

Дэвид С. Платт

Знакомство  
с **MICROSOFT<sup>®</sup>**



Microsoft  
.net

РУССКАЯ РЕДАКЦИЯ

**Microsoft<sup>®</sup>**



*Моей дочери Аннабел Роз Платт*

David S. Platt

Introducing  
**MICROSOFT<sup>®</sup>**  
**.NET**

---

---

**Microsoft** Press



Дэвид С. Платт

# Знакомство с MICROSOFT<sup>®</sup> .NET

Москва 2001

---

 РУССКАЯ РЕДАКЦИЯ

УДК 004  
ББК 32.973.26-018.2  
П45

Платт Д. С.

П45 Знакомство с Microsoft .NET/Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2001. — 240 с.: ил.

ISBN 5-7502-0186-4

Книга Дэвида Платта знакомит читателя с новейшей и многообещающей платформой — Microsoft .NET. В ней доступно описаны особенности архитектуры и компоненты этой системы. Вы узнаете, какие проблемы позволяет решить .NET, какие для этого используются подходы и как начать с ней работать. Книга содержит множество иллюстраций и примеров программ. Описанные здесь компоненты и технологии, такие как .NET Framework, ASP.NET, Web Forms, Web-службы и Windows Forms, позволят нам эффективно создавать программные продукты нового поколения для набирающей обороты платформы Microsoft .NET. Книга состоит из 5 глав и предметного указателя.

УДК 004  
ББК 32.973.26-018.2

Подготовлено к изданию по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.

Macintosh — охраняемый товарный знак компании Apple Computer Inc. ActiveX, BackOffice, JScript, Microsoft, Microsoft Press, MSDN, NetShow, Outlook, PowerPoint, Visual Basic, Visual C++, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, Win32, Windows и Windows NT являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

© Оригинальное издание на английском языке,  
Дэвид С. Платт, 2001

© Перевод на русский язык, Microsoft Corporation,  
2001

ISBN 0-7356-1377-X (англ.)  
ISBN 5-7502-0186-4

© Оформление и подготовка к изданию, издательско-торговый дом «Русская Редакция», 2001

# Оглавление

Вступление .....	IX
Предисловие .....	XI
<b>Глава 1 ВВЕДЕНИЕ В .NET</b> .....	<b>1</b>
Большой Интернет .....	1
Поднимаем планку: общие проблемы инфраструктуры .....	3
Планы, составленные наилучшим образом .....	5
И все же, что это за .NET? .....	7
Об этой книге .....	10
Предостережение: ознакомительное ПО .....	12
Песнь Интернету .....	13
<b>Глава 2 ОБЪЕКТЫ .NET</b> .....	<b>17</b>
Суть проблемы .....	17
Архитектура решения .....	22
Вот так... И почему все это? .....	26
Простейший пример .....	27
Подробнее о пространствах имен .NET .....	34
Сборки .....	38
Понятие сборки .....	38
Сборки и развертывание .....	41
Сборки и управление версиями .....	46
Особенности объектно-ориентированного программирования ...	50
Наследование .....	51
Конструкторы объекта .....	57
Управление памятью в .NET .....	59
Взаимодействие с COM .....	68
Использование объектов COM из программ .NET .....	68
Использование объектов .NET из COM .....	74
Транзакции в .NET .....	76
Структурная обработка исключений .....	79
Безопасность доступа к коду .....	86

<b>Глава 3 ASP.NET</b>	<b>97</b>
Суть проблемы	97
Архитектура решения	101
Простейший пример: создание простой страницы ASP.NET	105
Еще кое-что об управляющих элементах Web	109
Управление и настройка проектов	
Web-приложений: файл Web.config	118
Управление состоянием в ASP.NET	121
Безопасность в ASP	127
Аутентификация	128
Windows-аутентификация	130
Cookie-аутентификация на основе форм	131
Passport-аутентификация	135
Авторизация	140
Идентификационные данные	144
Управление процессом	149
<b>Глава 4 WEB-СЛУЖБЫ .NET</b>	<b>153</b>
Суть проблемы	153
Архитектура решения	158
Простейший пример: создание Web-службы	161
Самоописываемость Web-служб: WSDL-файлы	167
Создание клиента Web-службы	171
Случай 1: HTTP GET	171
Случай 2: HTTP Post	172
Случай 3: необработанный Soap	173
Случай 4: синхронная работа интеллектуального SOAP-прокси	175
Случай 5: асинхронная работа интеллектуального прокси SOAP	178
Поддержка Web-служб в Visual Studio.NET	181
Управление состоянием Web-службы	185

<b>Глава 5 WINDOWS FORMS</b> .....	<b>187</b>
Суть проблемы.....	187
Архитектура решения.....	190
Простейший пример.....	192
Более сложный пример: события и элементы управления.....	195
Создание собственных элементов управления Windows Forms ..	199
Размещение элементов управления ActiveX в приложениях Windows Forms.....	204
<b>Эпилог и благословение</b> .....	<b>209</b>
<b>Предметный указатель</b> .....	<b>211</b>
<b>Об авторе</b> .....	<b>219</b>



# Вступление

Я познакомился с Дэвидом Платтом в 1998 г., когда обдумывал свою книгу о COM. Мой предполагаемый редактор также редактировал книгу Платта «COM and ActiveX Workbook» и дал мне ее экземпляр, чтобы я посмотрел, что публиковалось на эту тему раньше. Я нашел ее замечательной и очень подробной. (На самом деле тогда я в глаза не видел Дэйва, но ведь прочитать что-то книгу — это все равно, что познакомиться с ее автором.)

Прошло несколько лет (а я так и не выпустил ни одной книги о COM и DCOM по причине болезни), и я встретил Дэйва в своем офисе в Microsoft, на сей раз во плоти. К тому времени у меня была возможность прочитать его прекрасную книгу о COM+, чего и вам советую. Я ждал этой встречи! Мне хотелось узнать такой ли он веселый и интересный, как и его последняя книга. Он превзошел мои ожидания.

Стиль Дэйва изменился со времен его ранних книг. В книге о COM+, как и в этой, много весьма непосредственных моментов и читать их очень весело. У него особый литературный стиль, хотя я и догадываюсь, что вы скажете: «Литературный стиль? Да ведь это компьютерные книги!»

Знаете, я иной раз задумывался о классификации книг по программированию и выделил три их типа.

- Основательные книги с большим объемом кода; в них обычно освещается одна конкретная тема. К этой категории относится книга Дэйва о COM и ActiveX. Такие книги отвечают на вопрос «как?»
- Книги с очень глубоким анализом рассматриваемого предмета и небольшим объемом кода, например, книга Дона Бокса (Don Box) о технологии COM. Такие издания отвечают на вопрос «зачем?»

- Книги-размышления. В них дается легкий для понимания обзор предмета и немного кода. При этом они не пытаются объяснить все и вся, как книги первой категории, и не погружают в предмет так глубоко, как книги второй. Их цель почти метафизическая: для понимания предмета вооружить вас подходом, подобным дзэн. Эти книги учат вас, когда задавать вопрос «как?», а когда — «зачем?»

«Знакомство с Microsoft .NET» — безусловно, книга последнего типа. Ее очень приятно читать, она информативна и содержит много интересного кода. Она формирует отношение к .NET и, поверьте, делает это отлично. Чтобы стать специалистом по .NET вам потребуется изменить свой образ мышления. Но это не так уж сложно, и книга, которую вы держите в руках, — первый шаг в этом направлении.

*Кэйт Боллингджер (Keith Ballinger)*  
Менеджер программы .NET Framework  
Microsoft Corporation

P.S. Меня всегда интересовало, что заставляет людей читать вступления. Если с вами это случилось, помогите мне в моих исследованиях, напишите мне по адресу [keithba@microsoft.com](mailto:keithba@microsoft.com). О результатах сообщу.



# Предисловие

Я всегда считал, что продукт, который сейчас называется Microsoft .NET, — это круто. Я помню статьи Мэри Киртлэнд (Mary Kirtland) в ноябрьском и декабрьском номерах *Microsoft Systems journal* за 1997 г. В них описывалось то, что тогда называлось COM+, — среда периода выполнения, предоставляющая разные полезные возможности вроде межъязыкового наследования и контроля доступа в период выполнения. Мне, помешанному на COM, понравились способы, которыми эта среда обещала решить массу бесивших меня проблем COM.

Затем в Microsoft решили, что следующая версия Microsoft Transaction Server будет называться COM+ 1.0 и будет интегрирована с Windows 2000, а то, о чем писала Мэри, будет COM+ 2.0. Позднее Microsoft переименовала COM+ в Microsoft .NET и для всей этой возни я выдумал новый термин — MINFU, Microsoft Nomenclature Foul-Up (Терминологическая Непразбериха Microsoft). Но звучит название продукта все же круто, и я был приятно удивлен, когда в Microsoft меня попросили написать о нем книгу, придерживаясь такого же общего подхода как для COM+ 1.0 в книге *Understanding COM+* (Microsoft Press, 1999). Вот эту книгу вы сейчас и держите в руках (надеюсь, что вы ее купили).

Я боялся, что Microsoft не позволит мне рассказать историю по-своему и заставит придерживаться линии партии. Но этого не произошло. Все, о чем я говорю в этой книге, — понравится вам это или нет — мое личное мнение. Конечно, мне нравится .NET, и я считаю, что она позволит обогатиться ее пользователям — Microsoft с этим не спорит. Когда предполагаемый работодатель просит у вас рекомендацию, чью рекомендацию вы дадите: того, кто считает вас полубогом или того, кто считает вас бараном? Большинство известных мне программистов дают немного тех и других. При обсуждении рукописи один менеджер, которому не понравились мои разглагольствования об административных средствах, написал мне: «По моему мнению, эта книга больше

похожа на беллетристику, чем на руководство к действию. Это входило в ваши планы?» Я ответил: «Я горжусь, что моя книга указывает как на хорошие, так и плохие стороны .NET, как на плюсы, так и на минусы. Иначе я был бы льстецом. По моему скромному мнению, это и позволит ей стать руководством к действию. Если вы обвиняете меня в том, что я называю вещи своими именами, признаю себя виновным».

Терпеть не могу скучного, сухого текста — как сухую рыбу или мясо. Помню, в колледже я попытался сдобрить отчет по лабораторной по химии шуточками и... завалился. «В науке не место легкомыслию, — изрек мой профессор — переделайте все заново. Используйте страдательный залог». У этого парня даже усы были будто нахмурены. Ты что, с наследством пролетел, Филипп? Расслабься — будешь жить дольше или хотя бы веселей! Может, он думал, что если будет скучным, то проживет дольше, но это ему не удалось.

По-моему, лучшие авторы — это хорошие рассказчики, даже в таких и особенно в таких областях, как наука или история. Например, я восхищен «Нашествием чумы» Лори Гарре (Laurie Carrett, *The Coming Plague*, Penguin, 1995), «Сыном Венеры» Ивэна Коннелла (Evan S. Connell, *Son of the Morning Star*, North Point Press, 1997) и биографией Уинстона Черчилля «Последний лев», которую написал Уильям Манчестер (William Manchester, *The Last Lion*, Little Brown, 1983). Вспомните свои учебники, написанные людьми вроде моего бывшего профессора. Что может быть гаже? А теперь прочитайте эту выдержку о поисках Эмилем Ру антитоксина от дифтерии и первых опытах на человеке во время эпидемии в Париже в 1894 г.:

Ру глянул на беспомощных врачей, а затем перевел взгляд на свинцово-серые лица, руки, в страхе сжимающие края одеял, тела, извивающиеся в поисках глотка воздуха...

Ру посмотрел на свой шприц — спасет ли эта сыворотка жизнь?

«Да!» — воскликнул Эмиль Ру-человек.

«Не знаю, нужен эксперимент», — прошептал Эмиль Ру — искатель истины.

«Но для эксперимента нам придется отказать в сыворотке минимум половине этих детей — мы не можем этого сделать». Так сказал Эмиль Ру — человек с большим сердцем — и голоса всех отчаявшихся родителей слились с молящим голосом этого Эмиля Ру.

О том, какой выбор предпочел Эмиль Ру и к чему это привело, вы можете узнать из книги Поля де Крюифа «Охотники за микробами» (Paul de Kruif, *Microbe Hunters*), вышедшую в 1926 г. и периодически переиздающуюся (последний раз в 1996 г. в издательстве Harcourt Brace]. В ней не так много академической точности, но что бы вы предпочли читать? Я знаю, что я предпочел бы писать. Я не претендую на красноречие де Крюифа и всерьез сомневаюсь, что кто-нибудь переиздаст эту книгу, пока мне не стукнет 112 лет. Но я сделал все, что мог, чтобы вы получили удовольствие, а кто из технических авторов хотя бы попытался это сделать?

Книга Поля де Крюифа заканчивается так: «Эта не приукрашенная история была бы не полной, если бы я не сделал признания: я люблю этих охотников за микробами, от старика Антония Левенгука до Пауля Эрлиха. Не за их открытия и благодеяния для рода человеческого. Нет. Я люблю их за то, какие они люди. Не были, а есть — в моей памяти каждый из них жив и будет жить, пока мне служит мой мозг». Как говорится в моем эпилоге (не надо сразу к нему переходить, прочитайте сначала книгу), Интернет заставляет эволюционировать человеческий вид — ни больше, ни меньше. Microsoft .NET — это продукт, который обещает широко распахнуть мир Интернета. И для меня очень много значит, что я могу рассказать о нем и рассказать по-своему. Причина моего переключения на Microsoft Press и ухода от прежнего издателя — возможность работать над проектом с командой и обсуждать будущее технологий. Первые читатели рукописей сказали мне, что смогли пробиться через мой текст. В общем, я на это и надеялся.

Каждая книга — дело коллективное, как полет на Луну, но в меньшем масштабе. Все лавры достаются автору, как и астронавтам (будете следить, как мои акции идут вверх?), но без всех остальных людей, много работавших над этим проектом, и читать было

бы нечего. Благодарностей им достается немного, как и тысячам участников программы «Аполлон» (хотя я думаю, что в фильме «Аполлон 13» облаченный в жилет заядлый курильщик контролер полетов Джин Кранц в исполнении Эда Харриса переплюнул Джима Ловела в исполнении Тома Хэнкса). Пока «Знакомство с Microsoft .NET» не экранизируют (кстати, прекрасная идея, надеюсь, меня слышит кто-нибудь из продюсеров) придется им за свои подвиги довольствоваться этими скромными благодарностями.

Первая премия — Джону Пиэсу (John Pierce), главному редактору этой книги. Много лет назад, в моей книге *Understanding COM+*, изданной Microsoft Press, он играл роль второго плана. Я очень рад, что в этой книге он сыграл главную роль. У нас с ним одинаково извращенное чувство юмора. Подписать его на то, чтобы он держал меня в рамках, — все равно, что пригласить Билла Клинтона присматривать за старшеклассниками из группы поддержки спортивной команды. Я знал, что он не будет корчить мой стиль и ломать мой голос, который — нравится вам это или нет — отличается от остальных. Он отточил мою прозу так, как не смог бы сделать я сам.

Дальше идет Марк Янг (Marc Young), технический редактор. Он находил ответы на все мои технические вопросы, обычно в условиях дефицита времени и ежедневных изменений в коде. Кроме Марка, многие другие члены команды разработки .NET находили окна в своем зверском графике, чтобы направить меня в нужном направлении. Я особенно признателен Сьюзан Уоррен (Susan Warren), Кэиту Болинджеру (Keith Ballinger), Марку Боултеру (Mark Boulter), Лорен Конфелдер (Loren Kohnfelder), Эрику Олсону (Erik Olson), Джону Риварду (John Rivard), Полу Вику (Paul Vick), Джеффри Рихтеру (Jeffrey Richter) и Саре Уильямс (Sara Williams).

С коммерческой стороны процесс был запущен год назад Бэном Райаном (Ben Ryan), а после его ухода подхвачен Даниэлем Бердом (Danielle Bird). Поддержку осуществляла Энни Гамильтон (Anne Hamilton). Тереза Фэган (Teresa Fagan), менеджер продуктов в Microsoft Press, услышав название «Знакомство с Microsoft .NET» подала идею; «Эй, ты должен зарегистрировать это как Web-адрес!» — и я помчался под дождем в свою одинокую писа-

тельскую мансарду (это был этаж для консьержек в отеле Club Hotel в Бельвью) и реализовал эту идею, пока никто другой до нее не додумался. Это была вишенка в водочке с мороженым.

Я также признателен Дюан Бэйкер (Duane Baker) и Крису Бэллу (Chris Bell) из Interland, Inc. за совместную работу по установке Web-сайта, где вы можете увидеть демонстрационные программы к этой книге.

И, наконец, я должен поблагодарить мою жену Линду, теперь мать моей дочери Эннабел.

Дэвид С. Платт

[www.rollthunder.com](http://www.rollthunder.com)

Ипсвич, Массачусетс, США

апрель 2001 г.



## Глава 1

# Введение в .NET

*Я нынче ночью не засну — болят года во мне.  
На вахте время потяну — с Тобой наедине.  
Опять сойду в машинный мрак: пойду к моей машине,  
В морях поднявшей кавардак в трехмесячной путине.<sup>1</sup>*

Р. Киплинг о важности доступа к системе  
24 часа в сутки, 7 дней 8 неделю.  
(«Молитва МакЭндрю», 1894 г.)

## Большой Интернет

Интернет большой. (Раздраженный Читатель: «Я заплатил за то, чтобы знать НАСКОЛЬКО большой!»). Подсчитывающий Гонорар Автор: «Я что-то не так сказал?»)

Сами по себе настольные ПК неинтересны. как неинтересна и одноклеточная амеба. Конечно, их можно использовать (ПК, а не амеб), чтобы сыграть в главную игру — Пасьянс — и компьютер не даст вам сжульничать (это достоинство?). Да и Блокнот всегда под рукой. Но в отличие от значения амебы в процессе эволюции, экономическое значение автономного ПК для общества не имеет мало-мальски достойного обоснования. Он просто не может делать многих полезных и интересных вещей, пока его кругозор ограничен его собственным ящиком. Однако когда вы через Интернет подключаете свой ПК ко всем

Автономные ПК гораздо менее полезны, чем ПК, соединенные в сеть.

---

<sup>1</sup> Здесь и далее стихотворения Р. Киплинга в переводе В. Топорова.

другим ПК з мире (и не только к ПК, но и другим интеллектуальным устройствам типа компьютеров на ладони или холодильников) — практически без затрат на дополнительную аппаратуру — начинают происходить забавные вещи. Одноклеточное эволюционирует и превращается в человекоподобное существо, способное сочинять, исполнять и ценить симфонии, взять вас с собой на Луну или уничтожать себе подобных. И на кой вам после этого Пасьянс?

Интернет продолжает  
менять общество все  
возрастающими  
темпами.

Web начиналась как средство просмотра скучных научных отчетов, но это давно позади. Она колоссально упростила распространение всех видов данных. Растущие возможности представления данных привлекали к Интернету все новых пользователей, а рост аудитории подталкивал к активным действиям провайдеров контента, и этот замкнутый процесс продолжает набирать обороты, даже когда я пишу эти строки. Вчера я просматривал в Web голевые моменты из хоккейного матча, который я пропустил. Потом с помощью Napster нашел несколько приятных мелодий из 500 000 песен, записанных на жестких дисках 10 000 различных пользователей (меня, конечно, интересовали только легальные копии). Затем я рассказал маме, как нужно настроиться на соединение с моей видеокамерой, чтобы она посмотрела на свою внучку, спящую в колыбельке за 500 миль от нее. Интернет открывает нам особый мир, о котором мы не могли подумать даже пять лет назад.

Аппаратура и услуги  
доступа в Интернет  
дешевы и продолжают  
дешеветь.

Аппаратура для подключения к Интернету и каналы передачи данных постоянно дешевеют. Web-камера, позволяющая моей маме смотреть на внучку, стоит всего несколько сотен долларов и не требует никаких дополнительных расходов при работе через установленный у меня кабельный модем. Прикиньте, сколько бы это стоило десять или даже пять лет назад: купить видеоаппаратуру и оплатить выделенный канал от Ипсвича, штат Массачусетс, до Орвигсбурга, штат Пенсильвания. Цены на аппаратуру и услуги Интернета скоро упадут до уровня лотерейных выигрышей, если еще не упали,



## Поднимаем планку: общие проблемы инфраструктуры

Аппаратура и услуги Интернета дешевы, но есть одна загвоздка. Согласно Второму Закону Платта, общее количество всякого рода неприятностей во Вселенной — величина постоянная.<sup>2</sup> Если кому-то приходится иметь дело с меньшим количеством неприятностей,

то не потому, что они исчезли, а потому, что он свалил их на чужую голову. Если аппаратура и каналы связи легче и дешевле, значит, по вселенскому закону программы для Интернета на столько же сложнее и дороже. Это бесспорно, и каждый может в этом убедиться. Проблемы Интернет-приложений не связаны с бизнес-логикой, которая не отличается от таковой в настольных приложениях (если некоторое число меньше 0 — расходы по вашему счету превышены). Между тем тот факт, что приложение должно работать на нескольких машинах, связанных Интернетом, порождает новый класс проблем, которые определяются открытой, неконтролируемой и разнородной природой Интернета. Подумайте, как просто (относительно) уследить за малышом в собственной гостиной и насколько труднее это сделать на вокзале, Тот же ребенок, те же цели (безопасность, развлечения) и совершенно другое окружение.

Рассмотрим, например, вопрос безопасности. Многие пользователи держат личную финансовую информацию на автономном ПК и обрабатывают ее с помощью Quicken или других подобных продуктов. Разработчики ранних версий Quicken не написали вообще никакого кода, обеспечивающего безопасность. Они были спокойны, точнее спокойны были их потребители, уверенные, что коль ПК физически недоступен, то никто их денежки не стянет. Параноики могли ку-

Программы для Интернета создают новые классы проблем, требующие сложных и дорогих решений.

Безопасность в настольных приложениях обычно вообще не реализована.

<sup>2</sup> Первый Закон Платта называется «Экспоненциальный рост оценочных значений» и утверждает, что каждый проект по разработке ПО требует атрое больше времени, чем при самых пессимистических оценках, даже если принимается во внимание сам этот закон.

пить продукты, защищающие всю систему с помощью паролей, но вряд ли многие это делали.

Однако, узнавая о различных новшествах, многие пользователи стали понимать, что толку от Quicken на автономном компьютере не так уж много. Он мало чем отличался от бумажного реестра платежных документов, разве что работать можно было быстрее и проще. Quicken не давал никаких преимуществ сетевым пользователям, пока не предоставил возможность подключаться к Интернету и взаимодействовать с другими участниками финансовых операций для таких действий, как получение и оплата счетов и автоматическое получение выписок по банковским счетам и кредитным картам. (То есть Quicken дал бы эти преимущества сетевым пользователям, если бы не его кривой пользовательский интерфейс. По-настоящему он не справлялся с этими новыми возможностями, загружая пользователя бесконечным числом разных параметров, которые нужно было выбирать, Но это не проблема Интернета.)

Интернет-приложения должны иметь средства защиты на всех фазах работы.

Но как только операции Quicken вышли за пределы безопасного кокona — одного ПК, на котором они выполнялись, — нужно было призадуматься о защите. Например, когда пользователь просит электронный центр оплаты счетов выписать чек телефонной компании, центру нужно быть уверенным, что запрос пришел от истинного владельца счета, а не от телефонной компании, отчаянно пытающейся избежать банкротства путем перечисления самой себе авансовых платежей за много месяцев вперед. Кроме того, пересылаемые данные нужно шифровать. Вряд ли вы захотите, чтобы соседский сынок-придурок узнал номер вашего банковского счета, подключив к линии вашего кабельного модема сетевой анализатор пакетов (что скажет ваша жена, увидев счет на 295 долларов за услуги сауны «Веселые девчонки»?).

Создавать безопасный код исключительно сложно.

Код, обеспечивающий безопасность, исключительно сложно писать (и проверять, и отлаживать, и разворачивать, и поддерживать, и сопровождать), когда сотрудники приходят и уходят. Вам нужно найти людей, знающих о безопасности все: как аутентифицировать пользователей, как решать, имеет ли

пользователь право делать то или это, как шифровать данные, чтобы их могли легко прочитать авторизованные пользователи, но не могли злоумышленники, как создать средства, позволяющие администраторам устанавливать и удалять права пользователей и т. д.

Интернет-вычисления порождают другие сходные классы проблем, которые я рассмотрю далее в этой главе. Как я объяснил в своей книге *Understanding COM+* (Microsoft Press, 1999), все эти проблемы сходны в том, что они напрямую не связаны с бизнес-процессами, хотя именно за реализацию последних вам и платят деньги клиенты. Эти проблемы относятся к инфраструктуре, — как, например, к автотрассам или электрической сети — к базе, на которой основана ваша повседневная жизнь.

Реализация инфраструктуры убивает проекты, Проекта, погибшего из-за бизнес-логики, я не видал. Вы знаете свои бизнес-процессы лучше всех, именно поэтому вы и пишете ПО, облегчающее управление ими. А инфраструктура вам неизвестна (если, конечно, не вы сами ее разрабатывали, как в случае с Microsoft). Экспертов по алгоритмам аутентификации и шифрования почти нет. Если вы попытаетесь сами написать такой код, произойдет одно из двух. Или ваша реализация будет увечной, потому что вы не понимаете, что делаете (вы легко отделаетесь, если о ней не узнают какие-нибудь нехорошие ребята), или вы не сможете довести дело до конца в рамках существующего бюджета. Когда я работал над распределенным приложением для торговли валютой (это было до Интернета, 12 лет назад; об этом тоже в моей книге *Understanding COM+*), мы получили оба этих удовольствия.

Безопасность и связанные с ней проблемы распределенных вычислений — неотъемлемая принадлежность инфраструктуры.

Инфраструктура, не бизнес-логика губит проекты.

## Планы, составленные наилучшим образом

Разработчики ПО всегда начинают проект с величественных замыслов и колоссальных обещаний. Как алкаш, стоящий у руля партии,

Разработчики ПО сами себя обманывают.

мы клянемся, что благодаря тщательному проектированию мы избежим ошибок, а наш код весь будет полностью документирован. Мы его всесторонне оттестируем, а после этого никаких дополнительных функций включать не будем. Более того, мы составим реалистичный график и будем его придерживаться. («Эта функция ОС делает не совсем то, что нам нужно, но я за неделю напишу получше, так что наши требования к программе менять не надо».) Создать надежное и полезное (в меру) приложение — в наших силах; все, что нам нужно, — дисциплина. Никаких пасьянсов. Во всяком случае до пяти часов. Ну, ладно, одну игру во время обеда. «Черт! Проиграл! Еще разок для ровного счета, идет? Ого! Уже 16:00?» Все проекты, которые я знал, начинались с таким угарным пылом.

Хоть кто-нибудь это сделал? Хоть один разработчик сдержал свои обещания? Нет. Такого никогда не было и не будет. Когда мы даем такие обещания, в глубине души мы знаем, что безбожно врем. Это не лечится. Как у наркоманов, выход здесь один (не считая летального исхода). Чтобы создавать успешные Интернет-приложения мы должны вести себя смиренно, как выздоравливающие пациенты, и на пути нашем к праведности мы должны:

1. признать себе в собственном бессилии — наша жизнь стала неуправляемой;
2. уверовать, что здравый рассудок вернет нам только сила, большая, чем наша.

Сами построить инфраструктуру вы не в состоянии.

Прикладные программисты должны осознать, что изменить инфраструктуру Интернета мы бессильны. Это сделает наши проекты неуправляемыми. Мы не сможем их реализовать, они будут длиться слишком долго, стоить слишком много — мы не знаем сколько. Не стоит и пытаться — это самообман и верный путь к краху. В совершенно невероятном случае, если мы даже напишем пристойный код, наши конкуренты не заставят себя ждать и слопают наши завтраки, обеды и ужины тоже. Вы не строите собственное шоссе, чтобы водить свою машину и не устанавливаете собственный электрогенератор (если не живете в Калифорнии в начале 2001 г.)

К счастью, требования вашего Интернет-приложения к инфраструктуре совпадают с требованиями всех остальных, так же как ваши требования к автострадам и электричеству такие же, как у других людей. На основании такой глобальной потребности правительство строит автомагистрали, а энергетические компании — электростанции, которыми вы пользуетесь за соответствующую плату (или непосредственно — оплачивая счета, или косвенно — платя налоги). Правительство и коммунальные службы связаны с экономикой другого масштаба — для достижения своих целей они могут нанять или подготовить самых талантливых людей и амортизировать свои затраты из огромного числа источников, которые вам и не снились.

Что нам на самом деле нужно, это чтобы кто-то сделал для распределенных вычислений то же, что правительство делает для автомагистралей (может не буквально то, что делает правительство в этом направлении, но суть вы уловили). Как выздоравливающий наркоман верит, что вернуть его к жизни может только Сила, сотворившая Вселенную, так и разработчикам нужна высшая сила в области компьютеризации, обеспечивающая инфраструктуру Интернета и возвращающая здравый смысл разработчикам.

Нужно использовать инфраструктуру, а не строить ее.

## И все же, что это за .NET?

Это и есть Microsoft .NET — готовая инфраструктура для решения общих проблем Интернет-приложений. Последнее время Microsoft .NET невероятно много — даже для нашей отрасли — рекламируется. Именно поэтому 5 000 чумовых фанатов, припухших от Jolt Cola<sup>3</sup>, собрались в Орландо, штат Флорида, в июле 2000 г. Не потому, что они не могут отказаться от покупки уцененных

Microsoft .NET предоставляет готовую инфраструктуру для решения общих задач при создании ПО для Интернета.

<sup>3</sup> Ходят слухи, что в качестве части приговора, вынесенного судьей Пенфилдом Джексонном, Microsoft впредь обязана подавать на своих конференциях только диетическую Jolt Cola, не содержащую кофеина. (Jolt Cola — напиток с повышенным содержанием сахара и кофеина. — Прим. перев.)

авиабилетов (даже если они не туда, куда им надо, да и вообще — не сезон) и не потому, что тащатся от жуткой жары и солнечных ударов. Они хотели первыми услышать о Microsoft .NET.

Microsoft .NET — это прикомпоновываемая среда периода выполнения, работающая в ОС Windows 2000. Наверное, в будущем Министерство юстиции США постановит, чтобы она была частью ОС. Последующие версии могут предлагать часть своих возможностей для других версий Windows, а, вероятно, как мы увидим, и для других ОС. Сервис, обеспечиваемый .NET мы и обсудим далее в этой книге.

- .NET Framework — среда периода выполнения, облегчающая написание полноценного надежного кода в сжатые сроки, управление, развертывание и модификацию этого кода. Написанные вами программы и компоненты выполняются в этой среде. Она дает программистам в период выполнения такие классные возможности, как автоматическое управление памятью (сборка мусора) и упрощенный доступ ко всем службам ОС. Она добавляет массу вспомогательных функций вроде простого доступа к Интернету и базам данных. Кроме того, она обеспечивает новый механизм повторного применения кода — более простой в использовании и в то же время более мощный и гибкий, чем COM. Развертывать .NET Framework проще, так как она не требует настройки реестра. Она также поддерживает на системном уровне стандартизированный механизм управления версиями. Все это доступно программистам на любом .NET-совместимом языке. .NET Framework мы обсудим в главе 2.

.NET предоставляет новую среду периода выполнения — .NET Framework.

- Хотя число специализированных программ растет, в ближайшей и среднесрочной перспективе для доступа к Интернету в качестве клиентского интерфейса мы будем использовать обычные браузеры. Для этого серверам нужно создавать страницы с помощью языка HTML, которые браузеры могут понимать и выводить пользователю. ASP.NET (следующая версия Active Server Pages) — это

.NET предоставляет новую программную модель для создания HTML-страниц — ASP.NET.

новая среда, работающая на Internet Information Server (IIS), заметно упрощающая написание кода для создания HTML-страниц. ASP.NET предлагает новый, не зависящий от языка способ создания кода и привязки его к запросам Web-страниц, — .NET Web Forms — управляемую событиями программную модель взаимодействия с элементами управления. Она делает программирование Web-страниц аналогичным программированию форм Visual Basic. ASP.NET содержит развитые средства управления сеансами и функции защиты. Она надежнее, и производительность ее значительно выше в сравнении с ASP. Про ASP.NET я расскажу в главе 3.

- Хотя стандартные браузеры продолжают играть важную роль, я думаю, что будущее за специализированными приложениями и устройствами. Из Web можно будет не просто получить информацию и отобразить ее в браузере. Специализированный клиент (скажем, Napster для поиска музыки) будет через Интернет вызывать функции сервера и получать данные для отображения через специализированный пользовательский интерфейс, а в случае межмашинного взаимодействия пользовательский интерфейс вообще не потребуется. Microsoft .NET предлагает новый набор служб, позволяющих серверу предоставлять свои функции любому клиенту на любой машине с любой ОС. Клиент запрашивает сервер, применяя минимальный общий знаменатель — XML и HTTP. Набор предоставляемых таким образом функций называется Web-служба .NET. Вместо того чтобы сидеть и ждать, когда потребителей озорит и они падут в объятия Единственно Достойной Операционной Системы (аллилуйя!), надо просто сказать: «Покупайте нашу ОС, потому что мы предоставляем множество готовых функций, облегчающих написание приложений, способных взаимодействовать с любыми другими приложениями по всему миру, независимо от того, что они делают и где работают.» Я расскажу о Web-службах .NET в главе 4.
- Специализированные пользовательские приложения, обращающиеся к Web-службам, нуждаются в хорошем пользователь-

.NET предлагает новый способ обслуживания любых клиентов Интернет-серверами — .NET Web Services.

NET предоставляет: 1  
Windows Forms — новый  
способ разработки  
шикарных клиентских  
• приложений с использо-  
ванием .NET Framework.

ском интерфейсе. Высококачественный ин-  
терфейс гораздо удобней, как, скажем, спе-  
циализированный интерфейс Microsoft Out-  
look лучше универсального Web-интерфейса  
Hotmail. Microsoft .NET предоставляет новый

Windows Forms,— облегчаю-

щий написание специализированных клиен-

тских приложений с помощью .NET Framework. Чтобы пред-  
ставить себе его возможности, вообразите Visual Basic, сгу-  
щенный на анаболических стероидах. .NET Windows Forms мы  
обсудим в главе 5.

- Ни одна среда программирования для Интернета не будет  
полной без средств доступа к базам данных. Большинство

ADO.NET обеспечивает  
прекрасную поддержку  
доступа к базам данных  
в рамках .NET  
Framework.

Интернет-программ, по крайней мере сейчас,  
расходуют львиную долю своего времени на  
сбор информации от клиента, выполнение  
запросов к БД и представление результатов  
клиенту. .NET обеспечивает развитую под-  
держку работы с БД с помощью ADO.NET.

Хотя здесь я эту технологию не обсуждаю, главу про ADO.NET  
можно загрузить с web-сайта этой книги [www.introducingmic-](http://www.introducingmicrosoft.net)  
[rosoft.net](http://www.introducingmicrosoft.net),

## Об этой книге

Пока я не написал *Understanding COM+*, все мои книги были под-  
робными руководствами с примерами на C++. Это очень нра-  
вилось продвинутым программистам на C++, но покупателей,  
увы, находилось немного, что очень удручало моих кредиторов.  
Я решил сделать эту книгу доступной для разработчиков, кото-  
рые не знают или не любят C++. Больше того, я обнаружил, что  
менеджеры абсолютно ничего не получают от моего C++-под-  
хода, поскольку никогда не работают с примерами программ  
(было одно исключение — этому человеку я бесплатно отослал  
экземпляр своей книги, чтобы он трудился и над последующими).  
Я хочу расширить аудиторию и не ограничивать ее программис-  
тами. Безграмотные (или еще хуже — полуграмотные) менедже-  
ры — твари исключительно опасные. Истребление этого вида  
было бы моим величайшим вкладом в развитие цивилизации.



В этой книге применяется стиль, с которым я экспериментировал в моем последнем опусе, приспособив формат, успешно использованный Дэвидом Чеппелом (David Chappel) в его *Understanding ActiveX and OLE* (Microsoft Press, 1996 — Технологии ActiveX и OLE. М.: «Русская Редакция», 1997): множество пояснений и диаграмм и совсем немного кода в тексте. Как бы мне ни нравилась книга Дэвида Чаппела, кода мне все равно не хватало (как зачастую мне нужен кусочек шоколадного торта после изысканного суши). Я ловил себя на том, что пишу код, чтобы лучше понять его идеи, как писал уравнения для понимания текста в книге Стивена Хокинга (Stephen Hawking) «Краткая история времени». (Ажно, допустим, я — чокнутый.) Итак, для всех глав есть примеры программ, кое-что я написал сам, а кое-что — на основе примеров Microsoft. Эти примеры находятся на web-сайте книги, адрес которого, естественно, <http://www.introducngmicrosoft.net>. Менеджеры и проектировщики могут читать эту книгу не захлебываясь кодом, а изголодавшиеся по коду программисты по-прежнему могут удовлетворить свой аппетит. Несмотря на весь шум вокруг C# (произносится «Си-шарп»), я написал код примеров на Visual Basic.NET, поскольку с этим языком знакомо большинство читателей. Если вы захотите запустить примеры программ, вам потребуется компьютер с Windows 2000 Server, как минимум бета-версия Microsoft .NET SDK и предварительная версия Visual Studio.NET. Подробно требования к системе и настройке описаны на web-сайте.

Примеры программ и инструкции по их установке находятся на web-сайте этой книги.

В каждой главе представлена одна последовательно рассматриваемая тема. Я начинаю с описания архитектурных проблем, подлежащих разрешению. Затем объясняю архитектуру верхнего уровня той инфраструктуры, которую .NET предоставляет, чтобы решить проблему, кодируя как можно меньше. После этого я перехожу к простейшему примеру решения проблемы. Менеджеры могут прекратить чтение после этого раздела. Затем идет обсуждение всяких тонких моментов: альтернативных вариантов, граничных условий и т. п. Я всюду пытаюсь следовать закону Порнелля, придуманного Джерри Порнеллем (Jerry Pournelle) в его рубрике «Владения хаоса» в журнале *Byte*, который формулируется очень просто: слишком много примеров не бывает.

В каждой главе •..  
представлена одна тема  
в нисходящем порядке.

## Предостережение: ознакомительное ПО

Выбрать момент для написания книги по программированию довольно сложно.

Современное ПО — наиболее быстро меняющаяся область человеческой деятельности, из всех когда-либо существовавших. При написании книги об актуальном ПО требуется компромисс между точностью и своевременностью. Если вы начнете писать, когда ПО действительно начнет поставаться, книга окажется на полках через полгода или через год после выхода ПО. А это может составлять половину жизненного срока программы, а то и больше. С другой стороны, если вы приступите к книге слишком рано, в процессе разработки, окончательная версия ПО слегка утратит сходство с тем, что описано в книге. Я всегда старался откладывать написание функциональной спецификации продукта до момента его поставки — это единственный способ гарантировать соответствие продукта спецификации.

Я написал эту книгу в первом квартале 2001 г., используя версию Beta 2 Microsoft .NET Framework и Visual Studio.NET. Я даю абсолютную гарантию, что некоторые функции в период между написанием этой книги и выходом окончательной версии ПО будут изменены: могут быть добавлены новые, ожидаемые удалены, а существующие изменены. Я ведь тоже, когда планирую направление разработки собственного проекта по мере его приближения к завершению или когда хочу добавить новые функции, действую по собственному усмотрению, не ставя об этом в известность Microsoft, чтобы они что-то переделали или просто учли этот факт. Если хотите оставаться в курсе моих соображений относительно .NET и знать об изменениях в этой книге, подпишитесь на бесплатную информационную рассылку *Thunderclap* («Ужасная новость») на моем web-сайте [www.rollthunder.com](http://www.rollthunder.com).

В какой-то момент вы, несомненно, скажете: «Черт возьми! Платт абсолютно прав (или не прав)! Надо бы, чтобы Microsoft изменила то или добавила это». В Microsoft будут рады услышать ваше мнение. Например, в главе 3 я объясняю, что в соответствии с текущими планами Microsoft предполагает, что вы должны использовать Блокнот (Notepad) для настройки ASP.NET. Я спорю с ними, объясняя, почему это просто неприемлемо для админист-

раторов в реальной работающей среде. Если вы со мной согласны и считаете, что это нужно исправить, направьте свои соображения в Microsoft. Мы все рады вашим замечаниям на всех стадиях разработки!

## Песнь Интернету

Современная поэзия сводит меня с ума. Почти вся она для меня неотличима от напыщенной политизированной прозы с рассыпанными там и сям возвратами каретки. Ни рифмы, ни ритма, — только автор (обычно с личными доходами или с грантом за счет налогоплательщиков, иначе он бы умер с голоду) у которого фатальная мания, что Он Должен Сказать Что-то Важное. Может, мой скромный интеллект просто не хочет предпринимать усилия, чтобы разобраться в его умственных отклонениях. Не знаю, как вы, а я найду другое применение немногим оставшимся у меня извилинам.

Современная поэзия —  
полный бред.

Вместе с тем я люблю старую поэзию, особенно Киплинга. По нынешним временам политкорректным его не назовешь (прочтите его поэму «Бремя белого человека», если хотите знать почему). В его защиту могу сказать, что все мы — продукты своей эпохи. И он получил Нобелевскую премию по литературе в 1907 г., так что кому-то он тогда должен был нравиться. Дедушка и бабушка подарили мне его книгу «Вот так сказки». Мои родители читали мне ее перед сном и по ней я сам учился читать. Я изучал поэзию Киплинга в школе, заметив, что читать раздел в хрестоматии с его стихами гораздо интересней, чем слушать учителя. Его стихи до сих пор звучат во мне как никакие другие ни до, ни после.

Поэзия Киплинга  
прекрасна.

Вы спросите, какое отношение это имеет к компьютерному безумию? Невероятное ускорение технологических инноваций в последние годы заставили меня вспомнить поэму Киплинга «Молитва МакЭндрю», опубликованную в 1894 г. Многие считают, что современность сильно отличается от прошлого, особенно, когда речь идет о технологиях. И все же я поражаюсь, насколько ощущения

Киплинг написал поэму,  
прославляя флотского  
инженера, и почти все,  
о чем он говорил, отно-  
сится к сегодняшним  
программистам.

МакЭндрю совпадают с моими сегодняшними. Главный герой — старый шотландский морской инженер — размышляет о наиболее блестящем технологическом достижении своих дней — корабельном паровом двигателе. Это было началом смерти расстояний — процесса, который я и вы, мои дорогие чудики, доведем до конца, прежде, чем обретем вечный покой. Мне так нравится эта поэма, что каждую главу я начинаю с цитаты из нее. Полностью вы ее можете прочитать на <http://home.pacifier.com/~rboggs/KIPLING.HTML>. Может, вы думаете, что прототипом шотландских инженеров является Скотти из фильма *Star Trek*, но я уверен, что Джени Роденберри взял за основу кипплинговского МакЭндрю,

Эта поэма содержит раннюю формулировку закона Мура.

Например, каждый программист знает формулировку закона Мура, так? Он гласит, что вычислительная мощность за данную цену удваивается каждые восемнадцать месяцев.

Многие знают и его опровержение — закон Гроша, который утверждает, что как бы ни старались ребята-железячники, ребята-программисты все равно сведут их усилия на нет. Многим известно даже следствие Яблокоу, которое формулируется очень просто: «И это не предел». Но МакЭндрю постиг это сто лет назад, куда раньше, чем некоторые плагиаторы прилепили к этой идее имя Мура и назвали ее законом. Когда я смотрю на старую добрую IBM PC с частотой 4,77 МГц (два флопа и 256 Кбайт памяти), я вспоминаю слова Кипплинга:

*Он начинал еще тогда — в машинном на подхвате!  
Не пароход был в те года — труба на самокате!  
Давление — десять фунтов. Смех! Иначе не назвать.  
Теперь мы выдадим на всех аж сто шестьдесят пять!  
Мы ковырялись так и сяк с убоищем мотора,  
Л ведь на тридцати узлах работать будем скоро!*

Подобно Родни Дэнжерфилду<sup>4</sup> мы жаждем почета и признательности. Человечество смотрит на нас косо с тех самых пор, как первый пещерный программист исследовал заостренный камень и сказал: «Классная фрактальная структура. Интересно, что если масштабировать его до размеров наконечника для копья?» Помните, как одноклассники толпились вокруг футболистов, боль-

<sup>4</sup> Популярный в США комик. — Прим. перев.

шинство из которых (не все, Брайан) были немые, как стена? Отличники, как правило, были не интересны (я не исключение), и даже мои шахматные награды не подкреплялись университетским дипломом. Хотя я теперь и знаю, что в перспективе я заработал гораздо больше денег, чем мои разгильдяи-одноклассники (которые, по мнению моего отца, пользуются большим успехом у противоположного пола), мне все еще больно. МакЭндрю в голос рыдал о том же, только гораздо красноречивей (выделено мной):

*Романтику ему подай! Проклятый первый класс! —  
Да в чистых томиках издай. — Л кто споет про нас?  
Любовь и кровь, любовь и кровь — тошнит от перегара!  
**О Боже, нужен новый Бернс, чтоб создал песню Пара!***

Я не Бернс, и даже моя матушка затыкает уши, когда я пою. Но я приложу все усилия, чтобы рассказать эту историю так, как я ее вижу. Надеюсь вы получите удовольствие от чтения.



# Объекты .NET

*Басами поршни зазвучат, а помпы — чуть визгливо,  
Эксцентрики заголосят, и заскрежещут шкивы,  
И передачи подадут свой глас в свой миг и час,  
И вал — услышишь! — подпоет, в великий хор включась.  
Вот это песня! В ней напор, и слаженность и сила.*

Р. Киплинг о чрезвычайном разнообразии типов  
компонентов, гармоничная работа которых необходима  
для функционирования каждого приложения.  
(«Молитва МакЭндрю», 1894 г.)

## Суть проблемы

Хорошие программы писать трудно — так было всегда. А сложность проблем, которые приходится решать разработчикам при со-

Хорошие программы  
писать нелегко.

здании полезных приложений, в современном распределенном и гетерогенном мире Интернета возросла. Иногда я ловлю себя на ностальгии по старым добрым временам, когда под разработкой ПО понимали создание процессора вводимой информации, который, анализируя считанные прямо с клавиатуры коды, преобразовывал их в понятные командному процессору символы. Теперь все не так. В настоящее время разработчиков постоянно преследует ряд сложнейших проблем.

