойство Text элемента управления TextBox m_txtLastName к столбцу au_lname таблицы Authors набора данных m_ds, а также элементов m_txtFirstName и m_txtPhone к столбцам au_fname и phone.

Для обхода списка в источнике данных мы будем использовать объект BindingManagerBase. Следующий фрагмент кода показывает, как получить диспетчер привязок для источника данных, привязанного к элементам управления формы. В нашем случае, поскольку данные представляют собой список, диспетчер привязок, возвращаемый BindingContext, – это CurrencyManager.[1]

```
// Получение диспетчера списка от контекста привязки

m_lm = (CurrencyManager)this.BindingContext[m_ds.Tables["tbl"]];
```

Чтобы продемонстрировать применение BindingManagerBase для обхода источника данных, мы добавили на форму две кнопки, btnNext и btnPrev. Затем мы привязываем события нажатия этих двух кнопок соответственно к обработчикам btnNext_onclick и btnPrev_onclick:

```
protected void btnNext_onclick(object sender, EventArgs e)
{
  m_lm.Position += 1;
}

protected void btnPrev_onclick(object sender, EventArgs e)
{
  m_lm.Position -= 1;
}
```

В случае применения BindingManagerBase для управления позицией в списке (в нашем случае – текущей записью в таблице Authors) элемен-

[1] Если источник данных возвращает только одно значение, BindingManagerBase указывает на объект типа PropertyManager. Когда источник данных возвращает список значений, объект имеет тип CurrencyManager.

ты управления TextBox будут обновлены для отображения новых значений. На рис. 8.3 показан пользовательский интерфейс для примера привязки данных к простым элементам управления.

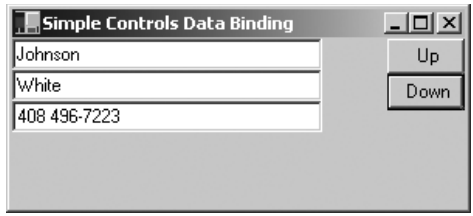


Рис. 8.3. *Привязка данных к простым элементам управления*

Теперь посмотрим на другой тип привязки данных. В этом примере мы будем привязывать всю таблицу Authors к элементу управления для табличного отображения данных data grid:

```
using System;
using System.Windows.Forms;
using System.Data;
using System.Data.OleDb;

public class MyForm : Form
{
  public static void Main()
  {
    Application.Run(new MyForm());
  }

  private Button m_btn1;
  private TextBox m_txt1;
  private DataGrid m_dataGrid1;

  public MyForm()
  {
    Text = "Hello World";

    m_txt1 = new TextBox();
    m_txt1.Text = "select * from authors";
    m_txt1.Dock = DockStyle.Top;
    this.Controls.Add(m_txt1);

    m_btn1 = new Button();
    m_btn1.Text = "Retrieve Data";
    m_btn1.Dock = DockStyle.Top;
    m_btn1.Click += new EventHandler(btn1_onclick);
    this.Controls.Add(m_btn1);

    m_dataGrid1 = new DataGrid();
    m_dataGrid1.Dock = DockStyle.Fill;
    this.Controls.Add(m_dataGrid1);
```

```
    this.AcceptButton = m_btn1;
  }

  protected void btn1_onclick(object sender, EventArgs e)
  {
    try {
      DataSet ds = new DataSet();
      string oConnStr =
        "provider=sqloledb;server=(local);database=pubs;Integrated
Security=SSPI";
      OleDbDataAdapter oDA =
        new OleDbDataAdapter(m_txt1.Text, oConnStr);
      oDA.Fill(ds, "tbl");

      /* Вы можете либо непосредственно указать таблицу:
       *
       *    m_dataGrid1.DataSource = ds.Tables["tbl"];
       *
       * либо задать отдельно источник данных и таблицу:
       */
      m_dataGrid1.DataSource = ds;
      m_dataGrid1.DataMember = "tbl";

    } catch(Exception ex) {
      MessageBox.Show("Произошла ошибка. " + ex.ToString());
    }
  }
}
```

Привязка данных к элементам управления типа List в Windows Forms и в Web Forms реализована аналогично. Однако вызывать метод DataBind элемента управления не надо. А надо лишь указать в свойстве DataSource элемента пользовательского интерфейса источник данных. Источник данных должен реализовывать интерфейс IEnumerable (или IList, реализующий IEnumerable). Выходит, что в качестве источника данных можно использовать сотни классов, в том числе DataTable, DataView, DataSet и все классы массивов и коллекций.

Процесс привязки данных к элементу управления DataGrid также прост: присвойте свойству DataSource объекта DataGrid нужный источник данных, и все. Мы указали таблицу `tbl` при добавлении ее к DataSet с помощью метода Fill() адаптера; таким образом, следующая часть кода просто обращается к коллекции таблиц объекта DataSet по имени таблицы:

```
m_dataGrid1.DataSource = ds.Tables["tbl"];
```

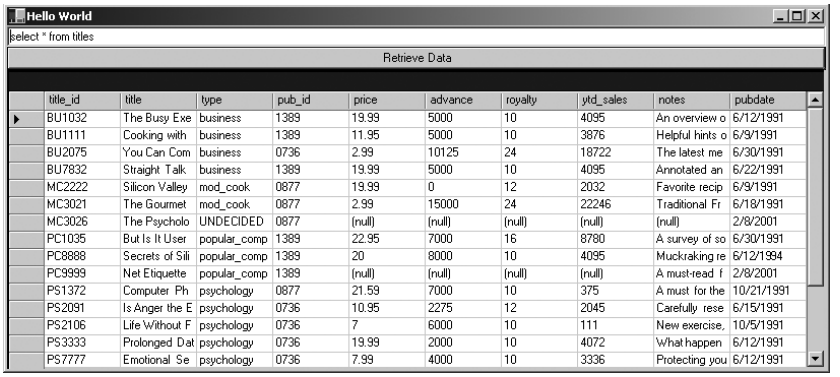
Если источник данных содержит более одной таблицы, то необходимо также задать имя таблицы, к которой требуется привязать элемент управления, в свойстве DataMember элемента управления:

```
m_dataGrid1.DataSource = ds;
m_dataGrid1.DataMember = "tbl";
```

Результат привязки двух таблиц к элементу управления DataGrid показан на рис. 8.4 и 8.5.

au_id	au_lname	au_fname	phone	address	city	state	zip	contract
172-32-1176	White	Johnson	408 496-7223	10932 Bigge	Menlo Park	CA	94025	☑
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #	Oakland	CA	94618	☑
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin L	Berkeley	CA	94705	☑
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland	San Jose	CA	95128	☑
274-80-9391	Straight	Dean	415 834-2919	5420 College	Oakland	CA	94609	☑
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi	Lawrence	KS	66044	☐
409-56-7008	Bennet	Abraham	415 658-9932	6223 Batema	Berkeley	CA	94705	☑
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde	Palo Alto	CA	94301	☑
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	95428	☑
486-29-1786	Locksley	Charlene	415 585-4620	18 Broadway	San Francisco	CA	94130	☑
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar H	Nashville	TN	37215	☐
648-92-1872	Blotchet-Halls	Reginald	503 745-6402	55 Hillsdale Bl	Corvallis	OR	97330	☑
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct.	Walnut Creek	CA	94595	☑
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl.	Ann Arbor	MI	48105	☑
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	46403	☑

Рис. 8.4. *Привязка таблицы authors к элементу управления DataGrid*



title_id	title	type	pub_id	price	advance	royalty	ytd_sales	notes	pubdate
BU1032	The Busy Exe	business	1389	19.99	5000	10	4095	An overview o	6/12/1991
BU1111	Cooking with	business	1389	11.95	5000	10	3876	Helpful hints o	6/9/1991
BU2075	You Can Com	business	0736	2.99	10125	24	18722	The latest me	6/30/1991
BU7832	Straight Talk	business	1389	19.99	5000	10	4095	Annotated an	6/22/1991
MC2222	Silicon Valley	mod_cook	0877	19.99	0	12	2032	Favorite recip	6/9/1991
MC3021	The Gourmet	mod_cook	0877	2.99	15000	24	22246	Traditional Fr	6/18/1991
MC3026	The Psycholo	UNDECIDED	0877	(null)	(null)	(null)	(null)	(null)	2/8/2001
PC1035	But Is It User	popular_comp	1389	22.95	7000	16	8780	A survey of so	6/30/1991
PC8888	Secrets of Sili	popular_comp	1389	20	8000	10	4095	Muckraking re	6/12/1994
PC9999	Net Etiquette	popular_comp	1389	(null)	(null)	(null)	(null)	A must-read f	2/8/2001
PS1372	Computer Ph	psychology	0877	21.59	7000	10	375	A must for the	10/21/1991
PS2091	Is Anger the E	psychology	0736	10.95	2275	12	2045	Carefully rese	6/15/1991
PS2106	Life Without F	psychology	0736	7	6000	10	111	New exercise,	10/5/1991
PS3333	Prolonged Dat	psychology	0736	19.99	2000	10	4072	What happen	6/12/1991
PS7777	Emotional Se	psychology	0736	7.99	4000	10	3336	Protecting you	6/12/1991

Рис. 8.5. *Привязка таблицы titles к элементу управления DataGrid*

Расстановка управляющих элементов

Элементы управления становятся полностью работоспособными после добавления их на форму, настройки обработки событий и привязки данных. Однако соображения оформления интерфейса приложения могут потребовать изменения расположения элементов управления на форме. Это можно сделать путем задания физических координат элемента управления относительно контейнера[1], к которому он принадле-

[1] Это аналогично программированию в VB. Элементы управления изначально имеют абсолютные позиции на форме, но они могут программно перемещаться и изменять размеры во время работы приложения.

жит, или использовать стыковку и прикрепление элементов управления внутри контейнера.

Стыковка (docking) элемента управления не представляет собой ничего сложного. Можно пристыковать элемент управления к верхнему, левому, правому или нижнему краю контейнера. Если пристыковать элемент управления к верхнему или нижнему краю, то ширина элемента управления будет равна ширине контейнера. Аналогично, если элемент управления пристыковать к левому или правому краю, то его высота будет равна высоте контейнера. Можно также установить свойство Dock равным `DockStyle.Fill`, заполнив весь контейнер данным элементом управления.

Прикрепление (anchoring) немного отличается от стыковки. Можно прикрепить элемент управления внутри контейнера к одной или нескольким сторонам последнего. Расстояние между стороной контейнера, к которой был прикреплен элемент, и элементом управления остается постоянным.

Кроме того, эти приемы можно комбинировать, группируя элементы управления на нескольких панелях и затем задавая расположение этих панелей на форме. В случае комбинирования стыковки и прикрепления нет необходимости программно рассчитывать или изменять размеры и положение управляющих элементов на форме.

Если вы когда-то имели дело с разработкой в Java Swing, то можете заметить, что текущая версия среды Microsoft .NET Windows Forms в отношении размещения элементов управления схожа с JFC, однако в ней отсутствуют такие классы Layout Manager, как GridLayout и FlowLayout, помогающие располагать элементы управления в контейнерах. Мы надеемся, что в будущих версиях .NET SDK будет включено что-то вроде Layout Manager. В данный момент разработка приложения Windows Forms в Visual Studio .NET обеспечивается более чем достаточным количеством средств управления расположением элементов управления на форме.

Визуальное наследование

Визуальное наследование никогда раньше не было возможно на платформе Windows с применением технологий Microsoft. До выхода Microsoft .NET (и здесь мы говорим только о разработке в VB) разработчики обеспечивали повторное использование форм посредством шаблонов VB. По существу, это причудливое название для программирования в стиле «cut-and-paste». Когда меняется сам шаблон, его копии и производные не изменяются. Приходится либо повторно вырезать и вставлять шаблон, либо просто оставлять все как есть.

После появления Microsoft .NET, где все стало объектно-ориентированным, можно создавать производные классы путем наследования любого базового класса. Поскольку форма в приложении Windows

Forms – это класс, производный от базового класса Form, фактически можно создать свой класс формы как производный от других классов форм.

Это особенно удобно для мастеров (wizards), где каждая форма похожа на другие. Можно создать форму общего вида в качестве базового класса и затем создавать каждую из форм, наследуя от этого базового класса.

MDI-приложения

Имеется два основных стиля пользовательского интерфейса приложений Windows – однодокументный интерфейс (Single Document Interface, SDI) и многодокументный интерфейс (Multiple Document Interface, MDI).[1] В SDI-приложениях каждый экземпляр программы может иметь только один документ. Для того чтобы открыть несколько документов, необходимо запустить несколько экземпляров программы. Интерфейс же MDI, напротив, позволяет иметь несколько одновременно открытых документов в одном экземпляре приложения. Еще одно достоинство MDI-приложения состоит в том, что, в зависимости от типа открытого в данный момент документа, главное меню приложения меняется, отражая операции, которые можно выполнять над документом.

В архитектуре Windows Forms легко разрабатываются как SDI- так и MDI-приложения, но в этом разделе мы приведем только пример MDI-приложения.

Архитектура MDI-приложения основывается на общей схеме архитектуры Windows Forms. По существу, имеется одна форма, действующая как форма-контейнер, и другие формы, действующие как дочерние.

Класс Form предоставляет несколько свойств и методов, облегчающих разработку MDI-приложений, включая IsMdiContainer, IsMdiChild, MdiParent, MdiChildren, ActiveMdiChild и LayoutMdi().

Сначала рассмотрим минимальную главную форму для MDI-приложения:

```
using System;
using System.Windows.Forms;

public class MdiMainForm : Form
{
  public MdiMainForm()
  {
    this.Text = "MDI App for Text and Images";
    // Это MDI-контейнер
    this.IsMdiContainer = true;
  }
```

[1] Другие стили – это Explorer, Wizard и т. д., но мы не будем обсуждать их все в этой книге.

```
  public static void Main(string[] args)
  {
    Application.Run(new MdiMainForm());
  }
}
```

Хотите верьте, хотите нет, но это практически все, что надо сделать для создания главной формы MDI-приложения! Для каждой из дочерних форм, порожденных главной формой, мы установим свойство MdiParent так, чтобы оно указывало на главную форму.

В следующем фрагменте кода мы загружаем дочерние формы:

```
...
  Form a = new Form();
  a.MdiParent = this;
  a.Show();

  Form b = new Form();
  b.MdiParent = this;
  b.Show();
...
```

Для порождения дочерней формы MDI-приложения требуется установить лишь одно свойство – MdiParent. В своем приложении вы замените типы форм `a` и `b` на собственные классы форм. (Как показано далее в этой главе, у нас это будут ImageForm и TextForm.)

Кроме того, MDI-приложения интересны тем, что в них имеется один набор главных меню, и дочерние формы могут присоединять свои меню к общему меню MDI-приложения. Мы также покажем, как задавать меню для главной MDI-формы, а далее в этом разделе – как меню дочерней формы объединяется с этим главным меню.

Вся архитектура меню в приложении Windows Forms вращается вокруг двух классов:[1] MainMenu и MenuItem. MainMenu представляет меню для формы в целом, а MenuItem обозначает один пункт меню; однако каждый пункт меню содержит дочерние пункты меню в свойстве MenuItems. Здесь мы снова видим шаблон из *элементов управления* и *контейнеров*. Так, для того чтобы иметь два меню верхнего уровня (например, File и Window), надо настроить объект MainMenu таким образом, чтобы он содержал в своем свойстве MenuItems два пункта меню. Это можно сделать с помощью метода Add свойства MenuItems, динамически вставив пункты меню в коллекцию. Если количество пунктов меню известно заранее, то можно декларативно присвоить этому свойству массив пунктов меню. В свою очередь, пункты меню File и Window могут иметь в своих свойствах MenuItems несколько пунктов подменю, аналогично конфигурации главного меню.