Во-первых, выбирая язык программирования, мы сразу же столкнемся с противоречием. Хотя теоретически на любом языке можно создать программу, которая задействует все преимущества ОС, уж слишком часто приходится слышать: «Эй, парень, если

Все системные функции должны быть доступны в любом языке программирования.

ты пишешь на COBOL'е, то не **сможешь** использовать автоматическое управление памятью. Пора бы тебе перейти на настоящий язык». Хотелось бы, чтобы выбор языка определялся тем, насколько хорошо он подходит

для реализации прикладной области, а не соответствием возможностям системы — хватит нам граждан второго сорта! Думаю, что, если все крутые возможности Java были бы доступны лишь в Java, это стало бы причиной его скорого заката. Терпеть не могу, когда кто-нибудь настойчиво утверждает, что, дескать, принимаю лишь Единственный Истинный Язык Программирования. Я считаю, что Спасение должно быть доступно адептам всех «вероучений» от программирования.

Технология COM позволила разрабатывать приложения путем сборки их из коммерчески доступных компонентов, освобождая от необходимости писать все «с нуля»

После выхода в 1993 г. технологии COM разработчики ПО для Microsoft Windows поняли, что им больше не надо писать весь код приложения «с нуля». COM позволяет клиенту вызывать функции сервера на уровне двоичного, а не б компиляции исходного текста. В COM прикладную логику ПО можно реализовать с помощью компонентов, куп-

ленных у сторонних поставщиков (например, элемента управления «календарь») и подключенных к приложению через сравнительно тонкий слой связующего кода. Это позволяет быстрее разрабатывать приложения, функциональность которых лучше, чем у самостоятельно написанных программ, а сторонним компаниям — намного повысить удельную стоимость для амортизации своих затрат на разработку. С помощью COM Microsoft также предоставила доступ к функциональности ОС (в частности, к организации очередей и транзакциям), что также ускоряет работу приложений и облегчает их создание. Идея была неплоха, и программные боги стали благосклонны. На время...

До сих пор мы обходились COM. Теперь же требуется абстрагироваться от различий в реализации.

: Как и в случае большинства архитектур ПО. с помощью COM удалось в определенной мере облегчить ситуацию. Но теперь ее внутренняя структура из вспомогательного фактора превратилась в источник затруднений. В COM две главные проблемы: во-первых,



она требует от приложения развитой инфраструктуры, например, фабрик классов и преобразователей интерфейсов. В каждой среде разработки эти механизмы реализованы по-своему, поэтому они несколько отличаются и не настолько совместимы, как хотелось бы. Во-вторых, клиент и сервер COM взаимодействуют на расстоянии. Их взаимодействие основано на внешних интерфейсах, а не на сходстве внутренней реализации. Можно сказать, что у клиента и сервера COM «любовь по телефону». Увы, отличия между реализациями интерфейсов COM коварны и трудно согласуемы. Например, в C++ строки реализованы иначе, чем в Microsoft Visual Basic, а строки в Java отличаются и от C++, и от Microsoft Visual Basic. При передаче строки от сервера COM, написанного на Visual Basic, клиенту, написанному на C++, кто-то должен сглаживать эти отличия. Обычно это делают приложения на C++, поскольку реализация из Visual Basic инвариантна. В программистов уходит куча времени на устранение отличий в реализации. При этом много времени пропадает зря (кроме того, это надоедает программистам, которые стремятся заняться чем-нибудь повеселее), и вы никогда не узнаете наверняка, будут ли все клиенты COM работать с вашим сервером, невзирая на их реализацию. Необходимо сгладить эти различия, допуская более тесное взаимодействие приложений.

Web по своей сути гетерогенна, гомогенная Web — это ничто. Это доминирующее свойство должно учитываться в архитектуре любой программы, претендующей на успех.

Хотя Microsoft и желала бы повсеместно видеть ПК под управлением Windows, стало очевидно, что этого не будет. Желательно, чтобы однажды написанное ПО работало на самых разных платформах. Этого ожидали от Java, но ожидания оправдались не полностью (только не надо электронных писем, полных праведного гнева и возражений, — это МОЯ книга). Даже если сегодня мы не можем использовать этот подход в полной мере, в будущем хотелось бы иметь возможность развития интероперабельности платформ.

Утечки памяти сегодня — одна из самых существенных причин сбоев программ, особенно тех, что работают в течение долгого времени. Случается, программист выделяет в ОС блок памяти с

Хотелось бы, чтобы наши программы работали на самых разных платформах.

А автоматическое управление памятью должно предотвращать утечки памяти.

намерением освободить его позже, но забывает это сделать и выделяет следующий блок.

Тогда говорят, что произошла «утечка» блока памяти, поскольку в дальнейшем его уже не используешь. Если приложение работает долго, «утечки» накапливаются, и ему не хватает памяти. Это не важно в случае таких программ, как Блокнот, которые пользователь запускает всего на несколько минут и после закрывает, но фатально для таких приложений, как Web-серверы, которые должны непрерывно работать в течение дней и недель. Можно следить за освобождением каждого выделенного блока, но все равно в программах со сложной логикой блоки часто теряются. Нужен такой механизм борьбы с утечкой памяти, о котором просто нельзя забыть; он должен быть как автоматический ремень безопасности, который невозможно забыть пристегнуть.

Требуется помощь при управлении различными версиями одного из того же пакета ПО.

Ни один продукт не достигает совершенства к моменту начала поставок. (Знаю, знаю: ваши достигают, но согласитесь, что все остальные — нет, да? Кроме того, как без обновлений получить дополнительные деньги за

уже проданное ПО?) Спустя некоторое время после выхода первой версии вы начинаете поставлять обновленную версию продукта с новыми функциями и исправлениями ошибок в старых. После этого начинается веселье: не важно, сколько усилий вы затратили на поддержку преемственности нового выпуска со всеми старыми клиентами, на деле это осуществить очень трудно и практически невозможно испытать. Как было бы здорово иметь механизм, используя который, серверы публиковали бы сведения о версии их ПО. Было бы неплохо, если бы эти механизмы позволяли клиентам читать сведения о версиях ПО всех доступных серверов и выбрать совместимый или, при отсутствии такового, точно определить отсутствующие компоненты.

Благодаря таким методикам, как классы и наследование, ООП проникло в мир разработки ПО- Программы, сложность которых превышает определенный не слишком высокий уровень, можно создавать лишь по этим методикам. К сожалению, все языки поддерживают очень разные сочетания функций ООП, которые в силу естественных причин несовместимы. Это значит, что взаи-

модействие различных языков возможно лишь на очень низком уровне абстрагирования. Например, COM не позволяет программистам на Visual Basic расширять функциональность объектов, написанных на C++ с помощью удобных механизмов наследования. Вместо этого приходится обходить эти

функции ООП были доступны не только во всех языках, но и чтобы функции ООП одного языка были доступны из другого.

ограничения, применяя громоздкие механизмы. Хотелось бы, чтобы функции ООП были доступны не только в каждом языке, но и чтобы функции ООП одного языка были доступны в другом.

Web очень быстро становится для пользователя главным средством получения ПО, что ведет к возникновению серьезных проблем в области безопасности. Хотя в текущих версиях Windows автор загружаемых фрагментов кода идентифицируется с помощью циф-

Из соображений безопасности нужна возможность ограничения действий фрагментов кода, к которым нет полного доверия.

ровых сертификатов, в настоящее время невозможно гарантировать, что какая-либо программа не нанесет ущерба системе, скажем, зашифровав файлы. Можно решить, устанавливать загруженную программу или нет, но после этого нет ни одного приемлемого способа ограничить ее действия. Такое решение типа «все или ничего» на самом деле неприемлемо. Хотелось бы, чтобы в нашем распоряжении был какой-то механизм, позволяющий устанавливать для различных программ разрешенные и запрещенные действия, а ОС могла бы реализовать эти ограничения. Скажем, только что загруженной программе можно было бы разрешить чтение файлов, но запретить запись.

По мере роста ОС Windows стала невообразимо сложной. Будучи вначале скромным сервером игры «Пасьянс» с парой сотен функций, она выросла в чудовищный сервер «Свободной ячейки», у которого более 5 000

Нужен лучший способ организации функций ОС для облегчения работы с ними.

функций. Нужную функцию просто невозможно отыскать в алфавитном списке — на это уходит слишком много времени. Программисты управляют сложными проектами, организуя ПО в логические объекты. Чтобы получить хоть какой-то шанс найти нужную функцию, нужен аналогичный способ организации функций ОС в логически связанные группы.

Новая объектная модель должна бесшовно взаимодействовать с COM как в качестве клиента, так и сервера.

В конце концов я вовсе не хочу принижать значение COM. Для своего времени это была революционная технология, и значительная ее часть останется с нами в обозримом будущем. Подобно первым цветным телевизорам, которые должны были принимать не только цветные, но и преобладавшие в то время черно-белые передачи, новая объектная модель должна бесшовно взаимодействовать с COM, как в качестве клиента, так и сервера. Должно быть ясно, что столь длинный список требований не в состоянии реализовать самостоятельно ни один отдельно взятый разработчик приложений. Здесь мы достигли потенциального предела. Чтобы перейти в мир Интернета, требуется больше сил, чем нам может дать мир, в котором мы живем.

## Архитектура решения

Решение заключается в использовании управляемого кода, исполняемого под управлением CLR (Common Language Runtime).

.NET Framework — это продукт для ОС, созданных Microsoft, который предоставляет готовые решения для перечисленных проблем при программировании. Ключевым понятием этой платформы является управляемый код (managed code). Управляемый код работает в среде CLR (Common Language Runtime), которая поддерживает более богатый набор служб, чем стандартная ОС Win32 (рис. 2-1). Среда CLR дает нам большую мощь, позволяющую нашим программам выжить в суровом и диком мире, которым является программирование для Интернета в настоящее время.

Но как может CLR, обладая такой архитектурой, работать с любым языком? Звучит не очень убедительно, но это зависит от вашего определения термина «язык». Любой инструмент разработки, совместимый с CLR, компилирует свой исходный текст в программы на стандартном языке *Microsoft Intermediate Language* (MSIL или, для краткости, IL) (рис. 2-2). Поскольку независимо от языка исходного текста все инструменты разработки выдают программы на одном и том же языке — IL, все различия в их реализации исчезают к моменту достижения CLR. Независимо от формы представления строк в исходном тексте на самом языке

программирования их внутренняя реализация одинакова в любых программах, так как в CLR все они используют объект String. То же самое верно в отношении массивов, классов и всего остального.



Рис. 2-1. Управляемое исполнение программы в CLR.

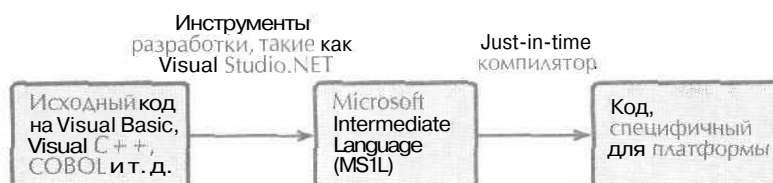


Рис. 2-2. Исходные тексты на различных языках компилируются в программы на MSIL.

Любая компания может создать язык, совместимый с CLR. В Microsoft Visual Studio.NET входят CLR-совместимые версии Visual Basic, C# (произносится как «Си-sharp»), JScript и C++. Сторонние компании также создали много других языков, среди которых APL, COBOL и Perl.

Код на IL, созданный инструментом разработки, не может сразу работать ни на каком компьютере. Для этого требуется следующий шаг — *компиляция по требованию* (just-in-time, JIT) (рис. 2-2). Утилита *компилятор по требованию* (*just-in-time compiler* или *JITter*) читает исходный текст на IL и производит настоящий машинный код, способный рабо-

Исходные тексты на любом языке, совместимом с CLR, компилируются в программы на едином промежуточном языке.

Для запуска на целевой машине исходный текст на IL компилируется по требованию (just-in-time).

тать на данной платформе. Благодаря этому .NET в некоторой степени независима от платформы, коль скоро для каждой платформы существует свой JITter. Microsoft не создает вокруг компиляции по требованию большой шумихи, какую создала Sun вокруг Java пять лет назад, поскольку эта функция еще не вышла из яслей. В настоящее время анонсирована лишь реализация CLR для Windows 2000, но я ожидаю, что через некоторое время появятся и другие. Эта стратегия рассчитана скорее на поддержку будущих версий Windows типа грядущей 64-разрядной версии и Windows XP, чем других ОС, например, Linux.

.NET framework поддерживает автоматическое управление памятью с помощью методики под названием «сбор мусора».

.NET Framework поддерживает автоматическое управление памятью с помощью механизма *сбор мусора* (garbage collection). Программа не обязана явно освобождать выделенную для нее память. CLR определяет, что память больше не используется программой и автоматически повторно использует ее. Хотел бы я себе прислугу, которая так же стирала бы мое белье!

.NET Framework поддерживает явное стандартизированное управление версиями.

Наконец, .NET Framework поддерживает управление версиями. Технология Microsoft .NET дает разработчикам серверов стандартизированный способ указания версии (эти сведения находятся в конкретном EXE или DLL-файле) и стандартизированный механизм, позволяющий клиенту указать нужную ему версию. ОС будет поддерживать запросы клиентами сведений о версии с помощью базового набора функций управления поведением, допуская их подмену разработчиком и позволяя в явном виде задавать поведение при управлении версиями.

.NET Framework расширяет сферу применения богатого набора функций ООП на все языки.

Поскольку в результате компиляции исходных текстов на любом языке получаются тексты на IL, все поддерживающие CLR языки потенциально поддерживают одинаковый набор функций. Хотя в принципе можно создать язык, совместимый с CLR, который не будет показывать программисту какие-либо лежащие в его основе функции CLR, думаю, что в этих безжалостных дарвиновских джунглях, коими

является современный рынок ПО, эта нездоровая идея быстро отомрет. CLR поддерживает богатый набор функций ООП, в частности, наследование и создание объектов с параметрами. Не пугайтесь, если на слух эти понятия покажутся вам сложными — зато они просты в использовании. Они сэкономят вам массу сил, избавят от многих потенциальных ошибок и со временем поправятся вам.

.NET Framework организует функциональность ОС посредством пространства имен System. Все объекты, интерфейсы и функции ОС теперь иерархически организованы, так что искать нужные вещи стало гораздо легче. Это также исключает конфликты имен между объектами и функциями, созданными вами, объектами и функциями самой ОС и созданными другими разработчиками.

.NET Framework организует функциональность системы в иерархическое пространство имен.

.NET Framework поддерживает безопасность доступа к коду. Можно разрешить выполнять определенные действия одной программе и запретить другой. Скажем, программе можно разрешить чтение файлов, но запретить запись. CLR реализует заданные ограничения и блокирует любые попытки выйти за их пределы. Это значит, что к полученному из разных источников коду можно применять разные уровни доверия аналогично тому, как вы по-разному доверяете людям, с которыми имеете дело. Это позволяет работать с кодом, загруженным из Web, не беспокоясь за целостность системы.

.NET Framework поддерживает безопасность кода.

Наконец, .NET Framework поддерживает бесшовное взаимодействие с COM как в качестве клиента, так и сервера. .NET помещает объект COM в объект-оболочку, в результате первый воспринимается как встроенный объект .NET. Это значит, что у кода .NET нет сведений об объектах, с которыми он работает, да это и не суть важно. С другой стороны, объекты .NET знают, как регистрироваться на определенном уровне абстрагирования, что позволяет клиентам COM воспринимать их как серверы COM.

.NET Framework обеспечивает бесшовное взаимодействие с COM как в качестве клиента, так и сервера.

Вот так... И почему все это?

Становится все сложнее создавать ОС, но вам на самом деле не стоит об этом беспокоиться.

А как быть со вторым законом Платта («Количество неприятностей во Вселенной постоянно», см. главу 1)? Если у меня стало меньше проблем (скажем, мне больше не нужно заботиться об освобождении выделенной

памяти), они не исчезли бесследно и обязательно падут на чью-то голову. В случае .NET эти проблемы свалились главным образом на две группы голов: сотрудников Microsoft и Intel (все мои читатели, работающие в Sun, вскочили с радостным воплем). Для Microsoft создание ОС усложнилось. Написать автоматический сборщик мусора вроде того, что содержится в .NET, на несколько порядков сложнее, чем простой диспетчер кучи вроде встроенного в Windows 2000. Поскольку Microsoft надеется продать несколько миллионов копий .NET, корпорация может позволить себе нанять много умных программистов и инженеров. Такое разделение труда — оставить Microsoft все заботы по разработке инфраструктуры — очень неглупо с экономической точки зрения.

Вычислительные задачи тоже усложняются — соответственно растут требования к вычислительной мощности компьютеров.

.NET Framework заставит Intel выпускать более быстрые процессоры и больше чипов памяти. Кое-кто из отдела маркетинга Microsoft утверждает, что .NET-приложения работают так же быстро, как прочие, но это не так, и никогда так не будет. Сложный меха-

низм сбора мусора требует больше вычислений, чем простое выделение памяти из кучи аналогично тому, как в автоматическом ремне безопасности больше деталей, чем в ручном. Кроме того, поскольку сбор мусора производится не так часто, придется увеличить память компьютера, чтобы ее хватило приложениям, в то время как остается достаточно «мусорных» объектов, подлежащих уборке. (Помните закон Гроша? Если нет — загляните в конец главы 1.) Не думаю, что дополнительная память и циклы процессора будут потрачены программой .NET впустую, как в Офисе на танцы дурацкой скрепки. Думаю, они будут разумно вложены, что сэкономит, таким образом, ваше время и деньги, позволяя писать программы быстрее и снижая число ошибок в них, поскольку большая часть грязной работы достанется ОС. Приложения, использующие обычную дисковую ОС, никогда не



будут работать так же быстро, как те, что программируют движения головок жесткого диска (указывая сектор и дорожку). Но это непозволительно, так как занимает слишком много времени, стоит чересчур дорого и не **позволяет** обработать достаточно много данных. Можно потратить **все свое время** на глупые операции с дисковыми секторами и никогда не кончить работы, за которую **платят**. Как только появляется возможность абстрагироваться от проблем, связанных с инфраструктурой, это становится потребностью. Если ваш диспетчер памяти работает довольно сносно, вы сэкономите время при отладке, отслеживая утечки памяти.

## Простейший пример

Я написал простейшую программу, чтобы на ее примере продемонстрировать принципы работы .NET Framework. В дальнейшем по ходу книги я буду поступать так же. Этот пример, как и другие программы из этой книги, доступны по адресу [www.introducingmicrosoft.net](http://www.introducingmicrosoft.net). Я написал объект-сервер .NET, .NET-альтернативу библиотеки ActiveX из Visual Basic 6 и соответствующий клиент. Сервер поддерживает единственный объект, имеющий единственный метод — *GetTime*, который в виде строки предоставляет текущее системное время с секундами или без них. Даже несмотря на то, что сервер написан на Visual Basic, а клиент — на C#, мне не пришлось задействовать Visual Studio. Честно говоря, оба приложения я писал в Блокноте и компоновал их, используя инструменты командной строки из состава .NET SDK. В других разделах этой главы я обязательно приведу примеры применения Visual Studio.NET. Копия .NET SDK доступна по адресу [www.msdn.microsoft.com/net](http://www.msdn.microsoft.com/net).

Ниже приведен пример объекта-сервера (рис. 2-3). По крайней мере на первый взгляд он напоминает программу, написанную на классическом Visual Basic. Однако в версию Visual Basic, которая вошла в .NET, Microsoft внесла ряд важных изменений, позволивших ей работать с классами CLR из .NET и корректно **взаимодействовать** с другими языками CLR. Подробное обсуждение этих **изменений** выходит далеко за рамки этой

Здесь начинается рассмотрение примера объекта .NET Framework.

Visual Basic.NET содержит ряд существенных отличий от Visual Basic 6.

книги, но для примера скажу: теперь индексация элементов массивов начинается не с 1, а с 0. Если объявить массив с 10 элементами (`Dim x(10) As Integer`), его члены будут пронумерованы числами от 0 до 9 вместо 1-10, как вы привыкли. В сражении между ребятами, программирующими на C++ и VB, за то, чей синтаксис массивов лучше соответствует CLR, победа за C++-никами. Увы, но эти изменения означают, что корректная работа существующих программ на VB после простой перекомпиляции в Visual Studio.NET невозможна. Вопреки вашим надеждам приведение программ в соответствие с требованиями .NET потребует определенных, но не слишком больших усилий (см. последние новости в примечании в конце этой главы).

### Примечание

В Visual Studio.NET входит средство обновления, которое автоматически запускается, когда вы открываете проект Visual Basic 6. Оно помечает обнаруженные необходимые изменения и предлагает исправления. Определенно, язык стал более мощным. Можно решиться на переход к .NET из-за крутых возможностей по поддержке Интернета. Даже если вы пишете только однопользовательские автономные приложения для обработки форм, причинами перехода могут быть поддержка управления версиями, облегченное развертывание и освобождение ресурсов.

При взгляде на исходный текст нашего объекта-сервера прежде всего заметна директива *Imports*. Эта новая директива языка Visual Basic.NET заставляет компилятор «импортировать пространства имен». Термином *пространство имен* (namespace) обозначают описание набора готовой функциональности, поддерживаемого некоторым классом в некотором окружении. Концептуально он идентичен ссылке в проекте Visual Basic 6. Список имен наборов функциональности, следующий после директивы *Imports*, задает наборы функциональности, ссылки на которые должны быть включены исполняющим ядром. В данном случае *Microsoft.VisualBasic* — это ссылка, содержащая определение функции *Now*, которая служит для получения значения времени. При использовании языка Visual Basic в Visual Studio.NET пространство

имен Visual Basic импортируется автоматически, для этого не требуется указывать оператор в явном виде.

```
' Импортировать внешнее пространство имен Visual
' Basic, чтобы получить возможность обращаться к
' функции Now по ее сокращенному имени.
```

```
Imports Microsoft. Visual Basic
```

```
' Объявить пространство имен, с помощью которого
' клиенты смогут получать доступ к классам этого
' компонента.
```

```
Namespace TimeComponentNS
```

```
' Объявить класс(ы), которые будет предоставлять
' клиенту эта DLL.
' Это делается так же, как и на VB6.
```

```
Public Class TimeComponent
```

```
' Объявить функцию(и), которую(ые) этот класс
' будет предоставлять клиенту.
' И это делается так же, как и на VB6.
```

```
Public Function GetTime (ByVal ShowSeconds As Boolean) _
    As String
```

```
' В силу некоторых причин форматирование дат и
' возврат значений функций в Visual Basic.NET
' различны.
```

```
If (ShowSeconds = True) Then
    Return Now. ToLongTimeString
Else
    Return Now. ToShortTimeString
End If
```

```
End Function
```

```
End Class
```

```
End Namespace
```

**Рис. 2-3.** Листинг исходного текста простейшего объекта-сервера.

Далее мы видим директиву *Namespace TimeComponentNS*. Она объявляет пространство имен моего компонента; с помощью этого имени клиенты смогут получить доступ к функциональности компонента. Пространства имен мы обсудим ниже в этой главе. Опять же, при использовании Visual Studio.NET пространство имен объявляется автоматически.

В этом разделе  
• приводится описание  
объекта-сервера.NET.

Далее идут объявления классов и функций, идентичные таковым на Visual Basic 6. Наконец, я разместил во внутренней логике получения времени операторы для форматирования его значения в строку и возврата строки клиенту. Эти операторы несколько различны. Свойство *Now* по-прежнему получает дату, но форматирование в строку теперь производит метод нового класса *DateTime*, а не отдельная функция. Значение, возвращаемое функцией Visual Basic, теперь задается новым ключевым словом *Return*, что отличается от синтаксиса, применявшегося в ранних версиях.

При компиляции исходных текстов «а языке Visual Basic получаются ОН, содержащие IL и метаданные.

Затем инструментами командной строки из .NET SDK скомпилировал исходный текст в DLL, которую назвал *timecomponent.dll*. Те, кому нужен образец синтаксиса командной строки, найдут его в программе-примере, загруженной с сайта книги. Полученная в результате библиотека может показаться обычной DLL, но внутри она устроена совершенно иначе. Компилятор Visual Basic.NET не преобразует код Visual Basic в машинные коды (команды, специфичные для вашего процессора). Вместо этого логика моего объекта-сервера в DLL выражена на IL, промежуточном языке, с которым мы познакомились в разделе «Архитектура решения». Все компиляторы языков CLR выдают вместо машинных кодов процессора исходные тексты на IL. Это позволяет CLR бесшовно работать с множеством различных процессоров. Полученная DLL также содержит метаданные (*metadata*), т. е. описание кода для системы CLR. Формат метаданных, необходимый для описания содержимого DLL, определяется CLR: они содержат сведения о том, какие классы и методы хранятся в DLL, какие внешние объекты ей нужны, какую версию программы она представляет и т. д. Ее можно рассматривать как библиотеку типов, накатан-

ную анаболиками. Главное отличие в том, что сервер COM иногда должен работать без библиотеки типов, тогда как сервер объекта .NET не может и подумать о том, чтобы работать без своих метаданных. Эти метаданные мы обсудим в разделе «Сборки».

Для проверки готового сервера нужен клиент. Чтобы продемонстрировать тот факт, что .NET работает с различными языками, я написал клиент на C#. Знаю, что в настоящее время этот язык знаком немногим, а может, и никому из вас. Но если взглянуть на следующий код (рис. 2-4), станет ясно, что на таком уровне сложности разобраться в нем можно. По правде говоря, после улучшений, внесенных в Visual Basic .NET для поддержки функции ООП, таких как наследование (см. ниже), мне доводилось слышать, как программисты после пары кружек пива называли C# «VB с ;» или даже «Java без Sun». За любой из этих эпитетов легко схлопотать, если произнести его слишком громко, перепутав бары в Редмонде и Саннивэйле.

Исходный текст примера клиента начинается с импорта пространств имен, для этого на C# требуется указать директиву *using*. Наш клиент импортирует пространство имен System (его подробное описание см. ниже), которое содержит описание функции *Console.WriteLine* и пространство имен нашего компонента time. Надо явно указать компилятору DLL, в которой расположено пространство имен, что можно сделать в пакетном файле компилятора. Для этого также служит удобный пользовательский интерфейс Visual Studio.NET.

Исполнение любой C#-программы начинается со статического метода *Main*. Видно, что в нем наша клиентская программа с помощью оператора C# *new* дает указание исполняющему ядру CLR отыскать DLL, содержащую класс *TimeComponent*, и создать его экземпляр. В следующей строке вызывается метод этого объекта *GetTime*. Затем системный метод *Console.WriteLine* выводит значение времени в окне командной строки. В этом случае компилятор C# создает EXE-файл. Подобно DLL сервера, этот EXE-файл содержит не машинные коды, а промежуточный язык и метаданные.

Степень сродства между Visual Basic и C# больше, чем это принято считать в сообществах программистов, использующих эти языки.

```

// Импортировать пространства имен, которые
// использует эта программа, чтобы получить
// возможность обращаться к их функциям по
// сокращенным именам.

using System ;
using TimeComponentNS ;

class MainApp
{
    // Статический метод "Main" является точкой
    // входа в приложение.

    public static void Main ()
    {
        // Объявить и создать новый компонент класса,
        // который поддерживается сервером,
        // написанным на VB.

        TimeComponent tc = new TimeCoinponent () ;

        // Вызвать метод сервера GetTime. Вывести
        // результирующую строку в консольном окне.

        Console. Write (tc. GetTime (true)) ;
    }
}

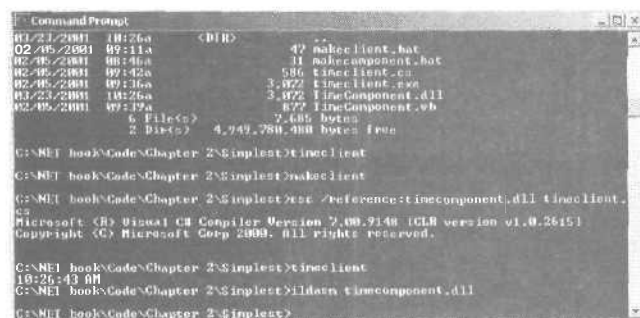
```

**Рис. 2-4.** Исходный текст на C# простейшего клиента объекта,

При компиляции клиента, написанного на C#, тоже получается файл с IL.

При запуске исполняемого файла клиента, написанного на C#, системный загрузчик определяет, что это управляемый код, и передает его ядру CLR. Ядро, обнаружив IL в EXE-файле, вызовет компилятор по требованию (just-in-time compiler или JITter). JITter — это системный инструмент, который преобразует IL в машинные коды любого процессора и ОС, под управлением которой он работает. Для каждой архитектуры есть собственный JITter, соответствующий особенностям конкретной системы. Благодаря этому один и тот же набор IL-инструкций может работать на системах различного типа. JITter создает машинные коды, которые начинает исполнять обработчик CLR. Когда клиент вызывает оператор *new* для созда-

ния объекта класса *TimeComponent*, обработчик CLR вновь вызовет JITter, чтобы по требованию скомпилировать IL из DLL компонентов, а затем выполняет вызов и возвращает результат. При этом выводится следующая информация (рис. 2-5).



**Рис. 2-5.** Информация, которую выводит на консоль программа *TimeClient*.

Модель компиляции в период выполнения неплохо работает в случае некоторых классов приложений, например, программы на Интернет-странице, на которую вы только что зашли. Но для других приложений она подходит хуже, например, для Visual Studio, которой вы пользуетесь семь дней в неделю и обновляете ее раз или два в год. Поэтому можно указать, что JIT-компиляция приложения должна осуществляться только раз при установке его на машину, а полученный при этом машинный код будет храниться в системе, как обычное приложение. Это делает утилита командной строки Ngen.exe (native image generator), которая в этом примере не показана.

Исходный текст на It  
 , компилируется  
 по требованию при  
 запуске клиента или его  
 компонентов.

Как загрузчик узнает местонахождение DLL сервера, когда клиент создает объект с помощью оператора new? Он просто ищет ее в каталоге, где находится клиентское приложение. Эта простейшая из моделей развертывания .NET называется *закрытой сборкой* (private assembly). На закрытую сборку можно ссылаться только из ее каталога. Она не поддерживает проверку версий и функции безопасности. Она не

Загрузчик находит запрошенную клиентом DLL в каталоге клиентского приложения.

требует записей в реестре, что необходимо серверу COM. Все, что нужно сделать для удаления закрытой сборки, — стереть ее файлы, и никакой дополнительной очистки. Очевидно, что, не смотря на эффективность таких простых решений в подобных ситуациях, они годятся не всегда. В частности, они не идеальны, когда одну серверную программу нужно предоставить в совместное использование нескольким клиентам. Более сложные сценарии мы обсудим ниже в разделе, посвященном сборкам.

## Подробнее о пространствах имен .NET

Легко выбирать элементы из коротких алфавитных списков.

Помню, как, программируя для Windows 2.0, я просматривал алфавитный список функций ОС (отпечатанный на бумаге — вот как давно это было!), пока не находил какую-нибудь функцию, чье имя напоминало мне предмет поиска. Я пробовал задействовать ее, иногда получалось, иногда — нет. Если функция не работала, я опять лез в список. Организовать в алфавитный список несколько сотен функций Windows 2.0 было довольно разумно. Обычно я с первого взгляда (ну, со второго или третьего точно) замечал все, чем мне может быть полезна ОС, что давало мне шанс написать достойную программу, хотя и не без ограничений.

-По мере того, как список становится длиннее, все усложняется.

В современных 32-разрядных Windows все функции ОС просто невозможно организовать в алфавитный список. Их жутко много — больше 5 000. Я не смогу найти функции для вывода на консоль в таком чересчур длинном списке. Они разбросаны среди слишком большого числа функций, не имеющих отношения к предмету, что затрудняет их поиск. Для создателей ОС это тоже проблема. Добавляя новую функцию, им приходится выбирать для нее имя, которое должно быть описательным и не конфликтовать с именами уже реализованных функций. Прикладные программисты также должны быть уверены, что имена их глобальных функций не конфликтуют с функциями ОС. Отношение «сигнал/шум» при таком подходе снижается по мере удлинения списка функций, и на сегодня оно приближается к абсолютному нулю. Можно сказать, что пространство имен (name-



space), набор имен, в пределах которого любое имя должно быть уникальным, стало слишком велико.

С большими списками легче обращаться, разбив их на меньшие «подсписки», — их легче анализировать. Классическим примером такого подхода может служить кнопка меню Start в Windows. Если бы приложения со всего компьютера были собраны в одно гигантское меню, вы бы никогда не нашли в нем нужное приложение. Вместо этого ужаса меню Start предоставляет относительно короткий список (что-то около 10 пунктов), который легко просматривать в поисках нужного элемента. В каждой группе есть список логических подгрупп, вложенных настолько глубоко, насколько это вам покажется оправданным. В итоге каждая из них раскрывается в короткий список реальных приложений. Чтобы запустить нужное приложение при путешествии по меню, в общей сложности приходится просматривать до 50 элементов, но их никогда не бывает больше дюжины (или около того) одновременно. Вы только подумайте, насколько это легче по сравнению с выбором нужного приложения из тысячи (примерно столько их установлено на большинстве компьютеров) с помощью единственного меню.

Лучший способ обработки больших списков — разбиение их на небольшие логические группы.

.NET Framework предоставляет лучший способ организации функции и объектов ОС. Этот механизм исключает конфликты между именами функций и объектов, написанными вами и другими разработчиками. Он основан на понятии пространства имен — логического подразделения функциональности ПО, в пределах которого все имена должны быть уникальны. Само по себе это понятие не ново, его десятилетиями использовали в объектно-ориентированных языках. Однако это первый известный мне случай, когда с помощью пространств имен в .NET организована вся функциональность ОС.

- В .NET используется понятие пространства имен — логической группы, в пределах которой все имена уникальны.

Все объекты и функции .NET CLR являются частью пространства имен System. При поиске в документации можно заметить, что все имена начинаются с символов «System». Мож :

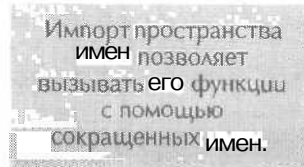
<sup>1</sup> Все объекты и функции .NET CLR «живут» в пространстве имен System.

но сказать, что все объекты и функции, имена которых начинаются с таких символов, «принадлежат к пространству имен System». Конечно, пространство имен System довольно велико, так как оно содержит имена каждого функционального элемента из их богатого набора в ОС. Поэтому оно разделено на несколько подчиненных пространств имен, скажем, *System.Console*, содержащее все функции, которые имеют дело с вводом-выводом в окно консоли. В некоторых из этих подпространств имен есть еще вложенные «подподпространства» и т. д. вглубь, пока не устанут разработчики. Полностью квалифицированным именем (fully qualified name) функции, которое иногда называют *квалифицированным именем* (qualified name) или *q-именем* (q-name), является имя, где перед именем собственно функции стоит полный путь к ней в пространстве имен, или *квалификатор* (qualifier). Например, *System.Console.WriteLine* — это полностью квалифицированное имя системной функции, которая выводит информацию в окне консоли. Полностью квалифицированное имя позволяет вызвать функцию из любого места программы.

Пространство имен System реализовано в нескольких отдельных DLL, поэтому надо удостовериться, что ваше средство разработки «знает», что нужно подключить все необходимые DLL.

Пространство имен System очень велико. Поэтому оно реализовано в нескольких отдельных DLL. Простого факта принадлежности функции или объекта к пространству имен System недостаточно, чтобы ваш редактор, компилятор или компоновщик смог ее найти. В общем случае при сборке придется указать инструментам разработки, какие DLL пространства имен System подключить к проекту. Скажем, я часто выполняю сборку компонентов .NET в Visual Studio и при их отладке использую сообщения. Объект *MessageBox* является частью пространства имен *System.Windows.Forms*, реализованного в библиотеке *System.Windows.Forms.dll*. При создании проекта «Библиотека компонентов» Visual Studio не включает эту DLL в список ссылок автоматически (возможно, потому, что авторы Visual Studio полагают, что эти компоненты будут работать «за кулисами», не взаимодействуя с пользователем). Если я хочу получить в проекте доступ к *MessageBox*, мне приходится добавлять эту ссылку в явном виде.

Подобный способ организации функций в логические группы очень удобен и позволяет найти нужную функцию с минимумом суеты. Его единственный недостаток в том, что полностью квалифицированные имена могут быть очень длинны. Так, функция *System.Runtime.InteropServices.Marshal.ReleaseComObject* сразу освобождает заданный объект COM, не выполняя полный сбор мусора (см. раздел «Управление памятью в .NET»). В большинстве .NET-приложений эта функций вообще не нужна, но в некоторых может использоваться многократно. Если набирать это длиннющее имя целиком каждый раз, когда нужно вызвать функцию, вы быстро выбьетесь из сил. Ведь моя жена обращается ко мне «Дэвид Сэмюэл Платт, сын Бенджамина, внук Джозефа», только когда о-о-очень сердита, а в этом состоянии она бывает лишь недолго. Поэтому аналогично тому, как люди, состоящие в близких отношениях, обращаются друг к другу по имени, CLR позволяет импортировать пространство имен (рис. 2-3 и 2-4). Импортируя пространство имен, вы как бы говорите компилятору о том, что будете использовать функции из этого пространства имен так часто, что хотите перейти с ними на «ты» и обращаться по имени. Импорт пространства имен на Visual Basic выполняет ключевое слово *Imports*, а в C#-программах — ключевое слово *using*. Так, выше я импортировал пространство имен *System*, благодаря чему смог вывести информацию на консоль, вызвав функцию *Console.WriteLine* (рис. 2-4). Если бы я не импортировал пространство имен *System*, эту функцию пришлось бы вызывать по полностью квалифицированному имени *System.Console.WriteLine*. С другой стороны, если импортировать пространство имен *System.Console*, я смог бы вызвать эту функцию просто как *Write*. Выбор пространства имен для импорта должен определяться комфортом при разработке и не влияет на конечный продукт. Он просто позволяет организовать мыслительный процесс так, чтобы получать наилучшие результаты. Поскольку вся идея отдельных пространств имен заключается в том, чтобы предотвратить конфликты путем помещения одноименных функций в разные пространства имен, нельзя одновременно импортировать все пространства имен. Во избежание путаницы я настоятельно рекомендую



определить непротиворечивый набор правил для выбора импортируемых пространств имен и следовать им на всем протяжении разработки проекта.

Ваш код также поселится в пространстве имен под тем именем, которое вы ему дадите. : При написании объекта-сервера .NET следует задать имя пространства имен, в котором обитает ваш код. Это делается с помощью директивы *Namespace* (рис. 2-3). Часто (но не обязательно) имя пространства имен совпадает с именем файла, в котором живет программа. В одном файле может быть несколько пространств имен, и, наоборот, одно пространство имен можно распространить на несколько файлов. Часто Visual Studio.NET и другие среды разработки автоматически присваивают проекту пространство имен. Но как убедиться, что ваше пространство не будет конфликтовать с пространством имен, избранным другим поставщиком для своего компонента? С помощью сборок.

## Сборки

В .NET широко используются новые единицы упаковки — сборки. В .NET Framework для хранения кода, ресурсов и метаданных широко используются сборки (*assemblies*). Весь код, исполняемый сред-ством поддержки периода выполнения .NET,

должен находиться в сборке. Кроме того, все функции безопасности, разрешения пространств имен и управления версиями работают для отдельной сборки. Поскольку сборки применяются часто и для самых разных нужд, мы обсудим их подробнее.

### Понятие сборки

Сборка — это логический набор, состоящий из одного или нескольких EXE- или DLL-файлов, в которых находятся код и ресурсы приложения. В сборку также входит декларация (*manifest*) — метаданные, описывающие код и ресурсы, которые находятся «внутри» сборки (чуть позже я объясню, почему «внутри» взято в кавычки). Зачастую сборки состоят из единственного EXE- или DLL-файла (рис. 2-6).

DLL, которую создал компилятор при компоновке программы-примера простого сервера времени (см. выше), на самом деле была сборкой. Вторая однофайловая сборка — это исполняемое

клиентское приложение, также скомпонованное для этого примера. При использовании таких инструментов, как Visual Studio.NET, каждому проекту скорее всего будет соответствовать одна сборка.

Наши программы-примеры создают две сборки, каждая из которых состоит из одного файла.

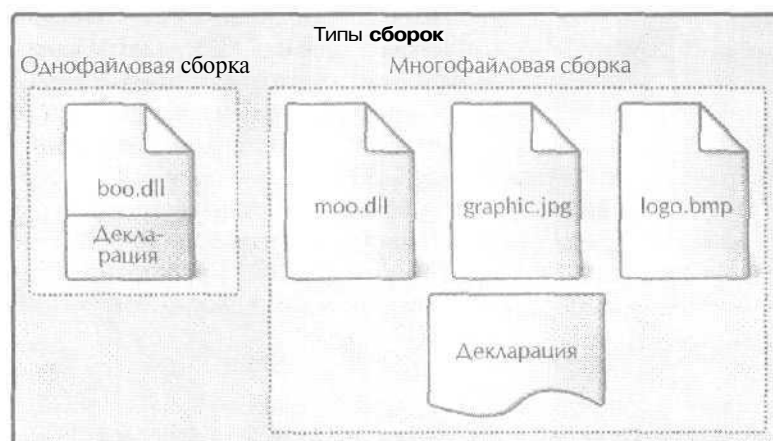


Рис. 2-6. Однофайловые и многофайловые сборки.

Хотя часто сборка располагается в одном файле, она может быть и логической (в противоположность физической) совокупностью нескольких файлов из одного каталога (рис. 2-6). Декларация с указанием файлов, составляющих сборку, может располагаться в одном из EXE- или DLL-файлов сборки, содержащих код приложения. Она также может жить в отдельном EXE- или DLL-файле, который при этом не содержит ничего, кроме декларации. Имея дело с многофайловыми сборками, надо помнить, что файлы сборки не связаны друг с другом средствами файловой системы. Ответственность за то, чтобы все перечисленные в декларации файлы были в наличии, когда их станет искать загрузчик, целиком и полностью лежит на вас. Единственный признак принадлежности файла к сборке — ссылка на него в декларации. В этом случае термин *сборка* не слишком удачен, так как подразумевает свинченные вместе металлические части. Возможно, термин «перечень» («roster») будет

Сборка также может быть логической совокупностью нескольких файлов.

более адекватным. Вот поэтому термин «внутри», упоминавшийся выше, я взял в кавычки. Утилита командной строки `AL.exe` [Assembly Generation Utility] из состава SDK позволяет добавлять/удалять файлы из многофайловых сборок. Интеллектуальные среды разработки типа Visual Studio делают это автоматически.

Декларацию сборки можно просмотреть с помощью программы ILDASM.exe/

Декларацию сборки позволяет просматривать дизассемблер IL (ILDASM.exe). Декларация нашего компонента `time` показана ниже (рис. 2-7). В ней перечислены внешние сборки, от которых зависит данная сборка. В нашем случае она зависит от `microsoft.dll`, главной библиотеки .NET CLR, а также от сборки `Microsoft.VisualBasic`, содержащей внутренние функции языка Visual Basic, такие как `Now`. Кроме того, там перечислены имена *сборок*, которые мы сделаем доступными из внешнего мира, в этом случае — `TimeComponent`.

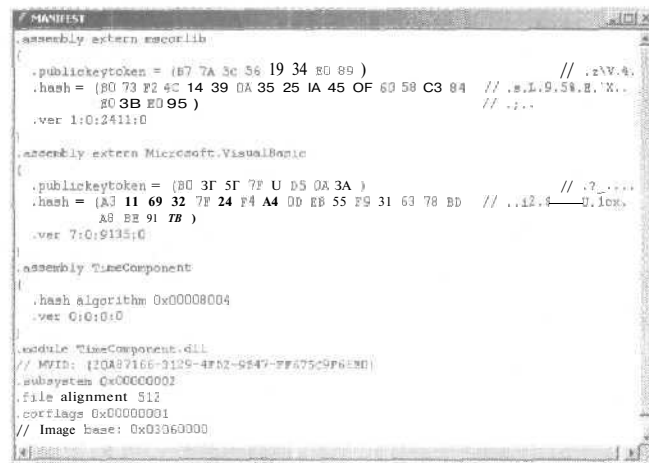


Рис. 2-7. Декларация сборки нашего компонента *Time*.

Кроме объектов программы, предоставляемых сборкой и необходимых ей, в декларации находятся сведения, описывающие саму сборку. Так, она содержит сведения о версии сборки в стандартизированном формате (о нем см. ниже). Она также может описывать «культуру» (culture), для которой написана сборка (в данном случае культура — забавное название человеческих язы-

ков и их диалектов, скажем, австралийского *английского*). В случае совместно используемой сборки (мы поговорим об этом подробнее чуть позже) в декларации содержится открытый ключ, позволяющий гарантированно отличить эту сборку от других независимо от имени ее файла. К декларации даже можно добавлять собственные атрибуты, которые будут игнорированы CLR. Атрибуты декларации устанавливаются с помощью вышеупомянутой Assembly Generation Utility или в Visual Studio.NET.

### Сборки и развертывание

При разделении кода по сборкам центральным является вопрос: как будет использоваться код вашей сборки — только вашим приложением или им и другими приложениями, которым он необходим. Microsoft.NET

поддерживает оба варианта, но в последнем случае придется попотеть. Вероятно, в случае кода, написанного для собственных приложений (скажем, вычислительного ядра для сложной финансовой программы) вы захотите сделать сборку закрытой. С другой стороны, объекты общего назначения, которые по логике могут использоваться несколькими приложениями (скажем, алгоритм сжатия файлов), лучше сделать совместно используемыми.

Полагаю, у вас возникнет желание сделать сборки закрытыми — в .NET это проще всего. По правде говоря, именно так я и поступил в приведенном выше примере. Для этого нужно всего лишь скомпоновать простую сборку как DLL и скопировать ее в каталог

или подкаталог, где находится использующая ее сборка-клиент. Не нужно делать записей в системном реестре или Active Directory, что необходимо при использовании компонентов COM. При этом ни одна программа не будет изменена, если только вы сами не измените ее, так что вы никогда не столкнетесь с общеизвестной ситуацией, когда при изменении версии совместно используемой DLL приложение без видимой причины выходит из строя.

Очевидной проблемой этого подхода является возрастание числа сборок (попыткой решения аналогичной проблемы стало использование DLL в Windows 1.0). Если каждому приложению, ко-

1. Тщательно подумайте, какими будут ваши сборки: закрытыми или открытыми.