[1] Третий класс – это ContextMenu, но в рамках этой книги мы его обсуждать не будем.

Рассмотрим исходный текст:

```
using System;
using System.Windows.Forms;

public class MdiMainForm : Form
{
  // Пункты меню File
  private MenuItem mnuOpen, mnuClose, mnuExit;
  // Пункты меню Window
  private MenuItem mnuCascade, mnuTileHorz, mnuTileVert,
                   mnuSeparator, mnuCloseAll, mnuListMDI;
  // Меню File и Window
  private MenuItem mnuFile, mnuWindow;
  // Главное меню
  private MainMenu mnuMain;

  public MdiMainForm()
  {
    this.Text = "MDI App for Text and Images ";

    // Пункт меню File
    mnuFile = new MenuItem();
    mnuFile.Text = "&File";
    mnuFile.MergeOrder = 0;
    // Пункт меню Window
    mnuWindow = new MenuItem();
    mnuWindow.MergeOrder = 2;
    mnuWindow.Text = "&Window";

    // Главное меню содержит пункты File и Window
    mnuMain = new MainMenu();
    mnuMain.MenuItems.AddRange(
        new MenuItem[2] {mnuFile, mnuWindow});

    // Присваивание главного меню форме
    this.Menu = mnuMain;

    // Пункты меню в подменю Файл
    mnuOpen = new MenuItem();
    mnuOpen.Text = "Open";
    mnuOpen.Click += new EventHandler(this.OpenHandler);
    mnuClose = new MenuItem();
    mnuClose.Text = "Close";
    mnuClose.Click += new EventHandler(this.CloseHandler);
    mnuExit = new MenuItem();
    mnuExit.Text = "Exit";
    mnuExit.Click += new EventHandler(this.ExitHandler);
    mnuFile.MenuItems.AddRange(
        new MenuItem[3] {mnuOpen, mnuClose, mnuExit});

    // Пункты меню в подменю Window
    mnuCascade = new MenuItem();
    mnuCascade.Text = "Cascade";
    mnuCascade.Click += new EventHandler(this.CascadeHandler);
```

```
    mnuTileHorz = new MenuItem();
    mnuTileHorz.Text = "Tile Horizontally";
    mnuTileHorz.Click += new EventHandler(this.TileHorzHandler);
    mnuTileVert = new MenuItem();
    mnuTileVert.Text = "Tile Vertically";
    mnuTileVert.Click += new EventHandler(this.TileVertHandler);
    mnuSeparator = new MenuItem();
    mnuSeparator.Text = "-";
    mnuCloseAll = new MenuItem();
    mnuCloseAll.Text = "Close All";
    mnuCloseAll.Click += new EventHandler(this.CloseAllHandler);
    mnuListMDI = new MenuItem();
    mnuListMDI.Text = "Windows...";
    mnuListMDI.MdiList = true;

    mnuWindow.MenuItems.AddRange(
        new MenuItem[6] {mnuCascade, mnuTileHorz, mnuTileVert,
                         mnuSeparator, mnuCloseAll, mnuListMDI});

    // Это MDI-контейнер
    this.IsMdiContainer = true;
  }
  public static void Main(string[] args)
  {
    Application.Run(new MdiMainForm());
  }
...
}
```

(Заметьте, что этот листинг должен завершаться обработчиками событий, исходный текст которых приведен ниже.)

Сначала мы объявляем все пункты меню, которые нам требуются, а также один экземпляр MainMenu в пределах класса. В основном конструкторе приложения мы затем создаем пункты меню и устанавливаем их свойства Text. Для двух пунктов меню верхнего уровня мы также устанавливаем свойство MergeOrder, чтобы мы могли определять, где именно дочерние формы будут вставлять свои меню в меню главной формы. В нашем случае мы устанавливаем для меню File порядковый номер 0, а для меню Window – номер 2. Как вы увидите позднее, для дочернего меню мы установим MergeOrder равным 1, чтобы оно оказалось между меню File и Window.

Затем мы добавляем меню File и Window к коллекции MenuItems главного меню с помощью метода AddRange():

```
mnuMain.MenuItems.AddRange(
        new MenuItem[2] {mnuFile, mnuWindow});
```

Заметьте, что на этом этапе меню File и Window все еще пусты. Затем мы присваиваем mnuMain свойству MainMenu объекта Form. Теперь мы можем увидеть на главной форме меню File и Window; однако выпадающих меню еще нет.

Так же, как мы создали пункты меню и добавили их к коллекции MenuItems главного меню, мы добавляем пункты к меню File и Window. Однако здесь есть одно отличие. Мы также привязываем обработчики к событиям Click пунктов меню. Возьмем для примера пункт меню Open:

```
mnuOpen = new MenuItem();
mnuOpen.Text = "Open";
mnuOpen.Click += new EventHandler(this.OpenHandler);
```

Заметьте, что синтаксис привязки обработчика OpenHandler к событию Click класса MenuItem схож с привязкой любых других обработчиков событий, которые мы уже видели. Конечно, мы должны предоставить тело функции в главном классе MDI.

Пока мы говорим о меню, отметим еще один интересный момент – это пункт меню mnuListMdi в конце меню Window. Мы установили свойство MdiList этого экземпляра MenuItem равным `true`, как это показано в следующем фрагменте кода, поэтому данный пункт меню будет автоматически показывать все открытые внутри MDI-приложения документы.

```
mnuListMDI.Text = "Windows...";
mnuListMDI.MdiList = true;
```

На рис. 8.6 показан пример реализации этой возможности во время выполнения приложения.

Рис. 8.6. *Автоматически сгенерированные пункты меню MdiList*

Следующий код – это обработчики событий, которые мы привяжем к различным пунктам меню главной формы (и этим завершим листинг класса MdiMainForm):

```
protected void OpenHandler(object sender, EventArgs e)
{
  //MessageBox.Show("Выбран пункт Open");
  OpenFileDialog openFileDlg = new OpenFileDialog();
  if(openFileDlg.ShowDialog() == DialogResult.OK)
  {
```

```
      try
      {
        String sFN = openFileDlg.FileName;
        String sExt = sFN.Substring(sFN.LastIndexOf("."));
        sExt = sExt.ToUpper();
        //MessageBox.Show(sFN + " " + sExt);
        if(sExt == ".BMP" || sExt == ".JPG" || sExt == ".GIF")
        {
          ImageForm imgForm = new ImageForm();
          imgForm.SetFileName(sFN);
          imgForm.MdiParent = this;
          imgForm.Show();
        }
        else if(sExt == ".TXT" || sExt == ".VB" || sExt == ".CS")
        {
          TextForm txtForm = new TextForm();
          txtForm.SetFileName(sFN);
          txtForm.MdiParent = this;
          txtForm.Show();
        }
        else
        {
          MessageBox.Show("Формат файла не поддерживается.");
        }
      }
      catch(Exception ex)
      {
        MessageBox.Show ("Ошибка: " + ex.ToString());
      }
    }
  }

  protected void CloseHandler(object sender, EventArgs e)
  {
     if(this.ActiveMdiChild != null)
    {
      this.ActiveMdiChild.Close();
    }
  }

  protected void ExitHandler(object sender, EventArgs e)
  {
    this.Close();
  }

  protected void CascadeHandler(object sender, EventArgs e)
  {
    this.LayoutMdi(MdiLayout.Cascade);
  }

  protected void TileHorzHandler(object sender, EventArgs e)
  {
    this.LayoutMdi(MdiLayout.TileHorizontal);
  }
```

```
protected void TileVertHandler(object sender, EventArgs e)
{
  this.LayoutMdi(MdiLayout.TileVertical);
}

protected void CloseAllHandler(object sender, EventArgs e)
{
  int iLength = MdiChildren.Length;
  for(int i=0; i<iLength; i++)
  {
    MdiChildren[0].Dispose();
  }
}
```

Функциональность обработчика события OpenHandler проста. По существу, мы открываем стандартное диалоговое окно, позволяющее пользователю выбрать файл. Для простоты мы будем поддерживать три формата изображений (BMP, GIF и JPG) и три расширения текстовых файлов (TXT, CS и VB). Если пользователь выбирает файл с изображением, мы открываем в качестве дочерней формы MDI-приложения ImageForm. Если выбран файл с текстом, мы останавливаемся на классе TextForm. Исходный текст ImageForm и TextForm приведен ниже.

Чтобы разместить дочерние формы, мы применяем метод LayoutMdi класса Form. Этот метод принимает в качестве параметра перечисляемое значение типа MdiLayout. Возможные значения – Cascade, ArrangeIcons, TileHorizontal и TileVertical.

Форма также поддерживает свойство ActiveMdiChild, указывающее на активную дочернюю MDI-форму. Мы используем эту информацию для обработки пункта меню File → Close, закрывающего выбранную в данный момент дочернюю MDI-форму.

Чтобы обработать выбор пункта меню Close All, мы используем цикл для перебора коллекции всех дочерних MDI-форм и их закрытия.

Ниже приведен исходный текст класса ImageForm:

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class ImageForm : System.Windows.Forms.Form
{
  private MenuItem mnuImageItem;
  private MenuItem mnuImage;
  private MainMenu mnuMain;

  private Bitmap m_bmp;

  public ImageForm()
  {

    mnuImageItem = new MenuItem();
    mnuImageItem.Text = "Image Manipulation";
    mnuImageItem.Click += new EventHandler(this.HandleImageItem);
```

```
        mnuImage = new MenuItem();
        mnuImage.Text = "&Image";
        mnuImage.MergeOrder = 1;    // Вставляем меню между File и Window
        mnuImage.MenuItems.AddRange(new MenuItem[1] {mnuImageItem});

        mnuMain = new MainMenu();
        mnuMain.MenuItems.AddRange( new MenuItem[1] {mnuImage});
        this.Menu = mnuMain;
      }

      public void SetFileName(String sImageName)
      {
        try
        {
          m_bmp = new Bitmap(sImageName);
          Invalidate();
          this.Text = "IMAGE: " + sImageName;
        }
        catch(Exception ex)
        {
          MessageBox.Show ("Ошибка: " + ex.ToString());
        }
      }

      protected override void OnPaint(PaintEventArgs e)
      {
        if(m_bmp != null)
        {
          Graphics g = e.Graphics;
          g.DrawImage(m_bmp, 0, 0, m_bmp.Width, m_bmp.Height);
        }
      }

      protected void HandleImageItem(object sender, EventArgs e)
      {
        MessageBox.Show("Обработка изображения.");
      }
    }
```

Класс ImageForm должен рисовать изображение из файла, поэтому мы включили ссылку на пространство имен System.Drawing. Для вывода изображения на форму мы используем классы Bitmap и Graphics. Сначала мы получаем имя входного файла и конструируем объект Bitmap из содержимого этого файла. Затем мы объявляем недействительной область экрана, чтобы она была перерисована. В перекрытом методе OnPaint мы получаем указатель на объект Graphics и запрашиваем от него перерисовку объекта Bitmap на экране.

Мы также хотим обратить ваше внимание на то, что свойство MergeOrder меню Image равно 1. Мы сделали это, чтобы продемонстрировать функциональность объединения меню в MDI-приложениях. Когда отображается форма, в главном меню MDI-приложения отображаются пункты File, Image и Window.

Чтобы завершить разработку примера, приведем исходный текст класса TextForm:

```
using System;
using System.Windows.Forms;
using System.IO;

public class TextForm : Form
{
  private MenuItem mnuTextItem;
  private MenuItem mnuText;
  private MainMenu mnuMain;

  private TextBox textBox1;

  public TextForm()
  {
    mnuTextItem = new MenuItem();
    mnuTextItem.Text = "Text Manipulation";
    mnuTextItem.Click += new EventHandler(this.HandleTextItem);

    mnuText = new MenuItem();
    mnuText.Text = "&Text";
    mnuText.MergeOrder = 1;    // вставляем после меню File и перед меню Window
    mnuText.MenuItems.AddRange(new MenuItem[1] {mnuTextItem});

    mnuMain = new MainMenu();
    mnuMain.MenuItems.AddRange(new MenuItem[1] {mnuText});
    this.Menu = mnuMain;

    textBox1 = new TextBox();
    textBox1.Multiline = true;
    textBox1.Dock = System.Windows.Forms.DockStyle.Fill;
    this.Controls.Add (this.textBox1);
  }

  public void SetFileName(String sFileName)
  {
    StreamReader reader = File.OpenText(sFileName);
    textBox1.Text = reader.ReadToEnd();
    reader.Close();
    textBox1.SelectionLength = 0;
    this.Text = "TEXT: " + sFileName;
  }

  protected void HandleTextItem(object sender, EventArgs e)
  {
    MessageBox.Show("Обработка текстового файла.");
  }

}
```

Аналогично классу ImageForm, класс TextForm также вставляет свое меню между меню File и Window. Когда TextForm становится активной дочерней MDI-формой, в меню MDI-приложения выводятся пункты File, Text и Window. Это объединение меню выполняется автоматически. Мы должны лишь установить свойства MergeOrder для пунктов меню.

Для функционирования TextForm у нас имеется простой объект TextBox. Мы устанавливаем свойство Multiline в `true` для имитации простого текстового редактора и свойство стыковки на заполнение всей формы. Когда главная форма передает имя текстового файла этой форме, мы читаем входной файл и помещаем его содержимое в текстовое поле.

На рис. 8.7 показано окно запущенного MDI-приложения. В этом экземпляре приложения одновременно открыто три окна TextForm и три окна ImageForm.

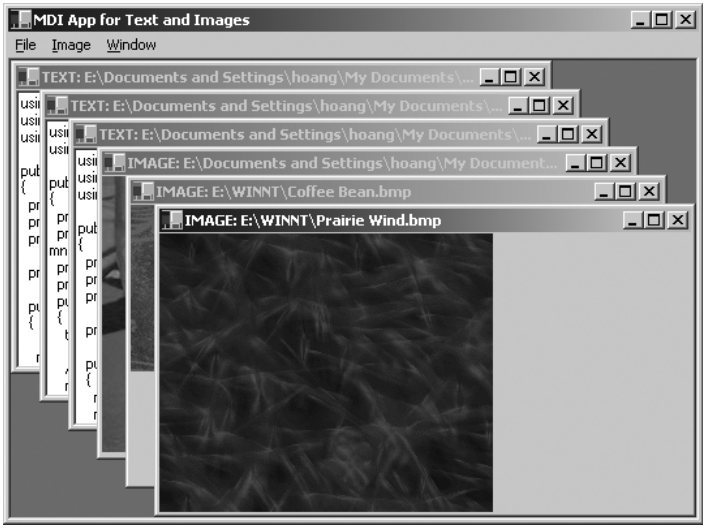


Рис. 8.7. *MDI-приложение*

Следующий сценарий предназначен для построения этого MDI-приложения. Как можно видеть, параметр, определяющий тип создаваемого файла, установлен в `winexe`, указывая, что результатом компиляции будет исполняемый файл, а задание типа `library` приведет к созданию DLL. Поскольку мы используем графический пакет для показа изображения, мы также должны добавить ссылку на сборку `System.drawing.dll`. В этом приложении у нас имеется три формы: главная, названная MDIApp, и две дочерние формы MDI – ImageForm и TextForm (убедитесь, что вы набираете эту команду в одну строку).[1]