Сборки могут быть закрытыми и предназначены для определенного приложения, что порой может облегчить вам жизнь.

Однако, иногда вам могут понадобиться совместно используемые сборки.

торое использует, например, текстовое поле, потребуется собственная копия содержащей его DLL, то сборки, размножаясь как бактерии, захлывнут весь компьютер. В мартовском номере журнала *MSDN Magazine* за 2001 год

Джеффри Рихтер выступил с возражениями. Он не считает это проблемой, поскольку сегодня жесткие диски объемом больше 40 ГБ продаются по цене меньше 200 долл., и каждый может позволить себе необходимое дисковое пространство. Таким образом, большинство сборок должны быть закрытыми, потому что тогда приложение никогда не выйдет из строя, если кто-то испортит совместно используемый код. Этот аргумент в стиле врачей скорой помощи, которые говорят, что мир был бы намного более приятным местом, если бы люди не напивались и не кололись. Хотя эти точки зрения абсолютно верны, ни одна из них не станет реальностью в ближайшее время. Идея Рихтера осуществима для разработчиков, у которых, как правило, есть быстрые компьютеры с большой памятью. Но клиент с обширным парком ПК двухлетней давности, который в данный момент невозможно модернизировать или просто заменить из-за отсутствия финансов, не будет покупать диски большего объема. Довольно скоро в процессе разработки вы ощутите необходимость совместного использования сборок несколькими приложениями, и в этом вам опять поможет .NET Framework.

.NET Framework допускает совместное использование сборок путем размещения их в глобальном кэше сборок (global assembly cache, GAC — звучит как вопли героев мультиков). GAC — это каталог на вашем компьютере (в данный момент `\\winnt\\assembly`), где должны жить все совместно используемые сборки. Утилита командной строки `GACUTIL.exe` из .NET SDK, которая хорошо работает при запуске из сценариев и пакетных файлов, поможет помещать сборки в кэш, просматривать их свойства и удалять их из кэша. Большинство пользователей предпочтут программу *Assembly Cache Viewer*, которая является расширением оболочки и устанавливается вместе с .NET SDK. Она автоматически подключается к Windows Explorer и позволяет просматривать GAC (рис. 2-8). При совместном использовании компьютерных файлов любого типа вы постоянно рискуете столкнуться с проблемой конфлик-



та имен. Поскольку все совместно используемые сборки .NET должны быть в GAC, чтобы ими можно было управлять, нужен какой-то способ присвоения гарантированно уникальных имен всем программным файлам, живущим в GAC (даже если исходно у них были одинаковые имена). Это делается с помощью *строгого имени* (strong name), или *совместно используемого имени* (shared name). С помощью криптографии по открытому ключу строгое имя гарантированно уникально и отличается от имен остальных сборок в системе. В декларации совместно используемой сборки содержится открытый ключ из пары открытого и закрытого ключей. Строгое имя формируется в результате комбинирования имени файла и части этого открытого ключа.

Гарантированная уникальность имен совместно используемых сборок достигается с помощью криптографии с открытым ключом.

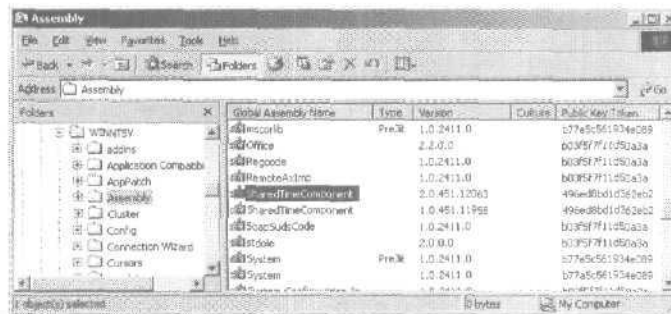
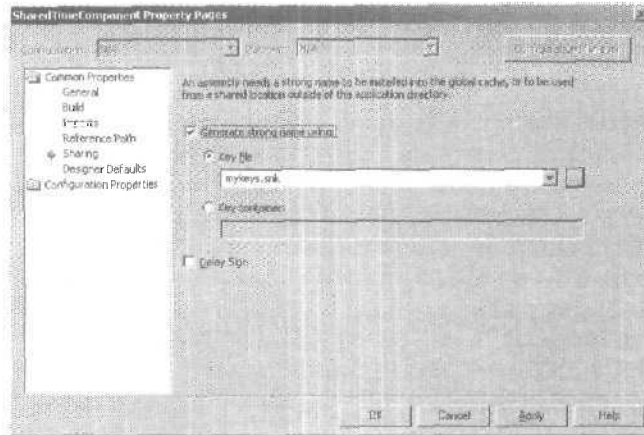


Рис. 2-8. Средство просмотра глобального кэша сборок.

Предположим, что нам требуется написать совместно используемую сборку, которая будет жить в GAC. Для этого примера я использую Visual Studio.NET, так как я нахожу, что эта среда разработки удобнее, чем инструменты командной строки. Я написал другой компонент Visual Basic.NET, выполняющий те же действия, что и простейшая программа-пример time, за исключением того, что он добавляет к строке результата символы «Shared example:», что позволяет отличить его от первого примера. После компоновки для этого компонента надо сгенерировать и присвоить строгое имя, или, иначе говоря, подписать компо-

Совместно используемые сборки живут в глобальном кэше сборок, администрирование которого осуществляется с помощью специальных утилит.

нент. Можно сконфигурировать Visual Studio.NET так, чтобы она выполняла это автоматически (рис. 2-9). Для этого нужен файл, содержащий пару криптоключей, который позволяет сгенерировать утилита командной строки из SDK — SN.exe. При компоновке Visual Studio.NET автоматически подписала этот компонент. После этого я вручную поместил его в GAC с помощью Windows Explorer.



**Рис. 2-9.** Visual Studio.NET, генерирующая строгое имя.

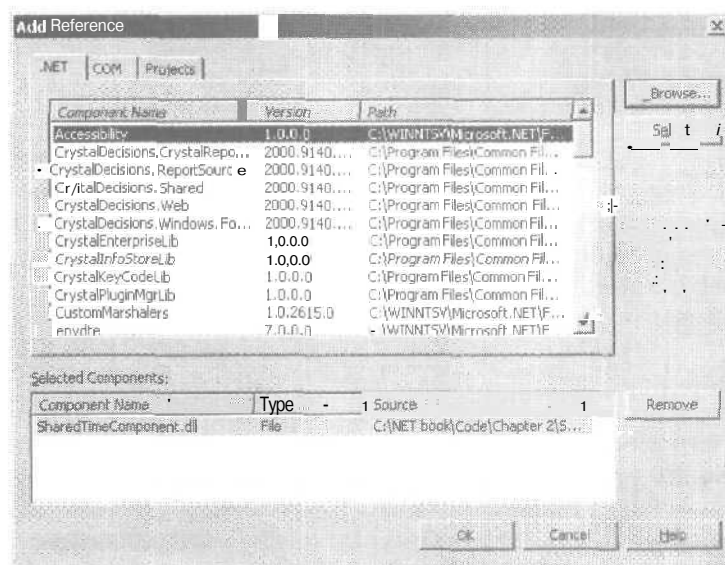
Здесь даются инструкции для генерации совместно используемой сборки.

Кроме того, я написал клиент (на этот раз с помощью Visual Studio.NET на Visual Basic), применяющий совместно используемую сборку. Я дал указание Visual Studio.NET генерировать ссылку на DLL сервера, щелкнув правой

кнопкой папку References в окне Solution Explorer (рис. 2-10), потом выбрал Add Reference, щелкнул кнопку Browse и нашел файл совместно используемой сборки, расположенный в стандартном каталоге.

В настоящее время Visual Studio не может добавлять ссылки на сборки в GAC, но эту функцию обещали включить в следующий выпуск. Дело в том, что ко времени разработки механизма ссылок Visual Studio конструкция GAC еще не устоялась. Поэтому если разработчики не создают клиент и сервер вместе как части одного проекта, приходится устанавливать по две копии компо-

нентов: одну в стандартный каталог для компиляции, а другую — в GAC для работы клиента. Пользователям понадобится лишь копия в GAC, Visual Studio.NET генерирует класс-ссылку, который обращается к этому пространству имен. Далее мне понадобится установить свойство *CopyLocal* как False, сообщив таким образом Visual Studio.NET, что локальная копия не нужна. При запуске клиента загрузчик ищет локальную копию и, если ее нет, проверяет GAC.



**Рис. 2-10.** Добавление ссылки на разделяемый компонент.

Дополнительным преимуществом подписи совместно используемых сборок с помощью открытого ключа является возможность проверки целостности файла сборки. Генератор сборки выполняет хэширование содержимого файлов, указанных в декларации. После этого он шифрует этот хэш вашим закрытым ключом и сохраняет зашифрованные результаты в декларации. Выбирая сборку из GAC, загрузчик хэширует ее файл(ы) по тому же алгоритму, открытым ключом расшифровывает хэш из декларации и сравнивает полученные результаты. Если они совпадают, загрузчик знает, что целостность

В этом абзаце даются инструкции для написания клиента, использующего объекты из GAC.

- Криптографический алгоритм также предоставляет возможность проверки целостности файлов сборки,

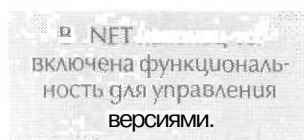
файлов сборки не нарушена. Это не позволяет идентифицировать владельца сборки, так как нельзя точно установить настоящего владельца открытого ключа, но дает твердую гарантию сохранности целостности сборки с момента ее подписи.

### Сборки и управление версиями

Управление версиями программ — огромная, мучительная и вовсе не привлекательная проблема.

Работа с обновлениями опубликованных программ исторически представляет собой громадную проблему, которую часто называют «адом DLL». Замена DLL, которую использует существующий клиент, на более новую версию ударит вас дважды. Во-первых, новый код иногда выводит из строя имеющиеся приложения, зависящие от оригинальной версии. Как бы вы ни пытались сохранить обратную совместимость нового кода со старым, узнать или проверить, что было с этим кодом после обновления, невозможно. Это особенно раздражает, когда после установки новой DLL старый клиент (уже не работающий после этого) не запускают целый месяц, а когда это наконец происходит, очень трудно сказать, что стало причиной его выхода из строя. Во-вторых, все обновления теряются, когда при установке приложение копирует старые DLL поверх новых, уже установленных на компьютере. При этом нарушается работа имеющегося клиента, зависящего от новых DLL. В 90% случаев на вопрос сценария установки: «Существует целевой файл *xxx*, более *новыи*, чем копируемый. Все равно копировать?» — пользователь отвечает «Да». Это особенно сводит с ума, когда причиной проблемы является совсем другое приложение, но в результате отказывается работать именно ваша программа, в результате ваша линия *технической* поддержки принимает дорогостоящие звонки и угрозы терактов, и остается лишь надеяться, что почтовая служба не купила ни *одной* копии программы. Проблемы с версиями обходятся чудовищно дорого и ведут к потерям производительности и замедлению отладки. Кроме того, эти проблемы заставляют *людей* воздерживаться от приобретения *обновлений* или даже от попыток их установки, поскольку они боятся, что после этого что-то обязательно перестанет работать, и часто они оказываются правы.

До сих пор Windows игнорировала проблему, связанную с управлением версиями, что вынуждало разработчиков самим решать эту проблему поэтапно. До .NET у разработчиков не было стандартизированного способа задать нужную линию поведения при управлении версиями, которую реализовывала бы ОС. Похоже, Microsoft осознала, что эту общую проблему можно решить лишь на уровне ОС, и представила в .NET систему управления версиями программ.

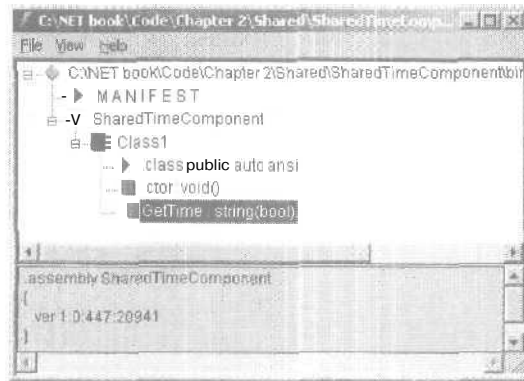


В декларации каждой сборки есть сведения о ее версии. Они включают номер версии для совместимости — набор из четырех чисел, позволяющий загрузчику CLR реализовать запрошенные клиентом действия по управлению версиями. Номер версии для совместимости состоит из старшего номера версии, младшего номера версии, номеров компоновки и ревизии. Инструменты разработки, которые создают сборку, помещают сведения о версии в декларацию. В Visual Studio.NET они хранятся в атрибутах, которые можно установить в файле проекта *AssemblyInfo.vb* (рис. 2-11). Чтобы задать номер версии инструментами командной строки, последним нужны сложные переключатели. Взгляните на номер версии, который выводит дизассемблер IL (рис. 2-12 внизу). Его также можно увидеть при установке сборки в GAC (рис. 2-8).

```
' Сведения о версии сборки состоят из следующих
' четырех значений:
'
' Старшая версия
' Младшая версия
' Ревизия
' Номер компоновки
'
' Можно задать все значения или оставить номера
' компоновки и ревизии по умолчанию с помощью '*'
' как показано ниже:
```

```
<Assembly: AssemblyVersion („2. 0.*“) >
```

**Рис. 2-11.** Файл *Assembly Info.vb*, который показывает версию сборки компонентов.



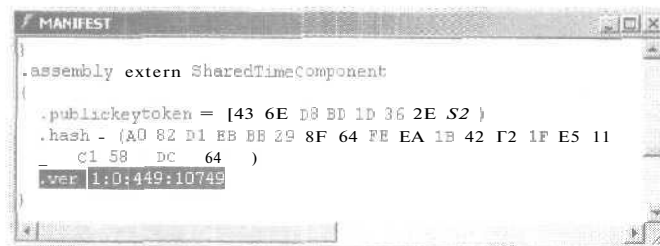
**Рис. 2-12.** ILDASM выводит версию компонента-сервера.

В каждой сборке есть информация, которая позволяет исполняющей среде узнавать номер версии сборки.

Каждая сборка-клиент содержит сведения о версиях сборок, от которых она зависит.

Декларация также содержит *информационную версию* (informational version) — строку, доступную для чтения человеком, например «Microsoft .NET Beta 1 Sept 2000». Информационная версия предназначена для просмотра людьми и игнорируется CLR.

При компоновке клиентской сборки вы видели, что в ней указаны имена внешних сборок, от которых она зависит. В ней указан и номер версии внешней сборки сервера (рис. 2-13).



**Рис. 2-13.** ILDASM выводит версию, нужную клиенту.

При запуске клиента CLR ищет сборку с нужным клиенту номером версии. Действия при управлении версиями, заданные по умолчанию, требуют точного совпадения версии загружаемой

сборки и версии, для работы с которой создавалась сборка-клиент, иначе загрузка окончится неудачей. Поскольку в GAC могут быть различные версии одной и той же сборки (рис. 2-8), вы не столкнетесь с проблемой, когда установка новой версии нарушает работу старых клиентов или вместо новой версии по ошибке записывается старая. Можно держать в GAC все нужные версии, при этом каждая сборка-клиент при запросе получит ту сборку, для работы с которой она создавалась и тестировалась.

Такой подход к управлению версиями может вас не устроить. Возможно, будет обнаружен фатальный побочный эффект, например, «дыра» в защите исходной версии сервера, что потребует ее немедленного обновления. Или в новом сервере обнаружится ошибка, и придется «откатить» клиенты к старой версии сервера. Вместо того чтобы перекомпилировать все клиенты для работы с новой версией, что пришлось бы делать в случае классической DLL, можно изменить действия системы по умолчанию, используя файл конфигурации (рис. 2-14). Имя файла конфигурации соответствует ПОЛНОМУ имени конфигурируемого им приложения (включая его расширение), но отличается от него дополнительным расширением «.config» в конце. В качестве примера приводится содержимое файла SharedTimeClient.exe.config. В случае каждой сборки можно задать новую версию, предназначенную для замены старой.

По умолчанию клиенту требуется точный номер версии сервера, для работы с которой он создан,

С помощью файла конфигурации можно изменить действия системы, заданные по умолчанию.

```
<configuration>
  <runtime>
    <assemblyBinding
      xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedTimeComponent"
          publicKeyToken="496ed8bd1d362eb2"/>
        <bindingRedirect oldVersion="1.0.451.11958"
          newVersion="2.0.451.19800"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

**Рис. 2-14.** Файл настройки управления версиями.

## Особенности объектно-ориентированного программирования

Организовать внутреннюю функциональность сложных проектов **по** не просто.

Когда сложность проекта ПО достигает определенного уровня, на организацию исходного текста и запоминание внутреннего устройства каждой функции приходится тратить больше усилий, чем на работу с прикладной

областью. Никто не сможет запомнить все действия, которые выполняют функции и как они взаимодействуют друг с другом. В результате начинается хаос. Для проектов, в которых задействовано не более пяти программистов, это ограничение не столь серьезно. Но для больших программ с богатым набором функциональности, например Microsoft Word, требуется лучший способ организации кода, чем использование глобальных функций для каждой мелочи. В противном случае будет затрачено больше усилий на то, чтобы пробиться сквозь дебри «макаронных» программ, чем на организацию обработки текстов.

Единственный способ достичь успеха в разработке проектов ПО с участием множества программистов — разделение их на классы объектов.

Для решения этой проблемы были разработаны методы ООП, благодаря чему появилась возможность создавать программы больше и посложнее. Трудно выяснить, что имеется в виду при употреблении термина «объектно-ориентированный». Его значение сильно зависит от контекста и особенностей

аудитории — в этом плане он похож на слово «любовь». Однажды я с удивлением наблюдал за тем, как два уважаемых автора полчаса яростно полемизировали, заслуживает ли некоторая методика программирования определения «объектно-ориентированная» или только «основанная на объектах». Но (как и в случае слова «любовь») есть некий компромисс: большинство разработчиков согласны, что объектно-ориентированное ПО — это хорошо, даже если в силу различных причин им не совсем ясно, почему и для кого. Большинство согласно с тем, что слово «любовь» означает по крайней мере то, что предмет вам очень нравится. Аналогично большинство программистов согласится, что ООП подразумевает как минимум разбиение программ на классы, в которых логически связанные наборы данных объединяются с функ-



циями, которые их обрабатывают. *Объект* — это отдельный экземпляр класса. *Кошка* — это класс, *моя кошка Симба* — экземпляр класса или объект. Если вы сделаете благое дело, осмысленно разделив функциональность программы на классы, вашим работникам не придется разбираться в функциональности *всей* программы. Они смогут сосредоточиться на конкретном классе/классах из порученного им подмножества *программы*, которые будут испытывать минимальное (будем *надеяться*) влияние других классов.

Предоставить программисту объектно-ориентированную функциональность — исторически эту задачу решал язык *программирования*, и разные языки решали ее в разной степени. Например, COBOL ее *вовсе* не ре-

Все языки .NET поддерживают такие методики ООП, как наследование и конструкторы.

шает. Visual Basic поддерживает минимальный набор объектно-ориентированной функциональности, в основном это классы и ничего более. C++ и Java обладают более продвинутой поддержкой функций ООП. Языки, нуждающиеся в бесшовном взаимодействии, должны поддерживать ООП в равной степени. Таким образом, Microsoft и разработчики оказались перед выбором: сделать Visual Basic (и другие языки, не поддерживающие ООП) умнее или C++ (и другие языки с поддержкой ООП) — глупее. Архитекторы .NET принадлежат к школе, девиз которой: в современной индустрии ПО без ООП ничего полезного сделать нельзя. Поэтому они решили, что функции ООП должны стать неотъемлемой частью окружения CLR и, таким образом, *быть* доступными всем языкам. Самые полезные из методик ООП, которые поддерживает CLR, — наследование и конструкторы. В следующих разделах я расскажу о каждой из них.

## Наследование

В современной промышленности практически ни один производитель (кроме стеклотрубок) не производит продукцию из «чистых» *природных* элементов — земли, воздуха, огня и воды. Почти все они используют уже созданные кем-то ранее компоненты, увеличивая их стоимость в процессе производства.

Практически в каждой отрасли современной экономики производится добавление стоимости к имеющимся компонентам.

Так, компания, продающая дома на колесах, не производит моторов и шасси — вместо этого она покупает небольшие грузовики у автомобильной компании и устанавливает на них специализированные кузова. Автомобильная компания в свою очередь купила ветровые стекла у производителя стекла, который приобрел песок для него у владельца карьера. Желательно, чтобы процесс разработки ПО соответствовал этой модели и начинался с готовой универсальной функциональности, к которой добавляются наши специализированные компоненты.

ООП реализует эту концепцию в ПО с помощью наследования.

Методика ООП под названием «наследование» намного облегчает написание объектов программ. Допустим, кто-то где-то на поддерживающем наследование языке написал

класс объектов *Класс*. Этот класс поддерживает полезную универсальную функциональность, скажем, чтение байтов из потока и запись их в поток. Нам хотелось бы создать свой класс, который не только осуществляет чтение и запись, как базовый класс, но и предоставляет некоторую статистику (например, длину последовательности байтов), взяв за основу функциональность базового класса и добавив несколько наворотов. Для этого нужно написать фрагмент программы, называемый *производным классом*, в котором функциональность базового класса некоторым образом модифицирована: часть кода добавлена, другая — изменена, а часть оставлена как есть. Для этого с помощью синтаксиса языка компилятору дается указание, что наш производный класс наследует функциональность базового класса. В результате с помощью соответствующей ссылки компилятор автоматически включит в производный класс функциональность базового класса. Это можно представить в виде операции вырезания/вставки, при которой не происходит на самом деле никаких перемещений. Говорят, что производный класс *наследует* от базового класса, *происходит от* него или *расширяет* его (рис. 2-15 — некоторые промежуточные классы опущены для ясности).

Все объекты .NET являются производными от системного базового класса *System.Object*.

Все виды функциональности .NET Framework, от простого преобразования строк до самых сложных Web-служб, предоставлены с помощью наследования. Давайте, как всегда, начнем изучение наследования с самого простого

го примера. Хорошая иллюстрация наследования в .NET — компонент `time`, созданный в предыдущей главе. Несмотря на отсутствие соответствующего кода, который указывал бы это в явном виде, класс нашего компонента `time` происходит от базового класса `System.Object`, предоставленного Microsoft. Убедиться, что это так, можно, изучив компонент с помощью ILDASM (рис. 2-16). Все без исключения объекты в системе .NET происходят от `System.Object` или производного от него класса. Если не указан иной базовый класс, подразумевается `System.Object`. При желании можно указать другой базовый класс ключевым словом `Inherits` (на Visual Basic, рис. 2-17) или оператором `<|>` (на C#).

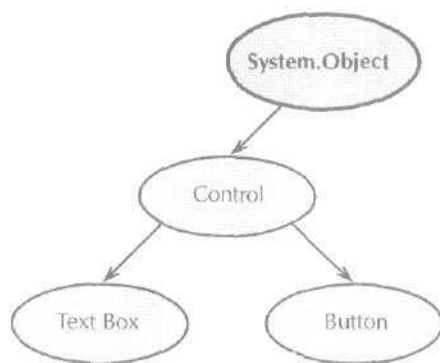


Рис. 2-15. Наследование в ООП.

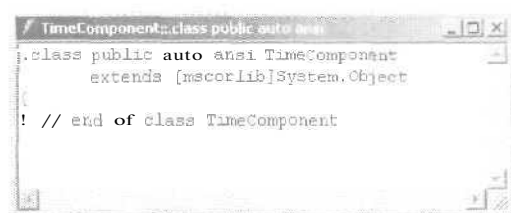


Рис. 2-16. Демонстрация наследования от `System.Object` с помощью ILDASM.

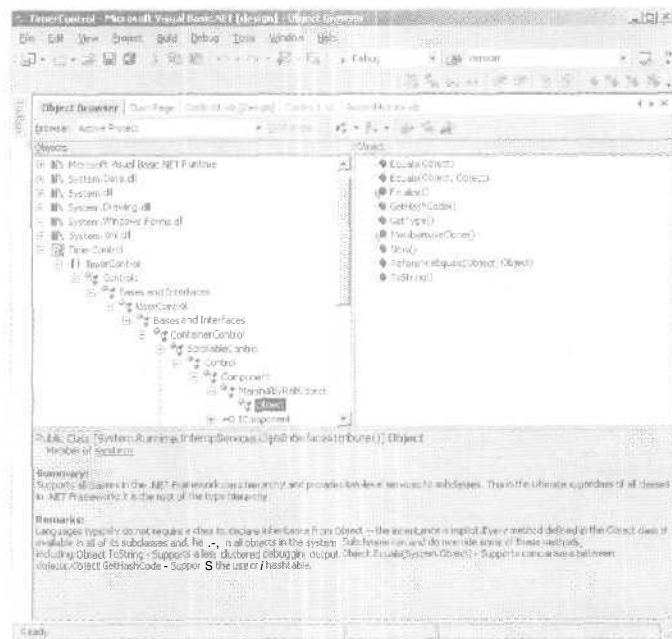
В более сложных случаях можно вывести дерево наследования, используя Object Browser из Visual Studio.NET (рис. 2-18). Этот же случай слишком прост, чтобы обращаться к Object Browser.

Значительная часть функциональности в .NET предоставляется с помощью наследования. „

Namespace VS7DemoTimeService

```
Public Class WebService1
    Inherits System.Web.Services.WebService
```

**Рис. 2-17.** Явное объявление наследования.



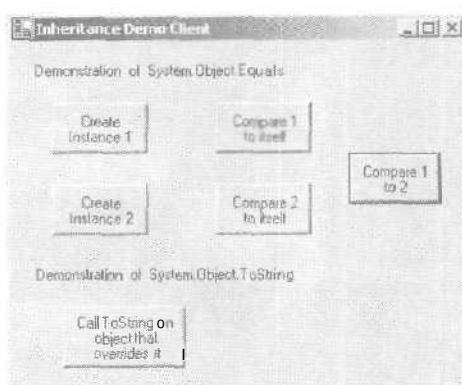
**Рис. 2-18.** Object Browser из Visual Studio.NET показывает деревонаследования.

Хорошо, пусть наш компонент time получил свою функциональность с наследством от *System.Object*, но как узнать, что же было из завещания? Это придется выяснять с помощью несколько старомодной технологии RTFM (Read The Funny Manual — почти буквально: «прочитайте это веселое руководство!»). В результате мы найдем, что у нашего базового класса несколько открытых методов (табл. 2-1). Это значит, что наш производный класс (компонент time) знает, как выполнять эти действия, даже если мы сами их не запрограммировали.

Табл. 2-1. Открытые методы *System.Object*

Имя метода	Назначение
<i>Equals</i>	Определяет, является ли этот объект тем же экземпляром, что и заданный объект.
<i>GetHashCode</i>	Быстро генерирует и возвращает целочисленное значение, с помощью которого можно идентифицировать этот объект в хэш-таблице или в другой схеме индексации.
<i>GetType</i>	Возвращает системные метаданные этого объекта.
<i>ToString</i>	Возвращает строку, представляющую собственно объект.

Метод *Equals* определяет, действительно ли две ссылки указывают на один и тот же физический экземпляр объекта. Это было на удивление трудно выяснить в COM, и некорректно реализованный сервер мог легко нарушить это. Но в .NET объекты наследуют такую функциональность от базового класса «бесплатно». Я написал клиентское приложение, которое создает несколько экземпляров компонента *time* и, среди прочего, демонстрирует работу метода *Equals* (рис. 2-19).



**Рис. 2-19.** Клиентская программа, демонстрирующая функции *System.Object*, унаследованные компонентом *time*.

Подобно реальному наследнику, иногда вашему компоненту не нужно все, что он может унаследовать от базового класса. Возможно, вам нравится антикварный столик, который оставила вам

Изменить часть функциональности базового класса можно, подменив его методы.

тетушка Софи, но этого не скажешь про ее страдающего газами бульдога (или наоборот). Наследование в программах в общем случае позволяет производному классу *подменять* методы, унаследованные от базового класса,

т. е, *заменять* их другими. Хороший пример — метод *System.Object.ToString*, который дает указание объекту вернуть строку программисту, занятому отладкой приложения. Реализация, унаследованная от *System.Object*, просто возвращает имя производного класса, что не очень информативно. Чтобы облегчить отладку компонента, этот метод должен возвращать более подробную информацию. Так, объект, представляющий открытый файл, может вернуть имя этого файла. Это делается *подменой* метода базового класса (рис. 2-20). В производный класс будет помещен метод с тем же именем и параметрами, что и у базового. Но для него будет указано ключевое слово *Overrides* (в случае C# — *override*), которое заставляет компилятор заменить реализацию из базового класса на новую реализацию из производного класса,

```
' Этот метод подменяет метод ToString
' универсального базового класса System.Object.
```

```
Public Overrides Function ToString( ) As String
```

```
' Вызвать метод базового класса ToString и
' получить результат. Если не хотите, можете этого
' не делать. Я сделал это для наглядности.
```

```
Dim BaseResult as String BaseResult = MyBase . ToString
```

```
' Построить строку результата из строки базового
' класса и добавленной мной информации. В итоге
' получается, что я частично использую базовый класс,
' а не заменяю его полностью.
```

```
Return "You have reached the overriding class. "+
      "The base class says: " + BaseResult
```

```
End Function
```

**Рис. 2-20.** Подмена методов базового класса.

Если производный класс хочет *предоставить* собственную функциональность *в дополнение* к функциональности базового класса, а не заменить ее, он может вызывать подменяемый метод базового класса в явном виде.

П оменяющий метод может обращаться к подменяемому им методу базового класса.

На Visual Basic базовый класс доступен через именованный объект *MyBase* (на C# это объект *base*). Компонент-пример вызывает базовый класс чтобы получить от него строку, а затем добавляет к строке базового класса собственную строку. В результате компонент использует функциональность базового класса вместе с производным классом, а не замещает ее полностью. Не все методы базового класса можно подменить. Чтобы разрешить подмену методов базового класса, нужно писать их с указанием ключевого слова *Overridable* на VB или *virtual* (на C#).

Большое значение *придается* способности .NET поддерживать перекрестное межъязыковое наследование. Иначе говоря, класс, написанный на одном языке (скажем, на Visual Basic) может быть производным базового класса, написанного на другом языке (скажем, на C#). В COM такая возможность не поддерживалась из-за слишком больших различий в реализации языков. Однако, стандартизированная архитектура CLR на основе IL позволяет приложениям .NET использовать эту функцию. По правде говоря, компонент-пример *time* это и делает, причем без всяких усилий с *моей* стороны. Уверяю вас: класс *System.Object* написан на одном языке без привлечения других, но, несмотря на это, от него наследуют *все* без исключения объекты .NET, невзирая на язык, на котором они написаны.

В .NET наследование работает перекрестно между различными языками.

## Конструкторы объекта

Как пели Good Rats много лет назад, «все мы как-то родились». Подобно тому, как рождение человеческих существ *сопровождается* различными ритуалами (религиозными обрядами, выпивкой в складчину), объектам требуется место для размещения ритуального кода для рождения. В ООП давно известен *конструктор класса* — функция, вызыва-

Объектам необходимо стандартное место для размещения инициализирующего кода.

емя при создании объекта. (В ООП также используется понятие *деструктора* класса — эта функция вызывается при уничтожении объекта. В .NET на смену деструкторам пришел системный сборщик мусора, описанный ниже.) В различных языках конструкторы реализованы по-разному: на C++ — с помощью имени класса, на Visual Basic — с помощью события *Class\_Initialize*. Как и в случае многочисленных функций, которые широко варьируют в различных языках, для корректной совместной работы нужно стандартизировать ритуалы создания объектов в программах на различных языках.

В .NET для этого служит конструктор

В .NET из Visual Basic исчезло событие *Class\_Initialize*, теперь его модель во многом похожа на модель C++, так как для поддержки наследования требуются параметризованные конструкторы. В каждом классе .NET может быть один или больше методов-конструкторов. Этот метод в Visual Basic.NET называется *New*, а в C# его имя совпадает с именем класса. Функция-конструктор вызывается, когда клиент создает объект оператором *new*. В этой функции можно разместить любой код, выполняющий инициализацию объекта (возможно, выделение ресурсов и перевод их в начальное состояние). Вот пример конструктора (рис. 2-21).

Конструкторы объектов могут принимать разные наборы параметров, позволяя создавать объекты в определенном состоянии.

С конструктором можно сделать еще одну интересную вещь — передать ему параметры, таким образом позволяя клиенту привести объект в определенное состояние сразу после создания. Так, конструктор объекта, представляющего точку на графике, должен

принимать два целочисленных параметра X и Y, задающие положение этой точки. В класса может быть несколько конструкторов с разными наборами параметров. Скажем, один конструктор объекта «точка» может принимать два параметра, другой принимает одну имеющуюся точку, а третий, вовсе без параметров, инициализирует члены объекта «точка» нулевыми значениями (рис. 2-22). Такая гибкость особенно полезна, если вы хотите создать объект, который нужно инициализировать перед вызовом. Допустим, у вас есть объект для представления пациента больницы, поддерживающий методы типа *Пациент.СогретьПобольшеДенег* и *Пациент.Ампутировать(какуюКонечность)*. Очевидно,



что жизненно важно знать, какого человека представляет каждый отдельный экземпляр этого класса. В противном случае пациентов можно перепутать. Предоставив конструктор, требующий идентификатор пациента, а не пустой конструктор по умолчанию, можно гарантировать, что никто не сможет прооперировать неустановленного пациента или по ошибке изменить идентификатор пациента после того, как он создан.

```
Public Class ConstructorDemoComponent

    Dim m_x, m_y As Integer

    Public Overloads Sub New ()

        MyBase.New ()

        m_x = 0
        m_y = 0
    End Sub

    Public Overloads Sub New (x As Integer, y As Integer)

        MyBase.New ()

        m_x = x
        m_y = y
    End Sub

End Class
```

**Рис. 2-21.** Пример конструктора.

```
Dim foo As New ConstructorDemoComponent ()

Dim bar As New ConstructorDemoComponent (4, 5)
```

**Рис. 2-22.** Пример конструктора.

## Управление памятью в .NET

Одним из главных источников неприятных, трудно обнаруживаемых ошибок в современных приложениях является некорректное использование ручного управления памятью. Старые языки, такие как C++, требовали от программиста удалять созданные объекты вручную. Это вызывало две главные проблемы. Во-пер-

Управление памятью вручную ведет к труд обнаруживаемым ошибкам, которые дорого обходятся.

вых, создав объект, программисты забывали до удалить его по окончании использования. Такие утечки в конечном счете полностью истощали память процесса и вызывали его крах. Во-вторых, удалив объект вручную, программист мог позже по ошибке попытаться вновь обратиться к этому адресу памяти. Visual Basic сразу обнаруживает ссылку на недействительную область памяти, а C++ — как правило, нет. Иногда в той области памяти, где раньше был удаленный объект, все еще остаются какие-то внешне нормальные значения, поэтому программа продолжит работу с испорченными данными. Эти ошибки до боли знакомы всем. Конечно, проще всего сказать: «не ошибайтесь, и все». Но в реальных программах объекты часто создаются в одной части программы, удаляются — в другой, а между ними располагается сложная логика, которая в одних случаях удаляет объект, а в других — нет. Все эти ошибки дьявольски трудно предотвратить, и еще труднее обнаружить. Дисциплина при программировании, конечно, помогает, но на самом деле хотелось бы как-то заставить программистов думать о прикладной логике, а не об управлении ресурсами. Могу побиться об заклад, что *гранд-дама* телевизионных поваров Джулия Чайлд нанимает кого-нибудь прибраться на кухне после того, как все дела там закончены, чтобы позволить себе сосредоточиться на тех аспектах кулинарии, которые необходимы для уникальной квалификации в ее прикладной области.

Автоматическое управление памятью и восстановление ресурсов типа того, что встроено в Visual Basic и Java, — весьма полезная вещь.

У современных языков, какими являются Visual Basic и Java, таких проблем нет. В этих языках управление памятью осуществляется автоматически по принципу «пальнул и забыл». Это стало одной из причин, по которой программисты выбирают их для разработки ПО. Программисту на Visual Basic 6.0, как правило, не обязательно помнить об удалении созданных объектов. Visual Basic 6.0 подсчитывает ссылки на каждый объект и, когда счетчик ссылок упадет до 0, автоматически удаляет объект, а освободившуюся память утилизирует. Среда разработки работает, как автоматическая посудомойка, которая берет грязные кастрюли и сковородки из раковины, и, вымыв, расставляет их

обратно по полкам. Эх, мне бы такую! Может, мое желание и сбывается, если вы посоветуете **всем** своим друзьям купить эту книгу... Microsoft **ввела** автоматическое управление памятью в .NET CLR, что делает его доступным для любого языка. Концептуально это просто (рис. 2-23).



**Рис. 2-23.** Автоматическое управление памятью со сборкой мусора,

Программист создает объект оператором `new` и получает ссылку на него. CLR выделяет для него память из *управляемой кучи* (managed heap) — части памяти процесса, зарезервированной CLR для этой надобности. С определенной частотой системный поток анализирует все объекты в управляемой куче, чтобы выяснить, на какие из них программа удерживает ссылки. Объект, на который не осталось ни *одной* ссылки, называется *мусором* (garbage) и удаляется из управляемой кучи. После этого оставшиеся в управляемой куче объекты *дефрагментируются*, а имеющиеся у программ ссылки устанавливаются на новые адреса объектов. Таким образом, вышеуказанные проблемы ручного управления памятью решаются без написания кода. Невозможно забыть удалить объект, так как система «убирает» самостоятельно. Невозможно и обратиться к удаленному объекту через недействи-

Сборщик мусора CLR делает автоматическое управление памятью доступным любому приложению.

тельную ссылку, поскольку объект не будет удален, пока на него есть хоть одна ссылка. Очевидно, для сборщика мусора потребуется больше циклов процессора, чем для стандартного `gcisnet`-чера кучи, даже если он написан так, чтобы не проверять один и тот же объект дважды и распознает циклические ссылки. Как сказано выше, я считаю, что это правильная нагрузка на процессор, поскольку при этом ускоряется разработка и уменьшается число ошибок.

Сборщик мусора работает лишь в удобное для него время, но его можно заставить выполнить сбор мусора в нужный момент.

Магический процесс сбора мусора имеет место, лишь когда сборщик наконец-то соберется это сделать. В отличие от алгоритма, позволяющего определить, что память управляемой кучи исчерпана, точного алгоритма запуска сборщика не знает никто. Более того, я гарантирую, что он еще не раз изменится

до окончательного выпуска продукта. Возможно, он даже будет отличаться в разных версиях одного выпуска. Функция `System.GC.Collect` позволяет вручную инициировать сбор мусора. Возможно, понадобится вызвать ее на определенных этапах, заданных логикой программы (скажем, чтобы собрать весь хлам после того, как пользователь сохранит файл, или чтобы отдраить палубу перед началом большой операции). В основном вы просто позволяете сборщику мусора делать свою работу тогда, когда он этого захочет.

Перед инициацией сбора мусора обычно размещается код, выполняющий освобождение ресурсов. Обычно он располагается в деструкторе объекта или методе `Class_Terminate`.

До сих пор автоматический сбор мусора показывал себя замечательно, но кое-что было упущено. А как насчет освобождения ресурсов при освобождении объекта? Приложения на `C++` обычно выполняли освобождение ресурсов внутри деструктора объекта, а классы `Visual Basic` — внутри их метода `Class_Terminate`. Это удачное место для размещения

кода, выполняющего освобождение ресурсов, так как клиент не сможет забыть вызвать его. Но как сделать это во время автоматического сбора мусора? Поймите, во-первых, что проблема стала значительно проще. Главной задачей освобождения ресурсов, выполняемой в деструкторах `C++`, было удаление дополнительных объектов, на которые ссылался удаляемый объект. Те-

перь об этом позаботится сборщик мусора. Но время от времени приходится выполнять освобождение ресурсов без участия механизма локального сбора мусора, скажем, при освобождении соединения с БД или выходе из удаленной системы.

Сборщик мусора CLR поддерживает *завершитель* (finalizer) — метод объекта, вызываемый при удалении объекта сборщиком мусора. В силу определенных причин завершитель аналогичен деструктору классов C++ и методу Visual Basic *Class\_Terminate*, заменителем которых он является. Однако завершитель существенно отличается от обоих этих механизмов, и некоторые отличия могут настораживать. Универсальный базовый класс CLR *System.Object* содержит метод *Finalize*, который можно подменять (рис. 2-24).

Когда объект удаляется сборщиком мусора, поток сборщика обнаруживает наличие у объекта метода *Finalize* и вызывает его, исполняя, таким образом, код освобождения ресурсов.

Protected Overrides Sub Finalize ()

’ Реализация необходимой логики завершения.

MessageBox.Show ("In Finalize, my number = " + \_  
MyObjectNumber.ToString ())

’ Перенаправить вызов базовому классу.

MyBase.Finalize ()

End Sub

**Рис. 2-24.** Поддержка объектом функции *Finalize*.

Помимо прочего, у завершителей есть несколько недостатков. Ясно, что этот механизм потребляет процессорное время; не стоит его использовать, если очистка не нужна. Невозможно дать гарантии относительно порядка вызова сборщиком мусора завершителей удаляемых им объектов, поэтому старайтесь исключить всякую зависимость от этих действий в своих программах. Завершители вызываются в отдельном потоке сборщика мусора. Поэтому вы их никак не упо-

Использование завершителей может быть сложнее, чем кажется на первый взгляд.

рядочите, чтобы навязать собственный порядок вызова этих методов. В противном случае вы разрушите всю систему сбора мусора своего процесса. Завершители не вызываются при завершении приложений, так как программа «понимает», что при завершении процесса нужно освободить все ее внутренние ресурсы, чего суетиться-то? В ранней версии .NET была функция, позволявшая заставить систему вызывать завершители при закрытии приложений, но из рабочей версии ее убрали. При использовании завершителей могут возникать и другие проблемы (см. статьи Дж. Рихтера).

---

### ПРЕДУПРЕЖДЕНИЕ

Несмотря на внешнюю простоту, поведение завершителей довольно сложно, и в нем легко запутаться. Если вы планируете их использовать, вам просто НЕОБХОДИМО прочитать описание сбора мусора в .NET в двух частях, написанное Дж. Рихтером и опубликованное в ноябрьском и декабрьском выпусках MSDN Magazine (сейчас они доступны по адресу <http://www.microsoft.com/msdnmag>). Сам факт, что для описания механизма сбора мусора потребовалось две статьи, позволяет судить о сложности его внутреннего устройства, несмотря на сравнительную легкость его подключения к программе (а может, все дело как раз в этом).

Завершители идеально подходят для задач, при решении которых нам не важно, когда будет выполнена очистка, т. е. если вас устраивают обещания типа «как только это действительно потребуется, это обязательно будет сделано». Порой это хорошо, а порой не очень, если ресурсы, которые должен освободить завершитель, дефицитны и лимитируют работу процесса (например, при соединении с БД). Эвентуальное освобождение не вполне приемлемо, если объект должен быть уничтожен прямо СЕЙЧАС, чтобы освободить занятые им дорогостоящие ресурсы, о которых не знает сборщик мусора. Как сказано выше, можно инициализировать немедленный сбор мусора, но для этого потребуется анализ всей управляемой кучи, что довольно расточительно, если не нужна очистка для других объектов. Поскольку объект, подлежащий разборке, известен, нужен способ очистить лишь

этот объект (подобно тому, как Дж. Чайлд не убирает всю кухню и не выносит мусор, лишь для того, чтобы был чистым ее любимый нож). Подобная операция имеет величественное имя: *детерминированное завершение* (deterministic finalization).

Объекты, которым требуется поддержка детерминированного завершения, реализуют интерфейс *IDisposable*, содержащий единственный метод *Dispose*. В этом методе можно разместить любой код, освобождающий дефицитные ресурсы. Клиент, вызывая этот метод, дает объекту указание немедленно освободить эти ресурсы. Примером может служить системный класс *System.Windows.Forms.DataGrid*.

Иногда имя метода детерминированного завершения, поддерживаемого объектом, может отличаться. Это позволяет определить понятное разработчику имя. Так, при вызове метода *Dispose* для объекта «файл» можно подумать, что при этом файл будет уничтожен. Поэтому разработчик такого объекта может дать этому методу более логичное имя — *Close*.

Детерминированное завершение кажется удачным, но и оно не без недостатков. Нет гарантии, что клиент не забудет вызвать ваш метод *Dispose*, поэтому в завершителе также надо включить поддержку функциональности освобождения ресурсов. Но если клиент все-таки вызывает *Dispose*, вероятно, стоит сэкономить время сборщика мусора, которое он потратит на вызов завершителя объекта, так как *Dispose* уже выполнил освобождение ресурсов. Системная функция *System.GC.SuppressFinalize* позволяет дать сборщику мусора указание не беспокоиться о вызове завершителя, даже при наличии такового. Если базовый класс поддерживает метод *Dispose*, нужно, чтобы объект в явном виде перенаправлял вызов своего *Dispose* базовому классу. Иначе вызов не дойдет до базового класса, и освободить занятые им ценные ресурсы будет невозможно. Пример метода *Dispose* показан ниже (рис. 2-25). Этот класс является производным класса *System.Object*, у которого нет метода *Dispose*, поэтому я опустил соответствующий код.

Объект, который должен предоставить клиенту способ детерминированного освобождения ресурсов, поддерживает метод *Dispose*.

```
Public Class Class1
    Implements System. IDisposable

    Public Sub Dispose () Implements System.IDisposable.Dispose
        ' Здесь будет любая логика, необходимая для немедленного
        ' освобождения ресурсов.

        MessageBox.Show ("In Dispose (), my number = "+ _
            MyObjectNumber.ToString < >)

        ' Если у базового класса есть метод Dispose,
        ' перенаправим ему вызов, сняв знак комментария
        ' со следующей строки.

        ' MyBase. Dispose ()

        ' Пометить объект как более не требующий завершения.

        System.GC.Suppress Finalize (Me)
    End Sub

End Class
```

**Рис. 2-25.** Пример метода Dispose, обеспечивающего детерминированное завершение.

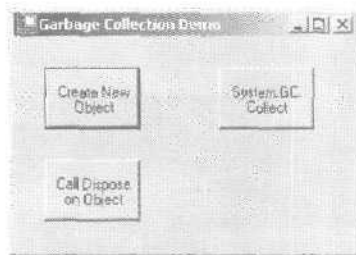
Необходимо писать программу с учетом тех случаев, когда клиент пытается обратиться к объекту после вызова его метода *Dispose*.

Я написал небольшую программу, иллюстрирующую понятие автоматического управления памятью и сбора мусора. Ее можно загрузить с Web-сайта этой книги. Вид клиентского приложения показан ниже (рис. 2-26). Заметьте: при вызове *Dispose* объект не превращается в мусор. На самом деле *Dispose* по

определению нельзя вызвать для объекта, который является мусором, поскольку в этом случае не будет нужной ссылки. И вообще объект не может стать мусором, пока на него есть ссылки. В этом случае я предлагаю организовать в объекте внутренний флаг, сигнализирующий о завершении объекта и позволяющий реагировать генерацией исключения на обращения после завершения объекта.

Автоматическая сборка мусора упрощает программирование выделения и освобождения объектов, благодаря чему внести путаницу в этот процесс стало сложнее, чем при аналогичных опе-





**Рис. 2-26.** Клиентское приложение, демонстрирующее управление памятью.

рации на C++. Но в результате программирование детерминированного завершения стало сложнее и запутаннее, чем в Visual Basic 6. Вероятно, программисты на C++ сочтут такой поворот событий за благо. Но их коллегам, работающим на Visual Basic и привыкшим к его автоматическому детерминирован-

ному поведению, практически с «защитой от дурака», это может показаться шагом назад. Причина того, что Microsoft отдала приоритет сбору мусора, в следующем: алгоритм подсчета ссылок в Visual Basic неверно обрабатывал циклические ссылки на объект. Это случается, когда у дочернего объекта есть ссылка на родительский. Допустим, объект A создает объект B, B — объект C, а C получает и удерживает ссылку на объект B. Когда A освобождает свою ссылку на объект B, последний не может быть разрушен, так как у C все еще имеется ссылка на B. Если до освобождения ссылки объектом A программист не поместит код, разрывающий циклическую ссылку, B и C становятся «забытыми объектами», у которых нет ссылок, кроме ссылок друг на друга. Алгоритм сбора мусора автоматически определяет и обрабатывает такие случаи с циклическими ссылками, чего не может алгоритм подсчета ссылок. После долгих обсуждений, сломав немало копий, Microsoft приняла решение, что защищенное от неправильного использования автоматическое предотвращение утечек памяти в любом случае важнее, чем простота реализации детерминизма. Одни программисты с этим согласятся, другие — нет (у меня еще нет определенного мнения, или, как говорит мой

По мнению Microsoft управление памятью с помощью сбора мусора сделано устойчивым к утечкам памяти ценой детерминизма.

архитектор, я решительный сторонник разных вариантов). Так или иначе, этот выбор — результат тщательного анализа, а не сиюминутного каприза.

## Взаимодействие с COM

Обратная совместимость имеет решающее значение при разработке любой новой системы,

Критическими факторами коммерческого успеха любой новой платформы являются ее интегрируемость с имеющимися платформами и одновременная поддержка новых средств разработки лучших приложений. Например, Windows 3.0 поддерживала не только приложения DOS и обеспечивала их многозадачность лучше, чем любой другой продукт за то время, но и предоставляла платформу для создания приложений Windows, которые были лучше, чем любое приложение для DOS. Но нам никогда не приходится разрабатывать что-либо «с нуля». Как только Господу Богу удалось создать мир всего за шесть дней? Дело в том, что у Него не было никакой базы. об обратной совместимости с которой Ему пришлось бы беспокоиться (мой редактор не преминул заметить, что Бог также поспешил на документацию).

Поэтому .NET поддерживает взаимодействие с COM.

С 1993 г. в области взаимодействия между приложениями Windows зависят от COM. COM пронизывает практически весь код среды Windows, и это надолго. .NET Framework приходится поддерживать COM, чтобы получить шанс на коммерческий успех. И это так: клиент .NET может использовать сервер COM, и наоборот. Поскольку более вероятно, что новым программам .NET придется взаимодействовать с уже имеющимися программами COM, а не наоборот, я опишу этот случай первым.

## Использование объектов COM из программ .NET

Клиент .NET может обращаться к объекту COM через вызываемую оболочку периода выполнения.

Клиент .NET обращается к серверу COM через вызываемую оболочку периода выполнения (runtime callable wrapper, RCW) (рис. 2-27). RCW является оболочкой объекта COM и работает как посредник между ним и средой CLR, позволяя клиентам .NET воспринимать объект COM просто как встроенный объект .NET, а клиент .NET

при этом рассматривается объектом COM просто как стандартный клиент COM.



Рис. 2-27. Взаимодействие между клиентом .NET и объектом COM через вызываемую оболочку периода выполнения.

У разработчика клиента .NET есть несколько способов генерации RCW. Если вы пользуетесь Visual Studio.NET, щелкните правой кнопкой раздел проекта References и выберите из контекстного меню Add Reference. При этом вы увидите диалоговое окно (рис. 2-28), в котором предложены на выбор все зарегистрированные объекты COM, найденные в системе. Выберите объект COM, для которого нужно генериро-

Генерировать RCW козвеляют различные инструментальные средства.

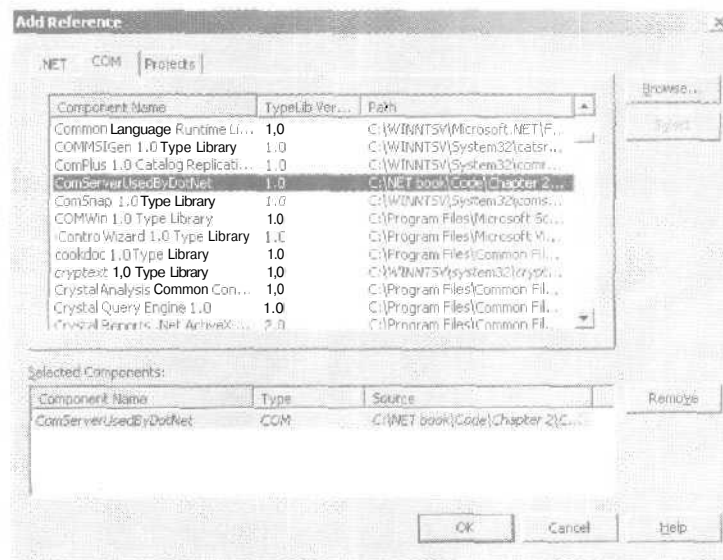


Рис. 2-28. Поиск объекта COM для генерации RCW.

вать RCW, и Visual Studio.NET выдаст ее вам. Без Visual Studio.NET аналогичную задачу позволяет выполнить TlbImp.exe (средство импорта библиотек типов) — инструмент командной строки из .NET SDK. Алгоритм, который читает библиотеку типов и генерирует код RCW, на самом деле находится в классе периода выполнения .NET под названием *System.Runtime.InteropServices.TypeLibConverter*. Как Visual Studio.NET, так и TlbImp.exe используют этот класс. Если вы создаете инструмент разработки или являетесь мазохистом, то также можете использовать этот класс.

Вот пример клиентской программы .NET, использующей объект-сервер COM (рис. 2-29). Эти примеры можно загрузить с Web-сайта книги и разбирать их по ходу изложения. В этом примере находятся сервер и клиент COM, а также клиент .NET, так что клиенты можно сравнить. Исходный текст показан дальше (рис. 2-30).

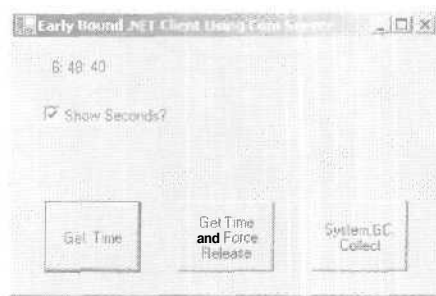


Рис. 2-29. Пример клиента .NET, использующего сервер COM.

RCW чудесным образом преобразует вызовы .NET в COM, а результаты COM — соответственно в .NET.

Сгенерировав RCW, следуя описанию в предыдущем абзаце, вы, возможно, пожелаете импортировать его пространство имен в клиентскую программу оператором *Imports*. Это позволит ссылаться на объекты по их сокращенному имени. Объект RCW можно создать просто оператором *new*, как и любой другой объект .NET. При создании RCW вызывает встроенную функцию COM *CoCreateInstance*. Таким образом, создается объект COM, для которого генерируется оболочка. После этого клиентская программа .NET может вызывать методы RCW так, как если бы она была встро-

енным объектом .NET. RCW автоматически приводит каждый вызов в соответствие с правилами вызовов COM, например, конвертирует строки .NET в строки BSTR, необходимые COM, и направляет их объекту. Прежде чем передать клиенту результат, который вернул объект COM, RCW конвертирует его во встроенные типы COM. Такая архитектура может показаться знакомой пользователям поддержки COM в Visual J++.

```
Protected Sub Button1_Click (ByVal sender As Object,
                             ByVal e As System.EventArgs)
```

```
    ' Создать экземпляр RCW - оболочку для нашего
    ' объекта COM object.
```

```
    Dim RuntimeCallableWrapper As New ComUsedByDotNet.CClass1 ()
```

```
    ' Вызвать метод, получающий значение времени,
```

```
    Label1.Text = RuntimeCallableWrapper.GetTimeFromCom( _
        CheckBox1.Checked)
```

```
    ' При выходе из области действия объект становится
    ' мусором, но реально освобождается
    ' только во время следующего сбора мусора,
```

```
End Sub
```

**Рис. 2-30.** Листинг клиента .NET, использующего RCW.

Запустив программу-пример клиента COM, вы заметите (по диалоговым окнам, вывод которых я разместил в программе), что при щелчке кнопки создается объект, который сразу разрушается. При запуске программы-примера клиента .NET вы увидите, что объект создается, когда пользователь щелкает кнопку Get Time, но его немедленного разрушения не происходит. Можно подумать, что так и должно быть, если объект-оболочка выходит из области действия, но это не так, даже если вы явно сделали ссылки на объект пустыми. Так работает реализованный в .NET способ отложенного освобождения ресурсов. Он описан выше в разделе, посвященном сбору мусора. После выхода из области действия RCW больше недоступна программе. Но на самом деле объект, оболочкой которого явля-

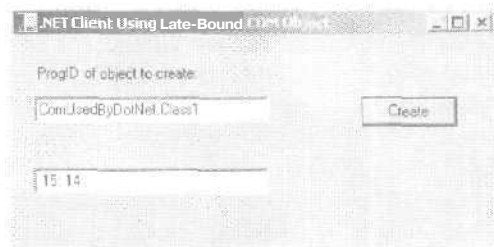
НА самом деле объекты COM разрушаются, только когда сборщик мусора удаляет их RCW.

ется RCW, не освобождается до тех пор, пока RCW не будет удалена и разрушена сборщиком мусора. Это может стать проблемой, так как большинство объектов COM создавалось без учета этого жизненного цикла, и поэтому могут удерживать ценные ресурсы, которые надо освободить сразу по завершении клиента. Есть два способа решения этой проблемы. Очевидно, первый из них — инициировать немедленный сбор мусора функцией *System.GC.Collect*. При ее вызове все системные ресурсы, которые в данное время не используются (включая RCW вне текущей области действия), будут собраны и реутилизированы. Недостатком этого подхода являются издержки, связанные с полным циклом сбора мусора (а они могут быть велики). Возможно, из-за разрушения всего лишь одного объекта такие расходы не имеют смысла. Если требуется избавиться от одного конкретного объекта COM, не трогая остальные, это можно сделать функцией *System.Runtime.InteropServices.Marshal.ReleaseComObject*. Для механизма RCW, описанного в предыдущих абзацах, нужен объект, который подвергнется раннему связыванию. Здесь я имею в виду, что во время разработки программист должен обладать глубокими знаниями объекта, чтобы создать для него класс-оболочку. Но так работают не все объекты. Например, ситуации, возникающие при работе со сценариями, требуют позднего связывания, когда клиент читает ProgID объекта и имя метода, который нужно вызвать, из кода сценария в период выполнения. Большинство объектов COM поддерживают интерфейс *IDispatch*, чтобы дать возможность получить к ним доступ через позднее связывание. В таких ситуациях создать RCW заранее невозможно. Справится ли с этим .NET?

- .NET также без особых проблем поддерживает позднее связывание.

К счастью, да. .NET Framework поддерживает позднее связывание с интерфейсом *IDispatch*, который поддерживается большинством объектов COM. Программа-пример, использующая позднее связывание (рис. 2-31), и ее исходный текст (рис. 2-32) показаны ниже. Системный тип на основе ProgID объекта создается статическим методом *Type.GetTypeFromProgID*. Если вместо ProgID имеется CLSID, то же самое делает статический метод *Type.GetTypeFromCLSID* (не показан). Объект COM

создается методом *Activator.CreateInstance*, а вызывает метод функция *Type.InvokeMember*. Здесь, конечно, придется потрудиться — это обычное дело при позднем связывании — но все это вполне осуществимо.



**Рис. 2-31.** Пример программы, использующей позднее связывание.

```
Protected Sub Button1_Click (ByVal sender As Object, _
    ByVal e As System.EventArgs)

    ' Получить имя системного типа на основе prog ID.

    Dim MyType As System.Type
    MyType = Type.GetTypeFromProgID (textBox1().Text)

    ' Создать объект этого типа с помощью активатора.

    Dim MyObj As Object
    MyObj = Activator.CreateInstance(MyType)

    ' Собрать параметры объекта COM в массиве.

    Dim prms () As Object = {checkboxBox1().Checked}

    ' Вызвать метод объекта по имени.

    label2 ().Text = MyType.InvokeMember("GetTimeFromCom", _
        Reflection.BindingFlags.InvokeMethod, Nothing, MyObj, _
        prms).ToString ()

End Sub
```

**Рис. 2-32.** Листинг программы, использующей позднее связывание.

## Использование объектов .NET из COM

Клиент COM обращается к объекту .NET через вызываемую оболочку COM (COM callable wrapper, CCW).

Допустим, у нас есть клиент, уже «говорящий на языке COM», который надо заставить использовать объект .NET. В силу некоторых причин это менее распространенный сценарий, чем обратный (см. выше), так как он предполагает разработку новых объектов

COM в мире .NET. Но нетрудно предвидеть ситуацию, когда вас есть клиент, использующий 10 объектов COM, при этом вам требуется добавить 11-й набор функциональности, который существует только в виде объекта .NET. При этом клиент должен их все воспринимать одинаково из соображений единообразия. Помощником COM callable wrapper (CCW) .NET Framework может справиться и с этой ситуацией (рис. 2-33). CCW служит оболочкой объекта .Net и играет роль посредника между ним и средой CLR, заставляя клиент COM воспринимать объект .NET так, как если бы это был объект COM.



Рис. 2-33. Вызываемая оболочка COM.

Компонент .NET должен быть подписан, обитать в GAC и предоставлять конструктор по умолчанию для работы с клиентом COM.

Для работы с CCW сборка компонента .NET должна быть подписана строгим именем. В противном случае исполняющая среда CLR не сможет однозначно идентифицировать ее. Кроме того, сборка должна находиться в GAC или, что менее характерно, в дереве каталогов клиентского приложения. Однако, как было ранее в случае компоновки клиента совместно используемого компонента, при регистрации компонент должен находиться в стандартном каталоге вне GAC. Любой класс .NET, который должен быть создан COM, должен предоставлять конструктор по умолчанию, т. е. конструктор, не требующий параметров. Функции COM для создания объектов не обладают сведениями о передаче парамет-

тов. Однако, как было ранее в случае компоновки клиента совместно используемого компонента, при регистрации компонент должен находиться в стандартном каталоге вне GAC. Любой класс .NET, который должен быть создан COM, должен предоставлять конструктор по умолчанию, т. е. конструктор, не требующий параметров. Функции COM для создания объектов не обладают сведениями о передаче парамет-



ров объектам, которые они создают. Поэтому нужно быть уверенным, что вашему классу это не потребуется. В вашего класса может быть столько конструкторов с параметрами, сколько нужно для использования клиентами .NET, но при этом должен быть хотя бы один, которому не требуются параметры (для использования клиентами COM).

Чтобы клиент COM нашел объект .NET, в реестре нужно создать записи для COM. Это позволяет сделать утилита RegAsm.exe из .NET SDK. Эта программа читает метаданные класса .NET и создает в реестре записи, которые указывают клиентам COM на объект .NET. Программа-пример предоставляет па-

Утилита RegAsm.exe из .NET SDK создает в реестре записи, указывающие COM местонахождение сервера класса .NET.

кетный файл, который выполняет эти действия. Ниже показаны созданные им записи реестра (рис. 2-34). Заметьте: сервером COM в этом случае является библиотека-посредник Mscoree.dll. Параметр *Class* в разделе *InProcServer32* указывает этой библиотеке, какой класс .NET должен быть создан и заключен в оболочку, а элемент *Assembly* — сборку, в которой находится этот класс.



**Рис. 2-34.** Элементы реестра, созданные программой REGASM.EXE.

Клиент COM обращается к объекту .NET так, как если бы он был встроенным объектом COM. Когда клиент вызывает для создания объекта функцию *CoCreateInstance*, реестр направляет запрос зарегистрированному серверу, Mscoree.dll. Эта DLL проверяет за-

Исходный текст клиента COM использующего объект .NET, находится среди примеров к этой главе.

прошенный CLS1D, читает реестр в поисках класса .NET, который должен быть создан, и на основе этого класса .NET создает CCW. CCW преобразует встроенные типы COM в их эквиваленты из .NET (например, строки BSTR — в объекты *String*, характерные для .NET), а затем перенаправляет их объекту .NET. Она также выполняет обратное преобразование результатов из .NET в COM, включая любые ошибки. Среди программ-примеров из этой главы есть клиент COM, который обращается к совместно используемой сборке компонента *time*, созданной ранее в этой главе.

Доступность частей функциональности .NET для клиентов COM можно задать через метаданные, а именно — в атрибуте *ComVisible*.

Разработчик программ для .NET может захотеть сделать какие-то методы, интерфейсы и классы доступными клиентам COM, а какие-то нет. Поэтому среди прочих метаданных .NET поддерживает атрибут *System.Runtime.InteropServices.ComVisible*. Его можно использовать для сборки, класса или интерфейса, а также для отдельных методов. Элементы, у которых этот атрибут установлен в *False*, будут невидимыми для COM. CLR по умолчанию устанавливает его в *True*, поэтому в отсутствие этого атрибута элемент будет видимым для COM. Но поведение Visual Studio.NET, заданное по умолчанию для сборок, подразумевает установку этого атрибута в файле *AssemblyInfo.vb* равным *False*. Параметры, заданные на более низких уровнях иерархии, заменяют таковые на более высоком уровне иерархии. В программе-примере я установил этот атрибут для своего класса в *True*, сделав его, таким образом, видимым COM, как показано в следующем фрагменте. Если бы я хотел, чтобы все содержимое сборки было видимо для COM, я изменил бы его значение в файле *AssemblyInfo.vb*.

```
<System.Runtime.InteropServices.ComVisible (True) > _
    Public Class Class1
```

## Транзакции в .NET

Транзакции нужны для защиты целостности данных в распределенных системах. Допустим, мы пишем приложение, которое принимает оплату счетов в диалоговом режиме. Для оплаты телефонного счета нужно снять некую сумму с моего счета, который находится в некоторой БД, и перевести ее на счет телефонной ком-

паники, который, возможно, расположен в другой БД на другой машине. Если операция списания со счета проходит успешно, но по какой-то причине зачисление на счет оканчивается неудачей, надо отменить списание со счета, иначе деньги будут потеряны, а целостность данных в системе — нарушена. Требуется уверенность в успехе или неудаче обеих операций. Именно этого можно достичь, объединив выполнение обеих операций в одной транзакции. Если обе операции завершаются успешно, транзакция фиксируется, и новые значения счетов сохраняются. Если хоть одна операция терпит неудачу, транзакция отменяется, и суммы на всех счетах откатываются к исходным значениям [о транзакциях вообще настоятельно рекомендую прочитать книгу Филиппа А. Бернштейна и Эрика Ньюкамера «Принципы обработки транзакций» (Philip A. Bernstein, Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997)].

Транзакции гарантируют целостность баз данных во время сложных операций.

В COM+ и его предка, Microsoft Transaction Server, была автоматическая поддержка, которая облегчала написание объектов, участвующих в обработке транзакций. Программист в административном порядке помечает CGOU объекты как требующие участия в транзакции. Затем, при активизации объекта, COM+ автоматически использовала транзакцию.

в COM+ хорошо реализована автоматическая поддержка транзакций.

Объекты пользовались менеджером ресурсов COM+ Resource Managers, с помощью которого такие программы, как Microsoft SQL Server, поддерживающие обработку транзакций в стиле COM+, вносили изменения в БД. После этого объект сообщал COM+, удовлетворен ли он полученными результатами. Если все объекты, участвующие в транзакции, оказывались удовлетворены, COM+ подтверждала транзакцию и давала Resource Managers указание сохранить все внесенные изменения. В противном случае COM+ отменяла транзакцию, приказывая Resource Managers отменить результаты всех операций над объектами и откатывая систему в исходное состояние. Подробнее о реализации обработки транзакций в COM+ см. мою книгу *Understanding COM+* (Microsoft Press, 1999).

Встроенные объекты .NET тоже могут принимать участие в транзакциях. Поскольку имеющаяся система обработки транзакций, разработанная Microsoft, основана на COM, объекты .NET могут делать это благодаря своей способности к взаимодействию с COM, описанной в предыдущем разделе. Необходимо зарегистрировать объект .NET как сервер COM. После этого через COM+ Explorer нужно установить этот компонент в приложение COM+ и настроить его требования к транзакциям точно так, как если это был встроенный объект COM. А инструментом командной строки Regsvcs.exe можно одновременно выполнить регистрацию объекта и настройку приложения COM+. Есть альтернативный путь: подобно встроенным объектам COM, которые иногда указывают свои требования к транзакциям в их библиотеках типов, можно задать требования объектов .NET к транзакциям через метаданные этих объектов. Следующий фрагмент демонстрирует написанный на Visual Basic объект .NET с атрибутом, указывающим, нужны ли объекту транзакции.

```
Public Class < TransactionAttribute _
    (TransactionOption.Required) > Class1
```

Если бы вы писали этот объект на C#, все было бы точно так же, кроме того, что вместо угловых скобок были бы использованы квадратные. Страница ASP.NET (см. главу 3) указывает требования размещенного на ней кода к транзакциям через такой атрибут:

```
<%@ Page Transaction="Required" %>
```

Web-службы (см. главу 4) указывают свои требования к транзакциям, помечая атрибутами отдельные методы:

```
Public Function <WebMethod (),
    TransactionAttribute(TransactionOption.Required) >_
    HelloWorld( ) As String
```

Объекты .NET могут участвовать в транзакциях COM+.

Объект .NET, принимающий участие в транзакции, должен «проголосовать» за тот или иной ее итог. Есть два способа сделать это.

В COM+ и MTS объект получает свой контекст, вызывая функцию API *GetObjectContext*, а затем вызывает в этом контексте метод, чтобы определить свой голос за тот или иной итог транзакции. Контекст объекта .NET находится в

объекте *System.EnterpriseServices.ContextUtil*, поддерживаемом системой. Этот объект предоставляет методы *SetAbort* и *SetComplete*, которые используются наиболее часто, а также их менее распространенные равноценные методы *EnableCommit* и *DisableCommit*. Эти методы устанавливают биты, показывающие, что объект завершил транзакцию и «доволен» ее итогами, точно так же, как они делают это в COM+. Контекст также содержит свойства *DeactivateOnReturn* и *MyTransactionVote*, которые позволяют читать и устанавливать эти биты по отдельности. Кроме того, через атрибут *AutoComplete* можно заставить объект .NET автоматически голосовать за итог транзакции по принципу «пальнул и забыл». В этом случае, если объект вернет управление штатным образом, автоматически вызывается метод *SetComplete*. Но если объект завершается из-за возникновения исключения, будет вызван метод *SetAbort*.

Компонент «голосует» за тот или иной итог транзакции посредством системного объекта *Microsoft.ComServices.ContextUtil*.

## Структурная обработка исключений

В период выполнения любых программ возникают ошибки. Различные действия, которые пытается выполнить программа, например, открытие файла или создание объекта, могут заканчиваться неудачей. Как же программе узнать, успешно или нет закончилась операция, и как написать код, чтобы справиться с последней ситуацией?

Любая программа требует обработки ошибок, возникающих в период выполнения.

В случае классического подхода функция, вызов которой заканчивается неудачей, возвращает специальное значение, свидетельствующее о сбое, например, *Nothing* (или NULL в случае C+ +). В этом подходе три недостатка: во-первых, программист должен написать код для проверки значения, возвращенного функцией, а на это при современных темпах разработки ПО часто нет времени. Подобно ремням безопасности или контрацептивам, коды ошибок работают, только когда их используют. Ошибки не перехватываются на уровне их источника, а распространяются на более высокие уровни программы, где их намного труднее раскрыть, и они

Индикация неудачного завершения функции с помощью возврата кода ошибки работает не лучшим образом.

часто остаются замаскированными до начала продажи программы. Во-вторых, вы могли заметить, что коды ошибок широко варьируются в зависимости от функций, что повышает риск ошибок при программировании. Например, функция *CreateWindow* при ошибке возвращает 0, *CreateFile* — (−1), а *LoadLibrary* в 16-разрядной Windows — любое значение меньше 32. На самом деле разнородной даже больше, поскольку при успешном вызове все функции, имеющие отношение к COM, возвращают 0, а при ошибках различных типов — ненулевые значения. В-третьих, при вызове функция может вернуть только одно значение, а оно несет очень мало сведений для отладчика (человека или машины), которые могли бы быть полезными для понимания источника ошибки и ее устранения.

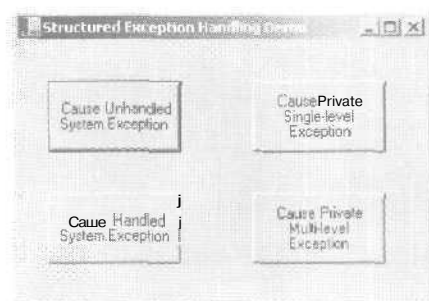
Ни одна другая методика не дает хороших результатов при работе с различными языками.

В разных языках были попытки использовать другие подходы для обработки ошибок периода выполнения. В Visual Basic применялся механизм *On Error Go To*, который был и остается жутким ляпом. Для *Go To* нет места в современном ПО в течение по крайней мере последнего десятилетия. В C++ и Java был более удачный механизм — *структурная обработка исключений* (structured exception handling), где сведения об ошибке содержит объект, который обрабатывается специальным блоком кода — обработчиком. К сожалению, подобно многим другим функциям языков, созданных до эпохи CLR, структурная обработка исключений работала лишь со «своим» языком. В COM была попытка предоставить богатый набор функций межъязыковой обработки исключений через интерфейсы *ISupportErrorInfo* и *IErrorInfo*, но такой подход оказалось трудно реализовать при программировании, и никто не мог дать гарантии, что остальные следовали тем же правилам, что и вы.

.NET поддерживает структурную обработку исключений как фундаментальную возможность, доступную в любом языке. Кроме того, функции SEH одного языка доступны из другого.

CLR .NET поддерживает структурную обработку исключений, похожую на таковую в C++ и Java, как фундаментальную функцию, доступную всем языкам. Такая архитектура решает многие проблемы, преследовавшие обработку ошибок в прошлом. Необработанное исключение приведет к аварийному завершению работы приложения, поэтому при

разработке их нельзя игнорировать. Функция, сообщающая об ошибке, помещает описание сбоя в объект .NET, который, таким образом, может содержать столько информации, сколько нужно сообщить. Так как эта инфраструктура встроена в CLR, чтобы задействовать ее преимущества, нужно написать совсем небольшой код. Как и остальная функциональность CLR, структурная обработка исключений .NET хорошо работает во всех языках. Я написал небольшую программу, демонстрирующую некоторые возможности структурной обработки исключений CLR (рис. 2-35). Можно загрузить исходный текст этой программы с Web-сайта этой книги и работать вместе со мной.



**Рис. 2-35.** Пример, демонстрирующий структурную обработку исключений.

Клиентская программа, прежде чем выполнить действие, которое, по ее мнению, может закончиться неудачей, готовит в программе блок обработки исключения (exception handler block) с использованием ключевых слов *Try* и *Catch*, как показано в листинге программы на Visual Basic.NET (рис. 2-36). Точный синтаксис структурной обработки исключений варьируется в зависимости от языка, но все, что я до сих пор видел, очень похоже на этот.

Когда программа при исполнении входит в блок *Try*, CLR записывает обработчик исключения в стек (рис. 2-37). Если вызванная функция, расположенная ниже в стеке, генерирует исключение, механизм обработки исключений CLR анализирует стек в направлении снизу вверх до тех пор, пока не найдет обработчик исключения. После этого стек

Блок *Try-Catch* позволяет задать в клиентской программе код для обработки исключений.

сжимается (все объекты удаляются из стека), и управление переходит к обработчику исключений. Исключение может возникнуть на любом уровне стека вызовов. В программе-примере я преднамеренно открыл файл, которого нет. Системный метод *File.Open* генерирует исключение, которое мой клиент перехватывает и выводит пользователю информацию об имевших место событиях.

```
Protected Sub btnHandled_Click (ByVal sender As Object, _
                                ByVal e As System.EventArgs)

    ' При входе в этот блок кода в стек записывается
    ' обработчик исключения.

    Try

        ' Выполнить операцию, которая заведомо приведет
        ' к возникновению исключения.

        Dim foo As System.IO.FileStream
        foo = System.IO.File.Open( _

            "Non-existent file".IO.FileMode.Open)

        ' При возникновении исключения на нижнем уровне
        ' программы, этот блок обработки перехватывает его.

    Catch x As System.Exception

        ' Выполнить любую необходимую очистку в ответ
        ' на перехваченное исключение.

        MessageBox.Show (x.Message)
    End Try
End Sub
```

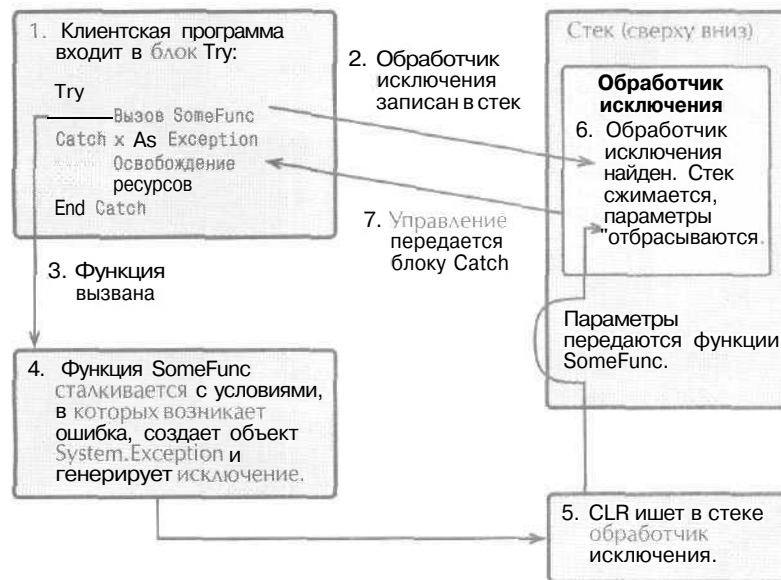
**Рис. 2-36.** Клиентское приложение, демонстрирующее структурную обработку исключений.

Исключение может быть сгенерировано любой программой. Как показано выше, CLR для всех сообщений об ошибках использует SEH. Вероятно, для согласования у вас возникнет желание через SEH уведомлять одну часть приложения об ошибках, возникших в другой. Фрагмент программы, которому требуется сгенерировать исключение, создает новый объект типа *System.Exception*.



Можно устанавливать свойства этого объекта таким образом, чтобы они служили для любых заинтересованных в ней перехватчиков поставщиками информации о ситуации, в которой возникло исключение. CLR автоматически производит трассировку стека, что позволяет обработчику исключений точно определить место возникновения исключения. Затем с помощью ключевого слова *Throw* можно сгенерировать исключение, как показано ниже (рис. 2-38). Вызов этой функции служит для системы сигналом к началу анализа стека для поиска обработчиков. Обработчик исключения может жить на любом уровне стека вызовов выше уровня, на котором возникло исключение.

Программа, которая собирается сгенерировать исключение, создает объект *System.Exception*, заполняет его поля и вызывает системную функцию *Throw*,



**Рис. 2-37.** Схема структурной обработки исключений.

Когда CLR передает управление обработчику исключения, содержимое стека программы между обработчиком и функцией, сгенерировавшей исключение, удаляется (рис. 2-37). При этом уничтожается любой объект или ссылка на объект. Автоматический

Блок *Try-Finally* позволяет освободить ресурсы после исключения.

сбор мусора в .NET позволяет не беспокоиться об утечках памяти, занятой объектами, что было большой проблемой при использовании встроенной обработки исключений C++.

Однако такое уничтожение объекта означает, что для объектов, которым требуется детерминированное завершение, у вас не будет шанса вызвать метод *Dispose*. Их завершители все-таки будут вызваны при следующем сборе мусора, но это может случиться слишком поздно. С подобной ситуацией поможет справиться обработчик *Try-Finally* (рис. 2-39). Команда из блока *Finally* выполняется по мере сжатия стека, поэтому в этот блок можно поместить код, необходимый для освобождения ресурсов. При желании можно объединить блоки *Catch* и *Finally* в одном и том же блоке *Try*.

```
Public Function BottomFunction () As String

    ' Создать новый объект исключение, установить
    ' его свойство "Message".
    ' Эти действия можно выполнить только в конструкторе.

    Dim MyException As New _
        Exception("Exception thrown by BottomFunction")

    ' Установить свойство Source нового объекта Exception
    ' Это можно сделать где угодно.

    MyException.Source = _
        "Understanding Microsoft .NET Chapter 2 ExceptionComponent"

    ' Сгенерировать исключение.

    Throw MyException

End Function
```

**Рис. 2-38.** Генерация исключения с помощью SEH.

Можно генерировать и перехватывать множество различных типов исключений.

Мощность SEH даже еще больше в тех случаях, когда возникают различные типы исключений, свидетельствующие о различных типах сбоев в программах. Для этого нужно создать собственный класс, производный от универсального базового класса *System.Exception*. К производно-

му классу «исключение» можно добавить любые дополнительные методы или свойства, которые, по вашему мнению, смогут прояснить ситуацию потенциальным перехватчиком. Даже если к этому классу ничего не добавлять сверх того, что уже имеется, простое наличие такого исключения позволяет определить его тип. В примере, который показан в начале раздела, при попытке открытия несуществующего файла система сгенерировала исключение типа *FileNotFoundException*. Я написал обработчик, который перехватывает исключения любого типа (рис. 2-35). Если бы я хотел, чтобы обработчик перехватывал *лишь* исключения типа *FileNotFoundException*, то заменил бы *Catch x As Exception* на *Catch x As FileNotFoundException*. При анализе стека CLR сравнивает тип сгенерированного исключения с типом, заданным в блоке *Catch*, и передает управление блоку, только если тип сгенерированного исключения точно совпадает с заданным для перехвата или является его производным. К блоку *Try* можно прикрепить любое число обработчиков *Catch*. CLR просматривает их по порядку, поэтому сначала лучше поместить самые конкретные.

```
Public Function MiddleFunction () As String

    ' При входе в этот блок в стек записывается
    ' обработчик исключения.

    Try
        BottomFunction ()

        ' Код из блока-обработчика Finally выполняется
        ' при выходе программы из блока Try независимо
        ' от причины. Больше всего нас беспокоит случай,
        ' в котором после того, как функция BottomFunction
        ' генерирует исключение, происходит
        ' сжатие стека. В отсутствие обработчика Finally
        ' у нас не было бы шанса выполнить освобождение
        ' ресурсов для этого исключения.

    Finally
        MessageBox. Show("Finally handler in MiddleFunction")
    End Try

End Function
```

**Рис. 2-39.** Использование блока *Finally*.

## Безопасность доступа к коду

Как правило, клиенты считают безопасным ПО, купленное в магазине.

В начале эры ПК лишь немногие пользователи устанавливали программы, купленные не в магазине. Тот факт, что продукция какого-либо производителя попала на полки CompUSA или, позднее, Egghead Software, убеждал

покупателя в том, что ПО из коробки не содержит вирусов, поскольку ни один мерзкий мошенник не мог позволить себе такие издержки на маркетинг. А пленочная упаковка, как на коробках с тайленолом, служила для клиента гарантией неприкосновенности продукта с момента отгрузки его производителем. Хотя в программах могли быть и, вероятно, были ошибки, которые время от времени служили причинами всяких проблем, вы чувствовали себя вполне комфортно и не опасались, что они разрушат данные на жестком диске (для которого вы еще не сделали резервную копию) просто из вредности.

Однако в настоящее время основная часть ПО поступает из Web.

Такая модель безопасности сегодня работает плохо, так как большая часть ПО теперь поступает не из магазинов. Самые большие пакеты вроде Microsoft Office или Visual Studio

устанавливаются с CD, хотя мне не ясно, как долго еще существует такая практика при распространении скоростных соединений с Интернетом. А как насчет обновлений, скажем, для Internet Explorer? Или новой игрушки типа тетриса? Производители любят распространять ПО через Web, поскольку это дешевле и легче, чем торговля через розничную сеть, а потребителям этот способ нравится из-за удобства и низких цен. Но мир программ Web не ограничен только тем, что традиционно считается прикладным ПО. Web-страницы могут содержать разные сценарии, и порой небезобидные. Даже в документах Офиса, пересылаемых по электронной почте, могут быть макросы. Вероятно, число загруженных из Web программ на вашем компьютере больше, чем установленных с купленных CD (не считая ОС), и это соотношение будет только расти,

Хотя, с точки зрения производителя, распространять ПО через Web замечательно, это ведет к возникновению проблем безопасности, с которыми мы раньше не сталкивались. Теперь злоумыш-

леннику намного проще сеять зло с помощью вирусов. Не проходит и месяца, чтобы по CNN не передавали предупреждение о новом вирусе, так что проблема достаточно остра, чтобы регулярно привлекать внимание попу-

лярных СМИ. Эксперты в области безопасности советуют работать лишь с программами, присланными лишь хорошо известными вам людьми, но через кого же еще будут распространяться почтовые вирусы, как не через них? И как опробовать ПО, полученное от компаний, о которых мы никогда раньше не слышали? Пользователю практически невозможно узнать, когда загруженный из Web код безопасен, а когда — нет. Даже посвященные могут повредить систему, запуская вредоносное или сбойное ПО. Можно запретить своим пользователям работу со всеми программами, кроме тех, которые установлены лично сотрудниками отдела ИТ. Но не попробуйте использовать такую политику хотя бы в течение одного рабочего дня, и вы увидите, как упадет производительность труда. Мы стали настолько зависимы от программ, загружаемых из Web, что поверить в это можно, лишь попытавшись жить без них. Единственное, что удерживает общество в том виде, в каком оно существует, от разрушения — это то, что люди, сочетающие в себе деструктивные наклонности и технические навыки для их реализации встречаются сравнительно редко.

Первой попыткой Microsoft обезопасить код в Web была система Authenticode, вошедшая в жизнь вместе с ActiveX SDK в 1996 г. С помощью Authenticode производители могли прикреплять к загружаемым элементам управления цифровые подписи. Это давало пользователям некоторую уверенность, что

этот элемент управления действительно был прислан лицом, указанным как его отправитель, и целостность его не нарушена с момента подписания. Система Authenticode неплохо работала, гарантируя, что последние обновления для Internet Explorer действительно идут от Microsoft, а не от какого-нибудь злобного шутника. Но в этой системе Microsoft пыталась воспроизвести меры безопасности розничного магазина, не отдавая себе отчета в том,

Для пользователя практически невозможно узнать, безопасен ли код, загружаемый из Web-»,

Система Authenticode не защищает от вредоносных действий, а просто позволяет идентифицировать лицо, нанесшее вам ущерб.

что их уже недостаточно в мире Интернета. Поверхностная проверка лиц, получающих цифровые сертификаты, не дает уверенности в отсутствии у производителей злых намерений (Verisign, как идиоты, продали мне сертификат всего за 20 долларов), которую з большей или меньшей степени давал факт наличия товара на полках магазина или в каталоге «товары — почтой». Что хуже всего, система Authenticode работала по принципу «все или ничего». Она позволяла более или менее определенно идентифицировать отправителя программы, но давала пользователю лишь единственный выбор: устанавливать программу или нет. После того, как программа попадала в систему, защититься от ее вредоносных действий было нельзя. Authenticode нельзя назвать системой безопасности — скорее это система отчетности. Она не защищает от вредительства, а лишь позволяет выявить вредителя.

Необходимо задавать уровень привилегий для отдельных программ, как для людей в реальной жизни.

Нам действительно нужен способ ограничения действий, которые может выполнить программа, основанный на уровне доверия. В жизни разным людям вы назначаете различные уровни доступа к вашим ресурсам в соответствии с уровнями доверия к ним: (те-

кущая) супруга может позаимствовать у вас кредитную карточку, друг — старую машину, сосед — шланг для полива газона. Хотелось бы, чтобы ОС поддерживала аналогичные различия. Например, ОС должна реализовывать ограничения, позволяющие загруженному из Интернета элементу управления обращаться к пользовательскому интерфейсу, но запрещающие доступ к дисковым файлам, если только он не прислан одним из немногих производителей, которым мы привыкли доверять. ОС Win32 не поддерживают функциональность такого типа, так как это изначально не входило в задачи их разработки. Но теперь у нас есть CLR, которая ее поддерживает.

.NET CLR поддерживает безопасность доступа к коду (code access security), позволяющую администратору задать привилегии для каждой сборки управляемого кода в зависимости от уровня доверия (если оно вообще есть).

Когда управляемый код вызывает CLR для обращения к защищенному ресурсу, скажем, чтобы открыть файл или получить доступ к Active Directory, CLR проверяет, предоставил ли администратор этой сборке соответствующую привилегию (рис. 2-40). При выполнении вызова CLR просматривает весь стек до верха, что не позволяет недоверяемой сборке, находящейся на верхнем уровне, обмануть систему безопасности через доверяемого «приспешника», расположенного в стеке ниже (даже если за вашей дочерью в школу придет монахиня, вы бы все равно хотели, чтобы учитель проверил, что она прислана вами, не так ли?). Хотя такая проверка снижает скорость доступа к защищенному ресурсу, другого приемлемого способа, который позволил бы избежать дыр в защите, нет. Когда CLR не может проконтролировать действия неуправляемого кода (например, объекта COM, который имеет дело непосредственно с ОС Win32, а не с CLR), предоставить или отнять привилегию доступа к неуправляемому коду может администратор.

.NET CLR поддерживает безопасность доступа к коду в период выполнения для отдельных сборок.

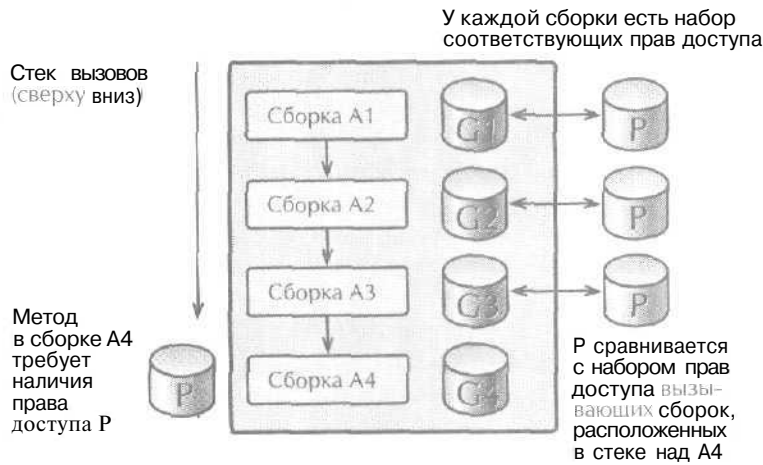


Рис. 2-40. Проверка прав доступа к коду в CLR.

Администратор устанавливает *политику безопасности* (security policy) — настраиваемый набор правил, определяющий, какие типы действий разрешены, а какие — запрещены и для каких сборок. Права доступа можно устанавливать на трех уровнях: предприятия, машины и пользователя. На более низком уровне можно ввести более строгие ограничения, чем на более высоком, но не наоборот. Так, если права доступа, заданные на уровне машины, разрешают сборке открывать файл, права доступа на уровне пользователя могут запрещать это, но не наоборот.

Администратор устанавливает политику безопасности доступа к коду путем редактирования XML-файлов конфигурации,

Администратор устанавливает политику безопасности, редактируя XML-файлы конфигурации, хранящиеся на диске машины. Точный адрес и внутренняя схема размещения этих файлов почти наверняка изменятся в конечном выпуске .NET Framework. Но в настоящее время параметры уровня машины

хранятся в файле `\\WINNT\\Microsoft.NET\\Framework\\[текущая версия]\\machine.config`. Пока предполагается, что администратор будет манипулировать этими файлами, используя программу командной строки `caspol.exe`. Она показалась мне очень сложной в использовании, даже по ужасным стандартам этой отрасли. Если Microsoft не будет поставлять утилиту с графическим интерфейсом, с которой будет легче работать, пользователь будет практически ограничен параметрами по умолчанию, так как никто и никогда не сможет их изменить. Тратить все усилия на разработку инструментов вроде IntelliSense для помощи программистам в ускорении создания приложений, заявлять об этом во всеуслышание, чтобы потом обложить их такими вот отвратительными «утилитами», — это в лучшем случае признак шизофрении. Увидев 2 года назад COM+ Explorer, я подумал, что наконец-то Microsoft усвоила мысль о необходимости полезных инструментов для администрирования. Наверное, эта мысль снова забыта.

Администратор задает права доступа, предоставляемые сборкам, в терминах наборов прав доступа (permission sets). Их смысл (хоть раз!) точно соответствует названию: это списки привилегий, которые могут быть предоставлены или отозваны как единое целое. Конфигурация безопасности по умолчанию состоит из нескольких наборов прав доступа (табл. 2-2).



Табл. 2-2. Права доступа в конфигурации по умолчанию.

Именованный набор прав доступа					
Право	<i>Everything</i>	<i>Local Intranet</i>	<i>Internet</i>	<i>Nothing</i>	
Dns Permission	Все встроенные права доступа без ограничений, кроме Security-Permission (чтобы пропустить проверку).	Без ограничений	—	—	
Environment		USERNAME TEMP TMP	—	—	
FileDialog		Без ограничений (доступ только для чтения)	Без ограничений (доступ только для чтения)	—	
FileIO		—	—	—	
Isolated Storage		AssemblyIsolationByUser, квота без ограничений	DomainIsolationByUser, 10240 байт, 365 дней	—	
Reflection		Без ограничений	—	—	
Registry		—	—	—	
Security		Execution, Assertion	Execution	—	
Socket Permission		Connect/accept, сайт происхождения	—	—	
UI		Без ограничений	SafeWinSubdows, Own-Clipboard	—	
Web Permission		Connect, сайт происхождения	—	—	
Зона	<i>Local (My Computer)</i>	<i>Intranet</i>	<i>Trusted Internet Sites</i>	<i>Restricted Sites</i>	<i>All Code*</i>

\* Учитываются все соответствующие группы кода,

При желании можно модифицировать эти или создавать собственные наборы прав доступа. Каждый набор состоит из одного или нескольких *прав доступа* (permissions). Право доступа — это право на выполнение одного конкретного действия, например, на открытие файлов, выбранных *пользователем* в системном диалоговом окне Open File; возможно, с оговоркой, что файлы будут открыты только для чтения. Пример XML-файла с описанием одного права доступа показан ниже (рис. 2-41).

A) Неограниченные права доступа ко всем переменным окружения

```
< Permission class=
    "System.Security.Permissions.EnvironmentPermission">
    <Unrestricted/>
</Permission>
```

B) Доступ только для чтения к переменным USERNAME, TEMP и TMP

```
<Permission class=
    "System.Security.Permissions.EnvironmentPermission">
    <Read>USERNAME; TEMP; TMP</Read>
</Permission>
```

**Рис. 2-41.** Выдержка из XML-файла, описывающего одно право доступа.

Администратор конструирует наборы прав доступа — списки прав, которые предоставляются и отзываются одновременно как единая группа.

В наборе Everything (раздел A на рис. 2-41) право на использование переменных окружения предоставляется без ограничений, но в наборе LocalIntranet (раздел B на рис. 2-41) оно ограничено правами на чтение переменных USERNAME, TEMP и TMP. Имейте в виду, что именно состав наборов прав доступа и их внутренняя XML-структура скорее всего подвергнутся изменениям за время, которое пройдет между написанием этой книги и выходом окончательной версии этого ПО.

Теперь вы знаете, какими наборами прав доступа может манипулировать администратор. Рассмотрим способ присвоения набора прав доступа сборке. Администратор привилегий программ может присвоить набор прав доступа определенной сборке так же, как администратор регистрации пользователей назначает

отдельным пользователям определенные привилегии при регистрации. Однако обе методики очень быстро стали громоздкими. Большинство администраторов регистрации создают группы пользователей (рабочие, служащие и т. д.), члены которых имеют привилегии одного уровня, и перемещают отдельных пользователей между этими группами. Аналогично большинство администраторов привилегий программ будет создавать группы сборок — *группы программ* (code groups) — и назначать им наборы прав доступа. Основное различие между задачами администраторов регистрации и привилегий программ в том, что первые определяют принадлежность к группе каждого пользователя вручную, так как новые пользователи появляются в системе не так часто.

Администратор присваивает группам программ наборы прав доступа.

Способ загрузки программ из Web не позволяет положиться на человека при принятии решения о доверии всякий раз, когда браузер встречает новую сборку. Поэтому администратор привилегий программ устанавливает условия членства (membership conditions), определяющие, каким образом сборка будет приписана к той или иной группе программ. В условиях членства обычно указана зона, откуда получена сборка, например, My Computer, Internet, Intranet и т. д. В условиях членства входят строгое имя сборки (например, «эта сборка написана нашими разработчиками») и открытый ключ, которым она подписана («сборка создана Microsoft»). Загружая сборку в период выполнения, CLR определяет группу программ, к которой принадлежит сборка, путем проверки ее условий членства и присваивает ей набор прав доступа, заданный для этой группы программ. Ниже показана сокращенная версия сложного XML-файла, согласно которому сборкам из зоны My Computer присваивается набор прав доступа FullTrust (рис. 2-42). Теперь ясно, что я имел в виду, говоря о хороших инструментах для администрирования?

Администратор устанавливает правила принадлежности сборки к группе программ,

Хотя основная часть работы с безопасностью доступа к программам достается системным администраторам, программистам тоже время от времени приходится писать программы, которые

CLR поддерживает много функций и объектов для программного взаимодействия с системой безопасности доступа к программам.

имеют дело с системой безопасности доступа к программам. Например у программиста может возникнуть желание добавить к сборке атрибуты, которые задают наборы прав доступа, необходимые для ее работы. Эти атрибуты не влияют на уровень прав доступа, предоставляемых сборке, так как на-

ходятся под контролем вышеописанных административных настроек. Однако они позволяют CLR сразу запретить загрузку сборки, а не ждать, когда сборка попытается выполнить запрещенную ей операцию. С другой стороны, при одних способах использования сборки надо выполнять запрещенные операции, а при других это не требуется. В последнем случае эти операции можно запретить. Программисту также может потребоваться прочитать уровень прав доступа, который был реально предоставлен сборке, чтобы проинформировать пользователя или администратора о том, чего не хватает. В CLR много функций и объектов, которые позволяют программисту писать программы, взаимодействующие с системой безопасности доступа к программам. Даже поверхностное рассмотрение этих функций выходит далеко за рамки этой книги. Но об их существовании все же следует знать, так как при желании с ними можно работать; но скорее всего такого желания у вас никогда не возникнет. Если установить административные права доступа, как здесь описано, то «хорошие» сборки смогут выполнять полезные действия, а «плохие» — не смогут нанести вреда.