```
csc /t:winexe
    /r:System.Windows.Forms.dll
    /r:system.drawing.dll
    MDIApp.cs
    ImageForm.cs
    TextForm.cs
```

[1] Можно также скомпилировать исполняемый файл без указания типа создаваемого файла и ссылок.

Частичное развертывание

Представьте MDI-приложение из предыдущего примера, поддерживающее сотни различных типов файлов. Для каждого типа файлов у нас, вероятно, будет класс, аналогичный классам ImageForm и TextForm. Неправда ли, было бы неплохо, если вместо компиляции и компоновки этих классов в приложение можно было бы загружать и устанавливать класс «на лету», после выбора соответствующего типа файла? В этом разделе мы покажем, как преобразовать предыдущее MDI-приложение, чтобы это было возможно.

Идея состоит в том, что главный исполняемый файл будет действовать в качестве управляющего модуля. Он будет первой сборкой, загруженной на клиентскую машину. Как только пользователь выберет для открытия определенный тип файла, этот модуль определит, какие вспомогательные DLL должны быть загружены для обработки этого типа файла, загрузит и установит DLL и затем попросит класс из загруженной сборки отобразить выбранный файл.

Чтобы управляющий модуль мог взаимодействовать со всеми вспомогательными классами, мы выделим общее из ImageForm и TextForm для создания абстрактного класса BaseForm.[1]

```
using System;
using System.Windows.Forms;
using System.Security;
[assembly:AllowPartiallyTrustedCallers]
namespace BaseForm
{
     abstract public class BaseForm : Form
     {
         abstract public void SetFileName(String sFileName);
     }
}
```

Мы перепишем классы ImageForm и TextForm, чтобы они были наследниками BaseForm. Любая другая форма, которую вам потребуется написать для других типов файлов, должна также наследоваться от BaseForm. Тогда управляющий модуль должен лишь знать, как взаимодействовать с BaseForm, и этого достаточно.

Далее последуют изменения в исходном файле *ImageForm.cs*. Здесь мы только заключаем класс в пространство имен под названием ImageForm, делаем класс ImageForm наследником BaseForm и предоставля-

[1] Поскольку далее мы будем демонстрировать запуск *MDIApp.exe* из броузера, в библиотеке *BaseForm.dll* должно быть явно разрешено ее использование из *MDIApp.exe* с помощью атрибута AllowPartiallyTrustedCallers. Это требуется для того, чтобы учесть изменения требований к безопасности в версии 1 Microsoft .NET Framework.

ем реализацию абстрактного метода SetFileName(). Оставшаяся часть кода совпадает с исходной версией *ImageForm.cs*.

```
//...
namespace ImageForm
{
  public class ImageForm : BaseForm.BaseForm
  {
    //...
    override public void SetFileName(String sImageName)
    {
      //...
    }
    //...
  }
}
```

То же самое, что и с *ImageForm.cs*, мы проделываем с *TextForm.cs*. Все эти изменения приведены ниже.

```
//...
namespace TextForm
{
  public class TextForm : BaseForm.BaseForm
  {
    //...
    override public void SetFileName(String sFileName)
    {
      //...
    }
    //...
  }
}
```

Теперь вместо компиляции MDIApp, сборки его вместе с *ImageForm.cs* и *TextForm.cs* и непосредственного применения классов ImageForm и TextForm в функции OpenHandler() мы будем использовать пространство имен Reflection для загрузки на лету каждой сборки с заранее заданного URL. Замените старый код обработки сообщений в OpenHandler():

```
ImageForm imgForm = new ImageForm();
imgForm.SetFileName(sFN);
```

на:

```
Assembly ass = Assembly.LoadFrom("http://localhost/MDIDLLS/ImageForm.dll");
BaseForm.BaseForm imgForm =
          (BaseForm.BaseForm) ass.CreateInstance("ImageForm.ImageForm");
imgForm.SetFileName(sFN);
```

Три строки кода, по существу, делают следующее:

1. Загружают сборку с указанного URL.
2. Создают экземпляр класса ImageForm в пространстве имен ImageForm и преобразуем его в тип BaseForm.
3. Просят производный от BaseForm класс открыть файл с изображением.

Для TextForm заменим старый код обработки текстовых файлов в OpenHandler():

```
TextForm txtForm = new TextForm();
txtForm.SetFileFile(sFN);
```

на:

```
Assembly ass = Assembly.LoadFrom("http://localhost/MDIDLLS/TextForm.dll");
BaseForm.BaseForm txtForm =
          (BaseForm.BaseForm) ass.CreateInstance("TextForm.TextForm");
txtForm.SetFileName(sFN);
```

Последнее, что мы сделаем, это укажем в MDIApp, что мы используем пространство имен Reflection, вставив следующую строку кода:

```
using System.Reflection;
```

Вам надо будет скомпилировать *BaseForm.cs*, *ImageForm.cs* и *TextForm.cs* в библиотеки *BaseForm.dll*, *ImageForm.dll* и *TextForm.dll* именно в этом порядке, т. к. ImageForm и TextForm являются производными от BaseForm.

Эту работу выполнят следующие командные строки:

```
csc /t:library /r:System.Windows.Forms.dll baseform.cs
csc /t:library /r:System.Windows.Forms.dll;baseform.dll imageform.cs
csc /t:library /r:System.Windows.Forms.dll;baseform.dll textform.cs
```

Нам также надо скомпилировать главный модуль *MDIApp.cs*. Заметьте, что единственное, о чем должен знать MdiApp.cs, – это о классе BaseForm.

```
csc /t:winexe
    /r:System.Windows.Forms.dll;system.drawing.dll;baseform.dll MDIApp.cs
```

После компиляции всех компонентов скопируйте их в виртуальный каталог под названием MDIDLLs, т. к. это то предопределенное местоположение, откуда должны загружаться эти компоненты.[1]

Последний шаг, который мы должны сделать, это с помощью утилиты .NET Framework Configuration задать для операций файлового ввода/вывода права на настройку зон безопасности и задать для зоны Local

[1] Убедитесь, что права доступа на этот виртуальный каталог не включают прав на запуск. Другими словами, просто создайте виртуальный каталог с установками по умолчанию.

Intranet уровень максимального доверия.[1] Такие настройки не рекомендуется использовать, не зная всех их последствий; однако это самое простое, что можно сделать для быстрой демонстрации эксперимента. Рекомендации для корпоративных приложений можно найти в документации по безопасности среды .NET.

Теперь все настроено. Надо лишь указать в вашем броузере адрес *http://localhost/MDIDLLs/MDIApp.exe* и увидеть, как автоматически загружаются и запускаются компоненты.

Еще одна утилита, которую вы захотите использовать для просмотра глобального кэша сборок (где находятся загруженные компоненты), – это `gacutil`. Выполните команду `gacutil /ldl` для получения списка всех загруженных сборок и команду `gacutil /cdl` для очистки загруженных сборок. Когда вы первый раз увидите главную форму MDIApp, единственной загруженной сборкой будет MDIApp. Но как только вы попытаетесь открыть файл изображения или текстовый файл, будут загружены сборки ImageForm или TextForm вместе с BaseForm.

Как видите, при такой конфигурации не надо иметь сотни DLL, когда требуется использовать только пару типов файлов. Замечательно также то, что когда DLL на сервере будут обновлены, вы автоматически получите последние версии компонентов.

Конечно, можно придать MDIApp более представительный вид, если главный модуль будет обращаться к веб-службе, выдающей список *типов файлов*, поддерживаемых этим приложением, и *имен классов*, а также *имен файлов сборок*, которые необходимо загрузить. Создав такую конфигурацию, можно написать условный код, основанный на типе файлов, передать *URL файла сборки* в метод Assembly.LoadFrom(); кроме того, можно использовать имя класса для создания экземпляра класса; все это сделает главный модуль более настраиваемым. Предположим, что веб-служба, выдающая список поддерживаемых приложением MDIApp имен файлов, читает информацию из базы данных или XML-файла. Когда потребуется ввести новый тип файла и вспомогательную сборку, работающую с этим типом, вам надо будет лишь добавить запись в базу данных или XML-файл и поместить файл сборки в виртуальный каталог. Мы думаем, что вам определенно следует попробовать выполнить такое упражнение.

Windows Forms и веб–службы

Раньше в Windows DNA стремились не использовать «толстые» клиенты в виде приложений Windows на PC из-за необходимости запуска на-

[1] Эта утилита расположена в Administrative Tools/Microsoft .NET Framework Configuration. Выберите Configure Code Access Security и затем Adjust Zone Security. Задайте для Local Intranet уровень полного доверия на время этого эксперимента.

вязчивой программы инсталляции. С появлением .NET и возможности развертывания путем простого копирования файлов такая проблема исчезла. Теперь «толстые» клиенты Windows могут легко взаимодействовать с бизнес-логикой среднего уровня по протоколам XML и HTTP. Действительно, клиентское приложение будет работать лучше, чем веб-интерфейс. Однако все зависит от требований приложения.

По-прежнему можно добавлять ссылки на веб-службы, автоматически генерируя классы-посредники для использования в своих приложениях Windows Forms точно так же, как это можно делать в Web Forms. Это еще одно свидетельство, что в целом среда .NET нацелена на охват Всемирной паутины. В Windows Forms привязка данных осуществляется автоматически, сразу после установки свойства DataSource элемента управления указывающим на соответствующий источник данных, как это видно в следующем блоке кода. Заметьте, что в нем отсутствует явный вызов метода DataBind, как при привязке данных в Web Forms.