```
<ICodeGroup class="System.Security.Policy.UnionCodeGroup">
  <IMembershipCondition class=
    "System.Security.Policy.ZoneMembershipCondition">
    <Zone>MyComputer</Zone>
  </IMembershipCondition>
  <PermissionSet class="System.Security.NamedPermissionSet">
    <Name>FullTrust</Name>
  </PermissionSet>
</ICodeGroup>
```

**Рис. 2-42.** Выдержка из XML-файла, описывающего группу программ и права членства для предоставления набора прав *FullTrust*.

---

### Примечание

Когда я вычитывал гранки книги за неделю до ее сдачи в печать, Microsoft объявила, что под сильным давлением Visual Basic-разработчиков пришлось снова изменить принципы работы массивов. Массивы все равно будут начинаться с нулевого элемента, но при этом Visual Basic.NET будет автоматически выделять дополнительный элемент наверху массива, что избавит вас от необходимости переписывать имеющиеся программы. Объявляя массив оператором *Dim X(5) As Integer*, вы на самом деле получите массив из 6 элементов, пронумерованных от 0 до 5. Это значит, что менять имеющиеся программы на Visual Basic вовсе не обязательно. Теперь проблема в том, что при совместном использовании программ, написанных на Visual Basic, с языками, в которых массивы начинаются с нулевого элемента (такими как C# и Java), другой разработчик будет знать, что нулевой элемент на самом деле есть, но не будет знать, пуст он или нет. Похоже, что проблема взаимозаменяемости программ, в которых массивы начинаются с 0 или 1 элемента, конца которой я ждал так долго, по чьей-то воле получила новую жизнь. Помоему, намного лучше было бы покончить с ней раз и навсегда. Возможно, Microsoft вновь корчит из себя мудреца в надежде, что разработчики, использующие массивы, которые начинаются с нулевого элемента, падут на колени с криком: «О, нет! Что угодно, только не эта путаница! Мы сдаемся, пусть все они начинаются с 1!» Надеюсь, эти изменения будут отвергнуты.



## Глава 3

# ASP.NET

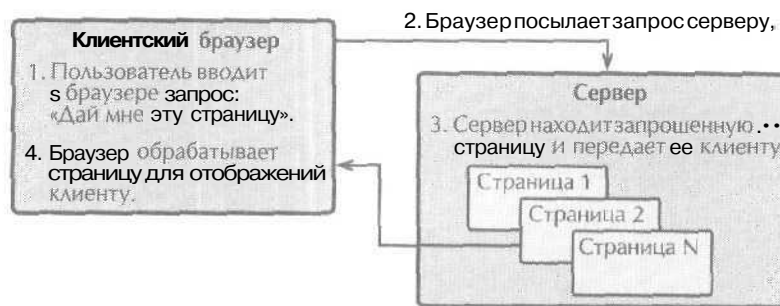
*И голоса наперебой — и все понятны мне —  
Звучат на скорости любой и при любой волне.  
От крышки люка до котла — единое стремление.  
Как зорька ясная, светла их Песнь Благодаренья.*

Р. Киплингo взаимодействию и масштабируемости.  
(«Молитва МакЭндрю», 1894 г.)

### Суть проблемы

Изначально Web служила для передачи статических страниц с текстом и изображениями. Запрограммировать сервер для этого было относительно просто — нужно просто принять URL, идентифицирующий файл, взять этот файл с диска сервера и передать этот файл клиенту (рис. 3-1). Даже такая простая архитектура позволяет многое. Например, наш местный кинотеатр имеет небольшой Web-сайт, на котором я могу посмотреть, что идет сегодня (чаще всего «Счастливы, Техас!», в котором двух беглых эзков облапошивают красотки в небольшом техасском городке), что будет скоро, и по ссылкам просмотреть анонсы и отзывы. Они используют Web как бумажную рекламную брошюрку, правда с более привлекательным содержанием, быстрой доставкой и меньшими затратами — словом (ну, может, двумя), более адекватно.

Сначала Web  
использовали для  
просмотра статических  
страниц, которые было  
относительно легко  
программировать.



**Рис. 3-1.** Сервер передает статические Web-страницы.

Сегодня Web-программисты должны уметь динамически создавать HTML-страницы в ответ на данные, полученные от пользователя, что порождает новые проблемы.

Этот подход был приемлем в доисторические времена, когда все данные в Web были статическими (отображая только созданное автором содержание без программной логики и возможности вмешательства со стороны пользователя) и открытыми (доступными всем, кто знал или мог найти адрес). Но со временем потребители стали спрашивать:

«Если я могу узнать, какой сегодня фильм, почему я не могу посмотреть текущее состояние своего банковского счета?» Но при статическом подходе это неосуществимо. Банк не может каждый день создавать новую страницу для любого возможного представления каждого счета — страниц окажется слишком много. Вместо этого пользователю нужно ввести некоторую информацию, такую как номер счета, а банковский компьютер должен по запросу создать HTML-страницу, отображающую баланс клиентского счета (рис. 3-2). Такие данные не являются ни статическими, ни открытыми, а это порождает новые проблемы.

Web-серверным приложениям нужна программная логика для генерации страниц.

Web-серверному приложению, динамически генерирующему страницы для клиента, требуется несколько вещей. Во-первых, ему нужен способ увязки программной логики с запросом страницы.

Когда пользователь запрашивает страницу, сервер не может просто извлечь ее с диска — до этого запроса ее просто нет. Вместо этого сервер должен выполнить некий алгоритм для генерации страницы. В при-



мере с банком нам, вероятно, потребуется просмотреть банковскую БД и найти текущий баланс пользовательского счета и последние транзакции. Нам бы хотелось реализовать этот алгоритм легко и быстро, с применением уже знакомых языков и инструментов. И хотелось бы, чтобы это работало настолько быстро и эффективно, насколько это возможно в реальной жизни.



**Рис. 3-2.** Сервер динамически генерирует Web-страницы на основе полученной от клиента информации.

Кроме того, нашему Web-серверу нужен способ передачи вводимой информации серверному алгоритму и возврата выходной информации от этого алгоритма пользователю. Пользовательский браузер передает серверу HTML-форму, содержащую элементы управления для ввода данных, которые определяют интересующую его информацию, например, номер счета и период времени, за который он хочет просмотреть транзакции. Извлечение данных с этой HTML-страницы — процесс нудный, поэтому нам бы хотелось иметь готовый способ запрограммировать это быстро и просто. Подумайте, насколько проще текстовое окно (для фанов C++ — управляющий элемент редактирования) позволяет читать введенные символы в сравнении с самостоятельной сборкой строки из отдельных символов. Нам хотелось бы иметь примерно такой же уровень готового сервиса для сборки HTML-страницы, передаваемой пользователю. Просто на HTML писать утомительно. Насколько проще иметь готовый элемент управления для текстового поля в пользовательском интерфейсе Windows, чем писать GDI-вызовы для установ-

Нашему Web-серверу нужен удобный способ получения вводимой пользователем информации и передачи выходных данных на браузер пользователя.

ки шрифта, цвета, текста и т. д. Что-то подобное нам нужно для создания HTML, передаваемого браузеру.

Нашему Web-серверу нужны средства защиты, чтобы неавторизованные пользователи не могли видеть или делать то, что им не положено.

В-третьих, поскольку по крайней мере некоторые данные теперь являются частными, наш Web-сервер должен знать, кем является пользователь, чтобы позволять ему просматривать и делать только то, что ему разрешено. Вам бы хотелось видеть свой счет, но не хотелось бы, чтобы его увидела злюка

бывшая жена или тем более сняла с него деньги. Написание такого кода — дело сложное и дорогое, и убедить недоверчивых заказчиков з его пуленепробиваемости ничем не легче и не дешевле. Нам нужна инфраструктура с готовыми средствами защиты.

Нашему Web-серверу нужны средства управления сеансами.

И, наконец, нашему Web-серверу нужен механизм управления пользовательскими сеансами. Для пользователя взаимодействие с Web-сайтом — это не последовательность

запросов, но диалог, «сеанс». Он ожидает, что Web-сайт может запоминать, что ему было сказано пару минут назад. Скажем, на сайте электронного магазина он хочет складывать товары в тележку так, чтобы они хранились там до расчета. Чтобы этого добиться, нужен соответствующий код — запрос к отдельной странице не способен поддерживать такую функциональность. Это опять же неотъемлемая часть большинства Web-приложений, так что хотелось бы иметь готовую, простую в использовании реализацию. В идеале это должно работать в многосерверной среде и быть устойчивым к сбоям.

Нам нужна законченная среда периода выполнения.

Короче, нашему серверу нужна исполняющая среда, предоставляющая готовые решения проблем, которые возникают при программировании всех Web-серверов. Хотелось бы, чтобы она упрощала программирование, администрирование и развертывание. Нужно чтобы ее можно было распространять хотя бы на несколько, а лучше — на множество серверов. И платить за это много не хочется. А больше нам ничего не нужно?

## Архитектура решения

В конце 1997 г. Microsoft реализовала относительно простую среду периода выполнения для Web — Active Server Pages (ASP) как часть сервера Internet Information Server (IIS), включенного в Windows NT 4 Option Pack. IIS об-

ASP была исполняющей средой, которую можно было легко применять для простых задач,

служивает Web-страницы, запрашиваемые пользователем. ASP позволяет программистам реализовывать алгоритмы динамического создания страниц на IIS, состоящих из статического HTML и кода сценариев (рис. 3-3). Когда пользователь запрашивает ASP-страницу, IIS должен ее найти и активизировать ASP-процессор. ASP-процессор должен прочитать страницу и один к одному скопировать содержащиеся на ней HTML-элементы в выходную страницу. В нашем примере атрибут `style` устанавливает голубой цвет текста. При этом также интерпретируются элементы сценариев, расположенные между ограничителями `<% %>`. Этот код должен выполнять алгоритм, выдающий в качестве результата HTML-строки, которые ASP-процессор должен скопировать в выходную страницу в те места, где были элементы сценария. Результирующая страница, собранная из статических HTML-элементов и HTML, динамически сгенерированного сценарием, должна быть передана клиенту (рис. 3-4). Для простых задач ASP применять относительно легко, что является признаком качества этой технологии.

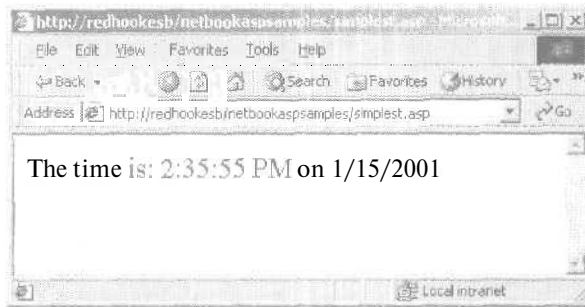
```
<html style="color:#0000FF;">  
  The time is: <% =time %> on <% =date %>  
</html>
```

**Рис. 3-3.** Смешение кода и HTML в ASP.

По мере расширения Web и увеличения потребностей пользователей Web-разработчикам потребовались совершенствование двух ключевых свойств исполняющей среды: простоты программирования и качества выполнения. ASP.NET и явилось таким усовершенствованием. ASP.NET похожа на оригинальную ASP и большая часть кода может быть переведена на нее практически без изменений. Но внутренняя

дер.NET — полностью переделанная ASP, сохранившая ее лучшие концепции.

реализация ASP.NET полностью переделана с тем, чтобы задействовать возможности .NET Framework. Вы обнаружите, что работает она лучше, рушится меньше и в ней проще программировать. И это приятно.



**Рис. 3-4.** Web-страница, созданная ASP после обработки смеси код/HTML, показанной на рис. 3-3.

Смесь HTML-элементов и сценарного кода может казаться логичной, но ее чертовски трудно реализовывать и сопровождать, кроме простейших случаев. Поскольку код и данные должны и появляются в любом месте страницы, развитые средства разработки, подобные Visual Basic, не могут этого переварить, так что никому не удастся создать среду разработки, полноценно поддерживающую ASP. Это значит, что писать ASP-код сложнее, чем другие виды кода, например приложения с пользовательским интерфейсом на базе форм Visual Basic. Я знаю, что я на этом помешан.

ASP.NET отделяет ваш код от HTML.

ASP.NET отделяет HTML от алгоритмов, создавая фоновый код (code-behind). Вместо того чтобы перемешивать HTML с кодом, вы пишете код в отдельном файле, на который есть ссылка на ASP-странице. Вы будете потрясены, насколько проще понять код, если убрать из него отвлекающий внимание HTML. Это все равно, как если ваше трехлетнее чадо вдруг перестанет орать, когда вы обсуждаете по телефону проблемы своих налогов. В результате такого разделения Microsoft смогла усовершенствовать среду разработки и отладки Visual Studio.NET, так что вы можете использовать ее при разработке Web-приложений.

В оригинальной ASP реализация ввода и вывода представляла сложности из-за невозможности отвлечься от HTML. Я имею в виду, что программист часто вынужден бороться с довольно неприглядными языковыми конструкциями HTML, вместо того чтобы думать о логике своей программы, а это не лучшее использование ресурсов. Скажем, получение данных из HTML-форм требует больше усилий, чем в настольных приложениях. Генерация выходной информации требует, чтобы программист собирал отдельные фрагменты HTML, а это опять же не то, на что программист должен тратить время. Чтобы написать код, показанный ниже (рис. 3-3), программисту нужно знать синтаксис HTML, позволяющий задать голубой цвет текста.

ASP.NET поддерживает *Web Forms* — архитектуру Web-страниц, делающую их программирование похожим на программирование форм настольных приложений. Вы добавляете на страницу управляющий элемент и пишете для него обработчик событий так же, как при написании настольного приложения на Visual Basic. Простота использования, сделавшая Visual Basic столь популярным, теперь доступна при построении Web-приложений. Как раз перед сдачей этой книги в печать я проводил занятия по первой бета-версии ASP-NET в группе бывалых ASP'ешников, и эта возможность заставила их буквально встать и зааплодировать.

Так же, как Visual Basic зависит от элементов управления Windows и элементов ActiveX сторонних производителей, Web Forms зависит от нового типа элементов управления — серверных элементов управления *Web Forms* (*Web Forms Server Controls*). В этой главе я буду для простоты называть их элементы управления Web. Это готовые фрагменты алгоритмов для реализации ввода и вывода, которые разработчик помещает на ASP.NET-страницы так же, как он поступал с формами для Windows-приложений. Эти элементы позволяют абстрагироваться от деталей работы с HTML, как Windows-элементы позволяли абстрагироваться от GDI. Например, вам больше не нужно помнить синтаксис HTML для установки основного цвета и цвета фона для строки текста. Вместо этого вы будете использовать элемент управления «надпись» и писать код для работы с

В ASP.NET есть готовые элементы управления, применяющиеся для HTML-страниц так же, как элементы управления Windows в настольных приложениях.

его свойствами, как вы это делали на любом языке программирования. Представьте, насколько легче стало программировать настольные приложения на Visual Basic. Элементы управления Web делают то же и для ASP.NET-страниц.

ASP.NET имеет готовые средства защиты приложений.

Изначально ASP слабо поддерживала защиту. В ASP.NET реализовать защиту гораздо проще. Работая в «чистой» Windows-среде, вы можете аутентифицировать пользователя

(проверить, что он тот, за кого себя выдает) автоматически, используя встроенные средства аутентификации Windows. Для большинства конфигураций, содержащих не только Windows-приложения, ASP.NET имеет готовые средства для создания собственных схем аутентификации. Кроме того, она поддерживает инициативу Microsoft Passport, если решите идти этим путем.

ASP.NET имеет новые функции управления сеансами, пригодные для многомашинных сред.

ASP поддерживала управление сеансами с помощью простого API. Главными недостатками были невозможность распространить управление сеансами на более чем одну машину и прерывание сеансов при перезапуске процесса. ASP.NET расширяет поддержку,

позволяя выполнять обе функции автоматически. Вы можете настроить ASP.NET для автоматического сохранения и последующего восстановления состояния сеанса или на определенной машине (если вы хотите использовать более одной машины, но вас не интересует постоянное хранение этой информации), или на Microsoft SQL Server.

ASP.NET имеет много функций, облегчающих работу и администрирование.

ASP.NET имеет также много новых функций периода выполнения. Первоначальная ASP выполнялась медленнее, так как код сценариев интерпретировался во время выполнения. ASP.NET автоматически компилирует страницы при их первоначальной установке или при первом обращении, что здорово ускоряет работу. Поскольку она применяет компиляцию по требованию, принятую в .NET Framework, вам не нужно останавливать сервер для замены компонента или страницы. Она также поддерживает утилизацию процессов. Вместо

того чтобы *оставлять* процесс работающим бесконечно, ASP.NET можно настроить на автоматическое прекращение и повторный запуск серверного процесса после определенного периода времени или числа сбоев (эти значения настраиваются). Это снимает большинство проблем с утечкой памяти в пользовательском коде. Потенциально ASP.NET проще администрировать, так как все параметры хранятся в читабельном виде в самой иерархии каталогов ASP. К *сожалению*, во время написания этой книги административные утилиты, облегчающие эту работу, находились в стадии разработки.

## Простейший пример: создание простой страницы ASP.NET

Рассмотрим *простейший* пример ASP.NET (рис. 3-5). Он показывает текущее время на сервере (с секундами или без) и пользователь может выбрать цвет текста. Вы можете скачать код с Web-сайта [www.introducingmicrosoft.net](http://www.introducingmicrosoft.net) и использовать его при обсуждении. На этом же сайте вы найдете и саму эту страницу и сможете исследовать ее поведение по мере того, как я буду рассказывать, что и как работает. Эту страницу вы легко установите на собственный ASP.NET-сервер.

Начало  
демонстрационной  
программы ASP.NET.

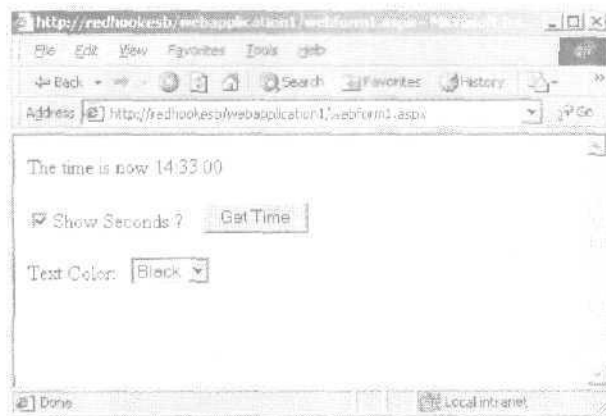


Рис. 3-5. Простейший пример ASP.NET.

Я слепил этот пример в Visual Studio.NET. Пытаясь написать примеры, не используя возможности среды разработки, я обнаружил, что Visual Studio.NET имеет столь мощную встроенную поддержку ASP.NET, что не задействовать ее — все равно, что писать в Блокноте настоящие Windows-приложения на Visual Basic (хотя, конечно, вы можете это делать, если ОЧЕНЬ этого хотите).

Ключ к пониманию этого примера — рассматривать вашу Web-страницу как форму Visual Basic. Я использую эту аналогию, так как предполагаю, что мои читатели с ним знакомы, но вы можете написать тот же код на любом другом языке Visual Studio.NET: C#, C++ или JavaScript. Я начал с создания проекта Web Application в Visual Studio.NET (рис. 3-6.). При этом, помимо всего прочего, была создана ASP.NET-страница WebForm1.aspx. Затем с инструментальной панели я перенес на форму элементы управления Web (рис. 3-7), добавив флажок, кнопку, выпадающий список и несколько надписей. Затем установил свойства этих элементов управления, используя окно Properties в правом нижнем углу экрана Visual Studio.NET, указав текст для каждого элемента, размер шрифта для надписей и элементы в выпадающем списке. Я установил свойство *AutoPostBack* выпадающего списка в True, чтобы форма автоматически пересылалась серверу, когда пользователь выберет новое значение в этом списке.

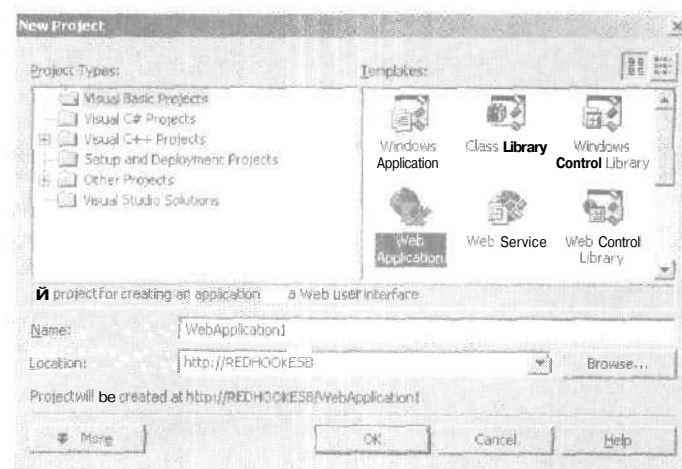
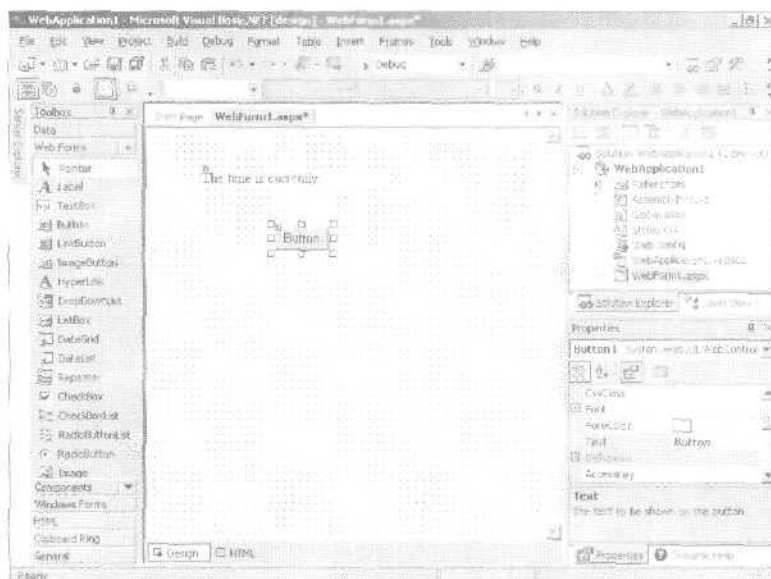


Рис. 3-6. Выбор WebApplication в VisualStudio.NET.





**Рис. 3-7.** Использование панели инструментов для перетаскивания элемента управления Web на форму.

Расположив свои элементы управления, мне нужно написать код, который их свяжет все вместе. При создании проекта Visual Studio.NET создала также класс для кода моей ASP.NET-страницы. Вы можете его

Вы помещаете элементы управления Web на свои ASPX-страницы, как на форму Visual Basic.

увидеть, щелкнув правой кнопкой страницу и выбрав в контекстном меню View Code, так же как вы это делаете сегодня с формами Visual Basic 6.0. Элементы управления Web уницируют события для своих форм, опять же аналогично с Visual Basic, так что мне нужно написать для них обработчики событий. Я могу добавить обработчик события, выбрав элемент управления в раскрывающемся списке в левом верхнем углу и событие в раскрывающемся списке в верхнем правом углу. Ниже приведен фрагмент моего Visual Basic-класса (рис. 3-8), а весь код можно загрузить с сайта этой книги. В этом простом примере, когда кнопка уницирует событие Click, я изменяю свойство текста надписи, показывающей текущее время. Я также добавил обработчик для события SelectedIndexChanged выпадающего списка. Когда пользо-

ватель выбирает цвет из выпадающего списка, я устанавливаю свойство цвета надписи разным значению, выбранному пользователю. При построении проекта Visual Studio.NET автоматически располагает его в корневом каталоге Web на моей машине,

```
Public Class WebForm1
    Inherits System.Web.UI.Page

    ' Пользователь нажал кнопку. Получаем время и отображаем
    ' его в элементе управления "надпись".

    Public Sub Button1_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs)
        If CheckBox1.Checked = True Then
            Label1.Text = "The time is now " + _
                now.ToLongTimeString
        Else
            Label1.Text = "The time is now " + _
                now.ToShortTimeString
        End If
    End Sub

    ' Пользователь выбрал другой цвет из списка.
    ' Изменяем цвет надписи в соответствии
    ' с выбором пользователя.

    Protected Sub DropDownList1_SelectedIndexChanged(ByVal _
        sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles DropDownList1.SelectedIndexChanged
        Label1( ).ForeColor = _
            Color.FromName(DropDownList1( ).SelectedItem.Text)
    End Sub

End Class
```

**Рис. 3-8.** Фрагмент кода для страницы WebForm1.aspx.

Обработчики событий  
элементов управления  
пишутся так же, как для  
форм Visual Basic.

Теперь, закончив писать код, я хочу увидеть его в действии. Простейший способ это сделать — щелкнуть правой кнопкой страницу и выбрать в контекстном меню Build and Browse. В результате в среде Visual Studio.NET откроется окно браузера. Когда браузер (или любой другой клиент) запрашивает .aspx-страницу, IIS загружает ее в исполняющее ядро

ASP.NET, которое затем анализирует страницу. Первый раз встретив страницу, ядро компилирует ее код. Последующие запросы той же страницы приведут к загрузке и выполнению этого скомпилированного кода. Затем ядро выполняет скомпилированный код класса, создавая элементы управления Web, которые я поместил на форму. Элементы управления работают только на сервере, создавая HTML, отображающий их текущее состояние, который встраивается в страницу, передаваемую клиенту. Ядро также исполняет код обработчиков событий и создает соответствующий HTML. Результирующий HTML передается клиенту, в результате чего получается показанная выше страница (рис. 3-5).

Исполняющее ядро вызывает класс, связанный с ASP.NET-страницей, и генерирует HTML из ее элементов управления и кода.

Вот и все. Гораздо проще, чем со старой ASP.

## Еще кое-что об управляющих элементах Web

Элементы управления Web — детище той же философии, что легла в основу создания пользовательского интерфейса Windows 15 лет назад. Та архитектура сделала Билла Гейтса самым богатым человеком в мире (на момент написания этой книги, хотя сейчас это, может быть, уже и не так, если на фондовом рынке произошли перемены). Это и есть мое определение успешной архитектуры — насчет вас не знаю.

Операции ввода/вывода повторяются часто. Сделать их готовыми — обернуть в оболочку, которую сможет использовать любой программист, — прекрасная идея. Вы, ребята, не знаете, какие вы счастливики, что вам не надо беспокоиться о написании, например, собственной кнопки и по пикселю ее разрисовывать, чтобы отличить нажатую от отжатой. Это не только экономит колоссальное количество дорогого времени, но и делает пользовательский интерфейс каждой программы более согласованным и, следовательно, более понятным. Помните (я хорошо помню) Windows 3.0, в которой не было стан-

Управляющие элементы служат для инкапсуляции многократно применяемой программной логики, имеющей дело с пользовательским интерфейсом.

дартного диалогового окна для открытия файла? Каждый прикладной программист должен был придумывать свое собственное. Это дорого стоило, и все прикладные реализации немного (или значительно) отличались. В Windows 3.1 диалоговое окно открытия файла стало частью ОС, что облегчило жизнь программистов и пользователей. В ASP.NET элемент управления Web — это любой алгоритм, связанный с пользовательским интерфейсом, который способен: а) показывать свою программную модель среде разработки, такой как Visual Studio.NET, и б) переводить свое представление в HTML для отображения в любом стандартном браузере (примерно так же, как элемент управления Windows переводит свое представление в последовательность вызовов Windows GDI).

Элементы управления Web богаче, многочисленнее и проще для программирования, чем стандартные элементы управления HTML.

«Но в HTML уже *есть* элементы управления, — скажете вы. — Кнопки, флажки и ссылки. Я их все время использую. Зачем мне изучать новые?» HTML поддерживает несколько элементов управления, которые являются частью языка, но здесь есть жесткие ограничения. Во-первых, они не столь многочисленны,

и возможности их не так богаты. Пока дело не пошло чуть дальше редактируемого поля и выпадающего списка. Сравните их количество с тем, что рекламируется в *Visual Basic Programmers Journal*, или их функциональность с сеткой, привязанной к данным в Visual Basic. Хотелось бы иметь гораздо больше функциональности, чем предоставлено этими несчастными элементами управления HTML. Во-вторых, для них трудно писать код. Возможность взаимодействия с ними в период выполнения ограничена. Среда Web Forms, в которой существуют элементы, имеет управляемую событиями программную модель, подобную Visual Basic. Их гораздо проще программировать, они лучше поддерживаются средами разработки и позволяют абстрагироваться от многих различий между разными браузерами. Web-элементы делают больше и программируются проще. Для меня это главное.

ASP.NET поставляется с набором основных следующих элементов (табл. 1). Я предполагаю, что после реализации программной модели сторонние поставщики разработают и представят на рын-

ке Web-элементы для всех мыслимых задач, как это произошло с элементами управления ActiveX, Вы можете написать их и сами, это не очень сложно (к сожалению, в этой редакции книги речь об этом не пойдет).

Элементы управления Web  
используются для мно-  
гих различных функций.

**Табл. 3-1.** Функции элементов управления Web.

Функция	Элемент	Описание
Отображение текста (только чтение)	Label	Отображает текст, который пользователь не может редактировать.
Редактирование текста	TextBox	Отображает текст, введенный в период разработки, который может быть отредактирован пользователем в период выполнения или изменен программно.  Примечание: хотя некоторые другие элементы тоже позволяют пользователям редактировать текст (например, DropDownList), обычно их основное назначение — не редактирование текста.
Выбор из списка	DropDownList	Позволяет пользователю выбрать значение из списка или ввести текст.
	ListBox	Отображает список вариантов выбора. Может позволять выбирать несколько значений.
Отображение графики	Image	Выводит изображение.
	AdRotator	Выводит последовательность (предопределенную или случайную) изображений.
Установка значения	CheckBox	Отображает окошко, щелкнув которое, пользователь устанавливает или сбрасывает значение.
	RadioButton	Кнопка-переключатель, может быть установлена/сброшена.
Установка даты	Calendar	Отображает календарь, позволяя пользователю выбрать дату.

(см. след. стр.)

Табл. 3-1. (продолжение)

Функция	Элемент	Описание
Команды	Button	Применяется для выполнения той или иной задачи.
	LinkButton	Работает как Button, но выглядит как гиперссылка.
	ImageButton	Работает как Button, но вместо текста содержит изображение.
Управление навигацией	HyperLink	Создает навигационную Web-ссылку.
Таблицы	Table	Создает таблицу.
	TableCell	Создает отдельную ячейку в строке таблицы.
	TableRow	Создает строку в таблице.
Группирование других элементов	CheckBoxList	Создает набор элементов CheckBox.
	Panel	Создает на форме панель без рамки, применяемую в качестве контейнера для других элементов управления.
	RadioButtonList	Создает группу кнопок-переключателей. Внутри этой группы может быть выбран только один переключатель.
Списки	Repeater	Отображает информацию из набора данных, используя набор указанных HTML-элементов и элементов управления для каждой записи из набора данных.
	DataList	Подобен Repeater, но с большими возможностями форматирования и позиционирования, включая возможность отображения информации в таблице. Позволяет также определить возможности редактирования.
	DataGrid	Отображает информацию в табличной форме. Обычно имеет привязку к данным.

Когда я расположил элементы управления на своей форме, Visual Studio.NET сгенерировала на .ASPX-странице операторы (рис. 3-9). Каждый оператор, начинающийся с `<asp:>`, — это директива синтаксическому анализатору ASP.NET создать элемент управления указанного типа, когда он будет генерировать файл класса для страницы. Например, в ответ на оператор `<asp:label>` анализатор создает в классе элемент «надпись». При выполнении страницы исполняющее ядро ASP.NET исполняет код обработчика событий, который взаимодействует с элементом (устанавливая время и цвет тек-

Исполняющее ядро создает и применяет элементы управления на сервере.

Элементы управления формируют собственный HTML для клиента.

```
<%@ Page Language="vb" AutoEventWireup="false" Codebehind=
    "WebForm1.aspx.vb" Inherits="WebApplication1.WebForm1"%>

<html>
  <body>
    <form id="WebForm1" method="post" runat="server">

<asp:Label id=Label1 runat="server" forecolor="Black">
    (время будет отображаться здесь)
</asp:Label></p>

<asp:CheckBox id=CheckBox1 runat="server"
    Text="Show Seconds ?"/>
<asp:Button id=Button1 runat="server" Text="Get Time" />

<asp:DropDownList id=DropDownList1 runat="server"
    autopostback="True">
  <asp:ListItem Value="Black" Selected="True">Black
</asp:ListItem>
  <asp:ListItem Value="Red">Red</asp:ListItem>
  <asp:ListItem Value="Green">Green</asp:ListItem>
  <asp:ListItem Value="Blue">Blue</asp:ListItem>
</asp:DropDownList></p>

    </form>
  </body>
</html>
```

**Рис. 3-9.** Фрагмент ASPX-страницы, созданной Visual Studio.NET с операторами элементов управления.

ста) (рис. 3-8). В завершение ядро указывает каждому элементу, чтобы он сформировал для себя HTML в соответствии с текущими свойствами, так же как элемент управления Windows формирует для себя вызовы GDI в соответствии со своими текущими свойствами. Затем ядро передает результирующий HTML на клиентский браузер. Ниже представлена схема процесса (рис. 3-10), а затем — фрагмент реального HTML, посылаемого клиенту (рис. 3-11).



**Рис. 3-10.** Последовательность обработки ASPX-страницы ядром.

Элементы управления для ввода могут управлять своим содержимым и состоянием между передачами.

Другая прекрасная возможность элементов управления для ввода (списков, полей редактирования, флажков, кнопок-переключателей и т. д.) — это их способность автоматически запоминать состояние, в котором они были до передачи на сервер. Microsoft называет

эту возможность *регистрацией ответных данных* (postback data). Допустим, сервер посылает браузеру страницу, содержащую сброшенный флажок. Затем человек, работающий с браузером, устанавливает этот флажок и отправляет форму серверу. Если сервер обрабатывает форму и ее же отправляет клиенту, флажок автоматически не запомнит свое состояние. Разработчику сервера придется писать код, который будет обеспечивать соответ-



ствие состояния флажка тому, что было при передаче от клиента. Элементы управления для ввода не требуют написания такого кода — они автоматически запоминают свое предыдущее состояние в пределах сеанса (подробнее об этом я расскажу ниже). На самом деле данные хранятся в выделенном поле заголовка HTTP. Это автоматическая функция, которую вы не можете отключить.

```
<span id="Label1" style="color:Black; ">
    (время отображается здесь)
</span>

<span>
<input type="checkbox" id="CheckBox1" name="CheckBox1" />
<label for="CheckBox1">Show Seconds ?</label>
</span>

<input type="submit" name="Button1" value="Get Time"
    id="Button1" />

<p>Text Color:

<select name="DropDownList1" id="DropDownList1"
    onchange="javascript:__doPostBack(' DropDownList1', ' ')">
    <option selected value="Black">Black</option>
    <option value="Red">Red</option>
    <option value="Green">Green</option>
    <option value="Blue">Blue</option>
</select>
```

Рис. 3-11. Фрагмент HTML, сгенерированного элементом управления.

Отображающие элементы управления Web, такие как надписи, списки и сетки данных поддерживают собственную версию фиксации своих свойств — *состояние отображения* (view state). Хотя пользователи и не могут установить их параметры, они все равно запоминают состояние, в котором их оставила программа при предыдущей передаче. Среда ASPX-страниц автоматически помещает скрытый элемент для ввода `__VIEWSTATE` на каждую страницу. Элементы Web автоматически сохраняют свое состояние в этом

Отображающие  
элементы управления  
также поддерживают  
свое состояние от  
, передачу к передаче.

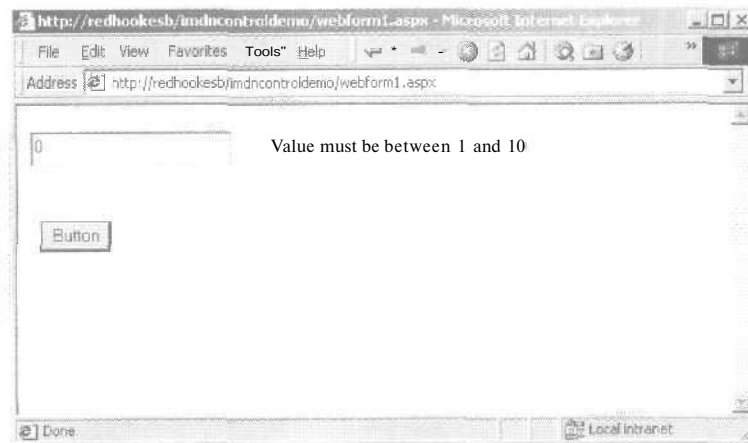
скрытом элементе при уничтожении страницы и восстанавливают его при последующем создании страницы, а это то, что вам чаще всего и нужно. Можно отключить эту возможность, установив свойство *MaintainState* элемента управления в *False*.

Зачем Microsoft использует два разных механизма управления состоянием элементов для ввода и отображающих элементов? Наверное, чтобы облегчить программистам возможность реализации Принципа Минимизации Шока, который гласит, что шокировать пользователя — нехорошо, так что делать это нужно как можно реже. Вводя что-то в поле, пользователь рассчитывает, что это останется у него перед глазами, пока он сам что-то не изменит. Если элемент управления для ввода не будет автоматически поддерживать свое состояние, соответствующий алгоритм придется реализовывать разработчику, иначе пользователь будет в шоке. Поскольку делать это нужно всегда, Microsoft встроила эту функциональность в элемент управления для ввода и не предоставила возможности ее отключения. С другой стороны, отображающие элементы показывают не то, что ввел пользователь. Обычно они отображают результаты вычислительных операций. Иногда вы захотите, чтобы отображающий элемент запоминал свое состояние между операциями, а иногда — нет. Отсюда дополнительный механизм — с возможностью отключения.

Элементы Web могут также определять конкретный браузер, на котором пользователь будет смотреть страницу, что позволяет им формировать HTML, использующий преимущества различных браузеров. Хороший пример — проверка попадания в диапазон (рис. 3-12). Проверять, находится ли введенное пользователем значение между некоторыми минимальным и максимальным значениями приходится так часто, что Microsoft создала для этих целей элемент управления Web. Вы помещаете его на форму и устанавливаете свойства, указывающие, откуда брать проверяемое, а также минимальное и максимальное значения. Формируя свой HTML, элемент управления проверяет версию браузера, на который будет передаваться страница. Если это новый браузер, поддерживающий DHTML (если точнее — MSDOM 4.0 или более позднюю версию и EcmaScript версии 1.2), элемент формирует HTML, включающий клиентский сценарий, который будет прове-

рять, что пользователь вводит только цифры, прежде чем форма будет отослана серверу. Это предотвращает лишние передачи по сети, в случае ввода пользователем неверных данных. Если данные проходят проверку на клиенте, форма передается на сервер. Затем та же операция проверки повторяется на сервере, хотя она уже прошла на клиенте. Таким образом гарантируется, что эта проверка всегда будет осуществлена перед запуском серверного кода, даже если клиентский сценарий искорежится на каком-нибудь чокнутом браузере. Если же элемент обнаруживает старый браузер, не поддерживающий DHTML, он автоматически генерирует HTML-код без клиентского сценария.

Элементы управления Web могут автоматически определять целевой браузер и формировать для него оптимальный код.



**Рис.3-12.** Элемент управления АЛЯ проверки вхождения в диапазон.

Написать свои элементы управления Web несложно, поскольку .NET Framework содержит готовые базовые классы, от которых вы можете унаследовать нужную элементу инфраструктуру (типа MFC на анаболиках). Пример, к сожалению, в этой книге не рассматривается.

Вы без особых хлопот можете написать собственные элементы управления Web.

## Управление и настройка проектов Web-приложений: файл Web.config

Мощная исполняющая среда подобная ASP.NET требует превосходных средств настройки.

Мощная исполняющая среда, подобная ASP.NET, содержит массу готовых служб. Как вы догадываетесь, любое отдельное приложение имеет уйму параметров. Механизмы настройки среды важны не меньше самих алгоритмов, лежащих в ее основе. Пользы от крутых функций, которые вы не можете настроить или не понимаете как настроить, столько же, сколько от венчальных клятв Билла Клинтона,

**ПРЕДУПРЕЖДЕНИЕ:** подробности, описываемые в данном разделе, подвержены влиянию изменений при последующих выпусках продукта.

ASP.NET держит конфигурационную информацию в отдельных файлах: с именами web.config.

Существовавшие до сих пор исполняющие среды использовали для хранения конфигурационной информации централизованные хранилища. Так, классическая COM использовала системный реестр, а COM+ — соответствующий каталог. В ASP.NET применяется децентрализованный подход. Каждое приложение хранит управляющую конфигурационную информацию ASP.NET в файле web.config в собственном каталоге приложения. Каждый файл содержит конфигурационную информацию в виде XML. Ниже вы видите фрагмент файла web.config для нашего примера (рис. 3-13), который показывает параметры компиляции и обработки пользовательских ошибок. Другие параметры мы обсудим далее в этой главе.

Файлы web.config могут также располагаться в разных подкаталогах приложения. Каждый файл web.config управляет операциями со страницами в данном каталоге и нижних подкаталогах, если там он не переопределяется. Записи, сделанные в web.config нижнего уровня переопределяют записи на предыдущем уровне. Кому-то это кажется противоестественным, кому-то — нет. Главный файл, определяющий умолчания для всей системы ASP.NET называется machine.config и лежит в каталоге [система, например, WINNT]\Microsoft.NET\Framework\[версия]. Записи, сделанные в файлах web.config в корневых каталогах отдельных Web-прило-

жений *переопределяют* записи, сделанные в главном файле. Записи файлов `web.config` в подкаталогах приложения переопределяют записи в корневом каталоге (рис. 3-14). При желании администратор может пометить записи как непереопределяемые.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <system.web>

    <!-- DYNAMIC DEBUG COMPILATION
      Set compilation debug="true" to enable ASPX debugging.
      Otherwise, setting this value to false will improve
      runtime performance of this application.
    -->
    <compilation defaultLanguage="vb" debug="true" />

    <!-- CUSTOM ERROR MESSAGES
      Set customErrors mode="On" or "RemoteOnly" to enable
      custom error messages, "Off" to disable. Add <error>
      tags for each of the errors you want to handle.
    -->
    <customErrors mode="RemoteOnly" />

  </system.web>

</configuration>
```

**Рис. 3-13.** Фрагмент файла `web.config` для программы-примера.

Все разделы файла `web.config` настраиваемы на всех уровнях подкаталогов. Так, раздел `<sessionstate>` может отсутствовать везде, кроме корневого каталога. Вам нужно будет проверить все отдельные разделы, чтобы определить степень детализации каждого.

На момент написания книги эту схему конфигурирования использовать трудно. Простой XML-файл можно редактировать любым редактором или с помощью Visual Studio.NET, где есть утилита чуть получше (рис. 3-15). Но

ей далеко до других административных утилит Windows 2000, таких как COM+ Explorer. Команда разработчиков ASP.NET подтверждает необходимость разработки лучшей утилиты, но они считают, что должны сначала закончить функциональную часть

Все административные утилиты до сих пор в процессе разработки.

продукта, а потом писать административные утилиты, даже если их придется поставлять с задержкой после выхода самого продукта. Хотя я и возражаю против этого (вежливо, но твердо), могу вас заверить, что проблема будет решена. Я видел некоторые прототипы, над которыми работают в Microsoft, и они очень даже ничего. Хотя основные решения еще обсуждаются и график выпуска не представлен, я убежден, что команда ASP.NET осознала, что хорошие средства администрирования критически важны для успеха продукта на рынке. Надеюсь, их выпуск не задержится после выхода .NET Framework. Я гарантирую, что к тому времени, когда начнутся поставки административных утилит, сама платформа еще не исчезнет.



Рис.3-14. Иерархическая природа файлов web.config.

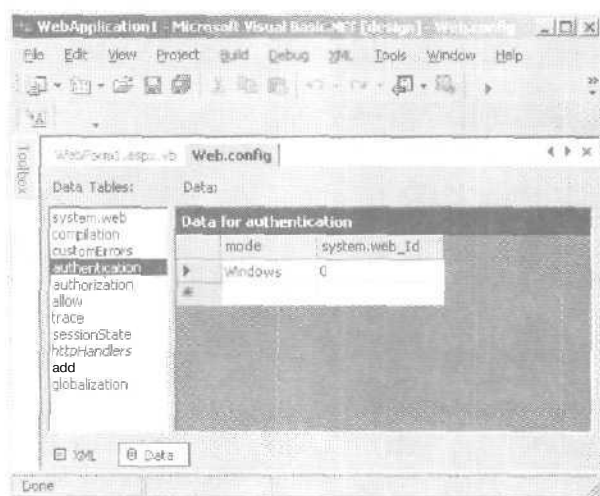


Рис.3-15. Утилита редактирования конфигурационных файлов в Visual Studio.NET еще сырая.

## Управление состоянием в ASP.NET

По умолчанию запросы Web-страниц не зависят друг от друга. Если я запрашиваю с сервера страницу Б, содержимому этой страницы неизвестно, просматривал ли я перед этим страницу А. Такой механизм хорошо работает для простых запросов только на чтение, как в примере с кинотеатром, который я приводил в начале главы. Сервер, реализующий такой механизм, легко написать, так как не нужно хранить данные от запроса к запросу.

Запросы, выдаваемые одним пользователем, могут быть не связаны друг с другом. Иногда это хорошо.

Однако по мере усложнения взаимодействия в Web такая конструкция перестает удовлетворять потребности пользователей. Мои действия на странице А должны оказывать воздействие на содержимое страницы Б. Скажем, я хочу заказать билеты на самолет прямо на Web-сайте авиакомпании, так как они предоставляют скидки постоянным клиентам. Для меня неприятно искать прямой рейс, выписывать его на бумажке, затем искать обратный, выписывать его тоже, а затем вручную вводить

Но при более развитом взаимодействии это может быть плохо, ,

номера обоих рейсов на совсем *другой* странице, чтобы купить билет. Я хочу, чтобы сайт авиакомпании автоматически запомнил, какой *прямой* рейс я выбрал, когда я выбираю обратный и *помнил* оба, когда я покупаю билет.

Web-программисты зачастую должны управлять отдельными данными для многих пользователей одновременно.

Запоминание данных при переходе от одной формы к *другой* никогда не было проблемой для настольных приложений, рассчитанных на одного пользователя. Программисту нужно просто хранить входные данные во внутренней памяти программы, не заботясь о том, какому *пользователю* они принадлежат, так как в каждый отдельный момент программу использует один человек. Но отслеживать пользовательские действия на многих страницах Web-приложения, к которому одновременно обращаются (как вам бы хотелось) многие, гораздо сложнее. Web-программист должен отделять мои данные от данных других *пользователей*, чтобы не перепутать наши рейсы (правда, если я неожиданно улечу на Гавайи в феврале, это будет здорово!). Мы называем это *управлением сеансом*, и Web-программисту нужны для этого эффективные и простые средства.

ASP.NET позволяет хранить данные, привязанные к конкретному пользователю в объекте *Session*.

Старая ASP обеспечивает простой механизм управления сеансами, который поддержан и расширен в ASP.NET. При каждом обращении к *.ASPX-странице* ASP.NET создает внутренний объект *Session* (сеанс). Это набор данных, который живет на сервере и привязан к конкретному активному пользователю. Web-программист может хранить в объекте *Session* любые интересующие его типы данных. Этот объект автоматически запоминает (помещая уникальный идентификатор в cookie-файл браузера или добавляя его в URL), к какому пользователю относятся данные, так что программисту не нужно писать для этого код. Скажем, когда я передаю форму с выбранным рейсом, программист страницы *.ASPX* может сохранить сведения о нем в объекте *Session*. Исполняющая среда *.ASPX* знает, кто является пользователем (я), так что программа автоматически выдаст мой *рейс*, а не чей-то еще (рис. 3-16). Объект *Session* может хранить произвольное число строк для каждого пользователя, но поскольку все они используют серверную па-



мать, советуя ограничить объем хранимых данных, связанных с сеансом минимально необходимым.



Рис. 3-16. Управление сеансом.

Это легко реализуемые и потому популярные функции. Каждая ASPX-страница имеет свойство с именем *Session*, которое дает доступ к объекту *Session* текущего пользователя. Вы храните данные в этом объекте в виде строк и соответствующим образом их выбираете. Взгляните на пример кода, который это делает (рис. 3-17). Этот пример вы также найдете на Web-сайте книги. Эти страницы также показаны ниже (рис. 3-18).

Естественно, если сервер поддерживал бы непрерывный сеанс для каждого пользователя, даже если бы тот смотрел всего одну страницу, ресурсы памяти быстро бы истощились. Механизм сеансов разработан для управления состоянием только активных пользователей, т. е. тех,

ASP.NET предоставляет простой API для установки и получения сеансовых переменных.

ASP.NET автоматически удаляет сеансы после заданного интервала времени.

кому требуется, чтобы сервер управлял их состоянием. ASP.NET автоматически удаляет объект *Session* пользователя, сбрасывая его содержимое, если пользователь не *проявляет* активности в течение некоторого времени. Этот интервал устанавливается в разделе `<sessionstate>` файла `web.config` (рис. 3-19). Вы можете также сами сбросить сеанс, вызвав метод `Session.Abandon`.

\* Код для *сохранения* данных сеанса.

```
Protected Sub Button1_Click(ByVal sender As System.Object, _
                             ByVal e As System.EventArgs) _
    Handles Button1.Click

    ' Сохраняем текущую текстовую строку.
    Session("DemoString") = TextBox1( ).Text

    ' Перенаправляем пользователя на другую страницу.
    ResponseC).Redirect("WebForm2.aspx")

End Sub
```

\* Код, получающий данные о сеансе.

```
Protected Sub Page_Init(ByVal Sender As System.Object, _
                         ByVal e As System.EventArgs) _
    Handles MyBase.Init
'CODEGEN: This method call is required by the Web Form Designer.
'Do not modify it using the code editor.
    InitializeComponent( )

    * Выбираем строку для сеанса и отображаем как надпись.

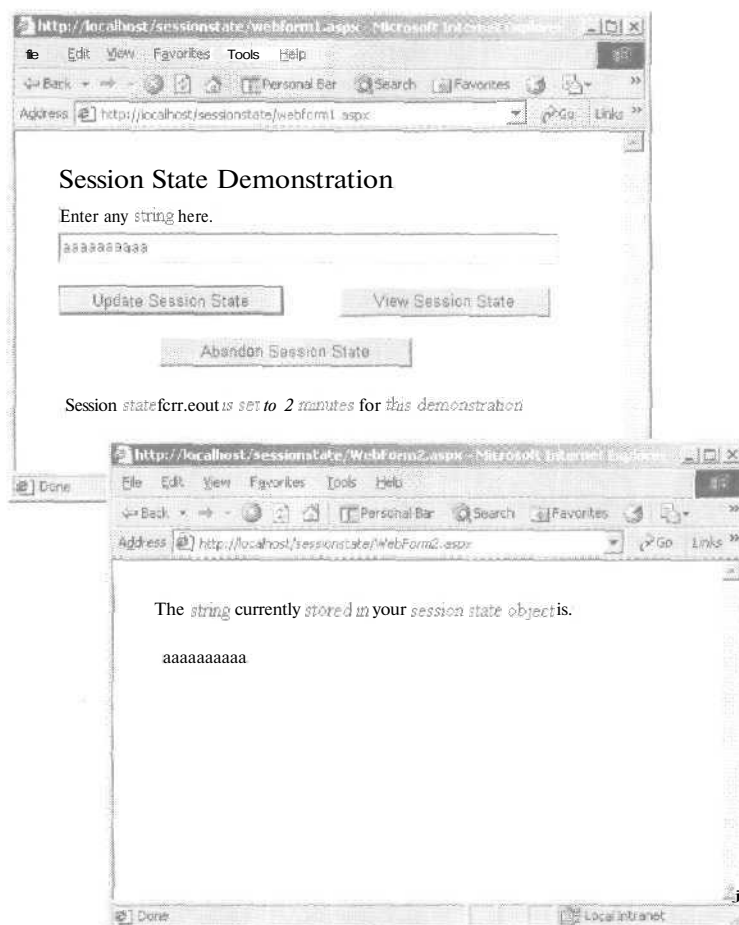
    Try
        Label2( ).Text = Session("DemoString").ToString

        ' Ссылка на несуществующую строку приведет к исключению.
        ' Корректно его обрабатываем.

    Catch ex As Exception
        Label2( ).Text = "(session string is empty)"
    End Try

End Sub
```

**Рис. 3-17.** Код АЛЯ управления сеансом.



**Рис. 3-18.** Пример приложения АЛЯ управления сеансом.

```
<sessionState
  mode="inproc"
  stateConnectionString="tcpip=127.0.0.1:42424"
  sqlConnectionString="data source=127.0.0.1;
    user id=sa; password="
  cookieless="false" timeout="2"
/>
```

**Рис. 3-19.** Записи ЛАЯ управления сеансом в файле web.config.



батывающая последующий запрос может иметь доступ к сеансовым значениям, сохраненным при предыдущем запросе, выполнявшемся другой машиной. Очевидно, что при этом растет нагрузка на сеть, так что может быть лучше направлять последовательные запросы одной и той же машине.

Для лучшего управления большими наборами данных вы можете хранить сеанс на SQL-сервере: установите атрибут *mode* в *sqlserver* и укажите строку для соединения с SQL-сервером в атрибуте *SqlConnectionstring*. Используемый по умолчанию сценарий создает временную БД для хранения сеанса. Это обеспечивает более быстрый доступ, так как данные не записываются на медленный железный диск, но это также означает, что сеанс не выживет при сбое. Если вам нужна надежность и вас не волнует, что за нее вы заплатите производительностью, можете сами изменить атрибуты БД, чтобы использовалась постоянная таблица.

Сеансовые значения можно также хранить в БД SQL-сервера.

ASP.NET помогает также управлять состоянием приложения, представляющего собой виртуальный каталог верхнего уровня IIS и все его подкаталоги. Состояние уровня приложения время от времени меняется, так что жестко указывать какие-то его значения в своих программах не стоит, но эти значения относятся ко всем пользователям, а не к какому-то отдельному. Примером может служить рекламное объявление, которое должно отображаться на каждой странице. Каждое приложение имеет один объект *Application*, который применяется подобно *Session* за исключением того, что он содержит данные для всех пользователей.

ASP.NET также обеспечивает управление данными на уровне приложения.

## Безопасность в ASP

Безопасность жизненно важна при программировании распределенной системы любого вида. Я попытаюсь обрисовать возникающие в этой области проблемы и рассказать, как ASP.NET предоставляет вашим Web-приложениям готовую функциональность, обеспечивающую нужный уровень безопасности без высоких затрат на программирование.

Безопасность жизненно важна для любой распределенной системы.

Различным уровням приложения требуется различный уровень безопасности.

Требования к безопасности Web-приложений чем-то напоминают таковые для мэрии. Много неизвестных людей посещают общедоступные места, такие как туристический офис, в котором раздают карты. Поскольку такие места не являются критическими, не хочется терять время и раздражать посетителей досмотром. Но в другие места той же мэрии, такие как этаж с кабинетом мэра, нельзя пускать тех, кто не может подтвердить свою личность (пройти аутентификацию) и кому здесь делать нечего (не прошедших авторизацию). Может потребоваться пропустить их через металлоискатель, чтобы убедиться, что они никого не покалечат.

### Аутентификация

Точная идентификация пользователя в целях безопасности называется аутентификацией.

Первая проблема безопасности — аутентификация. Кто ты такой и откуда я знаю, что ты действительно тот, за кого себя выдаешь? Аутентификация пользователя обычно представляет собой проверку удостоверения личности, предъявляемого пользователем, иногда согласованного между двумя сторонами (PIN-код или пароль), а иногда выданного третьей стороной, вызывающей доверие (паспорт или водительское удостоверение). Если удостоверение удовлетворяет сервер, он знает, кем является пользователь и может определить, какие действия может выполнять этот пользователь. Неаутентифицированный пользователь называется анонимным. Это не значит, что у него вообще не будет доступа к Web-сайту. Это лишь означает, что у него будет доступ только к тем функциям, к которым разработчик предоставил доступ анонимным пользователям, например, он сможет посмотреть расписание рейсов, а заказать билет со скидкой — нет.

Системы аутентификации сложные и дорогие.

Исторически аутентификация была наиболее сложной проблемой при проектировании системы безопасности. Большинство прикладных разработчиков не хотят иметь с ней дело, потому что при ее исключительной важности ее трудно правильно реализовать. Вам нужна постоянная команда очень умных программистов, которых не волнует ничего, кроме безопасности,

потому что есть *такие* же фанатичные плохие ребята, которых интересует только *взлом* вашего *сайта*, с которого они хотят бесплатно взять вашу первоклассную программу (или сделать что-нибудь похуже). Например, нельзя просто послать пароль по сети, даже если он зашифрован. Если сетевой пакет содержит даже *зашифрованный пароль*, не изменяемый при каждой *передаче*, взломщики могут его записать и потом им воспользоваться. Именно так я вошел в систему обмена валюты некоторого банка (название вы можете узнать, но только с их разрешения), *когда* выполнял контракт по проверке их системы безопасности. Они очень гордились своим алгоритмом шифрования паролей, который я даже не пытался взломать. Мне понадобилось всего 20 минут работы с анализатором пакетов, чтобы записать *идентификатор* и пароль пользователя и послать их, чтобы зайти на их *сервер*. Жаль, что у меня была почасовая оплата. В следующий раз я возьму недельку оплачиваемого отпуска, прежде чем объявить об успешных результатах.

Сложность и важность аутентификации заставили встроить эту *функцию* в безопасную (будем *надеяться*) ОС в первую очередь. ASP.NET поддерживает три механизма аутентификации (или четыре, если считать «попе») (табл. 3-2).

К счастью, в A S .NET встроено несколько различных готовых аутентификационных механизмов.

**Табл. 3-2.** Режимы аутентификации ASP.NET.

Название	Описание
None	Аутентификация в ASP.NET не используется.
Windows	Стандартная аутентификация Windows через IIS.
Forms	ASP.NET требует, чтобы заголовки запросов всех страниц содержали cookie-файлы, выданные сервером. Пользователи, пытающиеся получить доступ к защищенным страницам без этих файлов, переадресуются к странице регистрации, которая проверяет удостоверяющие данные и выдает cookie-файл.
Passport	Идея та же, что и в случае Forms, за исключением того, что идентификационные данные пользователя сохраняются и cookie-файлы выдаются внешней аутентификационной службой Microsoft Passport.

Нужно хорошо подумать, какие из ресурсов вашего сайта нуждаются в аутентификации.

Нужно хорошо подумать, где именно должна иметь место аутентификация. Как и в случае мэрии, нужно соблюсти баланс между защитой и доступностью. Так, финансовый Web-сайт будет требовать аутентификации

при просмотре счетов или переводе денег, но, видимо, должен предоставлять маркетинговые материалы и проспекты о своих фондах анонимной публике. Важно так построить сайт, чтобы защищенные функции оставались защищенными, но эта защита не мешала незащищенным операциям. Трудно представить что-нибудь более губительное для бизнеса, чем заставлять пользователя устанавливать и применять аутентифицированную учетную запись, прежде чем позволить ему посмотреть информацию о товарах, хотя некоторые сайты именно так и устроены.

Аутентификация в ASP.NET производится без шифрования. Об этом вам нужно позаботиться самим.

**ПРЕДУПРЕЖДЕНИЕ:** в отличие от DCOM, которая имеет собственный жесткий формат для поддержки конфиденциальности пакетов, ни одна из аутентификационных схем ASP.NET не шифрует передаваемые между клиентом и сервером данные. Эта проблема

связана не с ASP.NET как таковой, а с общим транспортным протоколом Web — HTTP. Если на вашем сайте есть данные, которые вы не хотите увидеть на первой странице газеты *USA Today*, вам нужно использовать протокол Secure Socket Layer (SSL) для шифрования. Обычно это делается только для наиболее чувствительных операций, поскольку шифрование замедляет транспортировку. Например, Web-сайт авиакомпании будет, вероятно, шифровать страницу покупки, где указывается номер кредитной карты, но не страницу поиска рейсов.

### Windows-аутентификация

Windows-аутентификация хорошо работает только в интрасетях на базе Windows.

ASP.NET поддерживает аутентификацию на базе Windows, что по сути означает делегирование процесса аутентификации IIS — базовой Web-серверной архитектуре, над которой надстроена ASP.NET. IIS можно настроить

так, чтобы он выводил диалоговое окно в пользовательском браузере и принимал идентификатор и пароль пользователя. Эти



идентификационные данные должны соответствовать пользовательской учетной записи Windows в домене, к которому принадлежит IIS. Другая возможность предполагает наличие Microsoft Internet Explorer 4 или более поздней версии, Windows-системы и отсутствие прокси. В этом случае IIS можно настроить на применение встроенных в Windows аутентификационных протоколов NTLM или Kerberos для автоматического согласования пользовательского идентификатора и пароля на основании текущего сеанса регистрации пользователя.

Windows-аутентификация неплохо работает в интрасетях, где используется только Windows, поскольку здесь у вас полный административный контроль. Для некоторых конфигураций, скажем, для больших корпораций, — это просто фантастика! Просто включай и работай! Но это совсем не так полезно в открытом Интернете, где серверу нужно общаться с системами любых типов (скажем, блокнотными компьютерами) с применением доступа любого типа (скажем, не IE) и где вы не хотите настраивать учетные записи Windows для всех пользователей.

### Cookie-аутентификация на основе форм

Большинство проектировщиков Web-приложения выберут аутентификацию на основе форм, или cookie-аутентификацию. Пользователь сначала устанавливает учетную запись с идентификатором и паролем. Некоторые Web-сайты, например, сайты авиакомпаний, позволяют сделать вам это через Web на странице, открытой для анонимного доступа. Другие, такие как сайты большинства компаний, оказывающих финансовые услуги, требуют подписанный документ, доставленный обычной почтой.

Первый раз обращаясь к странице защищенного Web-сайта, пользователь получает форму, в которой запрашивается его идентификатор и пароль. Web-сервер сравнивает их со значениями, хранящимися в его досье, и, если они совпадают, разрешает доступ. Затем сервер предоставляет браузеру cookie-файл, подтверждающий успешную регистрацию. Рассматривайте cookie-файл как метку, которую вам ставят в баре, после того как проверяют ваши документы и убеждаются, что вам уже стукнуло 21. Он со-

Аутентификация на основе форм начинается с идентификатора и пароля пользователя.

Сервер предоставляет браузеру cookie-файл — входной билет, идентифицирующий аутентифцированного пользователя.

держит идентификационные данные пользователя в зашифрованном виде. Браузер будет автоматически посылать этот cookie-файл в разделе заголовка всех запрашиваемых после этого страниц, чтобы сервер знал запрашивающего пользователя. Это избавляет

вас от необходимости вводить свой идентификатор и пароль для каждой посылаемой формы или запрашиваемой страницы (рис. 3-21).



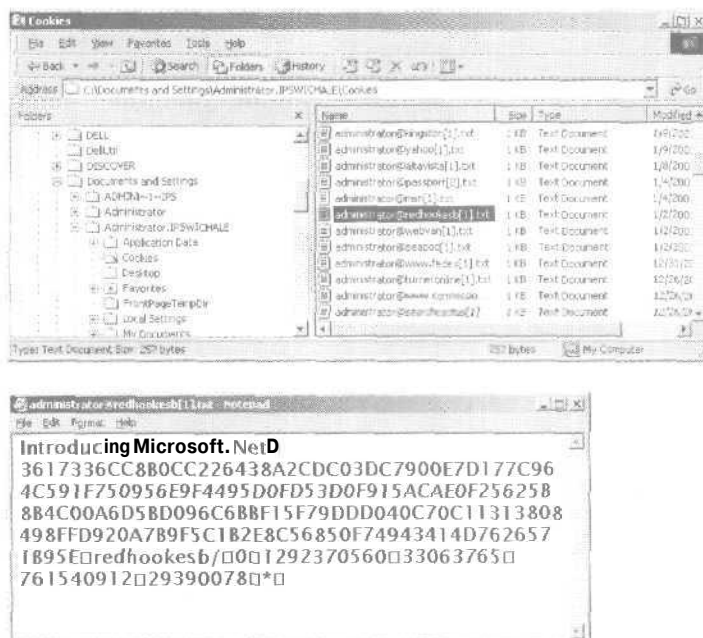
**Рис. 3-21.** Аутентификация на основе форм.

Cookie-файлы могут быть временным или постоянными.

Предоставляя cookie-файл, сервер указывает, выбросить его после завершения сеанса браузера или сохранить на жестком диске пользователя, чтобы пользователю не нужно

было проходить процедуру регистрации в следующий раз. Финансовые Web-сайты часто используют первый подход, чтобы быть сверхзащищенными от неавторизованного доступа, который может обойтись в тысячи долларов. Однако второй подход, очевидно, удобней пользователям. Большинство Web-сайтов, чьи данные не очень критичны, например, онлайн-е периодичес-

кие издания типа *Wall Street Journal*, часто позволяют пользователю выбрать вторую возможность (рис. 3-22).



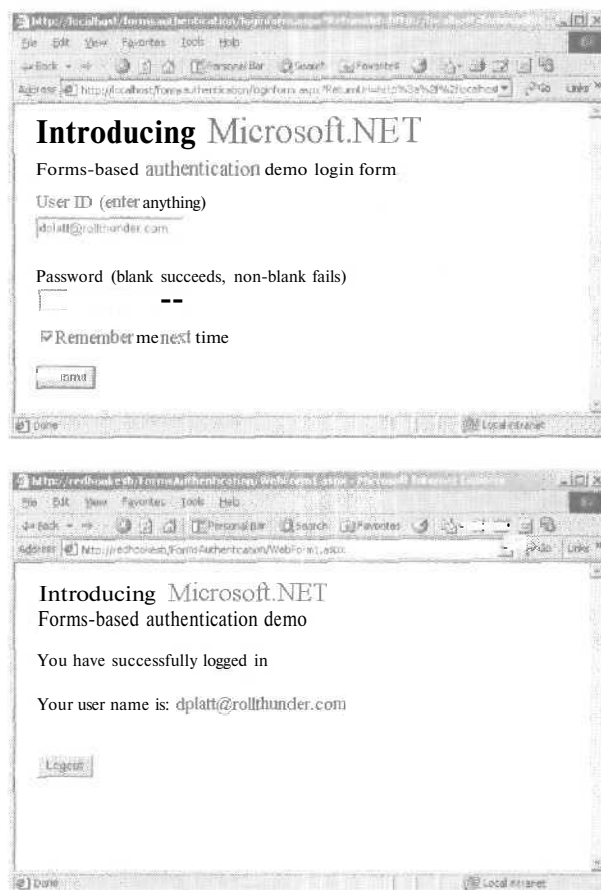
**Рис. 3-22.** Постоянный cookie-файл, созданный демонстрационным приложением.

ASP.NET имеет хорошую поддержку аутентификации на основе форм. Демонстрационное приложение (рис. 3-23) содержится на сайте книги. Чтобы ASP.NET использовала аутентификацию на основе форм, нужно сделать в файле `web.config` соответствующие записи (рис. 3-24). Элемент `authorization` указывает ASP.NET запретить доступ всем неаутентифицированным пользователям.

Это означает, что ASP.NET требует, чтобы любой запрос страницы из каталога или его подкаталогов должен содержать cookie-файл от браузера, говорящий, что пользователь аутентифицирован. При отсутствии cookie-файла, как это имеет место до первой регистрации, ASP.NET на-правляет пользователя на страницу, указанную в атрибуте `loginUrl`.

ASP.NET содержит раз-  
витую готовую под-  
держку аутентификации  
на основе форм. „

Эта страница служит для ввода пользователем идентификатора и пароля. Обработчик этой страницы содержит алгоритмы, необходимые, чтобы проверить, что пользователь является тем, за кого себя выдает. В примере принимается пустой пароль и отвергаются все непустые. Если сервер принимает входящего пользователя, функция *System.Web.Security.FormsAuthentication.RedirectFromLoginPage* отсылает его к странице, которую он пытается получить, в противном случае он переходит к экрану регистрации (рис. 3-25).



**Рис. 3-23.** Демонстрационное приложение аутентификации на основе форм.

```
<authentication mode = "Forms" >
    <forms name="IntroducingMicrosft.NET"
        loginUrl="loginform.aspx"/>
</authentication>

<authorization>
    <deny users="?" />
</authorization>
```

**Рис. 3-24.** Файл *web.config* для аутентификации на основе форм.

```
Public Sub Button1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs)

    * Принимаем пустой пароль. Вызываем функцию, посылающую
    * cookie клиентскому браузеру, и перенаправляем клиента
    * на начальную запрашиваемую им страницу,

    If txt_Password.Text = "" Then
        System.Web.Security.FormsAuthentication.RedirectFromLoginPage( _
            txt_UserID.Text, cb_RememberMe.Checked)

    * Отвергаем непустой пароль. Не обращаемся ни к каким
    * другим функциям.

    Else
        Label3.Text = "Invalid Credentials: Please try again"
    End If

End Sub
```

**Рис. 3-25.** Код формы АЛЯ аутентификации на основе форм.

ASP.NET можно также настроить на выполнение автоматической аутентификации, основанной на формах с применением пользовательского идентификатора и пароля, хранящегося в конфигурационном XML-файле. Видимо, этот подход хорош в небольших конфигурациях. Демонстрация такого подхода выходит за рамки этой книги, во всяком случае этой редакции.

### Passport-аутентификация

Третий вариант аутентификации — применение службы Passport. Схема, основанная на формах, прекрасно смотрится, и применять

Число Web-сайтов, для которых пользователь должен помнить идентификатор и пароль, стремительно растет. При этом возрастает угроза безопасности всех данных.

ее легко благодаря готовой поддержке в ASP.NET. Но фатальной проблемой для нее является неконтролируемый рост числа сайтов. Любой сайт, содержащий хотя бы бит критической информации требует реализации или иного механизма регистрации. Я, например, использую пять сайтов различных авиалиний для заказа билетов, и, поскольку все они принимают кредитные карты, везде требуется аутентификация. Это жуткая головная боль — отслеживать все идентификаторы и пароли, которые я ввожу на разных сайтах. На одном мое имя «dplatt»; когда я подписывался на другой, этот идентификатор уже был задействован, так что там я — «daveplatt». Я пробовал применить свой адрес электронной почты, который никем другим использоваться не должен, и иногда это работало, но некоторые сайты не позволяют вводить знак @, а другие не воспринимают такие длинные имена. В качестве пароля некоторые используют 4-символьные PIN-коды, другие требуют пароль длиной не менее 6 (а иногда 8) символов, включающий как цифры, так и буквы. Некоторые считают, что их данные настолько секретны (сведения о моих полетах? Бред!), что не позволяют хранить идентификационные данные в постоянных cookie-файлах. Единственная возможность отслеживать все свои пользовательские имена и пароли — выписать их на бумажку и держать поближе к компьютеру, где их и увидит любой злоумышленник. Это один из способов, которым физик Ричард Фейнман, Нобелевский лауреат, вскрыл секретный сейф и шулки ради оставил в нем бессмысленные заметки. Это происходило во время работы над проектом атомной бомбы в Лос Аламосе во время второй мировой войны. (О двух других способах вы можете узнать из его мемуаров «*Surely You're joking, Mr. Feynman*». Надеюсь, в Лос Аламосе эту проблему уже решили.) Как говорил мне один клиент еще десять лет назад, главная проблема безопасности — не анализаторы сетевых пакетов, а самоклеющиеся бумажки для заметок.

Microsoft Passport ([www.passport.com](http://www.passport.com)) — попытка обеспечить универсальную безопасную одношаговую процедуру регистрации. При этом применяется механизм, аналогичный аутентификации

на основе форм, но хранением и проверкой идентификаторов и паролей занимаются Web-серверы Microsoft Passport. Пользователь должен зайти на сайт службы Passport и создать паспорт — по сути идентификатор и пароль, хранящиеся на серверах Microsoft. Когда пользователь запрашивает страницу с сайта, применяющего Passport-аутентификацию, этот сайт ищет в запросе cookie-файл службы Passport. Если он его не находит, запрос перенаправляется на собственный сайт Passport, где пользователь вводит свой идентификатор и пароль и получает cookie-файл. Последующие запросы браузера к любым сайтам, применяющим Passport, содержат этот cookie-файл, пока пользователь не выйдет из системы (рис. 3-26). Таким образом, пользователь может применять один и тот же идентификатор и пароль на всех соответствующих сайтах. Этот механизм во многом решает проблему роста числа сайтов с необходимостью аутентификации, облегчая жизнь пользователям и повышая безопасность в сети.



Рис. 3-26. Passport-аутентификация.

Microsoft Passport предлагает единый идентификатор и пароль для всех сайтов.

Паспорт также позволяет пользователю хранить персональные сведения, которые он хочет предоставлять посещаемым Web-сайтам. Например, паспорт может содержать почтовый адрес, чтобы Web-сайты могли авто-

матизировать процедуру заполнения бланков заказов, и даже номер кредитной карты для осуществления покупок. Выглядит это так, как будто фантастически успешная (и запатентованная) технология заказов одним щелчком мыши, применяемая Amazon.com, доступна любым сайтам. На некоторых сайтах (например, Crutchfield.com или RadioShack.com) вы можете увидеть кнопку для осуществления быстрой покупки с помощью Passport.

В паспорте может содержаться такая информация как адреса и номера кредитных карт.

Концепция Passport-аутентификации кажется неплохой, но насколько привлекательной она будет для рынка — это вопрос. Пока что последователей у нее немного. Кроме собственных сайтов Microsoft (Hotmail и MSN), 0

настоящий момент (январь 2001 г.) на сайте Passport перечислены только 65 сайтов, применяющих Passport-аутентификацию, но по большей части крупными их не назвать. И похоже, что на некоторых из них (например на Victoria's Secret) эта возможность еще не реализована. Отчасти дело в том, что Microsoft требует лицензионного соглашения на применение Passport и оплаты своих услуг. Хотя сейчас оплачивается только базовая аутентификация, потенциальные партнеры опасаются ввязываться в это дело, ожидая увеличения расценок в будущем. Они могут также опасаться аутсорсинга. Поскольку они не контролируют этот сервис, как они могут быть уверены, что он будет работать или будет достаточно эффективным? Кому нужно повторение кошмара 23 января 2001 г., когда какой-то болван из Microsoft неправильно настроил DNS и заблокировал доступ ко всем сайтам Microsoft, включая Passport, почти на целый день? (Я не удивлюсь, если этот «специалист» редактировал XML-файлы в Блокноте, а ведь спе-

- Будущее Passport зависит от отношения к этой службе со стороны потенциальных партнеров.

циальная утилита позволила бы избежать проблемы.) И как быть, если такое повторится? Как может какая-то Интернет-компания послать Microsoft подальше, если Microsoft хранит их регистрационные данные?



Те же проблемы существуют и для потребителей. Решат ли пользователи, что паспорт настолько удобен, что нужно его применять? Это зависит от того, сколько сайтов принимает паспорт, что в свою очередь зависит от того, сколько потребителей его имеет. Так как все 60 миллионов пользователей Hotmail автоматически получили паспорта, начинается все не с нуля. А как быть с теми, кто испытывает стойкое отвращение к Microsoft? Я не думаю, что Ларри Эллисон или Жанет Рино когда-нибудь закажут паспорта. Так ли малочисленны эти потребители, чтобы сайты их игнорировали или они должны вместе с Passport-аутентификацией применять другие механизмы регистрации?

Отношение партнеров будет определяться отношением потребителей (и так по кругу).

Мне кажется, большие проблемы возникают из-за наличия в паспорте дополнительных данных. Доверят ли пользователи номера своих кредитных карт Microsoft, только чтобы облегчить себе процесс покупки? Я — нет.

Отношение потребителей во многом определяется возможностями электронной коммерции.

Это связано не столько с недоверием к Microsoft, которой я даю номер своей кредитной карты при каждой покупке бесполезного обновления, речь идет о всех онлайн-поставщиках, которые читают мой паспорт. Откуда я знаю, не прочитали ли в Radio Shack мой паспорт, если я у них ничего не покупал? Откуда мне знать, что они не прочитали мой почтовый адрес и не начнут меня бомбардировать своей дурацкой рекламой? Их обещания этого не делать не являются гарантией невмешательства в мою личную жизнь. Даже если Microsoft действительно сделает невозможным недозволенное чтение паспорта, сколько пользователей ей поверят? Универсальная регистрация — изящное решение, но готов поспорить, что возможности электронной коммерции сведут эту идею на нет. Потратить деньги и так довольно легко; у меня нет ни потребности, ни желания упрощать этот процесс.

Паспорт — интересная идея, и рынок определит, удачна она или нет. Я, пожалуй, не выбрал бы его как единственно возможный механизм аутентификации, но всерьез рассматриваю его как один из вариантов. Все, что упрощает использование сайта в сравнении с сайтами конкурентов, нужно исполь-

ASP.NET включает готовую поддержку Passport-аутентификации.

зовать. Если вы решите применять Passport-аутентификацию, вы обнаружите, что ее поддержка в ASP.NET позволяет сделать это довольно просто. Этот режим аутентификации устанавливается в файле web.config. Вам нужно будет переписать Passport SDK, подписать лицензионное соглашение и затем сотворить несложную программу, делегирующую процесс аутентификации этой службе.

### Авторизация

Мы должны проверить, имеет ли пользователь право делать то, что пытается.

Можно указать, какие страницы и кому доступны, создав записи в файле web.config для определенных страниц и каталогов.

После завершения процесса аутентификации мы знаем пользователя или знаем, что это анонимный пользователь. Теперь мы должны решить, разрешить ли ему смотреть запрашиваемую им страницу. Это называется *авторизацией*.

ASP.NET имеет хорошую поддержку контроля доступа .ASPX-страницам. Можно ограничить доступ к страницам, создав записи в файлах web.config вашего приложения (рис 3-27).

<!-- Авторизация

Пользователю Simba и всем пользователям, имеющим роль Doctors, позволено получать страницы из этого каталога, а также всех подкаталогов, в которых собственные файлы web.config не изменяют эти параметры.

->

```
<authorization>
  <allow users="Simba" roles="Doctors" />
  <deny users="*" />
</authorization>
```

**Рис. 3-27.** Авторизационные записи в файле web.config.

Раздел `<authorization>` содержит элементы `<allow>` и `<deny>`, которые соответственно разрешают и запрещают указанным пользователям/группам получать доступ к страницам в том каталоге, к которому относится данный файл web.config. Эти элементы могут также содержать атрибут *verb* (не показан), позволяющий пользователям смотреть страницу с помощью операций *get*,

но запрещающий отсылать данные на сервер. Хотя файл `web.config` относится ко всему каталогу (и к подкаталогам, если там он не переопределен), вы можете ограничить доступ к единственному файлу элементом `<location>` (не показан).

Когда пользователь запрашивает страницу.

ASP.NET применяет указанные в файле `web.config` правила в том порядке, в каком

Правила доступа газиме-  
няются в том порядке, в  
котором они приведены.

они там приведены, пока не определяет, разрешить или запретить запрашивающему пользователю доступ к

странице. Если из данного файла это определить нельзя, ASP.NET

просматривает файл `web.config` в родительском каталоге и так до тех пор, пока не переходит к файлу `machine.config`, о котором мы

уже говорили. Этот файл по умолчанию предоставляет доступ всем запросам. Так что если вы не хотите, чтобы пользователь

видел какую-то страницу, его нужно явно указать в элементе `<deny>`. Выше ASP.NET сначала применит элемент `<allow>` и,

если пользователя зовут «Simba» или он входит в группу «Doctors», доступ будет предоставлен, и процесс проверки завершится

(рис. 3-26). Если ни одна из этих проверок не проходит, ASP.NET применяет следующее правило, которое запрещает доступ всем

(символ \*). Запрос не будет выполнен, и проверка прекратится. Таким образом, только Doctors и Simba могут смотреть страни-

цы в этом подкаталоге. Заметьте: при Windows-аутентификации в домене перед именем пользователя или роли нужно указывать имя домена, чтобы ASP.NET могла его распознать (рис. 3-28).

```
<!-- Авторизация
```

```
    Здесь делается то же, что в предыдущем примере, только
    перед именем пользователя и роли указывается имя домена
    для корректной Windows-аутентификации в домене.
```

```
-->
```

```
<authorization>
  <allow users="REDHOOKESBO\Simba"/>
  <allow roles="REDHOOKESBO\Doctors"/>
  <deny users="*" />
</authorization>
```

**Рис. 3-28.** Авторизационные записи в файле `web.config` при Windows-аутентификации в домене.

Символ «\*» позволяет установить права доступа для неаутентифицированных пользователей.

Поскольку есть вероятность, что к нашему Web-сайту будут обращаться многие анонимные пользователи — те, кто не прошел аутентификацию, — нам нужен способ указывать, что они могут делать. Вопросительный знак

(?) указывает анонимных пользователей. Он применяется так же, как любое имя или символ «\*». Пример вы видели выше: файл `web.config` для аутентификации на основе форм (рис. 3-23). В данном случае неаутентифицированным пользователям не разрешается просматривать страницы в этом каталоге, что заставляет их регистрироваться с помощью соответствующей формы.

На практике при авторизации чаще всего применяются роли, т. е. особые группы пользователей.

ASP.NET позволяет авторизовать как отдельных пользователей, так и группы. На практике авторизацию отдельных пользователей применяют редко, обычно это делается для групп. Так, медицинское приложение может позволять ассистентам прописывать тайленол, но прописать наркотические вещества имеют право только лицензированные доктора. Эти права устанавливаются для каждой группы, а отдельные пользователи добавляются или удаляются из групп.

ASP.NET автоматически определяет принадлежность к группам при применении Windows-аутентификации.

В случае Windows-аутентификации ASP.NET автоматически распознает роль, если администратор настроил стандартные группы пользователей Windows. Каждый член группы имеет соответствующую роль. Программировать вам ничего не придется, распозна-

вание происходит автоматически. Однако если аутентификация производится с помощью cookie или Passport, определение роли значительно усложняется. ASP.NET не знает, как хранятся сведения о пользователях или что определяет принадлежность к роли в вашем приложении. Следовательно, вам нужно написать свой код и подключить его к среде ASP.NET, чтобы можно было определить принадлежность пользователя к группам. Вам придется написать класс, реализующий интерфейс *IPrincipal*, содержащий метод */IsInRole*. В этот метод вы помещаете код, определяющий принадлежность указанного пользователя к группе. (Вы также можете попробовать вызвать класс *GenericPrincipal*, позволяющий

для определения роли пользователей применить простой массив строк.) Ваш класс подключается к ASP.NET методом *OnAuthenticate* (если его имя не изменится) в файле *global.asax* вашего приложения. ASP.NET вызывает этот класс каждый раз, когда проверяет принадлежность пользователя к указанной группе (рис. 3-29). Хотя реализовать все это не так сложно, как кажется, я все же предпочел бы иметь для определения ролей простой механизм по умолчанию, чтобы писать собственный код, только если не устраивает стандартный механизм. Поскольку Microsoft сделала это для пользовательских идентификаторов и паролей (см. конец раздела об аутентификации на основе форм), можно было бы легко добавить атрибут принадлежности к роли к элементу, описывающего пользователя.

Чтобы принадлежность к группе в случае Passport-аутентификации и cookie-файлов, нужно написать собственный код.

```
Imports System
Imports System.Security
Imports System.Security.Principal

Public Class MyOwnRoleCheckerPrincipal : Implements IPrincipal

    Dim MyIdentity As IIdentity

    Public Sub New(ByRef id As IIdentity)
        MyIdentity = id
    End Sub

    Public Function IsInRole(ByVal role As String) As Boolean _
        Implements System.Security.Principal.IPrincipal.IsInRole

        If ( <role checking logic goes here> ) Then
            Return True
        Else
            Return False
        End If
    End Function

    <другие методы>

End Class
```

**Рис. 3-29.** Код, проверяющий принадлежность к группам при аутентификации с помощью форм или Passport.

## Идентификационные данные

Web-сайт обычно является средним слоем 3-уровневой системы, представляя собой связующее звено с уровнем данных.

Автоматическая авторизация, обеспечиваемая ASP.NET прекрасно подходит для управления доступом к страницам вашего Web-сайта. Однако ваши Web-страницы обычно находятся в середине 3-уровневой системы, под ними расположен уровень данных (БД).

Средний уровень реализует бизнес-логику, например, определяя, является ли пользователь, прописавший морфий, лицензированным доктором. Если это подтверждается, средний уровень должен связаться с уровнем данных, чтобы уменьшить имеющееся в наличии количество морфия и выписать пациенту очередной счет. Если не подтверждается, нужно, видимо, запустить у пользователя увлекательную игру, чтобы он сидел на месте, а в это время вызывать полицию.

Важно убедиться в подлинности процесса Web-сервера.

Уровень данных почти наверняка имеет собственные механизмы защиты. Все коммерческие БД позволяют настраивать права доступа с различным уровнем детализации, указывая, какие пользователи имеют доступ к различным БД и таблицам, а какие нет. Даже файловая система NTFS позволяет администратору указать, какие пользователи и группы могут выполнять те или иные операции с конкретными файлами и каталогами. Все это напрямую связано с тем, как уровень данных определяет подлинность пользователя, выдавшего запрос на обслуживание.

Уровень данных, как правило, доверяет авторизацию среднему уровню.

Первое и, вероятно, единственное, что вы захотите применить, называется моделью доверенного пользователя. В этой модели серверный процесс имеет конкретные идентификационные данные, например Pharmacy-

App, известные как доверенный пользователь. Уровень данных настроен так, чтобы разрешить ему выполнять те действия, которые позволят выполнить необходимую работу. Например, доверенному пользователю в фармацевтическом приложении нужно позволить делать записи в таблицы препаратов и счетов пациентов, но, видимо, не в таблицы с личными сведениями о пациентах и не в платежные ведомости сотрудников. Уровень

данных доверяет серверу среднего уровня в том, что касается авторизации, например, что лицензия доктора действительна и повторную авторизацию производить не нужно. Если вы доверяете супруге свой бумажник, вас не волнует, как будут потрачены деньги. Такая взаимосвязь показана ниже (рис. 3-30). Вы указываете идентификационные данные серверного процесса в элементе `<identity>` в файле `web.config` вашего приложения (рис. 3-31).