```
localhost.PubsWS ws = new localhost.PubsWS();
DataSet ds = ws.GetAuthors();
DataGrid1.DataSource = ds.tables[0].DefaultView;
```

Вы также сами можете сгенерировать исходный текст класса-посредника с помощью утилиты *wsdl.exe*, используя файл WSDL, полученный от веб-сервиса. Вы можете затем включить этот исходный текст в свой проект Windows Forms или скомпилировать его в DLL и добавить полученную библиотеку в проект в виде ссылки.

Заключение

Windows Forms предоставляет унифицированную модель программирования для разработки стандартных приложений Windows. Выбор языка программирования не оказывает влияния на скорость разработки, поскольку общая основа разработана так, чтобы выиграли все.

Windows Forms привносит в разработку приложений Windows с графическим интерфейсом пользователя истинно объектно-ориентированную модель программирования, обеспечивая расширяемую среду, значительно более ясную и простую в работе по сравнению с предыдущими попытками создать такую архитектуру.

В этой главе мы рассмотрели структуру приложения Windows Forms. Архитектура контейнеров и элементов управления, будучи очень простой, представляет собой мощную и гибкую среду для разработки приложений Windows. Мы также показали, как создавать приложения Windows Forms, начав с простой программы, не делающей ничего, затем создав приложение с привязкой данных и, наконец, приложение с многодокументным интерфейсом.

A

Языки .NET

В этом приложении содержится два списка языков (разработанные Microsoft и сторонними фирмами), для которых есть компиляторы, генерирующие IL-код для выполнения в CLR.

Языки для .NET, поддерживаемые Microsoft

В табл. A.1 перечислены коммерческие языки, поддерживаемые фирмой Microsoft. Дополнительную информацию по каждому из них вы можете получить, обратившись по указанным URL.

Таблица A.1. Поддерживаемые Microsoft языки для .NET

Язык	Ссылка
C#	*http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp*
Jscript	*http://msdn.microsoft.com/workshop/languages/clinic/scripting07142000.asp*
Managed C++	*http://msdn.microsoft.com/vstudio/nextgen/Technology/managedext.asp*
VB.NET	*http://msdn.microsoft.com/vstudio/nextgen/technology/languageasp*

Языки сторонних производителей для .NET

В табл. A.2 приведен список языков сторонних производителей, имеющих компиляторы для CLR. Некоторые из них представляют собой исследовательские проекты, другие – коммерческие языки, предназна-

ченные для .NET. На указанных веб-сайтах можно найти информацию об интересующих вас языках. Как отмечалось раньше, этот список может вырасти ко времени выхода книги на рынок, поэтому обязательно обратитесь на следующие сайты за наиболее актуальными списками: *http://www.gotdotnet.com* и *http://msdn. microsoft.com/net/languages/default.asp*.

Таблица А.2. Языки сторонних производителей

Язык	Ссылка
APL	*http://www.dyadic.com*
COBOL	*http://www.adtools.com/info/whitepaper/net.html*
Component Pascal	*http://www2.fit.qut.edu.au/CompSci/PLAS//ComponentPascal*
Delta Forth	*http://www.dataman.ro/dforth/*
Eiffel#	*http://www.eiffel.com/doc/manuals/technology/dotnet/eiffelsharp/ white_paper.html*
Fortran	*http://www.lahey.com/dotnet.htm*, *http://www.salfordsoftware.co.uk/compilers/ftn95/dotnet.shtml*
Haskell	*http://haskell.cs.yale.edu/ghc*
Mercury	*http://www.cs.mu.oz.au/research/mercury/dotnet.html*
Mondrian	*http://www.mondrian-script.org*
Oberon	*http://www.oberon.ethz.ch/lightning*
Perl	*http://www.activestate.com/ASPN/NET*
Python	*http://www.activestate.com/ASPN/NET*
RPG	*http://www.asna.com/pr2%5F20%5F01.asp*
Scheme	*http://rover.cs.nwu.edu/~scheme*
Smalltalk	*http://www.qks.com*
Standard ML	*http://www.research.microsoft.com/Projects/SML.NET*
TMT Pascal	*http://www.tmt.com/net.htm*

Дополнительную информацию можно найти на сайте O'Reilly .NET Center по адресу *http://dotnet.oreilly.com* и на сайте .NET DevCenter по адресу *http://www.oreillynet.comdotnet*.

B

Основные сокращения

В табл. B.1 приведен список основных сокращений и терминов, с которыми вы можете столкнуться в материалах и разговорах по .NET. Некоторые из этих сокращений и терминов имеют небольшое отношение к .NET, однако часто встречаются в этой книге, поэтому для вашего удобства мы указали их в этой таблице.

Таблица B.1. Список основных сокращений

Сокращение	Описание
.ASMX	Расширения для файлов с исходными текстами веб-служб.
.ASPX	Расширение для файлов с исходными текстами ASP.NET.
.config	Расширение для конфигурационных файлов приложений .NET.
ADO	ActiveX Data Objects.
ADO.NET	Аналогично ADO, ADO.NET – это способ доступа к источникам данных в .NET.
API	Application Programming Interface, интерфейс прикладного программирования.
AppDomain	Сокращение, обозначающее область приложения.
ASP	Active Server Pages, активные серверные страницы.
ASP.NET	Аналогично ASP, ASP.NET – это технология, поддерживающая быструю разработку веб-форм.
ATL	Active Template Library, библиотека активных шаблонов.
BLOB	Binary Large Object, большой двоичный объект.
CAB Files	Сжатые файлы.

Сокращение	Описание
CCW	COM Callable Wrapper, вызываемый упаковщик COM.
CLI	Common Language Infrastructure, общая языковая инфраструктура. Эта спецификация инфраструктуры и библиотек базовых классов, переданная компанией Microsoft в ECMA, чтобы сторонние разработчики могли создавать библиотеки времени выполнения .NET на других платформах. CLR – это реализация CLI от корпорации Microsoft.
CLR	Common Language Runtime, общеязыковая среда выполнения.
CLS	Common Language Specification, общая языковая спецификация.
CLSID	Идентификатор класса, используемый в COM.
COFF	Common Object File Format, общий объектный формат файлов.
COM	Component Object Model, компонентная объектная модель.
COM Interop	Сокращение для COM interoperation, возможность взаимодействия с COM.
COM+ 2.0	Начиная с версии Beta 1 .NET SDK этот термин не используется. Новый термин – .NET Framework.
COM+ Runtime	Начиная с версии Beta 1 .NET SDK этот термин не используется. Новый и общепринятый термин – Common Language Runtime.
CTS	Common Type System, общая система типов.
DB	База данных.
DCOM	Distributed Component Object Model, распределенная компонентная объектная модель.
DHTML	Dynamic HyperText Markup Language, динамический язык разметки гипертекста.
DISCO	Discovery of Web Services, открытие веб-службы. Веб-служба имеет один или более DISCO-файлов, содержащих информацию о том, как обращаться к его WSDL.
DISPID	Dispatch identifier, диспетчерский идентификатор. Используется в COM для идентификации метода или свойства для динамического вызова.
DLL	Dynamically Linked Library, библиотека динамического связывания.
DNA	Distributed interNet Applications Architecture, архитектура распределенных сетевых приложений.
DOM	Document Object Model, объектная модель документа.
DTD	Datatype Document, документ типа данных. Предшественник XSD-схем.

Сокращение	Описание
EXE	Исполняемый файл.
GC	Garbage Collector, сборщик мусора.
GDI	Graphical Device Interface, интерфейс графического устройства.
GDI+	.NET-библиотека, поддерживающая расширенное управление графикой.
Global.asax	Глобальный конфигурационный файл приложения ASP.NET.
GUID	Globally Unique Identifier, глобальный уникальный идентификатор.
HTML	HyperText Markup Language, язык разметки гипертекста.
HTTP	HyperText Transfer Protocol, протокол передачи гипертекста.
IDE	Integrated Development Environment, интегрированная среда разработки.
IDL	Interface Definition Language, язык определения интерфейса.
IE	Internet Explorer.
IID	Interface Identifier, идентификатор интерфейса.
IIS	Internet Information Services.
IJW	It Just Works, «это просто работает».
IL	Intermediate Language, промежуточный язык.
ILDASM	Intermediate Language Disassembler, дизассемблер промежуточного языка.
Inproc	Внутрипроцессный.
ISAPI	Information Server Application Programming Interface, прикладной интерфейс сервера IIS.
Machine.config	Конфигурационный файл административной политики для машины в целом.
MBR	Marshal-By-Reference, маршалинг по ссылке.
MBV	Marshal-By-Value, маршалинг по значению.
MFC	Microsoft Foundation Classes.
MSI	Microsoft Windows Installer Package, пакет инсталляции для Microsoft Windows.
MSIL	Microsoft Intermediate Language, промежуточный язык Microsoft.
MSVCRT	Microsoft Visual C++ Runtime, библиотека времени выполнения Microsoft Visual C++.

Сокращение	Описание
MSXML	Microsoft eXtensible Markup Language, расширяемый язык разметки Microsoft.
MTS	Microsoft Transaction Server, сервер транзакций Microsoft.
NGWS	Начиная с Beta 1 .NET SDK этот термин больше не используется. Новый и общепринятый термин – .NET.
NTFS	NT Filesystem, файловая система NT.
N-Tier	Multi-tier, многоуровневый.
NTLM	NT Lan Manager.
OBJREF	Object Reference, ссылка на объект.
Out of proc	Внепроцессный.
P/Invoke	Platform Invoke.
PE	Portable Executable, портируемый исполняемый файл.
perm	Разрешения.
RAD	Rapid Application Development, быстрая разработка приложений.
RCW	Runtime Callable Wrapper, вызываемый упаковщик времени выполнения.
REGASM	Register Assembly tool, утилита регистрации сборок.
RPC	Remote Procedure Call, удаленный вызов процедур.
SCL	SOAP Contract Language, язык контрактов SOAP.
SDK	Software Development Kit, набор разработчика программного обеспечения.
SEH	Structured Exception Handling, структурная обработка исключений.
SMTP	Simple Mail Transfer Protocol, простой протокол передачи почты.
SOAP	Simple Object Access Protocol, простой протокол доступа к объектам.
SQL	Structured Query Language, язык структурированных запросов.
Standard JIT	Оптимизированный двоичный код, включающий верификацию IL (генерируется стандартным JIT-компилятором).
STL	Standard Template Library, библиотека стандартных шаблонов.
TCP	Transport Control Protocol, транспортный управляющий протокол.
TLB	Type Library, библиотека типов.

Сокращение	Описание
TLBEXP	Type Library Exporter Tool, утилита экспорта библиотек типов.
TLBIMP	Type Library Importer Tool, утилита импорта библиотек типов.
UDDI	Universal Description, Discovery, and Integration Service. UDDI – это независимая от платформы среда описания и открытия веб-служб.
UDF	Uniform Data Format, унифицированный формат данных.
UI	User Interface, пользовательский интерфейс.
URI	Uniform Resource Identifier, унифицированный идентификатор ресурса.
URL	Uniform Resource Locator, унифицированный локатор ресурса.
URT	Универсальная библиотека времени выполнения. Начиная с Beta 1 .NET SDK этот термин больше не используется. Новый и общепринятый термин – .NET Framework.
VB	Visual Basic.
VBRUN	Visual Basic Runtime, библиотека времени выполнения Visual Basic.
VES	Virtual Execution System. VES – это подмножество CLR. VES не включает таких средств, как отладка, профилирование и взаимодействие с COM.
Visual Studio 7	Начиная с Beta 1 .NET SDK этот термин больше не используется. Новый и общепринятый термин – Visual Studio .NET.
VOS	Virtual Object System, виртуальная объектная система. Теперь называется CTS.
VS.NET	Visual Studio .NET.
WAP	Wireless Application Protocol, протокол беспроводных приложений.
web.config	Конфигурационный файл ASP.NET. В этом файле определяются HTTP-модули, обработчики, управление состоянием сеанса и другие конфигурируемые параметры ASP.NET.
WebForms	Начиная с Beta 1 .NET SDK этот термин больше не используется. Новый и общепринятый термин – это Web Forms (с пробелом).
WebServices	Начиная с Beta 1 .NET SDK этот термин больше не используется. Новый и общепринятый термин – это Web Services (с пробелом).
Win32	Windows, 32-разрядный.

Сокращение	Описание
WinForms	Начиная с Beta 1 .NET SDK этот термин больше не используется. Новый и общепринятый термин – Windows Forms (с пробелом).
WML	Wireless Markup Language, беспроводной язык разметки.
WSDL	Web Service Description Language, язык описания веб-службы. Можно рассматривать как IDL для веб-сервисов. В отличие от IDL, WSDL выражается исключительно XML-схемами. В Beta 1 .NET SDK использовался SDL, в Beta 2 и более поздних версиях SDL был заменен на WSDL.
XML	Extensible Markup Language, расширяемый язык разметки.
XPath	XML Path.
XSD	XML Schema Definition, определение XML-схемы.
XSL	Extensible Stylesheet Language, расширяемый язык таблиц стилей.
XSLT	Extensible Stylesheet Language Transformations, язык преобразования расширяемых таблиц стилей.

C

Общие типы данных

Каждый из языков среды .NET должен предоставлять свои собственные ключевые слова для поддерживаемых им типов. Например, ключевое слово для целого в VB – `Integer`, в то время как в C# или C++ это `int`, логическое значение в VB обозначается как `Boolean`, а в C# или в C++ – `bool`. В любом случае целому значению ставится в соответствие класс Int32, а логическому значению – класс Boolean пространства имен System. В табл. C.1 перечислены все простые типы данных, общие для .NET Framework. Типы, не совместимые с CLS, не обязательно будут работать со всеми CLS-совместимыми языками.

Таблица C.1. Общие типы данных

Тип	Описание
Boolean	Истина или ложь.
Byte	8-битовое целое без знака: от 0 до 255.
Char	Символ. 16-битовый символ Unicode.
DateTime	Представляет значение даты и времени.
Decimal	Может обозначать положительные или отрицательные значения с 28 значащими цифрами: от -79 228 162 514 264 337 593 543 950 335 до 79 228 162 514 264 337 593 543 950 335.
Double	Содержит 64-битовые значения с плавающей точкой: от -1.79769313486231570e308 до 1.79769313486231570e308.
Guid	Обозначает глобально уникальный идентификатор (GUID).
Int16	Содержит 16-битовое целое со знаком: от -32 768 до 32 767.

Таблица C.1 (продолжение)

Тип	Описание
Int32	Содержит 32-битовое целое со знаком: от –2 147 483 648 до 2 147 483 647.
Int64	Содержит 64-битовое целое со знаком: от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807.
SByte	Представляет 8-битовое целое со знаком от 128 до 127. Тип SByte не является CLS-совместимым.
Single	Представляет совместимое с IEEE 754f, 32-битовое целое с одинарной точностью: от –3.40282346638528859e38 до 3.40282346638528859e38.
String	Представляет строку символов Unicode.
UInt16	Представляет 16-битовое целое без знака. Тип UInt16 не является CLS-совместимым. От 0 до 65 535.
UInt32	Представляет 32-битовое целое без знака от 0 до 4 294 967 295. Тип UInt32 не является CLS-совместимым.
UInt64	Представляет 64-битовое целое без знака. Тип UInt64 не является CLS-совместимым. В типе данных UInt64 могут быть представлены положительные целые с 18 значащими цифрами: от 0 до 184 467 440 737 095 551 615.
Void	Void, фиктивное значение.

В табл. C.2 показано несколько полезных контейнерных типов, предоставляемых .NET Framework.

Таблица C.2. Контейнерные типы

Тип	Описание
ArrayList	Этот класс реализует интерфейс IList. Массив может динамически увеличивать размер.
BitArray	Этот класс представляет компактный массив битовых значений. Каждый элемент массива содержит значение типа Boolean (истина или ложь).
HashTable	Этот класс обозначает коллекцию связанных ключей и значений, упорядоченных по хеш-коду ключа.
Queue	Этот класс представляет конструкцию, работающую по принципу «первым вошел, первым вышел».
SortedList	Этот класс аналогичен HashTable, кроме того, все элементы отсортированы по их фактическим ключам (а не по хеш-кодам), а элементы доступны как по ключу, так и по индексу.
Stack	Этот класс представляет стековую конструкцию, работающую по принципу «первым вошел, последним вышел».

Примеры применения

В этом примере демонстрируется, как можно использовать преимущества контейнерных типов. Мы не будем иллюстрировать все их свойства и методы, а покажем самые важные характеристики этих типов. Все примеры этой главы написаны на C#, однако работать с этими CLS-типами можно из любого другого CLS-совместимого языка.

ArrayList

В следующем листинге мы демонстрируем несколько наиболее важных операций с классом ArrayList, таких как добавление данных в конец списка, вставка данных в произвольное место списка, перебор элементов списка и сортировка.

```
using System;
using System.Collections;
public class TestArrayList {

   public static void Main()  {

      ArrayList arrList = new ArrayList();
      arrList.Add("Понедельник");
      arrList.Add("Вторник");
      arrList.Add("Среда");
      arrList.Add("Четверг");

      // Мы попытаемся вставить пятницу позднее arrList.Add("Пятница");

      arrList.Add("Суббота");
      arrList.Add("Воскресенье");

      int i = 0;
      IEnumerator arrIterator = arrList.GetEnumerator();
      Console.WriteLine("В неделе: {0} дней.", arrList.Count);
      while(arrIterator.MoveNext()) {
         Console.WriteLine("[{0}] {1}", i++, arrIterator.Current);
      }

      Console.WriteLine("Вставляем пятницу");
      arrList.Insert(4, "Пятница");

      i = 0;
      arrIterator = arrList.GetEnumerator();
      Console.WriteLine("В неделе: {0} дней.", arrList.Count);
      while(arrIterator.MoveNext()) {
         Console.WriteLine("[{0}] {1}", i++, arrIterator.Current);
      }

      arrList.Sort();

      i = 0;
      arrIterator = arrList.GetEnumerator();
      Console.WriteLine("Сортируем как текст");
```

```
        while(arrIterator.MoveNext()) {
           Console.WriteLine("[{0}] {1}", i++, arrIterator.Current);
        }

        Object oDay = "Пятница";
        Console.WriteLine("Определяем индекс для пятницы "+
                          "с помощью BinarySearch: {0}",
                          arrList.BinarySearch(oDay));
        Console.WriteLine("Определяем индекс для воскресенья "+
                          "с помощью BinarySearch: {0}",
                          arrList.BinarySearch("Воскресенье"));
     }
  }
```

BitArray

Код примера для BitArray понятен без объяснений, что видно из следующего листинга. Мы используем битовый массив для хранения и извлечения прав доступа. Можно применять как методы Set и Get, так и оператор [].

```
  using System;
  using System.Collections;
  public class TestBitArray  {

     enum Permissions {canRead, canWrite, canCreate, canDestroy};

     public static void Main()  {

        BitArray bitArr = new BitArray(4);
        bitArr.Set((int)Permissions.canRead, true);
        bitArr[(int)Permissions.canWrite] = false;
        bitArr[(int)Permissions.canCreate] = true;
        bitArr[(int)Permissions.canDestroy] = false;

        Console.WriteLine("bitArr count: {0}\tlength: {1}",
                          bitArr.Count,
                          bitArr.Length);

        Console.WriteLine("Права доступа:");
        Console.WriteLine("Чтение: {0}",
                          bitArr[(int)Permissions.canRead]);
        Console.WriteLine("Запись: {0}",
                          bitArr[(int)Permissions.canWrite]);
        Console.WriteLine("Создание: {0}",
                           bitArr[(int)Permissions.canCreate]);
        Console.WriteLine("Уничтожение: {0}",
                           bitArr[(int)Permissions.canDestroy]);
     }
  }
```

HashTable

Тип HashTable аналогичен объекту-словарю, по сути, представляя собой ассоциированный массив. Каждый элемент, хранящийся в таблице, связан с ключом. Так как HashTable реализует интерфейс IDictionaryEnumerator, мы можем воспользоваться перечислителем для перебора коллекции данных. Как видно из примера кода, мы также можем перебирать данные с помощью коллекций `keys` или `values`.

```
using System;
using System.Collections;
public class TestHashtable  {

   public static void Main()  {

      Hashtable hashTbl = new Hashtable();
      hashTbl.Add("Param1", "Имя пользователя");
      hashTbl.Add("Param2", "Пароль");

      IDictionaryEnumerator hashEnumerator = hashTbl.GetEnumerator();
      Console.WriteLine();
      Console.WriteLine("Используем перечислитель:");
      while (hashEnumerator.MoveNext()) {
         Console.WriteLine("Ключ: {0}\t3начение: {1}",
                           hashEnumerator.Key,
                           hashEnumerator.Value);
      }

      Console.WriteLine();
      Console.WriteLine("Обход коллекции Keys:");
      foreach(string key in hashTbl.Keys) {
         Console.WriteLine(key);

      }

      Console.WriteLine();
      Console.WriteLine("Обход коллекции Values:");
      foreach(string val in hashTbl.Values) {
         Console.WriteLine(val);

      }

      Console.WriteLine();
      Console.WriteLine("Обход коллекции Keys:");
      foreach(string key in hashTbl.Keys) {
         Console.WriteLine("Ключ: {0}\t3начение: {1}", key, hashTbl[key]);

      }
   }
}
```

Queue

Чтобы продемонстрировать использование абстрактного типа данных (Abstract Data Type, ADT) очереди, мы создадим листинг фиктивного кода обработки заказов. Каждая единица, поставленная в очередь, обозначает строку типичного заказа. Затем мы извлекаем из очереди каждую строку и выполняем подсчет итога.