**Рис. 3-30.** Модель авторизации с доверенным пользователем.

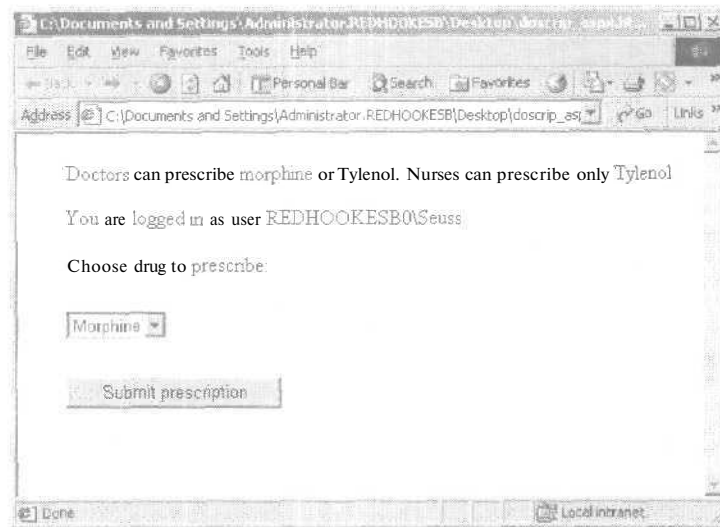
```
<identity impersonate="false"
  userName="MyDomain\MyUsername"
  password="MyPassword" />
```

**Рис. 3-31.** Файл `web.config` указывает идентификационные данные серверного процесса.

Код вашей `.ASPX`-страницы может выполнять собственную авторизацию, используя внутренний объект `User`. Метод `IsInRole` последнего позволяет определить, включен ли пользователь в административную группу. Концептуально это совпадает с применением аналогичного метода в `COM+`. Обычно этим можно ограничиться. Если вы хотите углубиться и работать с отдельными пользователями, можете задействовать свойство `Identity` объекта `User`. При этом вы получите интерфейс `Identity`, который укажет имя аутентифицированного пользователя и алгоритм, применявшийся для его аутентификации. Можете написать свой код, который будет обращаться к этим сведениям, чтобы решить, имеет ли пользова-

ASP.NET предоставляет объект, позволяющий вашему коду проверять идентификационные данные пользователя.

тель право делать то, что хочет. В рассмотренном ранее примере с Windows-аутентификацией показана такая программная авторизация. Пример и код страницы, осуществляющей авторизацию, показаны ниже (рис. 3-32 и 3-33).



**Рис. 3-32.** Пример Web-страницы с авторизацией на базе доверенного пользователя.

```
Public Sub Button1_Click(ByVal sender As Object, _
                        ByVal e As System.EventArgs)

    Session("DrugName") = DropDownList1.SelectedItem.Text
    Session("Result") = "OK"
    ' Если пользователь пытается прописать морфий и не входит
    ' в группу "Doctors", не выполнять запрос.
    If DropDownList1.SelectedItem.Text = "Morphine" Then
        If (User.IsInRole("Doctors") = False) Then
            Session("Result") = "BAD"
        End If
    End If
    Response.Redirect("ScriptResuH.aspx")
End Sub
```

**Рис. 3-33.** Код Web-страницы с авторизацией на базе доверенного пользователя.



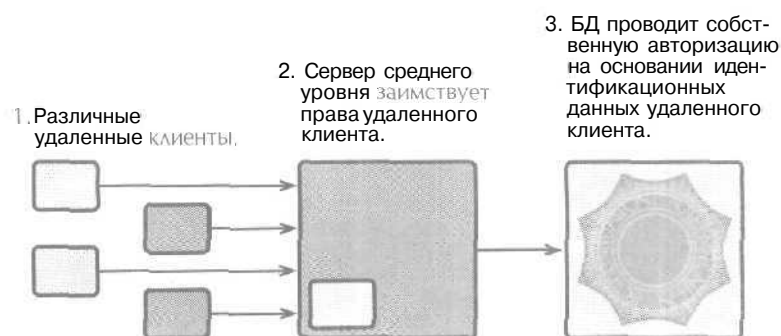
Не со всеми БД такой подход удобен. Многие базы были созданы, а их администраторы обучены до того, как 3-уровневая модель стала популярной. В особенности в старых системах уровень БД содержит проверки безопасности и ведение аудита, привязанные к конкретным клиентам.

Иногда модель с доверенным пользователем не подходит.

В этом случае, если используется Windows-аутентификация, можно вернуться к более старой модели заимствования прав и делегирования (impersonation-delegation), применявшуюся в старой ASP. В этой модели код .ASPX-страницы берет («заимствует») иденти-

В этом случае Web-сервер может заимствовать права клиента, если требуется Windows-аутентификация.

фикационные данные аутентифицированного пользователя (или применяет специальные идентификационные данные, предназначенные для анонимных пользователей). Затем код страницы пытается получить доступ к ресурсам уровня БД и сам ресурс, скажем, SQL-сервер, выполняет авторизацию, проверяет, имеет ли пользователь доступ (рис. 3-34).



**Рис. 3-34.** Модель авторизации с заимствованием прав и делегированием.

Этот внешне привлекательный подход дорого обходится в плане производительности. Он требует две аутентификации, а не одну: одну — когда ASP.NET аутентифицирует пользователя, прежде чем заимствовать его права, и вторую — когда БД аутентифицирует заимствованные

Модель заимствования прав и делегирования снижает производительность.

сервером идентификационные данные. Больше того, время теряется, даже когда пользователю отказывается в доступе. Приходится проходить все три уровня, а не два и делать два сетевых запроса, а не один. Лучше, если вы вспомнили, что забыли бумажник, когда пришли в магазин, а не после того, как взяли, что нужно, и отстояли очередь в кассу. Еще лучше, если вы вспомните об этом, садясь в машину.

& случае Windows-аутентификации можно программно передавать права клиента Web-серверу.

Web-сервер заимствует права клиента одним из двух способов. Старая ASP автоматически заимствует права клиента каждый раз, когда делается запрос. ASP.NET по умолчанию этого не делает, но можно разрешить эту

функцию, установив атрибут *impersonation* в True в файле *web.config* (рис. 3-35). Другой способ — вызов функции *System.Security.Principal.WindowsIdentity.GetCurrent().Impersonate*.

```
<identity>
  <impersonation enable="true" />
</identity>
```

Рис. 3-35. Параметры *web.config* для автоматического заимствования прав.

Web-страница, демонстрирующая автоматическое заимствование прав, выглядит так (рис. 3-36):

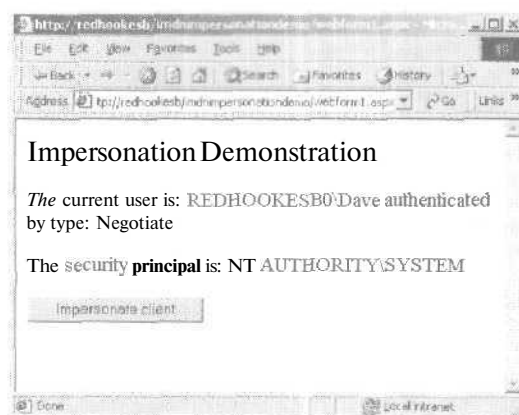


Рис. 3-36. Пример Web-приложения, демонстрирующего заимствование прав.

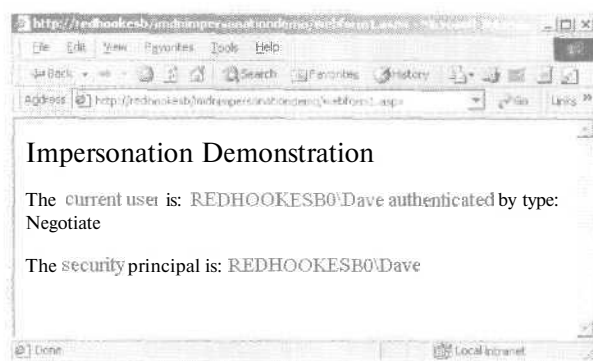


Рис. 3-36. (продолжение)

## Управление процессом

Одно из главных свойств, которое хочется видеть у Web-сервера — это его способность работать 24 часа в сутки, 7 дней в неделю без нашего вмешательства. С проблемой реализации сервиса такого уровня никогда не сталкивались разработчики настольных приложений. Так, утечки памяти в Пасьянсе, который каждый раз запускается на 5 минут (ладно, на 2 часа), вероятно, никогда не займут столько памяти, чтобы что-то испортить. Но если я оставляю Пасьянс работать целый год, все время пересдавая карты, любые утечки памяти приведут к тому, что адресное пространство процесса рано или поздно будет исчерпано и это приведет к аварии.

Такого рода надежность очень трудно реализовать отчасти потому, что требуется длительное тестирование. Единственная возможность проверить, работает ли что-то в условиях нагрузки в течение двух недель, — это проводить проверку в течение двух недель. И когда авария происходит, зачастую это связано с какой-то ерундой, которая произошла накануне, что решительно невозможно понять в момент аварии.

Однажды (12 лет назад) я писал приложение для солидного банка. Это была DOS-система (помните DOS?), которая могла работать целый день, но проработать без аварии неделю была не спо-

Важнейшее качество Web-сервера — его устойчивость при больших нагрузках.

Это чрезвычайно трудно реализовать. Да и описать.

Большая часть программ требует периодического перезапуска.

собна, и выловить ошибки мы были не в состоянии. У меня появилась блестящая идея поставить в машину плату со сторожевым таймером, который наша программа должна была периодически сбрасывать. Если таймер переполнялся, не будучи сброшенным, это приводило к перезагрузке системы. Банк не захотел использовать такое сляпанное наспех решение (надо им было о нем не говорить — просто сделать и жить спокойно) и вместо этого обязал администратора перезагружать систему каждый вечер.

ASP.NET поддерживает утилизацию процессов, чтобы продолжать работу, столкнувшись с неверным пользовательским кодом.

Старая ASP поддерживает бесконечную работу пользовательского процесса. Любые ошибки и утечки памяти в пользовательских программах аккумулируются и в какой-то момент приводят к аварии. В ASP.NET учитывается, что пользовательский код может быть неидеальным. Поэтому администратор может настроить сервер для периодического отключения и перезапуска рабочих процессов. Такая утилизация процесса настраивается элементом `<processModel>` в файле `machine.config` на уровне компьютера (рис. 3-37). Было бы здорово, если бы Microsoft позволила настраивать работу процессов на уровне приложения, но на момент написания этой книги этого сделано не было. Вы можете указать ASP.NET отключить рабочий процесс и вызвать новый через заданный интервал (атрибут `timeout`) после определенного количества запросов страниц (атрибут `requestLimit`) или при превышении лимита используемой памяти (атрибут `memoryLimit`). Хотя это и не значит, что мы можем расслабиться и писать ненадежный код, это все же позволяет работать с небольшими утечками памяти без ежедневной перезагрузки сервера. Как вы можете заметить, при настройке процесса можно использовать и другие параметры.

Утилизация процесса настраивается в файле `machine.config`.

```
<processModel
  enable="true"
  timeout="infinite"
  idleTimeout="infinite"
  shutdownTimeout="0:00:05"
  requestLimit="infinite"
  requestQueueLimit="5000"
  memoryLimit="80"
  webGarden="false"
  cpuMask="0xffffffff"
  userName=""
  password=""
  logLevel="errors"
  clientConnectedCheck="0:00:05"
/>
```

**Рис. 3-37.** Параметры утилизации процесса в файле *machine.config*.



## Глава 4

# Web-службы .NET

*Канон совместный — их и мой — в затверженном звучанье:  
«Закон, Порядок, Служба, Долг, Надежность, Послушанье!»  
Так их сработали навек, такую мысль внуша, —  
Подумать может человек, что есть у них душа.*

Р. Киплинг об устойчивости психики, необходимой  
для работы с высокими технологиями.  
(«Молитва МакЭндрю», 1894 г.)

### Суть проблемы

Описанная в этой книге базовая модель взаимодействия с пользователем не изменилась с тех пор, как в CERN (Женева) создали Web, чтобы просматривать скучные отчеты о физических экспериментах. Человек (или, как это было в популярном мультфильме, собака) запрашивает с сервера страницу с помощью универсальной программы-браузера, сервер декодирует запрос и передает страницу, браузер готовит ее для просмотра человеком, который, читая ее, мужественно борется со сном. Проблему скуки во многом удалось решить, улучшив информационное наполнение страниц (помещая на них итоги спортивных событий, порнографию, замысловатые музыкальные клипы Эла Янковича<sup>1)</sup>), но конечным потребителем запрошенных данных все равно остается человек, а не компьютерная программа.

Сегодняшняя структура Интернета такова, что в нем содержатся понятные людям сведения, а не данные для обработки клиентскими программами.

<sup>1)</sup> Его веселую пародию на Звездные войны можно увидеть по адресу [www.sagabegins.com](http://www.sagabegins.com).

Человечество получит громадную пользу, если специальные программы смогут запрашивать, «понимать» и использовать данные из Интернета так же легко, как это делают люди.

Интернет тем и хорош, что вездесущ. К нему подключены все (а если не все, то скоро остальные подключатся) интеллектуальные устройства на планете. Пользователи пожнут великую выгоду, если Web-серверы смогут обеспечивать данными программы, под управлением которых работают все многочисленные устройства, так же легко, как они

создают страницы для людей. Например, разработчики смогут существенно улучшить вид своих программ для пользователя, применяя вместо посредственного интерфейса (который работает в браузере), поддерживаемого сервером, замечательные специализированные интерфейсы, работающие на клиентской машине. Подумайте, насколько легче работать с электронной почтой через специализированный интерфейс Microsoft Outlook, по сравнению с универсальным интерфейсом Hotmail на основе браузера (но мы и это можем испортить дурацким интерфейсом — смерть танцующей скрепке!). Разработка специализированных пользовательских интерфейсов позволит повысить производительность серверов, если перенести форматирование данных для представления пользователю на клиентскую машину: представьте, что каждый из 1 000 клиентов будет сам форматировать для себя данные, а не единственный сервер будет заниматься форматированием данных для 1 000 клиентов. Обеспечение различных устройств данными через Интернет позволит создать программы вообще без пользовательского интерфейса, например, программу для банковского аудита. Это позволит повсеместно использовать связь через Интернет, не разделяя прикладную логику программ на страничные запросы. Аналогичным образом это обеспечит появление нового поколения некомпьютерных Интернет-устройств, например телефонов, которые будут выполнять все свои функции, используя Интернет вместо стандартных телефонных линий.

Сегодняшняя ситуация, когда доступ в Интернет осуществляется, главным образом, через универсальную программу-браузер, напоминает начало XX века, когда электричество только появилось в домах американцев. Тогда вся домашняя техника была еще без встроенных электромоторов. Поэтому компания Sears, например, продавала отдельный электродвигатель (по 8 долларов 75



центов)<sup>2</sup>, который можно было приладить ко всякой домашней технике: к швейной машине, миксеру или вентилятору. Ситуация была затруднительная, поскольку прежде чем использовать какой-нибудь аппарат, к нему приходилось подключать мотор и настраивать его. Как правило, денег хватало лишь на один мотор, поэтому, чтобы заштопать какую-либо вещь в жаркий день, приходилось выбирать между швейной машиной и вентилятором. А если к тому же приборов, к которым было нужно подключить мотор, было много, часто ни один из них толком не работал. Но по мере того, как моторы становились все меньше и дешевле, их стали встраивать в каждый аппарат. Эта тенденция дошла до того, что сегодня практически невозможно найти зубную щетку или разделочный нож без моторчика внутри. Современная бытовая техника проста в использовании, поскольку моторы и их инфраструктура (блоки питания, передачи и т. д.) оптимизированы для решения конкретной задачи и даже не видны снаружи. Не задумываясь о них, вы просто включаете прибор и взаимодействуете с его специализированным интерфейсом. Такого же рода революция начинается и в программировании для Интернета. Подобно тому, как моторы стали встраивать в каждый бытовой прибор, в любую написанную программу скоро будут внедрять средства доступа в Интернет. Вам не понадобится универсальный браузер, если только вы не собираетесь просто побродить по страницам. Вместо этого вы будете использовать специализированные программы, оптимизированные для решения конкретных задач. Вы не будете думать о том, как программа выходит в Интернет, как не задумывается о моторах в бытовой технике (пока они работают нормально и не ломаются, что зерно и для специализированных программ, использующих доступ в Интернет). Примером первых программ этого типа может служить Napster, который позволяет искать на жестких дисках тысяч участников программы музыкальные файлы согласно заданным критериям и загружать понравившиеся. Экран этой программы, которая ищет песни, бесплатно распространяемые ис-

Решать конкретные задачи с помощью специализированного аппаратного и программного обеспечения легче, чем делать все с помощью одного универсального устройства

<sup>2</sup> Изображение этого продукта от Sears находится в книге Дональда Нормана *The Invisible computer* (MIT Press, 1999) на стр. 50.

полнителями через Интернет, показан ниже (рис. 4-1). Другой пример программы со «скрытым» доступом в Интернет — специализированный интерфейс многопользовательской игры. Самая последняя редакция Microsoft Money делает полезные вещи, незримо объединяя данные из Web (текущие биржевые котировки и балансы) с содержимым рабочего стола (с локально созданными финансовыми расчетами).

Разработчики будут создавать приложения, komponya их из доступных в Web служб, так же, как они сейчас строят программы из готовых компонентов на локальных машинах.

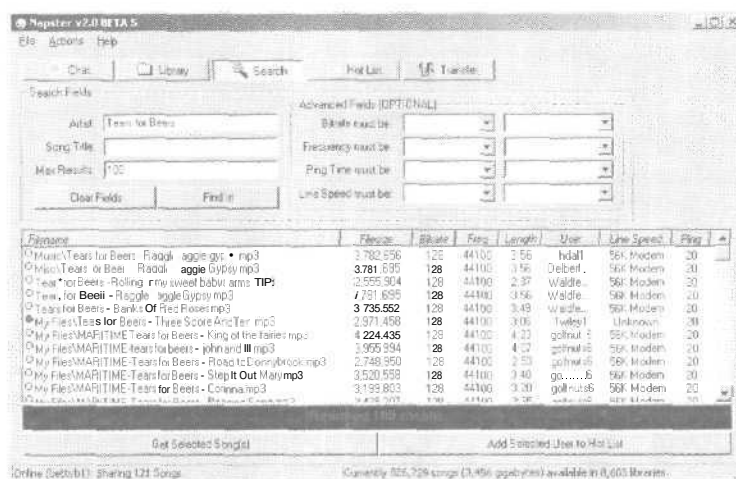
Программные средства, обеспечивающие легкий доступ в Интернет, позволят разработчикам компоновать приложения из доступных через Web служб так же, как они сегодня строят их из готовых компонентов на локальных ПК. Разработчик текстового процессора, который считается типичным автономным приложением, в этом случае может

включить в программу лишь элементарные средства проверки орфографии или вовсе не включать их. Для проверки сложной орфографии или при необходимости обратиться к толковому словарю текстовый процессор может подключаться к онлайн-редакции *Oxford English Dictionary*, расположенной на сайте *oed.com*. Этот словарь скромно называется «самым авторитетным и полным в мире словарем английского языка»<sup>3</sup>. За подписку на рабо-

<sup>3</sup> В настоящее время я пытаюсь заставить редакционную коллегию OED включить в словарь слово, которое я «родил» в статье, опубликованной журналом *Byte.com* 23 августа 1999 г. Это слово — MINFU, аббревиатура в стиле казарменных сокращений SNAFU и FUBAR (сокращения от довольно жестких американских ругательств. — Прим. перев.), которые прочно вошли в повседневное употребление. Словом MINFU в определенных кругах часто величают так называемый Microsoft Nomenclature Foul-Up (терминологическую неразбериху Microsoft). Например, называть активизацию на месте (in-place activation) внедренного объекта «визуальным редактированием» (я так понимаю, чтобы не путать его с осозательным и обонятельным) — это MINFU. Все терминологическое фиаско с COM-OLE-COM-ActiveX-COM было и по сей день остается одним гигантским MINFU. На мое письмо из OED пришел вежливый ответ, где, в частности, было сказано: «...если мы не найдем более общеупотребительный вариант, то рассмотрим возможность его включения в OED». В настоящее время пара авторов использует это слово, из которых самый известный — Дэвид Чеппел (David Chappel). Надеюсь, вы все будете использовать это слово в своих произведениях и пришлете мне ссылки на них, которые я перешлю в OED. Не надо благодарности — используйте это слово на здоровье (если я смогу протолкнуть его через Microsoft Press).

ту с этим сайтом надо платить, и пока что он доступен людям лишь через универсальные браузеры. Поддержка встроенного доступа к OED в программах позволила бы намного повысить объемы продаж подписки. Возможно, производители текстовых процессоров смогли бы снизить плату за подписку. Самым умным решением было бы давать пробную версию программы со встроенным доступом, которая работала бы несколько месяцев бесплатно. За это время у пользователя должна развиться зависимость от этой программы, а по окончании испытательного периода доступ закрывается, если пользователь не заплатит.

Доступ в Интернет должен быть встроен в каждую специализированную программу, а не доступен только через универсальный браузер.



**Рис. 4-1.** Пользовательский интерфейс программы Napster, специализированного приложения, использующего доступ в Интернет.

Чтобы создавать такие программы, разработчикам нужна возможность быстро и легко (т. е. без особых затрат) писать код, взаимодействующий с другими программами через Интернет. Идея не нова, существует несколько методик такого взаимодействия: RPC, DCOM и MSMQ. Каждая из них сама по

Устройства, использующие наиболее захватывающие коммуникационные технологии, работают исключительно друг с другом.

себе замечательна, а два-три года назад они представляли собой неплохие идеи. Но у них у всех есть одно фатальное ограничение: любая из этих систем может работать только с аналогичной системой — MSMQ может «разговаривать» только с MSMQ, клиент DCOM — только с сервером DCOM и т. д.

Необходимо универсальное межпрограммное взаимодействие с помощью функций через существующие каналы Интернета.

Что нужно на самом деле, так это универсальный программный доступ в Интернет, позволяющей программе из одного устройства вызвать функцию, написанную кем угодно, из другого устройства. Такой доступ не должен зависеть не только от ОС, но и от

внутренней реализации программы. (То есть не важно, на каком языке она написана — на C++ или BASIC'e, кто ее производитель, какая версия программы. Нам еле-еле удастся справиться с этим даже на единственном автономном компьютере. Этот доступ должен быть простым в использовании, иначе никто не захочет тратить время на написание программ, использующих его.

## Архитектура решения

При управлении международными полетами и универсальным доступом в Интернет возникают сходные проблемы.

Проблема универсального программного доступа в Интернет аналогична проблеме, с которой сталкиваются разработчики систем управления международными полетами. Члены случайноменяющегося набора гетерогенных узлов (самолеты различных типов) должны взаимодействовать с фиксированными серверами (диспетчерскими управления полетами) и между собой. Внутренние способы работы программ могут быть всевозможными и несовместимыми (как мысли летчиков, говорящих на тайском, норвежском и калифорнийском английском). При этом взаимодействие между экипажем, самолетом и диспетчерской должно протекать без сбоев, иначе могут случиться ужасные вещи, как например, 3 марта 1977 г., когда в аэропорту Тенерифе из-за неверно понятых указаний диспетчера столкнулись при взлете два Боинга 747, в результате погибли 335 человек.

Чтобы решить проблему, все страны, осуществляющие полеты, должны прийти к соглашению. При данных масштабах (совокупности всех операторов воздушных перевозок) и неоднородности (при наличии авиалиний для богатых и бедных, транспортной, универсальной и военной авиации, контрабандистов и т. д.) этой группы единственным под-

Решить проблему с управлением полетами удалось с помощью «наименьшего общего знаменателя», т. е. достигнув соглашения об использовании английского языка.

ходом, обеспечивающим работу системы, является стандартизация на основе «наименьшего общего знаменателя» двух критических областей. Во-первых, это физический механизм передачи информации. В авиации для этого применяются радиостанции, работающие на выделенной частоте в метровом диапазоне. Во-вторых, для управления полетами требуется стандартизированное кодирование передаваемой информации. Согласно международному соглашению все пилоты и диспетчеры должны вести переговоры только по-английски, даже если самолет авиакомпании Эйр Франс садится в международном аэропорту им. Шарля де Голля в Париже (хотя французы в гробу видели английский).

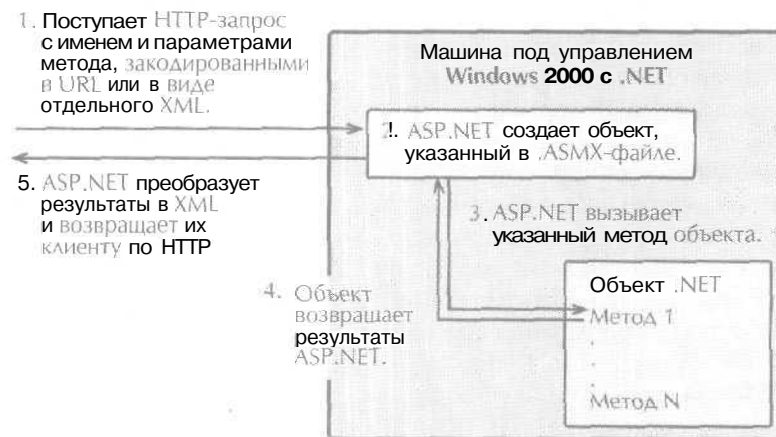
Как и в случае решения проблемы управления полетами, единственный способ справиться с огромным числом гетерогенных сущностей в Интернете — это использовать их «наименьший общий знаменатель». Чтобы принять решение о способе пересылки данных с одного устройства другому, нужно учесть, что для этого следует использовать то, что есть в каждом подключенном к Интернету устройстве. Аналогично тому, как на каждом самолете установлена метровая радиостанция, в Интернете наиболее распространен протокол HTTP (Hypertext Transfer Protocol), который сейчас используется для получения страниц практически всеми Web-браузерами. Кроме того, нужен «наименьший общий знаменатель» в области кодирования информации, которая передается по протоколу HTTP. Подобно тому, как при управлении полетами все переговоры ведутся на английском, в нашем случае в рамках универсальной схемы все пересылаемые данные будут кодироваться с помощью XML (Extensible Markup Language).

В результате объединения этих идей в Microsoft создано понятие *Web-служб* (Web Services). С помощью Web-служб объекты на

сервере могут бесшовно принимать запросы, поступающие от клиентов, применяя «наименьший общий знаменатель Интернета» — HTTP/XML. Чтобы создавать Web-службы, не обязательно изучать новые способы программирования. Объекты при этом пишутся так же, как если бы они были предназначены для работы напрямую с локальными клиентами. Просто они помечаются атрибутами, указывающими, что эти объекты надо сделать доступными Web-клиентам, а остальное делает ASP.NET. ASP.NET автоматически подключает готовую инфраструктуру, которая принимает поступающие через HTTP запросы и переправляет их объектам (рис. 4-2). Объединенные в составе Web-службы объекты могут работать с любым элементом Web, который «говорит» на HTTP и XML. Иными словами, с кем угодно в этой вселенной,

Web-службы бесшовно соединяют объекты .NET и запросы, поступающие через HTTP.

независимо на тип их ОС и окружения периода выполнения, в котором они работают. При этом писать инфраструктуру для связи через Web не нужно — она предоставлена .NET Framework.



**Рис. 4-2.** Взгляд на Web-службы ASP.NET со стороны сервера.

На стороне клиента .NET поддерживает прокси-классы, обеспечивающие простой доступ к Web-службам, которые предоставляет любой принимающий HTTP-запросы сервер (рис. 4-3). Средство разработки читает описание Web-службы и генерирует про-

кси-класс с функциями на любом языке, на котором ведется разработка на клиенте. При вызове одной из этих функций прокси-класс создает HTTP-запрос и посылает его на сервер. Когда с сервера приходит ответ, прокси-класс выполняет синтаксический разбор результатов и возвращает их в виде значения функции. Это обеспечивает бесшовное взаимодействие клиента с любым сервером, «говорящим» на HTTP и XML, т. е. фактически с любым сервером.

.NET также поддерживает прокси-классы, которые облегчают создание клиентов Web-служб, скрывая детали реализации связи через Интернет.

0. В период разработки программист генерирует код прокси-объекта на основе описания Web-службы.

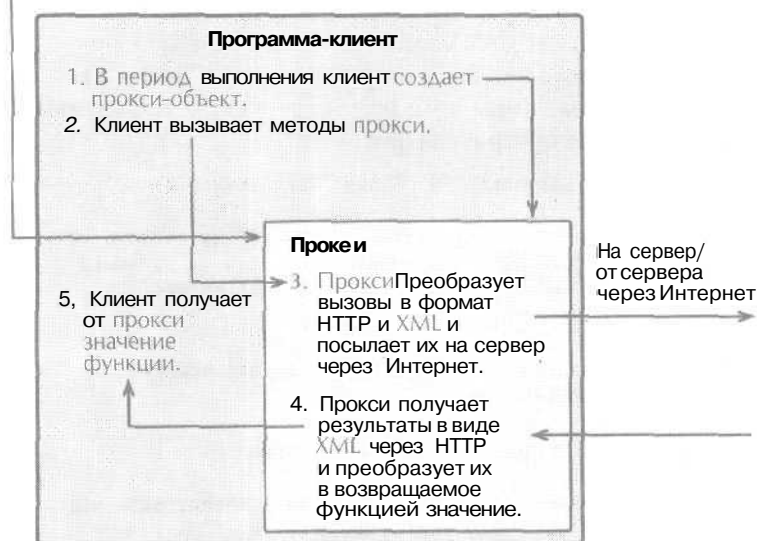


Рис. 4-3. Взгляд на Web-службы ASP.NET со стороны клиента.

## Простейший пример: создание Web-службы

Собираясь изучить новую технологию или научить кого-нибудь ей, я всегда пишу простейшую программу-пример. Мне кажется, она поможет продемонстрировать создание Web-службы. Она под-

Здесь начинается рассмотрение примера Web-службы.

держивает единственный метод, *GetTime*, который получает значение текущего времени с машины-сервера в виде строки с разрядами секунд или без них. Исходный текст этой службы можно загрузить с web-сайта этой книги ([www.introducing-microsoft.net](http://www.introducing-microsoft.net)) и разбирать его, следуя дальнейшему описанию.

Написать Web-службу можно и без Visual Studio. Я, например, написал ее в Блокноте.

Эту XWeb-службу я написал в виде страницы ASP.NET. Сначала я загрузил (это можно сделать бесплатно) и установил .NET SDK. После этого, используя обычные инструменты администрирования IIS (Internet Information Services) из Windows 2000, я настроил виртуальный каталог, указывающий на папку, в которой будет файл Web-службы. Затем созданный код ASP.NET я записал в файл *TimeService.asmx* и поместил его на сервер. Все эти действия можно выполнить и без Visual Studio; на самом деле все это я написал в Блокноте. Вот исходный текст программы (рис. 4-4):

```
<%@ WebService Language="VB" Class="TimeService"%>
```

```
' Первая строка заголовка указывает ASP.NET на то, что
' в этом файле находится Web-служба, написанная на языке
' Visual Basic, а также что эта служба предоставляется
' классом TimeService.
```

```
' Импорт пространств имен (считайте, что это ссылки),
' необходимых Web-службе.
```

```
Imports System Imports System.Web.Services
```

```
' Объявляем новый класс службы. Он должен наследовать код
' от системного базового класса WebService.
```

```
Public Class TimeService : Inherits WebService
```

```
' Помещаем в класс функции.
' Пометить их атрибутом WebMethods.
```

```
Public Function <WebMethod( )> GetTime (ShowSeconds as _
Boolean) As String
```



```

' Реализация прикладной логики функции:
' получить текущее время, отформатировать в соответствии
' с запросом и вернуть строку.

Dim dt As DateTime

If (ShowSeconds = True) Then
    GetTime = dt.Now.ToLongTimeString
Else
    GetTime = dt.Now.ToShortTimeString
End If

End Function

End Class

```

**Рис. 4-4.** Простая Web-служба.

Хотя эта программа довольно проста, некоторые ее конструкции, вероятно, вам незнакомы, поэтому разберем ее шаг за шагом.

Программа начинается со стандартного приветствия ASP.NET, `<%@%>`. В нем расположена директива *WebService*, которая сообщает ASP.NET, что код страницы должен рассматриваться как Web-служба. Атрибут *Language* задает язык этой страницы. Я использовал Visual Basic, поскольку с ним знакомо подавляющее большинство моих читателей. ASP.NET будет компилировать этот исходный текст с помощью Visual Basic.NET. Visual Studio для этого ставить не надо, все компиляторы входят в .NET SDK. Атрибут *Class* задает класс объектов, который должен быть активизирован для обработки входящих запросов, адресованных этой службе.

Остальной код страницы содержит реализацию класса. Я применил директиву *Imports* (это новинка Visual Basic.NET), которая указывает компилятору «импортировать пространство имен». Пространство имен (namespace) — замечательный способ ссылки на описание набора готовой функциональности. Концептуально он идентичен ссылкам в проектах Visual Basic 6. Поскольку при по-

На странице ASP.NET задан класс .NET, объекты которого используются в Web-службе.

При поступлении первого запроса ASP.NET автоматически скомпилирует код, написанный на Visual Basic.

ступлении запроса от клиента ASP.NET «оперативно» («just-in-time») компилирует эту программу, проект, в котором можно было бы задать эти ссылки, отсутствует. Ссылки придется поместить в программу явно. Имена, перечисленные после директивы *Imports*, задают обработчику наборы функциональности, ссылки на которые надо включить в программу. В нас будут включены ссылки на *System* и *System.Web.Services* — готовые механизмы, используемые в Web-службах. Остановите меня, если я чересчур углубился в технические подробности!

Теперь Visual Basic поддерживает наследование, что позволяет получить доступ к готовым системным функциям.

В следующей строке определено имя класса. Вы видели и писали сами массу классов на Visual Basic, и этот мало чем от них отличается. В моей программе в конце строки *Public Class TimeService : inherits WebService* есть новое ключевое слово. Одним из многих

улучшений Visual Basic стала поддержка методики объектно-ориентированного программирования, называемого *наследованием* (inheritance) (я рассказал о ней в главе 2 — со временем она вам обязательно понравится). Словами «мой класс *TimeService* наследует от класса *WebService*» я приказал компилятору взять весь код системного класса *WebService*, который в этом случае называется *базовым* (base) классом, и включить его в класс *TimeService* [*производный* (derived) класс]. Наследование можно представить в виде операций копирования и вставки; при этом никаких перемещений на самом деле не происходит. Честно говоря, самые ярые адепты C++ и Java даже физическую операцию копирования/вставки кода часто называют «наследованием через редактор». Класс *WebService* поддерживается новой библиотекой периода выполнения .NET, чем он напоминает Visual Basic 6, многие объекты которого поддерживала ОС. В этом классе в готовом виде содержатся все коммуникационные механизмы, нужные для обработки входящих HTTP-запросов и маршрутизации их соответствующим методам объекта.

Студенты часто спрашивают, чем различаются импорт пространства имен и объявление наследования, т. е. ключевые слова *Imports* и *Inherits*. *Imports* лишь вводит описание набора функциональности, но реально его не использует. Как сказано выше, это

ключевое слово подобно установленной ссылке. *Inherits* реально берет часть кода, на который она ссылается, включает его в описание и использует. Оно работает как оператор *Dim as*, накачан- ный стероидами.

Определив класс, надо определить его мето- ды и функции. И это делается почти так же, как на Visual Basic 6, но с одним новым наво- ротом: к каждому методу, который должен быть доступен Web-клиентам, надо добав- лять новый атрибут `<WebMethod(>`. Этот атрибут указывает ASP.NET, что помеченный им метод должен быть доступен клиентам в виде Web-службы. В Visual Basic 6 вы могли видеть массу элементов, похожих на атрибуты: *Public*, *Private* и *Const*. Многие из них применяются и в Visual Basic.NET, и .NET Framework, но с новым синтаксисом. Если у класса могут быть открытые и закрытые методы, то точно так же одни из его методов могут быть доступны Web-клиентам, а другие — нет.

В завершение мы доберемся до внутренне- го устройства этого метода. Как и для меня, для вас работа со временем может оказать- ся несколько новой. Но не стоит сейчас об этом беспокоиться — обработку времени и дат выполняет Visual Basic.NET.

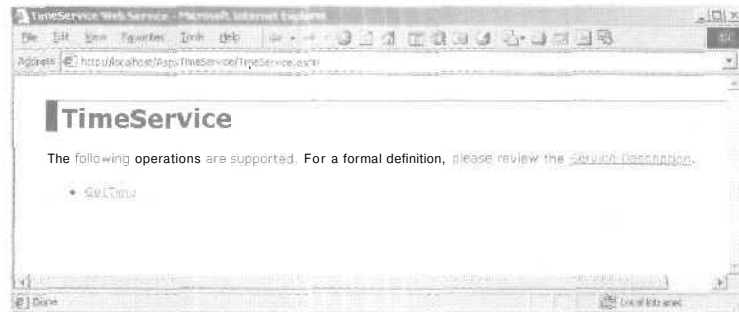
А теперь попробуем обратиться к нашей Web-службе с клиента. Хорошие средства поддержки для этого также имеются в ASP.NET в готовом виде. Если запустить Internet Explorer 5.5 и запросить страницу, которую мы только что написали на ASP.NET, резуль- тат будет таким, как показано ниже (рис. 4-5).

ASP.NET обнаруживает факт обращения к самой странице Web-службы (а не к одному из ее методов) и в ответ выводит страни- цу со списком функций, которые поддерживает эта служба. В данном случае в списке будет лишь одна функция, *GetTime*. В ответ на щелчок имени функции ASP.NET покажет страницу, ко- торая позволит проверить этот метод (рис. 4-6). Чтобы заставить *GetTime* показывать секунды или запретить это, введите TRUE или FALSE, соответственно, и щелкните кнопку Invoke. В итоге тестов- ая страница вызовет указанный метод и вернет результат

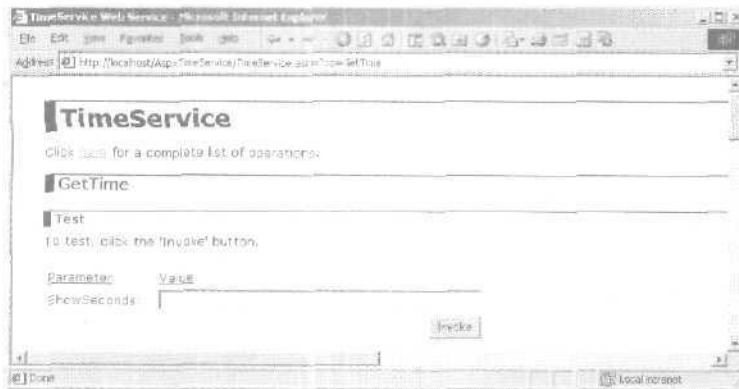
С помощью соответст- вующего атрибута можно указать ASP.NET предоставлять метод как Web-службу.

Всю прикладную логику можно писать с по- мощью классов Visual Basic почти так же, как обычно.

(рис. 4-7). Можно видеть, что в составе URL в поле Address передаются параметры, а результат возвращается в форме XML. Также обратите внимание, что страницу следует открывать так, как это делается в IIS, набирая ее URL: `http://localhost/[имявашего виртуального каталога]/TimeService.asmx`. Если просто дважды щелкнуть страницу при просмотре содержимого жесткого диска через Internet Explorer, вы минуete IIS и ASP.NET и получите просто текст страницы .ASMХ, но это вовсе не то, что нужно, не так ли?



**Рис. 4-5.** Экран по умолчанию, сгенерированный ASP.NET при запросе базовой страницы Web-службы.



**Рис. 4-6.** В ответ ASP.NET выводит такую страницу.

Удобные свойства логистики и развертывания объектов .NET (см. главу 2) есть и у моей Web-службы. Например, чтобы найти мой

объект, ASP.NET не требуется реестр, который нужен для поиска объектов COM: местоположение моего объекта задает URL. Обновлять код совсем не трудно: нужно просто скопировать новый файл поверх старого (попробуйте сделать это с примером на своей локальной машине). Он не будет заблокирован, что случилось бы на его месте с сервером COM. Мне даже не придется перезапускать сервер IIS, чтобы уведомить его о новом файле. Он автоматически обнаружит, что файл изменился, при необходимости оперативно скомпилирует его «по требованию» и в дальнейшем при обработке запросов к объекту будет использовать новую версию.

Web-службу, как и другие объекты .NET, легко администрировать и развертывать.

Для проверки Web-службы ASP.NET автоматически предоставляет простой клиент на основе браузера.

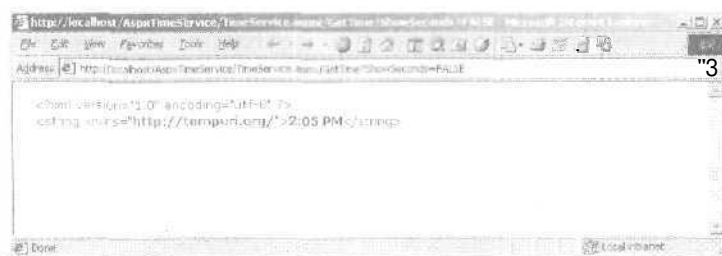


РИС. 4-7. Результаты, которые вернула Web-служба.

Вот и все, что потребовалось сделать, чтобы написать мою Web-службу. Ее исходный текст занимает всего лишь 13 строк, считая *Else* и *If*. Куда уж легче!

Этот пример демонстрирует массу мощнейших функций, использовать которые очень легко.

## Самоописываемость Web-служб: WSDL-файлы

Программистам, которые создают клиентские приложения, использующие Web-службы, потребуется описание выполняемых ими действий, а также сведений, которые следует сообщить службе, чтобы заставить ее работать. Так, клиенту нужно знать предоставляемые службой методы, их параметры и поддерживаемые протоколы. Концептуально эта информация аналогична инфор-

Web-службы должны предоставлять описание своей функциональности заинтересованным клиентам.

мации в библиотеке типов, которую использует стандартный компонент COM. Однако проблема с библиотеками типов в том, что они специфичны для Microsoft COM. Хотелось бы, чтобы клиентами Web-служб могли бы быть не только Microsoft-системы. Желательно также иметь возможность создавать описания для служб других авторов, кроме Microsoft, чтобы они работали в системах не только под управлением ОС Microsoft, но и других. Но при всем при этом клиентские приложения, разработанные Microsoft, должны иметь возможность использовать эти службы. Требуется универсальный подход к описанию службы, не ограниченный миром Microsoft. (Могли бы вы помыслить, что прочитаете подобные слова в книге, изданной Microsoft? Лично я — нет. Да, мир сегодня уже не тот...) Этот подход должен быть доступен для машинного анализа, что позволило бы использовать его в интеллектуальных средах разработки (так же, как библиотеки типов).

Описание Web-службы дается в виде WSDL-файла.

Инфраструктура ASP.NET может генерировать такое описание путем анализа метаданных (см. главу 2) в коде, реализующем службу. Описание хранится в XML-файле, использующем словарь под названием WSDL (Web Service Descriptor Language)<sup>4</sup>.

<sup>4</sup> Обращает на себя внимание тот факт, что многие акронимы в .NET не являются трехбуквенными сокращениями (TLA или Three-Letter Acronyms, TLA само по себе является TLA), которые все привыкли видеть в большинстве компьютерных проектов. Дело в том, что существует всего лишь 17 576 уникальных TLA и Microsoft исчерпала их все, не дойдя и до половины проекта. Я предвидел эту проблему два года назад и предложил решение в виде свертывания TLA в CCT (Clever Compound TLA — «умные составные TLA»; кстати, CCT также само по себе является CCT). Хотя этот подход, по-видимому, прижился в XML (возьмем, к примеру, полунасыщенные CCT: XSL — XML Stylesheet Language или SAX — Simple API for XML), Microsoft явно открещивается от использования CCT, предпочитая наращивать длину слов, производя акронимы SOAP, WSDL и UDDI. Эти терминологические объекты еще не получили имени, поэтому я объявляю, что они будут называться FLAP (Four-Letter Acronym Packages). Естественно, что аббревиатура FLAP — это тоже FLAP.

WSDL-файл Web-службы иногда называют *контрактом* (contract), поскольку в нем перечислены все функции, которые может выполнять служба, и указаны способы их вызова. WSDL-файл можно получить от ASP.NET, добавив к URL при запросе .ASMX-файла символы *?WSDL*. Например, на своей локальной машине я получил WSDL-файл с описанием примера Web-службы, запросив URL <http://localhost/AspxTimeService/TimeService.asmx?WSDL>. Именно эту операцию выполняет ссылка Service Description со страницы, показанной выше (рис. 4-5).

Инфраструктура .NET может генерировать описание программы на основе ее исходного текста.

При создании COM-компонентов на Visual Basic 6 или Visual C++ 6 иногда сначала пишут компонент, а затем описывающую его библиотеку. В других случаях начинают с библиотеки типов, описывающей интерфейс, а затем пишут реализующий ее код. WSDL может работать по любому из этих сценариев. Можно сначала написать код, как я поступил здесь. В этом случае ASP.NET сгенерирует WSDL-файл для заинтересованных клиентов. С другой стороны, я мог бы сначала написать WSDL-файл (или получить его из третьих источников), описывающий возможности службы, а потом с помощью утилиты из SDK сгенерировать шаблон, реализующий описанный WSDL-файлом службу (этот подход аналогичен использованию ключевого слова *Imports* в программах на Visual Basic 6).

В качестве альтернативы можно сначала создать описание, а затем генерировать для него код.

В силу определенных причин WSDL-файлы довольно сложны, поэтому ниже я привожу лишь выдержку из него, иллюстрирующую содержимое WSDL-файлов (рис. 4-8). В элементе *<service>* находятся элементы *<port>* для каждого протокола, позволяющего получить доступ к Web-службе. В нашем случае есть выбор между HTTP GET, HTTP POST и SOAP, являющимся гибридом HTTP/XML. В свою очередь каждый элемент *<port>* содержит сведения относительно других элементов документа (они не показаны), описывающие форматы входного запроса и выходных данных для каждого протокола.

Здесь в качестве примера показана часть WSDL-файла.

```

<service name="TimeService">
  <port name="TimeServiceSoap"
        binding="s0:TimeServiceSoap">
    <Soap:address location=
      "http://localhost/AspxTimeService/TimeService.asmx" />
  </port>
  <port name="TimeServiceHttpGet"
        binding="s0:TimeServiceHttpGet">
    <http:address location=
      "http://localhost/AspxTimeService/TimeService.asmx" />
  </port>
  <port name="TimeServiceHttpPost"
        binding="s0:TimeServiceHttpPost">
    <http:address location=
      "http://localhost/AspxTimeService/TimeService.asmx" />
  </port>
</service>

```

**Рис. 4-8.** Выдержка из WSDL-файла.

Интерпретировать содержание WSDL-файла позволяют различные инструменты разработки.