```
using System;
using System.Collections;

public class TestQueue  {

   public static void Main()  {
      string sLineItem1, sLineItem2;
      Queue myQueue = new Queue();

      sLineItem1 = "123\tItem 123\t4\t3.39";
      sLineItem2 = "ABC\tItem ABC\t1\t9.49";
      myQueue.Enqueue(sLineItem1);
      myQueue.Enqueue(sLineItem2);

      Console.WriteLine("\nОбработка заказа:\n");
      String sLineItem = "";
      String [] lineItemArr;
      Decimal total = 0;
      while(myQueue.Count > 0) {
         sLineItem = (String)myQueue.Dequeue();
         Console.WriteLine( "\t{0}", sLineItem);
         lineItemArr = sLineItem.Split(new Char[] {'\t'});
         total += Convert.ToInt16(lineItemArr[2]) *
                  Convert.ToDecimal(lineItemArr[3]);
      }
      Console.WriteLine("\nИтог по заказу: {0}\n", total);

   }
}
```

SortedList

В следующем коде демонстрируется абстрактный тип данных отсортированного списка. Отсортированный список похож на хеш-таблицу или словарный тип. Каждая единица данных связана с ключом, по которому отсортирован список. Заметьте, что строки добавляются в список не по порядку. Однако когда мы перебираем элементы списка, все строки уже отсортированы по связанным с ними ключам.

```
using System;
using System.Collections;
public class TestSortedList  {

   public static void Main()  {
```

```
        SortedList mySortedList = new SortedList();
        mySortedList.Add("AA", "Hello");
        mySortedList.Add("AC", "!");
        mySortedList.Add("AB", "World");

        Console.WriteLine("\nРучной обход списка:\n");
        for(int i=0; i< mySortedList.Count; i++) {
           Console.WriteLine("Ключ: {0}\t3начение: {1}",
                             mySortedList.GetKey(i),
                             mySortedList.GetByIndex(i));
        }

        IDictionaryEnumerator myIterator = mySortedList.GetEnumerator();
        Console.WriteLine("\nЦикл с помощью перечислителя:\n");
        while (myIterator.MoveNext()) {
           Console.WriteLine("Ключ: {0}\t3начение: {1}",
                             myIterator.Key,
                             myIterator.Value);
        }
     }
  }
```

Stack

Следующий код демонстрирует характерное свойство стекового абстрактного типа данных – принцип «первым вошел, последним вышел». Результатом операции `pop` **сначала является четвертый элемент, потом третий и т. д.**

```
using System;
using System.Collections;
public class TestStack{

   public static void Main()  {

      Stack myStack = new Stack();
      myStack.Push("Item 1");
      myStack.Push("Item 2");
      myStack.Push("Item 3");
      myStack.Push("Item 4");

      while(myStack.Count > 0) {
         Console.WriteLine(myStack.Pop());
      }
   }
}
```

D

Основные утилиты

Microsoft .NET Framework предоставляет много утилит, помогающих разработчику эффективнее использовать среду. В следующих разделах мы приведем описание подмножества наиболее часто применяемых утилит .NET, которые мы упоминали в этой книге.

- Утилита генерации сборок (*al.exe*)
- Утилита регистрации сборок (*gacutil.exe*)
- Ассемблер MSIL (*ilasm.exe*)
- Дизассемблер MSIL (*ildasm.exe*)
- Компилятор C++ (*cl.exe*)
- Компилятор C# (*csc.exe*)
- Компилятор VB (*vbc.exe*)
- Программа просмотра PE-файлов (*dumpbin.exe*)
- Утилита экспорта библиотек типов (*tlbexp.exe*)
- Утилита импорта библиотек типов (*tlbimp.exe*)
- Утилита определения XML-схем (*xsd.exe*)
- Утилита разделяемых имен (*sn.exe*)
- Утилита веб-служб (*wsdl.exe*)

Утилита генерации сборок (al.exe)

Утилита *al.exe* обычно применяется для генерации сборок с манифестами. Следующая таблица показывает несколько основных параметров утилиты генерации сборок.

Параметр	Описание
/flags:*флаги*	Задает значение поля Flags в сборке. 0x0000: допустимо одновременное выполнение версий. 0x0010: не может выполняться с другими версиями в той же области приложения. 0x0020: не может выполняться с другими версиями в том же процессе. 0x0030: не может выполняться с другими версиями на том же компьютере.
/help или /?	Используется для получения справки по этой команде.
/keyfile:*имя_файла_ключей* или /keyf:*имя_файла_ключей*	Используется для создания разделяемых компонентов. Файл *имя_файла_ключей* содержит пару ключей, сгенерированных утилитой разделяемых имен (*sn.exe*). Компилятор вставляет открытый ключ в манифест сборки и затем подписывает сборку закрытым ключом.
/keyname:*контейнер_ключей* или /keyn:*контейнер_ключей*	Предназначен для создания разделяемых компонетов. *Контейнер_ключей* содержит пару ключей, сгенерированных и помещенных в контейнер ключей с помощью утилиты разделяемых имен (*sn.exe*). Компилятор вставляет открытый ключ в манифест сборки и затем подписывает сборку закрытым ключом.
/main:*точка_входа*	Задает имя метода, который будет являться точкой входа при преобразовании модуля в исполняемый файл.
/out:*имя_файла*	Задает имя выходного файла.
/target:lib|exe|win или /t:lib|exe|win	Задает формат выходного файла (lib для библиотеки, exe для консольного исполняемого файла и win для исполняемого файла win32). Значение по умолчанию – lib.
/version:*major.minor.revision.build*	Задает информацию о версии для сборки. Значение по умолчанию – 0.

Утилита регистрации сборок (gacutil.exe)

Утилита *gacutil.exe* может применяться для установки и удаления сборки, а также для просмотра содержимого глобального кэша сборок (GAC). В следующей таблице перечислены наиболее распространенные способы использования утилиты регистрации сборок.

Параметр	Описание
/l	Вывести содержимое GAC.
/ldl	Вывести содержимое кэша загруженных файлов.
/cdl	Удалить содержимое кэша загруженных файлов.
/i *имя_файла*	Установить сборку, находящуюся в файле с именем *имя_файла* в GAC.
/u *имя_сборки*	Удалить сборку из GAC по указанному имени. Если существует несколько версий одной и той же сборки, они все должны быть удалены, если вместе с *именем сборки* не указан номер версии (например, gac -u myAssembly,ver=1.0.0.1).
/h или /help или /?	Вывести описание синтаксиса команды и ее параметров.

Ассемблер MSIL (ilasm.exe)

Эта утилита получает файл с кодом MSIL и генерирует портируемый исполняемый файл (PE), содержащий MSIL и метаданные, необходимые для запуска в среде .NET. Это наиболее полезно для разработчиков, которые хотят создавать MSIL-совместимые компиляторы. Все, что им нужно сделать – это написать компилятор, переводящий код исходного языка на MSIL. Утилита *Ilasm.exe* выполнит второй этап преобразования MSIL в формат PE, который может исполняться в .NET Framework. Общий синтаксис запуска ассемблера MSIL следующий:

```
ilasm [параметры] имя_MSIL_файла
```

В следующей таблице приведены основные параметры ассемблера.

Параметр	Описание
/debug	Обеспечивает включение в результирующий PE-файл отладочной информации, такой как локальные переменные, имена аргументов и номера строк. Полезна для отладочных версий.
/dll	Указывает необходимость создания dll-файла.
/exe	Указывает необходимость создания exe-файла.
/listing	Приводит к выдаче листинга результирующего файла на STDOUT.
/output=*имя_файла*	*Имя_файла* задает имя результирующего файла.
/?	Параметр используется для получения описания параметров командной строки.

Дизассемблер MSIL (ildasm.exe)

Эта утилита извлекает MSIL-код из PE-файла, предназначенного для среды .NET. Общий синтаксис этой утилиты:

```
Ildasm [параметры] имя_PE_Файла
```

Следующая таблица показывает несколько основных параметров дизассемблера:

Параметр	Описание
/linenum	Включает ссылки на строки исходного текста.
/output=*имя_файла*	Результат записывается в файл вместо выдачи графического диалогового окна.
/source	Показывает строки исходного текста в комментариях.
/text	Результат выдается в консольном окне.
/tokens	Показывает маркеры метаданных классов и членов классов.

Компилятор C++

В следующей таблице показано несколько основных параметров компилятора C++.

Параметр	Описание
/CLR	Параметр указывает компилятору на необходимость компиляции «управляемого» кода для библиотеки времени выполнения .NET.
/entry:*имя_метода*	Для управляемого кода C++ эта ссылка должна указывать на функцию, являющуюся главной точкой входа.
/link	Параметр объединяет этапы компиляции и компоновки.
/subsystem:|windows|windowsce|console|	Параметр задает тип результирующего файла.
/out:*имя_файла*	Параметр позволяет задать имя результирующего файла.

Компилятор C# (csc.exe)

В следующей таблице приведено несколько основных параметров компилятора C#.

Параметр	Описание
/debug	Обеспечивает включение в результирующий файл отладочной информации.
/define:*символ* или /d:*символ*	Параметр аналогичен директиве препроцессора в C++. Используйте его для определения символических имен для препроцессора.
/doc:*имя_документа*	*Файл с именем Имя_документа* – это результирующий XML-файл автоматически сгенерированных XML-комментариев из кода C#.
/help	Выдает справку по параметрам командной строки.
/main:*имя_класса*	Если имеется несколько методов Main в различных классах, необходимо указать метод Main того класса, который вы хотите сделать точкой входа приложения.
/out:*имя_класса*	Параметр задает имя результирующего файла.
/reference:*имя_библиотеки* или /r:*имя_библиотеки*	Параметр позволяет задать одну или несколько библиотек, которые должны быть включены при компиляции. Если библиотек несколько, то в качестве разделителя выступает точка с запятой.
/target:exe|library|winexe|module или /t:exe|library|winexe|module	Параметр позволяет указать тип результирующего файла: exe для консольных приложений, library для DLL и winexe для приложений Windows Form. Если задать параметр /target:module, компилятор выдаст двоичный файл модуля, а не сборки .NET. Модули могут быть позднее добавлены в сборку .NET.
/unsafe	Если в коде C# присутствуют ключевые слова unsafe, надо указать этот параметр при компиляции исходного текста.

Компилятор Visual Basic (vbc.exe)

В следующей таблице показаны несколько основных параметров компилятора Visual Basic.

Опции	Описание
/debug	Обеспечивает включение в результирующий файл отладочной информации.
/define:*символ* или /d:*символ*	Используйте этот параметр для определения символических имен препроцессора.

Опции	Описание
/help или /?	Выводит описание параметров командной строки компилятора Visual Basic.
/keycontainer:*контейнер_ключей*	Параметр *Контейнер_ключей* задает контейнер ключей, содержащий пару ключей для цифровой подписи сборки. Информацию по генерации контейнера ключей см. в описании утилиты ***sn.exe***.
/keyfile:*файл_ключей*	Параметр *файл_ключей* задает файл ключей, содержащий пару ключей для цифровой подписи сборки. Информацию по генерации файла ключей можно найти в описании утилиты ***sn.exe***.
/main:*имя_класса*	Если имеется более одной точки входа Main в различных классах, необходимо указать класс, метод Main которого вы хотите сделать точкой входа приложения.
/out:*имя_файла*	Параметр задает имя выходного файла.
optionexplicit[+/-]	Включите или выключите параметр optionexplicit, чтобы задать явное или неявное объявление переменных. По умолчанию параметр включен.
optionstrict[+/-]	Включите или выключите параметр optionstrict, чтобы разрешить или запретить преобразование с усечением. По умолчанию параметр включен.
/reference:*имя_библиотеки* или /r:*имя_библиотеки*	Параметр позволяет задать одну или несколько библиотек, которые должны быть подключены при компиляции. Библиотеки в списке разделяются при помощи точки с запятой.
/target:exe|library|winexe|module или /t:exe|library|winexe|module	Параметр позволяет указать тип результирующего файла: exe для консольных приложений, library для DLL и winexe для Windows Form-приложений. Если задан параметр /target:module, компилятор выдает двоичный файл модуля, а не сборки .NET. Модули могут быть позднее добавлены в сборку .NET.

Программа просмотра PE-файлов (dumpbin.exe)

Утилита *Dumpbin.exe* не является новой. Однако поскольку .NET Framework хранит внутри расширенного PE-формата IL-код, эта старая утилита все еще очень полезна для просмотра структуры исполняемых

файлов и DLL, а также списков импортируемых и экспортируемых из двоичных файлов символов. Общий синтаксис этой утилиты:

```
Dumpbin [параметры] Имя_PE_файла
```

В следующей таблице приведены основные примеры применения *dumpbin*.

Параметр	Описание
/all	Показывает всю информацию из PE-файла.
/exports	Показывает все экспортируемые PE-файлом символы.
/header	Показывает информацию о заголовке PE-файла.
/imports	Показывает все импортируемые PE-файлом символы.

Утилита экспорта библиотек типов (tlbexp.exe)

Утилиты экспорта и импорта библиотек типов необходимы для взаимодействия с COM-компонентами. Утилита экспорта генерирует библиотеку типов для сборки .NET Framework, чтобы другие COM-компоненты могли взаимодействовать с .NET-компонентами. Общий синтаксис *tlbexp.exe*:

```
tlbexp Имя_Сборки [параметры]
```

В следующей таблице приведено несколько основных параметров *tlbexp.exe*.

Параметр	Описание
/nologo	Подавляет вывод графической заставки tlbexp.
/out:*имя_файла*	*Имя_файла* – это имя файла библиотеки типов.
/silent	Подавляет все сообщения утилиты tlbexp.
/verbose	Показывает дополнительную информацию при преобразовании компонентов.
/? или /help	Показывает информацию об использовании утилиты.

Утилита импорта библиотек типов (tlbimp.exe)

Эта утилита по действию обратна утилите экспорта библиотек типов. Утилита импорта генерирует посреднический компонент .NET для COM-компонента, чтобы .NET-компоненты могли использовать существующие COM-компоненты. Общий синтаксис *tlbimp.exe*:

```
tlbimp PE_файл [параметры]
```

В следующей таблице показано несколько основных параметров *tlbimp.exe*.

Параметр	Описание
/keycontainer: *контейнер_ключей*	Параметр подписывает результирующую сборку закрытым ключом, находящимся в *контейнере_ключей*. Открытый ключ из *контейнера_ключей* будет использоваться в манифесте сборки. Информацию по генерации контейнера ключей можно найти в описании утилиты *sn.exe*.
/keyfile:*файл_ключей*	Параметр подписывает результирующую сборку закрытым ключом из *файла_ключей*. Открытый ключ из *файла_ключей* будет использоваться в манифесте сборки. Информацию по генерации файла ключей можно найти в описании утилиты *sn.exe*.
/nologo	Подавляет вывод графической заставки утилиты tlbimp.
/out:*имя_файла*	Параметр *Имя_файла* задает имя файла библиотеки типов.
/silent	Параметр подавляет вывод сообщений утилиты tlbimp.
/unsafe	Параметр приводит к генерации интерфейсов без проверок на безопасность .NET Framework.
/verbose	Параметр показывает дополнительную информацию при преобразовании компонента.
/? или /help	Параметр выводит справку об утилите.

Утилита определения XML-схем

Утилита определения XML-схем (XSD) полезна при работе с XML-схемами, написанными на языке XSD. XSD позволяет выполнять следующие преобразования:

- XDR в XSD
- XML в XSD
- Классы в XSD
- XSD в классы
- XSD в DataSet

XDR в XSD

Для преобразования файла, отформатированного как XDR, в XSD применяется следующий синтаксис:

```
xsd [параметры] file.xdr
```

Заметьте, что расширение *.xdr* диктует выполнение преобразования из XDR в XSD.

XML в XSD

Синтаксис преобразования XML-файла в XSD:

```
xsd [параметры] file.xml
```

Заметьте, что расширение файла *.xml* диктует выполнение преобразования из XML в XSD.

Классы в XSD

Можно преобразовать классы в XSD, указав в имени файла в параметре утилиты исполняемый файл сборки (расширение *.exe* или *.dll*). Также можно указать определенный тип из сборки, который вы хотите преобразовать в XSD с помощью параметра `/type`. Параметр `Имя_типа` может содержать символы подстановки. Если параметр `/type` не указан, будут преобразованы все типы из сборки. Ниже приведен синтаксис:

```
Xsd [/TYPE:имя_типа] файл_сборки
```

или:

```
Xsd [/T:имя_типа] файл_сборки
```

XSD в классы

Для преобразования XSD обратно в классы применяется параметр `/classes` или `/c`. Можно указать определенный элемент XSD-схемы, который должен быть преобразован в класс. Также можно указать язык исходного файла класса. Общий синтаксис таков:

```
xsd /CLASSES
    /ELEMENT:элемент
    /NAMESPACE:пространство_имен
    /LANGUAGE:язык /URI:uri file.xsd
```

или:

```
xsd /C E:элемент /N:пространство_имен /L:язык /U:uri file.xsd
```

Заметьте, что `пространство_имен`, `язык` и `uri` могут быть указаны только один раз.

XSD в DataSet

Для преобразования XSD в набор данных применяется параметр `/dataset` или `/d`. Преобразование можно ограничить определенным элементом XSD-схемы. Общий синтаксис таков:

```
xsd /D [/DATASET] file.xsd
```

Утилита разделяемых имен (sn.exe)

Утилита *sn.exe* обеспечивает уникальность имен разделяемых компонентов, т. к. компоненты в конечном итоге оказываются в GAC. Каждый разделяемый компонент подписывается закрытым ключом и публикуется с открытым ключом. В следующей таблице приведены несколько основных параметров *sn.exe*.

Параметр	Описание
/?	Выводит справку о параметрах командной строки.
-d *контейнер_ключей*	Параметр предназначен для удаления *контейнера_ключей* из CSP.
-i *файл_ключей контейнер_ключей*	Параметр читает пару ключей из *файла_ключей* и устанавливает их в *контейнер_ключей* в Cryptographic Service Provider (CSP).
-k *файл_ключей*	Параметр генерирует новую пару ключей и записывает ее в *файл_ключей*.
-v *сборка*	Параметр служит для проверки разделяемого имени *сборки*.

Утилита веб-служб (wsdl.exe)

Утилита *wsdl.exe* облегчает создание веб-служб ASP.NET и посредников для их клиентов. Чаще всего *wsdl.exe* применяется для генерации классов-посредников для веб-служб:

```
wsdl
     /language:язык
     /namespace:пространство_имен
     /out:выходной_файл
     /protocol:протокол
     путь
```

Параметр *путь* – это локальный путь к файлу описания службы или URI, откуда может быть извлечен SDL-файл. Параметр *язык* задает язык для результирующего файла-посредника. Он может быть C#, VB или JS. Сгенерированный класс будет в указанном пространстве имен. Результирующий файл исходного текста задается параметром *выходной_файл*. Параметр *протокол* задает протокол, который будет использовать посредник для взаимодействия с веб-службой. Выбор протоколов, предоставляемый .NET Framework – это SOAP, HttpGet и HttpPost. Можно также указать здесь свой собственный протокол, если вы расширили класс WebClientProtocol или HttpWebClientProtocol.

В случае применения кратких имен параметров синтаксис утилиты следующий:

```
wsdl
     /l:язык
     /n:пространствоимен
     /o:выходнойфайл
     /protocol:протокол
     путь
```

Алфавитный указатель

Специальные символы

#import, директива в C++, 26
#using, директива, 26
%windir%, параметр, 91
<‰ ... ‰>, отметка блоков отображения кода, 218

A

ABC (Abstract Base Class), абстрактные базовые классы, 53
abstract
  атрибут IL-декларации .class, 75
  ключевое слово, указание компилятору C#, 72
.assembly extern, ключевое слово, 44
.assembly, блоки, 45
AcceptChanges(), метод, 123
ACL (Access Control List), список управления доступом, 108
Activate(), метод, 106
Activator, вызов метода GetObject, 98
Active Server Pages, см. ASP
Active Template Library, см. ATL
ActiveMdiChild, свойство, 274, 280
ActiveX Data Objects, см. ADO
Add(), метод, 104
AddMessageFilter, метод класса Application, 261
AddRange(), метод, 277
Administrators, группа, 108
ADO (ActiveX Data Objects), 117, 291
ADO.NET, 23, 119
  архитектура, 118
  возможность взаимодействия, 119
  компоненты сущностей, 122
  масштабируемость, 120
  продуктивность, 121
  производительность, 122
  управляемые провайдеры, 136
Agents, роль, 108
al.exe (Assembly Generation Utility), утилита генерации сборок, 304
  использование сборок, 45
AlternatingItemStyle, управление визуальным форматированием DataGrid, 244
AlternatingItemTemplate, 245
ansi, атрибут, 76
AOL Instant Messanger, 163
API (Application Programming Interface), интерфейс прикладного программирования, 291
APL, язык, 290
AppDomain, класс, 64, 291
AppendChild(), формирование дерева XML-документа, 150
Application Center 2000, серверные продукты, 15
Application, класс, 260
Array, класс, 64, 241
ArrayList, класс, 64, 241, 298
asmx-файлы, 291
ASP (Active Server Pages), активные серверные страницы, 202, 291
ASP.NET, 204
  серверные элементы управления, 229
  веб-службы, 238, 241
  разработка приложения, 222
  событийное программирование, 237
  специальные серверные элементы управления, 230
aspx, расширение файла, 222, 291
Assembly Generation Utility (al.exe, утилита генерации сборок)
  использование сборок, 45
Assembly, класс, 64
assemblyIdentity, XML-тег, 94
AssemblyKeyFile, атрибут XML, 89

AssemblyVersion, атрибут XML, 89
ATL (Active Template Library), прикладная среда, 18, 255, 291
auto, атрибут в CIL, 76
AutoResetEvent, класс, 64

B

Basic Authentication, базовая аутентификация, 200
Basic over SSL Authentication, базовая аутентификация поверх SSL, 200
BeginTransaction(), метод, 138
Binary Large Object, см. BLOB
Binding, формат привязки данных для веб-служб, 167
BitArray, класс, 298
BizTalk, серверы, 15, 149
BLOB (Binary Large Object), большой двоичный объект, 291
boolean, тип данных, 297
Bottom, свойство элемента управления, 257
BufferResponse (boolean), свойство, 178
Button, элемент управления Windows, 266
Byte, тип данных, 297

C

C#, язык, 27, 204, 289
  код, 71
  межъязыковая интеграция, 81
  провайдеры веб-служб, создание, 176
C++, язык, 26, 289
  межъязыковая интеграция, 79
  разработка Windows-приложений, 255
CAB-файлы, формат архивирования дистрибутивов ПО, от Microsoft, 292
Cache, свойство
  Page, класс, 207
CacheDuration, свойство, 178
CanBePooled(), метод, 106
Cascade, значение правила ForeignKeyConstraint, 133
catch-блоки
  в C++, 68
CCW (COM Callable Wrapper), вызываемый упаковщик COM, 292
cfg, расширение файла, 56
Char, класс, 64, 297
ChildRelations, свойство объекта DataTable, 132
CIL (Common Intermediate Language), общий промежуточный язык, 23, 46
  кодирование, 75
cl, компилятор, 41
cl.exe (компилятор C++), 304, 307
.class, IL-декларация, 75
class, атрибут директивы WebService, 238
Class, ключевое слово VB.NET, 70
CLI (Common Language Infrastructure), общеязыковая инфраструктура, 23, 292
Click, событие элемента управления, 258
Client Certificates Authentication, аутентификация с помощью клиентских сертификатов, 200
Clone(), формирование дерева XML-документа, 150
CLR (Common Language Runtime), общеязыковая среда выполнения, 16, 22, 25, 292
  исполняемые файлы, 26, 29
  метаданные, 31
  сборки и манифесты, 39
  среда, 25
CLS (Common Language Specification), общая языковая спецификация, 18, 54, 292
  JIT-компиляторы, 57
  верификаторы, 56
  загрузчики классов, 55
CLSID (Class identifier), идентификатор класса в COM, 292
COBOL, язык, 290
codebehind, атрибут директивы WebService, 238
COFF (Common Object File Format), общий объектный формат файлов, 29, 292
COM (Component Object Model), компонентная объектная модель, 14, 16, 66, 119, 292
  COM+ Services, 99
  разработка Windows-приложений, 255
COM Callable Wrapper, см. CCW

COM Interop (COM interoperation), возможность взаимодействия с COM, 292
COM+ Explorer (Component Services Explorer), проводник COM+, 103
COM+ Services, службы COM+, 99
  безопасность на уровне ролей, 108
  поддержка транзакций, 103
  программирование на основе атрибутов, 100
Commerce Server 2000, серверный продукт, 15
Common Intermediate Language, см. CIL
Common Language Infrastructure, см. CLI
Common Language Runtime, см. CLR
Common Language Specification, см. CLS
Common Object File Format, см. COFF
Common Type System, см. CTS
CommonAppDataRegistry, свойство класса Application, 261
Component Pascal, язык, 290
Component Services Explorer, см. COM+ Explorer
config-файлы, 291
connectionString, метод класса SessionState, 249
Console, класс, использование метода WriteLine(), 27
Context, свойство класса Control, 205
contractRef, элемент, 172
@Control, директива, 215
Control, класс, 205
Controls, свойство класса Control, 205
cookie, управление состояниями сессий, 248
Cookieless, свойство класса SessionState, 249
CPModel, класс, 63
CreateChildControls, метод, 206, 235
CreateHtmlTextWriter, класс UserControl, 209
CreateHtmlTextWriter, метод, 209
CreateWindow, функция Windows API, 254
Cryptography, дочернее пространство имен, 64
csc, компилятор, 41
csc.exe (компилятор C#), 304, 307
.ctor(), метод, 35
CTS (Common Type System), система общих типов, 18, 49, 292

D

DataAdapter, объект, 118
DataBind(), метод, 220
DataBind, метод, 206
DataColumn, класс, 118
DataColumn, объект, 132
DataGrid, элемент управления, 243
DataList, элемент управления, 245
DataReader, объект, 139
DataRelation, класс, 118, 134
DataRow, класс, 118
DataSet, класс, 118, 123, 271
  XML, 148
  создание в C#, 124
DataSet, объект
  XML, 126
DataSetView, класс, 271
DataTable, класс, 118, 131, 271
Datatype Document, см. DTD
DataView, класс, 133, 241, 271
DateTime, тип данных, 297
DB (database), 292
DbDataAdapter, иерархия, 142
DCOM (Distributed Component Object Model), объектная модель распределенных компонентов, 18, 95, 292
  веб-службы, 164
  разработка Windows-приложений, 255
Deactivate(), метод, 106, 107
Decimal, тип данных, 297
definitions, элемент WSDL, 167
Delete(), метод, 111
DELETE, SQL-команда, обновление данных, 143
Delta Forth, язык, 290
Description, свойство, 178
DHCP (Dynamic Host Configuration Protocol), протокол, 99
DHTML (Dynamic HTML), динамический HTML, 217, 224, 292
DictionaryBase, класс, 64
Digest Authentication, краткая аутентификация, 200
Directory, класс, 64

.disco, файлы открытия, 172
DISCO (Web Service Discovery), открытие веб-службы, 292
discoveryRef, элемент, 172
DispatchMessage, функция Windows API, 254
DISPID (Dispatch identifier), диспетчерский идентификатор, 292
Distributed Component Object Model, см. DCOM
DLL (Dynamic Link Library), библиотека динамической компоновки, 292
DNA (Distributed interNet Applications), распределенные сетевые приложения, 119, 292
Dns, классы, 64
DoEvents, метод класса Application, 261
DOM (Document Object Model), объектная модель документа, 292
  XmlDocument, генерация, 151
Double, тип данных, 297
drag-and-drop в графических веб-интерфейсах, 23
DTD (Datatype Document), документ типа данных, 293
dumpbin.exe, утилита просмотра PE-файлов, 30, 304, 309, 310
Dynamic Host Configuration Protocol, см. DHCP
Dynamic HTML, см. DHTML
Dynamic Link Library, см. DLL

E

EditItemStyle, управление визуальным форматированием DataGrid, 244
EditItemTemplate, настраиваемый шаблон, 245
Eiffel#, язык, 290
Enabled, свойство элемента управления, 257
EnableSession (boolean), свойство, 178
EnableViewState, флаг класса Control, 205
.entrypoint, директивы, использование метода Main(), 48
Equals(), метод, 62
ErrorPage, свойство, 208, 209
EventArgs, параметр, 265
Exception, класс, 64
Exchange Server 2000, серверный продукт, 15
EXE, формат исполняемых файлов, 293
  построение с учетом межъязыковой интеграции, 83
ExecuteNonQuery(), метод, выполнение команд, 139
ExecuteReader(), метод, выполнение команд, 139
Exit, метод класса Application, 261
ExitThread, метод класса Application, 261
extends, ключевое слово в C#, 76
Extensible Markup Language, см. XML
Extensible Stylesheet Language Transformation, см. XSLT
Extensible Stylesheet Language, см. XSL

F

file, HTML-тег, 212
FileStream, класс, 64
Fill(), метод, 143, 271
Focus, метод элемента управления, 257
FooterStyle, управление визуальным форматированием DataGrid, 244
FooterTemplate, настраиваемый шаблон, 245
Form, класс, 258, 262, 274
  LayoutMdi(), метод, 280
Fortran, язык, 290
Framework (.NET), инфраструктура разработки
  цели разработки, 16

G

GAC (Global Assembly Cache), глобальный кэш сборок, 19, 43
  просмотр, 91
  разделяемые компоненты, 87
gacutil.exe, утилита регистрации сборок, 304, 305
__gc, ключевое слово C++, 66
GC, класс, 64
GDI (Graphical Device Interface), интерфейс графического устройства, 293
GetAuthors(), метод, 178

GetBooks(), метод, 178
GetChanges(), метод, 123, 130
GetHashCode(), метод, 62
GetMessage, функция Windows API, 254
GetObject(), метод, 98
GetType(), метод, 62
GET-метод протокола HTTP, 166
Global Assembly Cache, см. GAC
Graphical Device Interface, см. GDI
GUI (Graphical User Interface), графический пользовательский интерфейс, 23
GUID (Globally Unique Identifier), глобально уникальный идентификатор, 35, 293
Guid, тип, 297

H

HailStorm, набор веб-служб, 16, 163
HasChanges(), метод, 123, 130
HasErrors, метод, 123
HashTable, класс, 241, 298
Hashtable, класс, 64
Haskell, язык, 290
HeaderStyle, управление визуальным форматированием DataGrid, 244
HeaderTemplate, настраиваемый шаблон, 245
Height, свойство элемента управления, 257
hidden, атрибут, 212
Hide, метод элемента управления, 257
hidebysig, атрибут, 48
Host Integration Server 2000, серверный продукт, 15
HTML (Hypertext Markup Language), язык разметки гипертекста, 293
  ASP-страницы, 202
    пространство имен System.Web.UI.HtmlControls, 210
  DataList, формирование списков, 245
  Repeater, элемент управления, 246
  жизненный цикл веб-формы, 226
  компоненты веб-форм, 222
  синтаксис элементов управления, 214
  стандартная гиперссылка, 213
  стандартная таблица, 213
  стандартное поле ввода текста, 213
  стандартный флажок, 213
HtmlAnchor, объект, 213
HtmlButton, объект, 213
HtmlControls, класс, 213, 226
HtmlForm, объект, 213
HtmlGenericControl, объект, 211, 213
HtmlImage, объект, 212
HtmlInputButton, объект, 213
HtmlInputCheckBox, объект, 213
HtmlInputFile, объект, 212
HtmlInputHidden, объект, 212
HtmlInputImage, объект, 212
HtmlInputRadioButton, объект, 213
HtmlInputText, объект, 211, 213
HtmlSelect, объект, 213
HtmlTable, объект, 213
HtmlTableCell, объект, 213
HtmlTableRow, объект, 213
HtmlTextArea, объект, 213
HTTP (Hypertext Transfer Protocol), протокол передачи гипертекста, 293
HTTP GET, метод, 166
  для ASP, 203
  клиенты, 180
HTTP POST, метод, 166
  для ASP, 203
  клиенты, 183
  не-.NET клиенты, 187
HttpWebRequest, классы, 64
Hypertext Markup Language, см. HTML

I

ID, свойство класса Control, 205
IDE (Integrated Development Environment), интегрированная среда разработки, 16, 293
IDL (Interface Definition Language), язык определения интерфейсов, 100, 293
IE (Internet Explorer), броузер, 293
IID (Interface Identifier), идентификатор интерфейса, 293
IJW (It Just Works), «заклинание», 293
IL (Intermediate Language), промежуточный язык, 41, 46, 293
  JIT-компиляторы, 57
ilasm.exe, ассемблер MSIL, 304, 306
ildasm.exe, дизассемблер MSIL, 34, 293, 304, 307
img, тег HTML, 212

implements, ключевое слово в C#, 76
Implements, ключевое слово в VB.NET, 70
@Import, директива, 215
Import, ключевое слово VB.NET, 28
in, атрибут, 100
Init, событие, 225
InnerHtml, свойство, 211
Inproc (In-Process), сокращение, 293
INSERT, SQL-команда, обновление данных, 143
InsertAfter(), формирование дерева XML-документа, 150
InsertBefore(), формирование дерева XML-документа, 150
instance, атрибут для объявления .method, 76
Int16, тип данных, 297
Int32, тип данных, 298
Int64, тип данных, 298
Integrated Windows Authentication, интегрированная аутентификация Windows, 200
__interface, ключевое слово в C++, 66
Interface Definition Language, см. IDL
Interface Identifier, см. IID
interface, атрибут IL-объявления .class, 75
Interface, ключевое слово VB.NET, 70
Intergrated Development Environment, см. IDE
Intermediate Language, см. IL
Internet Explorer, см. IE
Internet Security and Acceleration Server 2000, серверный продукт, 15
Invalidate, метод элемента управления, 257
IPAddress, классы, 64
ISAPI (Information Server Application Programming Interface), прикладной интерфейс сервера IIS, 293
IsCallerInRole(), метод, 103
IsInTransaction, свойство, 103
IsMdiChild, свойство, 274
IsMdiContainer, свойство, 274
IsPostBack, свойство класса Page, 208
IsValid, класс UserControl, 209
ItemStyle, управление визуальным форматированием DataGrid, 244
ItemTemplate, настраиваемый шаблон, 245

J

Java Virtual Machine, см. JVM
JavaScript, использование в ASP, 203
JIT-компиляторы, использование метаданных, 33
JScript, язык сценариев, 289
JVM (Java Virtual Machine), виртуальная машина Java, 22, 46

K

– k, опция генерирования/сохранения информации о ключах, 88
KeyDown, событие элемента управления, 258
KeyPress, событие элемента управления, 258
KeyUp, событие элемента управления, 258

L

Label, элемент управления Windows, 266
language, атрибут директивы WebService, 238
LayoutMdi(), метод, 274, 280
ldstr, инструкция, использование метода Main(), 48
Lease, вызов метода Renew(), 99
Left, свойство элемента управления, 257
.locals, директива в C#, 78
Load, событие, 225
Load, метод, 152
LoadControl, метод, 208, 233
LoadPageStateFromPersistenceMedium, метод класса Page, 209
LoadViewState, метод класса Control, 207
LoadXml, метод, 152

M

Machine.cfg configuration file, 293
Main(), 27, 35, 48
  кодирование на VB.NET, 71
MainApp, класс, 48
MainMenu, класс, 275
Manager, роль, 110
MapPath, метод класса Page, 209

MarshalByRefObject, класс, 96
.maxstack, директивы, использование метода Main(), 48
MBR (MarshalByReference), маршалинг по ссылке, 96, 293
MDI (Multiple Document Interface), многодокументальный интерфейс, 274
MdiChildren, свойство, 274
MdiParent, свойство, 274
MemoryStream, класс, 64
MenuItem, класс, 275, 278
Mercury, язык, 290
Merge(), метод, 130
MergeOrder, свойство, 277, 281
Message, сообщения от веб-методов, 167
MessageName, свойство, 178
MessageQueue, класс, 115
MethodInfo, класс, 64
MFC (Microsoft Foundation Classes), библиотека классов, 18, 255, 293
Microsoft
  .NET, обзор, 13
  Passport, служба аутентификации, 162
Microsoft Common Object File Format, см. COFF
Microsoft Foundation Classes, см. MFC
Microsoft Intermediate Language, см. MSIL
Microsoft Message Queuing, см. MSMQ
Microsoft Passport, веб-служба, 16
Microsoft Transaction Server, см. MTS
Microsoft Visual C++ Runtime, см. MSVCRT
Microsoft's eXtensible Markup Language, см. MSXML
mode, свойство класса SessionState, 249
Module, класс, 64
Module, ключевое слово VB.NET, 28
Mondrian, язык, 290
MouseDown, событие элемента управления, 258
MouseMove, событие элемента управления, 258
MouseUp, событие элемента управления, 258
MouseWheel, событие элемента управления, 258
mscorlib, сборка, 49
MSDAORA, драйвер, 138
msdata:PrimaryKey, атрибут XML, 129
MSI (Microsoft Windows Installer Package), пакет инсталляции для Microsoft Windows, 293
MSIL (Microsoft Intermediate Language), промежуточный язык Microsoft, 293
MSIL, дизассемблер (ildasm.exe), 34, 293
MSMQ (Microsoft Message Queuing), служба очереди сообщений, 113
MSN Messenger Service, программа для обмена сообщениями, 163
MSVCRT (Microsoft Visual C++ Runtime), библиотека времени выполнения Microsoft Visual C++, 294
MSXML (Microsoft's eXtensible Markup Language), анализатор XML от Microsoft, 294
MTS (Microsoft Transaction Server), сервер транзакций, 99, 294
Multiline, свойство, 283
Multiple Document Interface, см. MDI
MustInherit, ключевое слово в VB.NET, 70
MustOverride, ключевое слово в VB.NET, 70
Mutex, класс, 64
My Services (.NET), пакет служб, 163
MyTransactionVote, свойство, 103

N

name, атрибут тега assemblyIdentity, 94
.namespace, IL-декларация, 75
NAT (Network Address Translation), трансляция сетевых адресов, 18
  использование DCOM, 95
NDR (Network Data Representation), представление сетевых данных, 18, 120
.NET Framework
  архитектура, 22
  обзор, 13
  платформа, 15
  цели разработки, 16
.NET My Services, пакет служб, 163
Network Address Translation, см. NAT
Network Data Representation, см. NDR

newslot, атрибут для объявления .method, 76
NextRecordset(), метод, 123
NGWS (Next Generarion Windows Services), новое поколение служб Windows, 294
NTFS (NT Filesystem), файловая система, 294
N-Tier, сокращение, 294
NTLM (NT Lan Manager), протокол аутентификации, 294

O

Oberon, язык, 290
Object Windows Library, см. OWL
ObjectPooling, свойство, 106
OBJREF (Object Reference), ссылка на объект, 294
OLE DB, управляемый поставщик, 137
OleDbCommand, свойство объекта Connection
  в управляемом поставщике OLE DB, 137
  выполнение команд, 139
OleDbConnection, объект, в управляемом поставщике OLE DB, 137
OleDbDataAdapter, класс, в иерархии класса DataAdapter, 142
OleDbDataAdapter.Update(), метод, изменение базы данных, 143
OleDbParameter, объект, в управляемом поставщике OLE DB, 137
OpenHandler, обработчик события, 280
OrderID, поле, 125
out, атрибут, 100
output, объект HtmlTextWriter, 235
@OutputCache, директива, 217
OWL (Object Windows Library), библиотека объектов Windows, 255

P

P/ Invoke (Platform Invoke), средство вызова функций Windows API, 294
@Page, директива, 215
Page, класс, 207
  события веб-формы, 225
Parent, свойство класса Control, 205, 257
ParentRelations, свойство объекта DataTable, 132
Passport, служба аутентификации, 162, 180
Path, класс, 64
PE (Portable Executable), переносимый исполняемый файл, 25, 29, 294
  компиляция файлов с помощью JIT-компиляторов, 57
Perl, язык, 290
Permissions, дочернее пространство имен, 64
plug-and-play, 17
Policy, дочернее пространство имен, 64
Port, конечная точка веб-служб, 167
Portable Executable, см. PE
portType, операции, 167
portType, элемент WSDL, 168
POST-метод протокола HTTP, 166
pragma, директива компилятора, 29
PreJIT-компиляторы, 58
PrependChild(), формирование дерева XML-документа, 150
PreRender, событие, 225
PrimaryKey, свойство, 132
Principal, дочернее пространство имен, 64
private, атрибут, 76
private, ключевое слово, указание компилятору C#, 72
ProcessCmdKey, метод элемента управления, 258
public, ключевое слово, указание компилятору C#, 72
publicKeyToken, атрибут тега assemblyIdentity, 94
Python, язык, 255, 290
p-код, уровень языковой абстракции, 46

Q

Query Analyzer, загрузка и запуск instpubs.sql, 141
Queue, класс, 64, 241, 298, 302

R

RAD (Rapid Application Development), быстрая разработка программ, 69, 122, 294

Rapid Application Development, см. RAD
RCW (Runtime Callable Wrapper), вызываемый упаковщик времени выполнения, 294
Read(), метод, 154
ReadXml(), метод, 126
ReadXmlData(), метод, 127
ReferenceEquals(), 62
Reflection, пространство имен, 89
Refresh, метод элемента управления, 257
REGASM, regasm.exe, утилита регистрации сборок, 37, 294
@Register, директива, 216
  пейджлеты, 230
  производные элементы управления, 236
RegisterClass, функцияWindow API, 254
RegisterWellKnownServiceType(), метод, 97
RejectChanges(), метод, 123
Relations, отношения между таблицами базы данных, 123
Remote Procedure Calling, см. RPC
RemoveMessageFilter, метод класса Application, 261
Render, метод класса Control, 207
Renew(), метод, 99
Repeater, элемент управления, 246
Right, свойство элемента управления, 257
Rogue Wave, фирма, 255
RPC (Remote Procedure Calling), удаленный вызов процедур, 294
RPG, язык, 290
Run, метод класса Application, 261
runat, атрибут, 211, 218
Runtime Callable Wrapper, см. RCW

S

SavePageStateToPersistenceMedium, метод класса Page, 209
SaveViewState, метод класса Control, 207
SAX (Simple API for XML) спецификация для анализатора, 148
SByte, тип данных, 298
Scheme, язык, 290
SCL (SOAP Contract Language), язык контрактов SOAP, 294
SDI (Single Document Interface), однодокументный интерфейс, 274
SDK (Software Development Kit), набор разработчика ПО, 294
SEH (Structured Exception Handling), структурная обработка исключений, 294
Select, метод элемента управления, 257
SelectedItemStyle, управление визуальным форматированием DataGrid, 244
SelectedItemTemplate, настраиваемый шаблон, 245
Send(), метод, 113
SeparatorTemplate, настраиваемый шаблон, 245
ServerChange, событие веб-формы, 226
ServerClick, событие веб-формы, 226
Service, коллекция сетевых конечных точек, 167
ServicedComponent, класс, 103
SessionState, свойства класса, 249
SetAbort(), метод, 103
SetComplete(), метод, 103
SetDefault, значение правила ForeignKeyConstraint, 133
SetNull, значение правила ForeignKeyConstraint, 133
Shell Cache Viewer, расширение оболочки, 91
  добавление новых версий компонентов, 93
Show, метод элемента управления, 257
ShowWindow, функция Windows API, 254
SID (Security Identifier), идентификатор безопасности, 108
Simple API for XML, см. SAX
Simple Mail Transfer Protocol, см. SMTP
Simple Object Access Protocol, см. SOAP
Single Document Interface, см. SDI
Single, тип, 298
Size, свойство элемента управления, 257
Smalltalk, язык, 290
SMTP (Simple Mail Transfer Protocol), простой протокол передачи почты, 294

sn.exe, утилита разделяемых имен, 304, 312
SOAP (Simple Object Access Protocol), простой протокол доступа к объектам, 166, 294
SOAP Contract Language, см. SCL
Software Development Kit, см. SDK
SortedList, класс, 241, 298, 302
SQL (Structured Query Language), язык структурированных запросов, 23
SQL Server, 15, 294
  режим управления состоянием сессии, 253
  управление состояниями сессий, 250
SQL, управляемый поставщик, 137
  SqlClient-объекты, 137
SqlCommand, свойство объекта Connection, выполнение команд, 139
sqlConnectionString, свойство класса SessionState, 249
SqlDataAdapter в иерархии объекта DataAdapter, 142
SQLOLEDB, драйвер, 138
SQL-команда SELECT, 143
Stack, класс, 64, 241, 298, 303
Standard JIT, оптимизированный двоичный код, 294
Standard ML, язык, 290
Standard Template Library, см. STL
StartupPath Application, свойство класса, 261
STL (Standard Template Library), 294
String, класс, 64, 298
Structured Exception Handling, см. SEH
Structured Query Language, см. SQL
System, пространство имен, 64
System.Attribute, создание специальных атрибутов, 100
System.Collections, пространство имен, 64
System.Data, пространство имен, 64, 243
System.Data.Common, пространство имен для управляемых поставщиков, 137
System.Data.OleDb, пространство имен, 243
System.IO, пространство имен, 64
System.Messaging, пространство имен, 113
System.Net, пространство имен, 64
System.Object, 35, 48, 62
System.Object, корневой класс, 18
System.Reflection, пространство имен, 64
System.Runtime, пространство имен, 96
System.Runtime.Remoting, пространство имен, 97
System.Runtime.Remoting.Channels.Tcp, пространство имен, 96, 97
System.Security, пространство имен, 64
System.Threading, пространство имен, 64
System.Web.Services, пространство имен, 64
System.Web.UI, пространство имен, 64, 205
  Control, класс, 205
  Page, класс, 207
  сравнение с пространством имен System.Windows.Forms, 256
System.Web.UI.HtmlControls, пространство имен, 210, 229
System.Web.UI.WebControls, пространство имен, 213, 229
System.Windows.Forms, пространство имен, 64

T

Tables, таблицы базы данных, 123
TCP (Transport Control Protocol), транспортный управляющий протокол, 294
TextBox, элемент управления Windows, 266
Thread, класс, 64
ThreadPool, класс, 64
Timeout, свойство класса SessionState, 249
tlbexp.exe, экспортер библиотек типов, 37, 304
tlbimp.exe, импортер библиотек типов, 304, 310
tlb-файлы, 37, 295
TMT Pascal, язык, 290
Top, свойство элемента управления, 257

ToString(), метод, 62
Trace, свойство класса Page, 208
TransactionOption, свойство, 179
TranslateMessage, функция Windows API, 254
Transprot Control Protocol, см.TCP
try-блоки
  в C#, 82
  в C++, 68
Types, контейнер типов в структуре WSDL, 167
types, элемент WSDL, 168

U

UDDI (Universal Description, Discovery, and Integration), независимая от платформы среда описания и открытия веб-служб, 173, 295
UDF (Uniform Data Format), унифицированный формат данных, 295
UI (User Interface), пользовательский интерфейс, 295
UInt16, тип данных, 298
UInt32, тип данных, 298
UInt64, тип, 298
Uniform Data Format, см. UDF
Uniform Resource Identifier, см. URI
Uniform Resource Locator, см. URL
UniqueConstraint, ограничение на данные, 132
Universal Description, Discovery, and Integration, см. UDDI
Universal Runtime, см. URT
Unload, событие, 225
UPDATE, SQL-команда, обновление данных, 143
Update, метод элемента управления, 258
URI (Uniform Resource Identifier), унифицированный идентификатор ресурса, 295
URL (Uniform Resource Locator), унифицированный локатор ресурса, 295
URT (Universal Runtime), универсальная библиотека времени выполнения, 295
User, класс UserControl, 209
UserAppDataRegistry, свойство класса Application, 261
Users, группа пользователей, и безопасность на уровне ролей, 108
using namespace, оператор C++, 27

V

Validate, класс UserControl, 209
Validate, метод класса Page, 209
Validators, класс UserControl, 209
Validators, свойство класса Page, 208
value, ключевое слово, 52
VB (Visual Basic), 295
VB.NET (Visual Basic.NET), 28, 289
  код, 69
  межъязыковая интеграция, 80
vbc, компилятор VB, 28, 41, 304, 308
VBRUN (Visual Basic Runtime), библиотека времени выполнения VB, 295
VBScript, использование в ASP, 203
VES (Virtual Execution System), подмножество CLR, 295
ViewState, свойство класса Control, 205
Virtual Object System, см. VOS
Visible, свойство класса Control, 205, 257
Visual Basic, см. VB
Visual Basic Runtime, см. VBRUN
Visual Studio .NET, см. VS.NET
Void, тип данных, 298
VOS (Virtual Object System), виртуальная объектная система, 295
vptr, указатель на vtbl, 68
VS.NET (Visual Studio.NET), интегрированная среда разработки, 16, 295
  ADO.NET, 204
    и наследование классов, 122
  использование веб-служб, 240
vtbl, таблица указателей, 68

W

WAP (Wireless Application Protocol), протокол беспроводных приложений, 295
Web Forms, 295
@WebService, директива, 238
Web Service Discovery, см. DISCO
Web Services, среда, 164, 295
Web Services Description Language, см. WSDL

web.config, файлы, 179
настройки, 208
управление сессиями, 248
WebControl, объект, 213
WebMethod, атрибут, 239
WebMethodAttribute, класс пространства имен System.Web.Services, 174
WebService, класс пространства имен System.Web.Services, 174
WebServiceAttribute, класс пространства имен System.Web.Services, 174
Width, свойство элемента управления, 257
Win32, 295
Extensions (Python), 255
Windows 2000, 15
безопасность на уровне ролей, 112
загрузчики классов, 55
средства безопасности, 21
форматы PE-файлов, 31
Windows 98
загрузчики классов, 55
форматы PE-файлов, 31
Windows CE, 15
Windows Forms, 254
Control, 256
Form, класс, 258
веб-службы, 287
приложение, 262
разработка приложений, 262
частичное развертывание, 284
элементы управления, 264
Windows Me, 15
загрузчики классов, 55
форматы PE-файлов, 31
Windows NT, средства безопасности, 21
Windows XP, 15
Windows, аутентификация, 180
WinForms, 296
Wireless Access Protocol, см.WAP
Wireless Markup Language, см.WML
WM_CLOSE, сообщение, 254
WM_CREATE, сообщение, 254
WM_SIZE, сообщение, 254
WML (Wireless Markup Language), беспроводной язык разметки, 296
WndProc, метод элемента управления, 258
WriteLine(), метод, 27, 49
WriteXml(), метод, 126
WriteXmlSchema(), метод, 126
WSDL (Web Service Description Language), язык описания веб-службы, 15, 164, 166, 171, 183, 296
wsdl.exe, утилита веб-служб, 304, 313

X

XML (Extensible Markup Language), расширяемый язык разметки, 117, 296
DataSet, объект, 126, 148
Schema Definition (XSD), 296
анализаторы, 148
генерация в C#, 73
классы, 149
наборы данных, 148
определение схемы XSD, 15, 39
XmlAttribute, тип узла, 149
XmlDocument, базовый класс, 151
XmlDocument, тип узла, 149
XmlElement, тип узла, 149
XmlNamedNodeMap, объект-коллекция, 151
XmlNode, базовый класс, 149
XmlNodeList, объект-коллекция, 151
XmlNodeReader, объект, 153
XmlReader, объект, 153
XmlText, тип узла, 149
XmlTextReader, объект, 153
XmlWriter, объект, 155
XPath (XML Path), 296
XSD (XML Schema Definition), определение XML-схемы, 15, 39, 296
классы/DataSet, 312
xsd.exe, утилита определения XML-схем, 304, 311
XSL (Extensible Stylesheet Language), расширяемый язык таблиц стилей, 296
XSLT (Extensible Stylesheet Language Transformation), язык преобразования расширяемых таблиц стилей, 296
XslTransform, класс, 155

Z

zApp, библиотека классов, 255

А

абстрактные базовые классы (abstract base classes, ABC), 53
абстракция, концепция проектирования программного обеспечения, 46
анализаторы XML, 148
архитектура
  ADO.NET, 118
  среды .NET, 22
ассемблер MSIL (ilasm.exe), 304, 306
атрибуты
  COM+, программирование, 100
  поддержка транзакций, 103
  проверка, 101
аутентификация
  с помощью клиентских сертификатов, 200
  различные режимы, 179

Б

базовая аутентификация, 200
  поверх SSL, 200
байткод, уровень языковой абстракции, 46
безопасность, 59
  .NET Framework, 21
  веб-службы, 199
  на уровне ролей, 108, 110
  свойств в утилите администрирования IIS, 200
библиотеки типов, 32
  tlb-файлы, 37
блоки генерации кода, 218
блоки объявления кода, 217
брандмауэры, использование DCOM, 95
броузеры, генерация ASP-страниц, 203
быстрая разработка программ (RAD), 69

В

веб-интерфейсы, графические GUI, 23
веб-методы, 175
веб-службы, 161
  ASP.NET, 238, 241
  Microsoft, 16
  Windows Forms, 287
  WSDL, 166, 171
  безопасность, 199
  использование, 240
  клиенты, 180
  открытие, 171
  поставщики, 175, 180
  пространства имен, 174
  форматы связи, 164
веб-формы, 202
  ASP-NET, 204
  System.Web.UI, пространство имен, 205
  жизненный цикл, 226
  компоненты, 222
  конфигурирование, 228
  отображение, 226
  разработка приложения, 222
  серверные элементы управления, 229
  синтаксис, 214
  события, 225
  специальные серверные элементы управления, 230
верификаторы, 56
верификация кода, 56
версии компонентов
  добавление новых, 92
взаимодействие кода, 59
взаимодействие компонентов
  поддержка, 37
визуальное наследование, 273
визуальные элементы стандартных форм, 259
виртуальная Java-машина, 46
внепроцессное управление состоянием сессии, 249, 253
возможность взаимодействия, ADO.NET,119
выражения для привязки данных, 220

Г

глобально уникальный идентификатор, см. GUID, 35
графический пользовательский интерфейс, см. GUI, 23

Д

данные, 117
  метаданные, 31
делегаты, 53

динамические сборки, 41
директивы, 215
дизассемблер MSIL (ildasm.exe), 304, 307
диспетчер
  аренд, 99
  кода, 58

З

заголовок в стандартных формах, 258
загрузчики классов, 55
  использование метаданных, 33
закрытые
  компоненты, 85
  сборки, 41
зарегистрированные сборки, 87
значащие типы, 49

И

идентификация (уникальная), 39
идентичность кода, 42
импортер библиотек типов (tlbimp.exe), 304, 310
индексаторы
  поддержка в CLR, 51
инкапсуляция
  как концепция программирования, 65
инструментарий, использование метаданных, 33
интеграция языковая, 18, 78, 79
  C#, 81
  VB.NET, 80
интегрированная аутентификация Windows, 200
интегрированная среда разработки, IDE, 16
интерфейсы
  как концепция программирования, 65
  поддержка в CLR, 53
исполнение одновременное, 42
исполняемые файлы (CLR), 26

К

каталоги, хранение закрытых компонентов, 85
классы
  Control, 205
  Page, 207
  XML, 149
  конвертация из XML-схем, 38
  межъязыковая интеграция, 78
  определение в C#, 72
  поддержка в CLR, 51
клиенты (веб-служба)
  HTTP GET, 180
  не-.NET, 187
кнопки в HTML, 213
команда-запрос (тип выполнения), 139
команды, объекты, 118
комментарии (XML), 74
компиляторы, 41
  C# (csc.exe), 304, 307
  C++ (cl.exe), 304, 307
  JIT-компиляторы (Just-in-Time), 57
  VB (vbc.exe), 304, 308
компонентная инфраструктура среды .NET, 16
компонентное представление, 14
компоненты, 84, 122
  веб-формы, 222
  добавление новых версий, 92
  закрытые, 85
  и сборки, 39
  очереди сообщений, 113
  распределенные, 95
  сущностей, 118
    DataRelation, 134
    DataSet, 123
    DataTable, 131
    DataView, 133
контейнеры, 256, 275, 298
конфигурационные файлы (.cfg), 56
конфигурация на уровне машины
  управление сессиями, 248
конфигурация на уровне приложения
  управление сессиями, 248
концепции программирования, 65
корпоративные службы, 14
краткая аутентификация, 200

М

манифесты, 20, 39
  IL-код, 41
  исполнение одновременное, 42
  контроль версий, 41
  метаданные сборки, 44
  развертывание, 42
  разделение и повторное использование кода, 43

масштабируемость, 247
ADO, 120
ADO.NET, 119
межъязыковая отладка в VS.NET, 16
метаданные, 31
манифесты, 44
поддержка взаимодействия компонентов, 37
получение, 34
просмотр и генерация, 35
методы
Control, класс , 206
Page, класс, 207
определение в C#, 72
многомодульные сборки, 45
многострочное поле ввода текста, 213

Н

наборы записей, 123
надежность среды .NET, 20
наследование
как концепция программирования, 65
номер
построения, добавление новых версий компонентов, 92
ревизии, добавление новых версий компонентов, 93

О

обработка
исключительных ситуаций, 59
как концепция программирования, 65
событий веб-формы, 228
обработчики
ошибок, механизмы, 21
событий
привязка к элементам управления Windows, 265
общая модель программирования, 61
общие сборки, 41
объекты
классы, 62
параметры, 265
пул, 106
удаленные, 96
одномодульные сборки, 45
ООП (объектно-ориентированное программирование), 140
операционная система (ОС), 15
определение схем (см. XSD), 15
опубликованные сборки, 87
основные сокращения, 291
открытие веб-служб, 171
открытые ключи, 40
открытые методы класса Object, 62
отношения между таблицами в базе данных, 132
очереди сообщений, 113
постановка в очередь, 113
удаление из очереди, 115

П

пейджлеты, 230
платформа .NET, 15
поддержка отладки, 59
поддержка и управление исполнением, 58
поле со списком, 213
полиморфизм, 65, 79
пользовательский интерфейс, 222
поставщики веб-службы, 175, 180
постановка в очередь, 113
потоки (XML-файлы), 148
привязка данных, 241, 266
приложения, разработка в ASP.NET, 222
программа просмотра PE-файлов (dumpbin.exe), 304, 309, 310
производительность ADO.NET, 119, 121, 122
производные элементы управления, 234
пространства имен, 63
системные веб-службы, 174
пул объектов, 106

Р

развертывание
в нескольких каталогах, 86
в одном каталоге, 85
в среде .NET, 84
приложений в среде .NET, 19
разделяемые сборки, 40, 41, 90
добавление новых версий, 92
создание компонентов, 88
распаковка (объекты ссылочного типа), 51

распределенные
вычисления, 14
распределенные компоненты, 95
написание приложения Hello, 95
редактор XML-схем, 16

С

C#, язык
DataSet, создание, 124
C++, язык, 66
сборка мусора, 21, 59
распределенная, 99
сборки, 39
IL-код, 41
и компоненты, 39
исполнение одновременное, 42
использование, 46
контроль версий, 41
манифесты, 44
общие, 41
развертывание, 42
разделяемые, 40
разделение и повторное использование кода, 43
создание, 45
свойства
определение в C#, 72
поддержка в CLR, 51
серверные элементы управления, 204, 229
серверы
ADO.NET, масштабируемость, 120
распределенное приложение Hello, 95
синтаксис веб-форм, 214
системная безопасность, 199
событийное программирование, 237
соединения
сервер, организация пула в ADO.NET, 120
объекты, 118
сокращения, 291-296
составные элементы управления, 260
специальные атрибуты, разработка/использование, 100
специальные серверные элементы управления, 230
пейджлеты, 230
производные, 234
синтаксис, 219
спонсор, 99
среда (CLR), 25
ссылочная сборка, идентификация, 44
ссылочные типы, 50
статические сборки, 41
статическое открытие, 172
стили, управление визуальным форматированием DataGrid, 244

Т

таблица, 235
теги серверных объектов, 221
типы данных, 297

У

удаление из очереди, 115
удаленные объекты, 96
узлы (XML), 148
уникальная идентификация, 39
упаковка (объекты ссылочного типа), 50
управление
доступом к коду, 42
состоянием, 247
управляемые поставщики
для ADO.NET, 118, 136
соединение, 138
управляющие элементы, см. элементы управления
уровни языковой абстракции, 46
утилита
веб-служб (wsdl.exe), 304, 313
генерации сборок (al.exe), 304
определения XML-схем (xsd.exe), 304, 311
разделяемых имен (sn.exe), 304, 312
регистрации сборок (gacutil.exe), 304, 305

Ф

факторы зрелости .NET, 15
файлы открытия, 180
форматы связи (веб-службы), 164
формы
Web/Windows, 23
добавление элементов управления, 264
класс, 274

Ч

частная очередь, 113
читатели данных, объекты, 118

Ш

шаблоны
шаблоны
  DataList, 245
  Repeater, 246
  управление отображением данных, 241

Э

элементы управления, 256, 275
  Control, класс, 256
  добавление на форму, 264
  помещение на страницу, 236
  расстановка, 272
  специальные, помещение на страницу, 236
экспортер библиотек типов (tlbexp.exe), 37, 304

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-054-5, название «Платформа .NET. Основы» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.