Вместо того чтобы при работе с библиотеками типов копаться в их двоичном содержимом, обычно применяют интерпретирующие средства просмотра. Аналогично при работе с WSDL-файлами зам скорее всего не придется иметь дело с их необработанным содержанием, если только вы не пишете программу для анализа WSDL-файлов. Вместо этого вы будете работать с WSDL-файлами, используя интерпретирующие средства. Например, таким средством является проверочная страница, которую предоставляет ASP.NET. Эта страница не только проверяет (рис. 4-6), но и интерпретирует WSDL-файл, показывая поддерживаемые им протоколы и способы их использования для доступа к службе. Следующий снимок экрана показывает часть страницы с описанием способа доступа к службе TimeService через протокол HTTP SET (рис. 4-9).



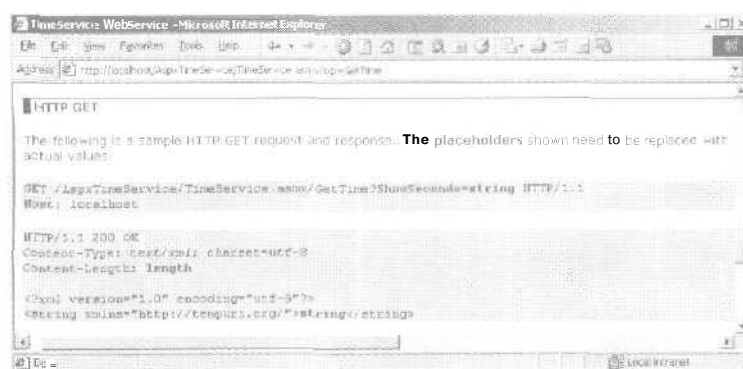


Рис. 4-9. Клиентская программа, которая обращается к Web-службе через HTTP GET.

## Создание клиента Web-службы

Мне показалось, что написать клиент не труднее, чем саму службу. Слушатель запросов ASP.NET, маршрутизирующий входящие запросы к объектам Web-службы, может работать с запросами по трем протоколам: HTTP GET, HTTP POST и SOAP. Концептуально это напоминает ситуацию при управлении самолетом, когда диспетчеры говорят на различных диалектах английского: британском, американском и австралийском. Первые два протокола поддерживаются в основном из соображений преемственной совместимости, так как сегодня практически все программы в Web используют один из этих протоколов. Однако при осуществлении новых проектов, вероятно, будет легче использовать SOAP. В частности, из-за преимуществ, которые дают готовые средства поддержки этого протокола из Visual Studio.NET. Чтобы изучить способы написания клиентов, применяющих эти механизмы, рассмотрим каждый из них.

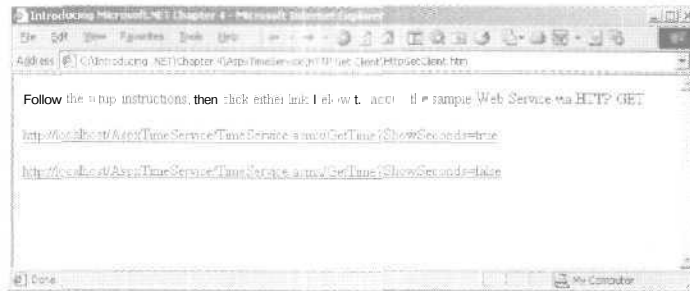
### Случай 1: HTTP GET

Моя Web-служба будет принимать запросы в виде простых запросов HTTP GET с параметром *ShowSeconds* в строке URL. В предыдущем разделе мы видели фрагмент WSDL-файла с описанием поддержки для этого протокола. А вот пример Web-страни-

цы, предоставляющей доступ к таким запросам (рис. 4-10), — эта страница также есть на сайте среди других примеров этой главы. Когда запрос приходит на сервер, ASP.NET выполняет синтаксический разбор параметров в строке URL, создает объект *TimeService* и вызывает метод *GetTime*. ASP.NET принимает значение, возвращаемое этим методом, форматирует его как XML и возвращает клиенту результат в форме XML. Экран с результатами показан выше (рис. 4-7).

Вызов web-службы будет осуществляться по протоколу HTTP GET.

значение, возвращаемое этим методом, форматирует его как XML и возвращает клиенту результат в форме XML. Экран с результатами показан выше (рис. 4-7).



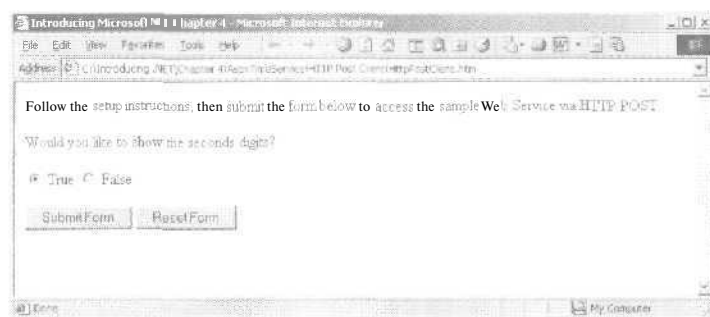
**Рис. 4-10.** Web-страница, которая обращается к нашей демонстрационной Web-службе.

### Случай 2: HTTP Post

Web-служба также будет принимать все вызовы, поступающие в виде запросов HTTP Post. Чтобы выяснить, как отформатировать запрос *Post*, чтобы Web-служба смогла его распознать и обработать, я изучил интерпретированный WSDL-файл, показанный выше. Затем с помощью Microsoft Frontpage 2000 я написал форму, выполняющую все, что нужно (рис. 4-11). HTML-код показан ниже (рис. 4-12) (соответствующий файл также есть на Web-сайте книги среди других примеров из этой главы). Когда запрос приходит на сервер, ASP.NET создает объект *TimeService*, получает от элементов управления формы параметры и вызывает метод

Web-служба будет принимать вызовы по протоколу HTTP Post.

*GetTime*. Затем ASP.NET принимает значение, которое возвращает этот метод, форматирует его как XML и возвращает клиенту результат в виде XML.



**Рис. 4-11.** Клиентская форма, которая обращается к Web-службе.

```
<form METHOD="POST" ACTION=
  "http://localhost/aspxtimeservice/timeservice.aspx/GetTime">
  <p>Would you like to show the seconds digits?</p>
  <blockquote><p>
    <input TYPE="RADIO" NAME="ShowSeconds" VALUE="True"
      CHECKED>
      True
    <input TYPE="RADIO" NAME="ShowSeconds" VALUE="False">
      False<br>
  </p></blockquote>
  <input TYPE="SUBMIT" VALUE="Submit Form">
  <input TYPE="RESET" VALUE="Reset Form">
</form>
```

**Рис. 4-12.** HTML-код клиентской формы.

### Случай 3: необработанный Soap

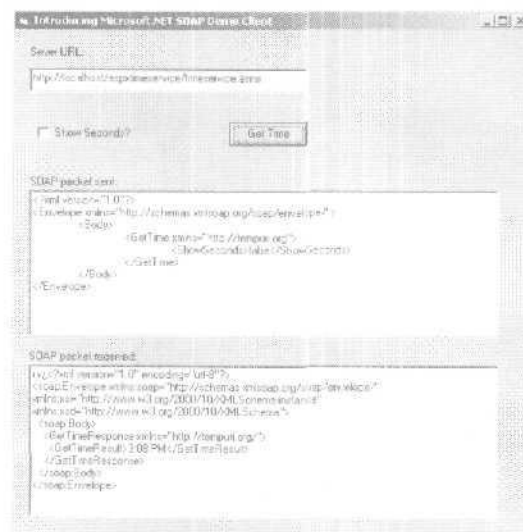
В дополнение Web-служба будет принимать вызовы, поступающие через HTTP Post, закодированные в пакеты SOAP. Нет-нет, SOAP — это не мыло<sup>5</sup>. SOAP означает Simple Object Access Protocol (Простой протокол доступа к объектам). Это богатый и гибкий словарь XML, описывающий функции и их параметры, для которого есть встроенная поддержка в Visual Studio.NET. Поэтому я думаю, что в большинстве случаев при разработке новых

<sup>5</sup> Soap (англ.) — мыло. — Прим. перев.

клиентов будет избран именно SOAP, а не HTTP GET или HTTP Post, работу которых я только что продемонстрировал. Я написал небольшую программу, которая с помощью SOAP вызывает метод Web-службы *GetTime* (рис. 4-13). При этом реальное взаимодействие осуществляется по протоколу HTTP. Поскольку код получился довольно громоздким, здесь он не приводится, но его можно найти среди других примеров на сайте этой книги. В отличие от вышеприведенных примеров, где URL указывал метод службы, в этом случае URL указывает на страницу .ASMX, содержащую все методы Web-службы. Пакеты SOAP, посланные на сервер, содержат имя и параметры функции, закодированные в XML согласно принятой схеме (верхнее текстовое поле на рисунке). Когда пакет SOAP достигает сервера, ASP.NET распознает его, выполняет синтаксический разбор пакета, получая имя и параметры

Web-служба будет принимать вызовы через SOAP.

метода, создает объект и вызывает метод. Затем ASP.NET принимает значение, которое вернул метод, форматирует его как XML и возвращает результат клиенту.



**Рис. 4-13.** Пример приложения, демонстрирующий доступ к Web-службе через SOAP.

#### Случай 4: синхронная работа интеллектуального SOAP-прокси

Писать пример из предыдущего раздела, использующий SOAP, было довольно утомительно. Это занятие напомнило мне написание клиента *IDispatch* на классическом COM вручную в том смысле, что там была куча

стереотипов, которые почти не отличались от метода к методу, критичных для правильной работы программы (ошибись хоть в одном символе — и капут). Visual C++ поддерживает класс-оболочку, который здорово упрощает доступ к объектам автоматизации (многие программисты на Visual Basic даже и не догадывались, что для адептов C++ этот процесс связан со многими страданиями; другим было безразлично, или они это поощряли). Теперь .NET SDK поддерживает набор классов-оболочек, которые делают написание клиента SOAP заурядной операцией.

Прокси-классы генерирует утилита командной строки Wsdl.exe из состава .NET SDK. В Visual Studio.NET ее можно запустить из среды разработки при добавлении ссылки на службу. Утилита читает описание Web-службы из WSDL-файла и генерирует прокси, дающий доступ к методам службы на заданном языке. В настоящее время поддерживаются Visual Basic, C# и JavaScript, но не C++. Можно создать прокси, использующий любой из поддерживаемых протоколов, но по умолчанию задан SOAP. Wsdl.exe запускает команда:

```
wsdl /l:vb http://localhost/aspxtimeservice/  
timeservice.asmx?WDSL
```

Классы-прокси в Visual Basic содержат функции, позволяющие получать время от нашей Web-службы как синхронно, так и асинхронно. Вот синхронная версия, которую мы и обсудим (рис. 4-14).

Прокси-класс наследует от базового класса *System.Web.Services.Protocols.SoapHttpClientProtocol* (нам часто придется использовать такие длинные, очень длинные имена, так что привыкайте — они хоть и наглядны), содержащего реальный код. В прокси-класса есть свойство *Url*, унаследованное от базового класса. Оно

.NET SDK поддерживает прокси-классы SOAP, что заметно облегчает написание клиентских приложений.

Прокси-класс, сгенерированный утилитой Wsdll.exe, предоставляет готовую функциональность для обработки запросов SOAP.

: задает URL сервера, на который направляется вызов. Из WSDL-файла это свойство получает значение по умолчанию, но при желании его можно изменить в период выполнения (программа-пример демонстрирует соответствующую методику). Клиент вызывает указанный метод у прокси с помощью метода *invoke*, который прокси также унаследовал от базового класса. Затем этот метод создает пакет SOAP, содержащий имя и параметры метода (рис. 4-13), и посылает его на сервер по HTTP. Когда возвращается с сервера ответный пакет SOAP, базовый класс осуществляет его синтаксический разбор, извлекает результат и возвращает его прокси, который передает его клиенту.

```
Public Class <System.Web.Services.WebServiceBindingAttribute
(Name:="TimeServiceSoap", _
[Namespace]:="http://tempuri.org/")> TimeService Inherits _
System.Web.Services.Protocols.SoapHttpClientProtocol

Public Sub New( )
    MyBase.New
    Me.Url = _
        "http://local host/aspxtimeservice/timeservice.asmx"
End Sub

Public Function _
    <System.Web.Services.Protocols.SoapMethodAttribute( _
        "http://tempuri.org/GetTime", MessageStyle:= _
        System.Web.Services.Protocols.SoapMessageStyle. _
        ParametersInDocument)>
    GetTime (ByVal ShowSeconds As Boolean) As String
        Dim results( ) As Object = _
            Me.Invoke("GetTime", NewObject ( ) {ShowSeconds})
        Return CType(results(0), String)
    End Function
End Class
```

**Рис. 4-14.** Класс-прокси SOAP на языке Visual Basic, сгенерированный Wsdll.exe.

Блок атрибутов в объявлении функции (символы в угловых скобках) содержит сведения для базового класса о форматировании вызова, в частности, имена методов. В Visual Studio.NET метаданные этих атрибутов интенсивно используются для передачи информации к средствам готовой функциональности системного кода. Вероятно, раньше это пришлось бы делать с помощью членов-переменных базового класса, при этом было бы трудно различить постоянные атрибуты периода выполнения и атрибуты, изменяемые во время исполнения программы. Новый подход более устойчив к путанице, что в общем случае хорошо.

Ниже показаны реальный код клиента на Visual Basic (рис. 4-15) и вид клиентского приложения (рис. 4-16). Видно, что ничего необычного в нем нет. Всю рутинную работу сделал прокси-класс.

' Пользователь щелкнул кнопку GetTime.

```
Protected Sub button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles button1.Click

    ' Создать новый объект класса-прокси,

    Dim foo As New TimeService( )

    ' Установить значение свойства URL, заданное пользователем.

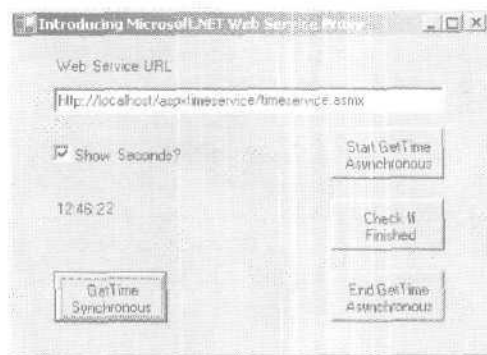
    foo.Url = textBox1( ).Text

    ' Реально вызвать функцию. Поместить результат функции
    ' в текстовое поле, где он будет виден пользователю.

    label1( ).Text = foo. GetTime(checkBox1( ) .Checked)

End Sub
```

**Рис. 4-15.** Программа на Visual Basic, которая вызывает синхронный метод SOAP-прокси.



**Рис. 4-16.** Окно программы-примера, использующей SOAP-прокси.

### Случай 5: асинхронная работа интеллектуального прокси SOAP

Как правило, клиент стремится вызывать Web-службу а асинхронном режиме.

Интернет может быть очень даже забавным, но он всегда так густо населен, что нет никакой надежды насладиться его прелестями в отсутствие разумной стратегии, позволяющей справиться с его хронической перегруженностью — в этом он напоминает Disney World. Обработка элементарного вызова Web-службы может занять 5-10 секунд даже при удачном стечении обстоятельств, так что полученное время станет неточным. В таком случае нельзя блокировать пользовательский интерфейс дольше, чем на пару секунд. Приложение промышленного масштаба не может просто вызывать синхронный метод прокси во время исполнения обрабатывающей нажатие кнопки формы функции, как это сделано в примере выше. Необходимо как-то разместить вызов в другом потоке, который может ждать ответа, не блокируя пользовательский интерфейс. Позже, когда поток вернет результаты вызова, их вывести пользователю.

До сих пор большинству программистов приходилось писать собственную программную инфраструктуру для решения этой проблемы. Код такого рода всегда был печально известен как очень ненадежный. С финансовой точки зрения, потраченное на него время можно занести в чистые убытки. Поскольку такое положе-



ние дел стало универсальным, классы «интеллектуальный прокси» теперь оснащаются готовым кодом, позволяющим решить эту проблему. Вместо того чтобы блокироваться

в ожидании завершения вызова, программа может вызвать метод, который передает данные запроса на сервер и сразу возвращает управление. Позже, в более удобное время, программа может вызвать другой метод и получить результаты с сервера. Это очень, очень облегчает жизнь, так как освобождает от писания кода для распределения процессорного времени. Кроме того, этот метод работает с любыми серверами, а не только с теми, что поддерживают асинхронный доступ.

Ниже показана асинхронная версия вызова Web-службы (рис. 4-17). Прокси содержит метод *Begin[имя\_метода]*, в данном случае *BeginGetTime*. В начале списка параметров стоит параметр, который есть у синхронного метода (у меня это булева переменная *ShowSeconds*).

В случае обратного вызова используются еще два параметра, но этот случай выходит за рамки этой книги. Следующая программа (рис. 4-18) производит асинхронный вызов, при этом для обоих параметров передается *Nothing* (эквивалент *null* из Visual Basic). Цепочка взаимодействия начинается с вызова метода *BeginGetTime*, который отправляет запрос и сразу же возвращает управление. При этом возвращается объект типа *IAsyncResult*, который позже понадобится для получения результата. Теперь мы свободны и можем заняться другими делами. Чтобы имитировать долгую операцию во время лабораторного тестирования, я вставил в страницу *.ASMX* цикл, который просто считает от одного до миллиарда. Если хотите проверить эту программу, посчитайте вместе с ней...

Позже в определенный момент потребуется получить результаты операции и узнать, сколько времени было на самом деле, когда сервер послал результаты обратно. Это делает прокси-метод *End[имя\_метода]* (в этом случае — *EndGetTime*), которому передается

объект *IAsyncResult*, полученный от метода *Begin*. Таким образом, код инфраструктуры может узнать, какой именно результат вер-

Класс-оболочка прокси поддерживает асинхронную версию каждого метода Web-службы.

Следует вызывать тот метод, что возвращает управление сразу после начала запроса.

Теперь нужно вызвать другой метод, чтобы получить результаты после завершения операции.

нуть, если **клиент** сделал несколько необработанных запросов одновременно. Метод *End* получает результат и возвращает его.

```
Public Function BeginGetTime(ByVal ShowSeconds As Boolean, _
    ByVal callback As System.AsyncCallback, _
    ByVal asyncState As Object) _
    As System.IAsyncResult
    Return Me.BeginInvoke("GetTime", New Object( ) _
        {ShowSeconds}, callback, asyncState)
End Function

Public Function EndGetTime(ByVal asyncResult As _
    System.IAsyncResult) As String
    Dim results( ) As Object = Me.EndInvoke(asyncResult)
    Return CType(results(0), String)
End Function
```

**Рис. 4-17.** Асинхронные методы SOAP-прокси.

```
' Начать асинхронный вызов, чтобы получить время

Dim AsyncTimeServiceProxy As TimeService
Dim AsyncResult As IAsyncResult

Private Sub button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles button2.Click
    AsyncTimeServiceProxy = New TimeService( )
    AsyncTimeServiceProxy.Url = textBox1( ).Text
    AsyncResult = AsyncTimeServiceProxy.BeginGetTime _
        (checkBox1( ).Checked, Nothing, Nothing)
End Sub

' Проверить, завершилась ли операция.

Private Sub button3_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles button3.Click
    If (AsyncResult.IsCompleted = True) Then
        MessageBox.Show("Is complete")
    Else
        MessageBox.Show("NOT complete")
    End If
End Sub

' Определить время, полученное в результате асинхронного вызова.
```

```

Private Sub button4_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
    Handles button4.Click
    Label1( ).Text = _
        AsyncTimeServiceProxy.EndGetTime(AsyncResult)
    AsyncResult = Nothing
    AsyncTimeServiceProxy = Nothing
End Sub

```

**Рис. 4-18.** Программа на VisualBasic, вызывающая асинхронные методы SOAP-прокси.

Но как узнать, завершена ли операция и готовы ли результаты? Для этого у объекта *AsyncResult* есть метод *IsCompleted*, возвращающий TRUE или FALSE. Чтобы выяснить, завершилась ли операция, можно периодически опрашивать этот метод. Если вызвать *EndGetTime* до завершения операции, он будет заблокирован до того момента, пока операция реально не закончится. Поскольку скорее всего вы не станете вызывать этот метод в главном потоке пользовательского интерфейса, имеет смысл делать это в рабочем потоке, который не сможет продолжить обработку своих данных без результатов вызова Web-службы.

Опрос свойства позволяет выяснить, завершена ли операция.

## Поддержка Web-служб в Visual Studio.NET

Показанный выше пример с Web-службой продемонстрировал, что для написания Web-службы среда разработки не нужна. Но поскольку сроки — это второе по значимости ограничение при разработке ПО (если первым считать тупоголовых менеджеров, которые даже не знают, с какого конца брать паяльник), имеет смысл писать Web-службы с помощью инструментария, который позволил бы «выдавать» их быстрее. Поскольку у нашего старого приятеля Блокнота нет таких возможностей, как встроенный отладчик, лучше писать Web-службы в Visual Studio.NET. Я покажу, как написать аналогичную службу с помощью бета-выпуска 2 Visual Studio.NET.

Если в главном меню Visual Studio выбрать File/New/Project, выводится диалоговое окно (рис. 4-19). В левой панели этого окна я выбрал Visual Basic Projects, а в правой — значок Web Service, ввел в нужную строку имя проекта и щелкнул OK. В результате



Интересно писать программу, которая хранится в файле `Service1.aspx`, где на самом деле «живет» код, реализующий Web-службу (рис. 4-21). Такие функции среды разработки, как автозавершение, заметно облегчают написание программ в *Visual Studio.NET* по сравнению с Блокнотом.

Шаблон проекта Web Service генерирует остальные файлы (рис. 4-20). `Web.config` — это файл с данными XML, содержащий сведения о параметрах конфигурации, которые сообщают ASP.NET, как работать с Web-службой в период выполнения. Этот файл мы уже видели в главе 3. В частности, он содержит значение таймута сеанса ASP.NET. В настоящее время этот файл придется редактировать вручную, но для этого определенно требуется хорошая программа конфигурирования, возможно на основе MMC (Microsoft Management Console). Надеюсь, что такая программа будет создана перед окончательным выпуском этого ПО.

`WebService1.vsdisco` — это XML-файл данных, который служит для управления динамического обнаружения Web-службы клиентами. Элементы конфигурации по умолчанию, которые можно видеть в загруженном коде, просто указывают ASP.NET исключить из сгенерированного WSDL-файла некоторые подкаталоги (созданные в процессе опубликования). На данном этапе разработки проекта этот файл пока еще слабо документирован.

`.asmx`-файл — это целевой файл для запросов клиентов. В отличие от предыдущего примера, когда в этом файле «жил» код, в нашем примере здесь стоит ссылка. Атрибут `Codebehind` указывает клиенту, где находится код, реализующий службу:

```
<%@WebService Language="vb" Codebehind="Service1.aspx.vb"
    Class="WebService1.Service1"%>
```

Функции обработки событий уровня проекта (такие как `Application_Start` и `Application_End`), «живут» в файле `Global.asax`. В эти функции следует поместить собственный код, который будет вызван ASP.NET для обработки указанных событий. Файл `Global.asax` соединяет ASP.NET и остальные файлы.

При сборке *Visual Studio.NET* автоматически публикует проект на заданном сервере (в этом случае заданном по умолчанию). Чтобы протестировать Web-службу, просто запустите отладчик из

главного меню Visual Studio.NET. Он выводит проверочную страницу, похожую на показанную выше (рис. 4-5), с помощью которой очень удобно решать эту задачу. Также можно устанавливать контрольные точки и всячески отлаживать Web-службу. Таким образом, разработка приложений проходит гладко, быстро и даже, не побоюсь этого слова, весело,

```
Imports System.Web.Services

Public Class Service1
    Inherits System.Web.Services.WebService

    # Region " Web Services Designer Generated Code "

    'Required by the WebServices Designer.
    Private components As System.ComponentModel.Container

    Public Sub New( )
        MyBase.New()

        'CODEGEN: This procedure is required by
        ' the WebServices Designer.
        'Do not modify it using the code editor.
        InitializeComponent()

        'После вызова InitializeComponent добавьте
        'собственный инициализационный код.
    End Sub

    Private Sub InitializeComponent( )
        'CODEGEN: This procedure is required by
        'the WebServices Designer.
        'Do not modify it using the code editor.
        components = New System.ComponentModel.Container( )
    End Sub

    Overrides Sub Dispose( )
        'CODEGEN: This procedure is required by
        'the WebServices Designer.
        'Do not modify it using the code editor.
    End Sub

#End Region
```

```

<WebMethod( )> Public Function GetTime(ByVal ShowSeconds As Boolean) As String
    If (ShowSeconds = True) Then
        Return Now( ).ToLongTimeString
    Else
        Return Now( ).ToShortTimeString
    End If
End Function

End Class

```

**Рис. 4-21.** Код Web-службы из файла *Service1.asmx*.

## Управление состоянием Web-службы

В объектах Web-служб в их естественном состоянии (здесь следует тяжкий вздох) состояние попросту отсутствует. Это означает, что для обработки каждого входящего запроса ASP.NET создает новый экземпляр объекта и уничтожает его по завершении обработки вызова. Результаты предыдущего вызова недоступны последующему, если только не отступить от этого правила и не сделать их доступными. Эта ситуация в некотором смысле аналогична активизации по требованию в COM+.

Иногда это то, что надо, иногда — нет. В случае показанной в этой главе службы такая ситуация всех устраивает. Будет или не будет текущий клиент показывать секунды — от этого никак не зависят результаты, которые получит следующий клиент. Каждый вызов функции самодостаточен, у любого вызова нет причин помнить хоть что-нибудь о предыдущем. Можно возразить, что здесь нет объекта в его классическом определении (совокупность данных и обрабатывающих их команд). Клиент просто осуществляет вызов, который больше ни с чем не связан, в силу чего это не объект, а функция. Можно спорить о семантике, однако все работает. Если такое поведение не годится, можно сохранять состояние объекта между вызовами. Физически экземпляры объекта по-прежнему создаются при необходимом-

По умолчанию Web-службы находятся в неопределенном состоянии.

В ASP.NET Web-службы могут хранить сведения о состоянии в контейнерах состояния на уровне сеанса или приложения.

ти и уничтожаются после, но ASP.NET позволяет сохранить данные, которые обрабатывались объектом в прошлой жизни, подобно тому, как завещание позволяет оставить наследство потомкам. В обсуждении ASP.NET (см. глазу 3) сказано, что существуют два различных типа состояния объектов, которые используются страницами ASP.NET, каждый со своими преимуществами и недостатками. Те же возможности управления состоянием доступны и для объектов Web-служб. В базовом классе *WebService*, производным от которого является наша Web-служба, сведения о состоянии хранятся в двух наборах, один из которых называется *Application*, а другой — *Session*. Можно поместить туда данные, а потом извлечь их, обращаясь к наборам с помощью строк с именами элементов.



## Глава 5

# Windows Forms

*Убавь! Залей! Куда глядишь? Вы слишком много жжете!  
Ты на посудине, а вишь, жжешь как на пакетботе!  
За «думал» — денег не дают! Не думай, а трудись!  
О Господи, попробуй тут разок не чертыхнись!*

Р. Киплинг о трудности обучения системных администраторов и необходимости обеспечения их надежными инструментами.  
(«Молитва МакЭндрю», 1894 г.)

### Суть проблемы

В предыдущих главах рассказывалось о создании серверного кода на платформе Microsoft .NET — безусловно, она очень хорошо подходит для этой цели. Однако какая ОС установлена на всех компьютерах вашего офиса, на домашней рабочей машине и на компьютере, на котором играют ваши дети?

На ноутбуке сидящего рядом пассажира самолета? За исключением нескольких фанатиков Macintosh и группы странноватых любителей Linux, это Microsoft Windows. На мой взгляд, разработчики часто забывают, что число клиентских компьютеров с Windows несравнимо больше числа серверов под управлением данной ОС — возможно, больше, чем в 100 раз. Думаете, пользователи этих компьютеров остановились на Windows из-за ее возможностей по обработке транзакции или развертыванию много-

Рынок программ для Windows является самым большим рынком ПО в мире и останется таковым в ближайшем будущем.

серверных комплексов? Нет, они поступили так из-за удобного пользовательского интерфейса и большого количества настольных приложений для этой ОС. Рынок программ для Windows — самый большой рынок ПО в мире и останется таковым в ближайшем будущем. Мы станем создавать приложения, отличные от разрабатывавшихся 10 лет назад. Программы для работы с электронными таблицами и текстовые процессоры давно заняли свое место под солнцем, однако мы по-прежнему будем программировать для настольных ПК Windows. Например, мы разработаем несколько приложений с богатым пользовательским интерфейсом типа программы Napster для поиска музыкальных записей, упоминавшейся в главе 4.

Разработчики настольных приложений часто сталкиваются с теми же проблемами, что и разработчики серверного ПО.

1 Некоторые архитектурные проблемы, с которыми приходится сталкиваться разработчикам настольного ПО, отличаются от проблем разработчиков серверных приложений. Например, разработчик настольных программ рассматривает на отдельного человека, реагирующего на события гораздо медленнее компью-

тера, и поэтому всегда имеет в своем распоряжении множество циклов процессора и в отличие от разработчика серверного ПО может не волноваться о масштабируемости. С другой стороны, проблемы, встречающиеся тем и другим программистам, как правило, одни и те же. Например, разработчики стараются повторно использовать исходный код, написанный другими. Все они могут работать с кодами разных типов (скажем, с кодом графической анимации и кодом для доступа к БД), но в итоге сталкиваются с экономической целесообразностью использовать результаты труда своих коллег. Всех их волнует контроль версий приложения, абстрагирование от различий в реализациях языков программирования и совместимость с COM. До недавнего времени разработчики настольных программ по большей части игнорировали безопасность, считая, что она нужна лишь серверным приложениям. Но теперь, когда большой объем кода передается по Web, они столкнулись с проблемами аутентификации и авторизации, уже долгое время терзающих их коллег, разработчиков серверных программ. Добро пожаловать в реальный мир.

Разработчикам настольных приложений, равно как и разработчикам серверного ПО, требуются готовые решения этих проблем.

Помимо общих с разработчиками серверных приложений проблем, у создателей настольного ПО имеются и собственные трудности. Вспоминаю, как десять лет назад я писал приложения для Windows на C. Большая часть времени уходила на создание кода, реализующего стандартные элементы пользовательского интерфейса Windows-приложений. Тогда не было готовых строк состояния и панелей инструментов, и поэтому мне приходилось писать собственный код. Точно так же поступал любой, кто желал использовать эти элементы интерфейса в своих приложениях. Я вынужден был создавать собственные обработчики команд, позволявшие получать сообщения от элементов управления. Меня все еще мучают кошмары, когда я вспоминаю, как бездумно пообещал клиенту добавить в приложение функцию предварительного просмотра печатаемых страниц за фиксированную сумму. Экономически (уже имеющийся код пишется повторно, в третий раз, в четвертый и т. д.) и эргономически (непредсказуемое поведение каждой новой панели инструментов сводит пользователей с ума) нецелесообразно, чтобы каждый программист писал собственный код для стандартных элементов пользовательского интерфейса. Нам нужны готовые решения общих проблем разработки интерфейса.

Чтобы обеспечить программистов стандартами многократного использования, различные среды разработки реализуют разнообразные подходы к готовым пользовательским интерфейсам. Visual Basic предоставляет модель программирования, основанную на событиях и формах, которая благодаря простоте использования стала довольно популярной.

В Visual C++ реализованы классы Microsoft Foundation Classes (MFC), предоставляющие готовую функциональность (например, функцию предварительного просмотра, с которой в свое время мне пришлось помучаться) на основе системы наследования C++. «Плясать от кода», как в Visual C++, гораздо труднее, чем работать с основанной на формах парадигмой программирования Visual Basic, но меня привлекают широкие возможности язы-

Большинству приложений с графическим интерфейсом пользователям нужен стандартный набор возможностей, что открывает дорогу готовой функциональности.

.. Разработчик зачастую останавливается на конкретном языке из-за имеющихся в нем готовых элементов пользовательского интерфейса, а не из-за того, что этот язык подходит для решения стоящих перед ним задач.

ка Visual C++. Программистам приходится выбирать между мощным языком и возможностью быстрой разработки пользовательского интерфейса. В Visual Basic я обычно создаю интерфейсные приложения, поскольку мне нравится его редактор форм и удобство подключения элементов управления. Однако слабый язык, в котором синтаксис COM-объектов непонятен, обработка ошибок реализована идиотски, и вызов API-функций для выполнения стандартных задач (работа с реестром и т. д.) затруднен, меня жутко злит. Visual C++ я люблю за мощный язык программирования, но здесь меня доканивает низкий уровень абстракции (например, трудности с обработкой и генерацией событий).

Как всегда, мы (и я не исключение) хотим, чтобы нам оставалось реализовать только бизнес-логику.

: Как и всем остальным людям, разработчикам настольных приложений хочется получить все сразу и не отдавать чего-нибудь взамен. Нам нужны готовые решения общих с разработчиками серверного ПО проблем. Мы хотим быстро создавать пользовательский интерфейс, но нам хочется делать это на мощном языке программирования (в идеале — выбранном нами). Для обеспечения согласованности везде должна применяться единая модель программирования. И чтоб все это было недорого. Что ж, когда мы пишем Санта-Клаусу, попросить у него лошадку ничего не стоит.

## Архитектура решения

Многие функции CLR платформы .NET помогают разработчикам как серверного, так и настольного ПО.

Как вы уже знаете, .NET предоставляет стандартную функциональность для решения общих проблем разработчиков серверных и настольных приложений (подробнее о них см. главу 2). Разработчики настольного ПО любят и станут работать с предоставляемой CLR моделью многократного использования кода, функциями управления версиями и памятью, упрощенным развертыванием закрытых сборок, организацией пространства имен, функциями защи-

ты и совместимостью с COM. Web-службы (см. главу 4) дадут разработчикам настольных приложений возможности и способы подключения к серверам.

В Microsoft .NET разработчикам доступен богатый набор функциональности под удивительно бессмысленным названием *Windows Forms*. Windows Forms предоставляет .NET-классы с готовыми компонентами пользовательского интерфейса, пригодными для большинства настольных приложений. Если вы думаете о гибриде Visual Basic и MFC, который реализован в .NET и благодаря этому доступен любому языку программирования, вы думаете практически в правильном направлении. Windows Forms поддерживает меню, панели инструментов и строки состояния, печать и предварительный просмотр, размещение элементов управления ActiveX, а также упрощенный доступ к БД и Web-службам. Это настолько богатый и обширный набор функциональности, что в данной главе я смогу привести лишь поверхностный его обзор. Отдавая должное Windows Forms, нужно сказать, что описание этой технологии займет целую книгу, и, насколько я знаю, Чарльз Петцольд сейчас пишет именно такую книгу для Microsoft Press (англоязычное издание планируется к выходу осенью 2001 г.).

Windows Forms • это пакет, предоставляющий готовые элементы пользовательского интерфейса в составе CLR.

---

### Внимание

Как и все мощные инструменты, средства разработки пользовательского интерфейса в Windows Forms являются обоюдоострыми. Они позволяют создавать как отличные, так и ужасные интерфейсы. Воина слишком важна, чтобы предоставить ее генералам, а интерфейс слишком важен, чтобы оставить его программистам. Любой, кто занимается разработкой пользовательского интерфейса без изучения специализированной литературы (*About Face*, Alan Cooper, IDG, 1995; *Design of Everyday Things*, Donald Norman, Doubleday, 1990), преступно небрежен. А прочитав книгу о непривлекательных Web-узлах *Web Pages That Suck* (Vincent Flander, Michael Willis; Sybex, 1998) вы, возможно, не станете создавать такой узел.

## Простейший пример

Как всегда, я начал изучать Windows Forms, написав простейший пример, который только смог придумать (рис. 5-1). Исходный код можно загрузить с Web-узла данной книги (<http://www.introducingmicrosoft.net>) и работать вместе со мной. В Visual Studio.NET очень удобно создавать проекты такого типа, но чтобы подчеркнуть языковую и инструментальную независимость Windows

Пример приложения Windows Forms, написанного на Visual Basic в Блокноте.

Forms, я написал пример на Visual Basic в Блокноте. Хотя приложение весьма тривиально и включает немного кода, оно все же демонстрирует часть самых основных и важных функций Windows Forms.

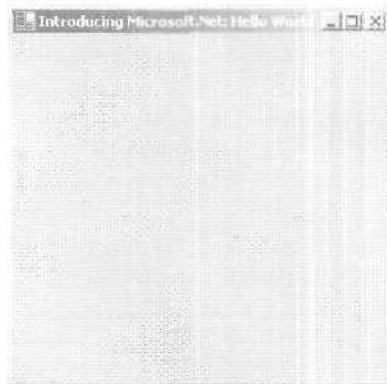


Рис. 5-1. Пример приложения Windows Forms.

В данном примере не импортируются какие-либо пространства имен.

Ниже приведен исходный код моей программы (рис. 5-2). Заметьте: в данном примере я не импортировал пространств имен (подробнее о них см. главу 2). Без нужды их можно не импортировать — это делается, лишь что-

бы с уверенностью вызывать объекты по коротким именам вместо полных. Я решил поступить так, чтобы вы четко видели, потомком какого класса является каждый объект. При компиляции вам придется включить некоторые системные DLL по ссылке, которая имеется в пакетном файле компиляции, прилагаемом к загруженному вами исходному коду. При работе с проектом Win-

dows Forms в Visual Studio .NET это автоматически сделает за вас сама среда разработки.

```
Namespace IMDN.SimplestHelloWorld

' Объявляем новый класс - потомок базового класса CLR
' под названием System.Windows.Forms.Form.

Public Class SimplestHelloWorld : Inherits _
    System.Windows.Forms.Form

    ' Конструктор класса. Передаем вызов базовому классу
    ' для его инициализации и затем задаем
    ' заголовок нашего окна.

    Public Sub New( )
        MyBase.New
        Me.Text = "Introducing Microsoft.Net: Hello World"
    End Sub

    ' Данная функция является точкой входа приложения
    ' Windows Forms. Создаем новый экземпляр объекта нашей
    ' формы и передаем его системной функции,
    ' запускающей приложение.

    Shared Sub Main( )
        System.Windows.Forms.Application.Run(New _
            SimplestHelloWorld( ))
    End Sub

End Class

End Namespace
```

**Рис. 5-2.** Листинг простейшего примера,

В Windows Forms окно верхнего уровня называется *формой* (form); у каждого приложения должно быть *хоть* одно такое окно. Программистам на Visual Basic это требование знакомо. Программисты на C++ могут считать форму основным окном своего приложения. В Windows Forms функциональность, необходимая каждому окну верхнего уровня, реализована в базовом классе CLR по имени *System.Windows.Forms.Form*. Этот класс позволяет размещать элементы управления, поддерживает закрепление дочернего окна, реагирует на события и т. д.

Класс нашей формы является производным от базового класса CLR, предоставляющего основную готовую функциональность формы.

Я начал разработку приложения с создания нового класса *SimplestHelloWorld*, происходящего от *System.Windows.Forms.Form*. Сообщая о наследовании, я указываю компилятору получить всю функциональность базового класса, разработанную Microsoft, и включить ее в мой производный класс (подробнее о наследовании см. главу 2).

Мы подменяем конструктор класса, чтобы добавить новую функциональность при создании формы.

: Теперь нужно дополнить класс, представляющий окно верхнего уровня, кодом, определяющим отличия нашего класса от базового. В данном случае мы подменяем метод *New* (который является конструктором; см. главу 2) базового класса, вызываемый при каж-

дом создании экземпляра нашего объекта. В этом методе мы сначала вызываем конструктор базового класса, тем самым позволяя последнему завершить инициализацию перед выполнением нами каких-либо действий в производном классе. В ООП невыполнение этого — распространенная причина ошибок. Архитектура .NET не вызывает конструктор базового класса автоматически, поскольку иногда вам требуется пропустить этот вызов или осуществить другой (например, если вы целиком заменяете часть функциональности базового класса, вместо того чтобы тупо на него полагаться, как мы делаем здесь). В базового класса есть свойство *Text*, содержащее строку в заголовке формы. Мы устанавливаем в этом свойстве недвусмысленный текст.

Мы создаем экземпляр нашего объекта «форма» и указываем загрузчику запустить его.

Теперь нам нужно подключить форму верхнего уровня, чтобы указать системному загрузчику окно, которое следует вывести при запуске приложения. Если бы мы работали с Visual Studio .NET, среда разработки автома-

тически добавила бы весь необходимый код. Но ведь мы работаем в Блокноте, и код придется добавить самостоятельно. При запуске программы загрузчик ищет функцию *Main*, которую я добавил в наш объект. Эта функция может присутствовать лишь в начальной форме приложения, и загрузчик вызывает ее лишь раз. Спецификатор *Shared* (для фанатов C++ — *static*) означает, что все объекты класса используют лишь один экземпляр *Main*. За-

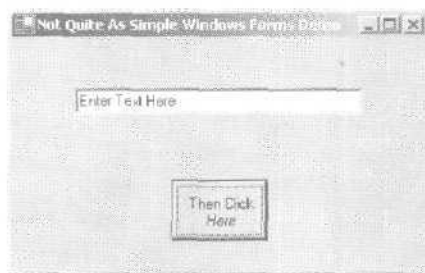


грузчик CLR вызывает *Main* для запуска приложения. Вы можете дополнить ее любым кодом. Приложение Windows Forms требует, чтобы один из потоков принимал от ОС сообщения пользовательского интерфейса (например, нажатия клавиш и щелчки мыши) и передавал их корректным обработчикам событий. Статическая функция CLR — *System.Windows.Forms.Application.Run* — создает и запускает именно такой поток. Передав ей новый экземпляр нашего объекта «форма», мы указываем, какой форме передавать сообщения.

### Более сложный пример: события и элементы управления

Наше простейшее приложение демонстрирует несколько необходимых функций, но и только. На форме нет даже кнопки ОК. Теперь перейдем к менее тривиальному приложению (рис. 5-3). Хотя это и не требовалось, здесь я работал в Visual Studio.NET, так как встроенный редактор данной среды разработки значительно упрощает взаимодействие с элементами управления.

Пример более сложного приложения Windows Forms.



**Рис. 5-3.** Более сложная форма Windows Forms, созданная в Visual Studio .NET.

Вы увидите, что модель программирования Windows Forms напоминает модель программирования Visual Basic 6.0, хотя и доступна в любом языке CLR. Это хорошая идея, поскольку модель программирования Visual Basic 6.0 была и остается весьма популярной. Многие программисты, включая меня, не выносят скрывающегося за этой моделью языка, и теперь мы можем использо-

вать любой CLR-совместимый язык по собственному выбору. Кроме того, сторонники C++ выиграли большинство схваток по поводу модели ООП, обсуждавшейся в главе 2. Оптимист скажет, что Microsoft смешала лучшие возможности двух принципов проектирования. Пессимист станет уверять, что программистам на Visual Basic следует дать утешительный приз.

На формах обычно имеются взаимодействующие с пользователем элементы управления, которые получают и выводят ему данные. Как и Visual Basic 6.0, пакет Windows Forms среды CLR включает массу элементов управления. Создавая это приложение, я перетаскивал элементы управления с панели инструментов Visual Studio .NET на форму (рис. 5-4). При этом Visual Studio .NET генерировала в коде формы обращения к CLR, которые создают элементы управления, задают их свойства и размещают их в нужном месте при начальном отображении формы. Вот сокращенная версия одного из таких методов (рис. 5-5).

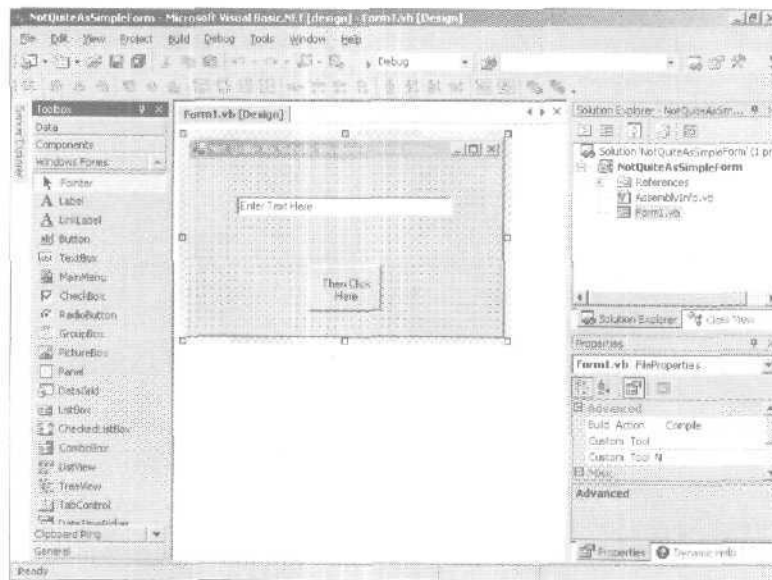


Рис. 5-4. Панель инструментов Visual Studio .NET.

```

Private WithEvents Button1 As System.Windows.Forms.Button
Private WithEvents TextBox1 As System.Windows.Forms.TextBox

' Примечание: следующая процедура необходима
' Windows Forms Designer, и ее можно изменить
' с его помощью. Модифицировать процедуру
' с помощью редактора кода не следует.

Private Sub InitializeComponent( )

    ' Создаем элементы управления

    Me.Button1 = New System.Windows.Forms.Button()
    Me.TextBox1 = New System.Windows.Forms.TextBox()

    ' Задаем свойства кнопки

    Me.Button1.Location = New System.Drawing.Point(112, 96)
    Me.Button1.Size = New System.Drawing.Size(75, 40)
    Me.Button1.TabIndex = 1
    Me.Button1.Text = "Then Click Here"

    ' Задаем свойства текстового поля

    Me.TextBox1.Location = New System.Drawing.Point(48, 48)
    Me.TextBox1.Text = "Enter Text Here"
    Me.TextBox1.TabIndex = 0
    Me.TextBox1.Size = New System.Drawing.Size(200, 20)

    ' Задаем свойства формы.

    Me.Text = "Not Quite As Simple Windows Forms Demo"
    Me.Controls.AddRange(New System.Windows.Forms.Control() _
        {Me.TextBox1, Me.Button1})

End Sub

```

**Рис. 5-5.** Сокращенная версия метода *InitializeComponent*, демонстрирующая создание элемента управления.

Данный код размещается в закрытом методе *InitializeComponent*, вызываемом из конструктора формы. В Visual Basic .NET элементы управления на форме создает код, кото-

Элементы управления  
на Форме создаются  
функциями CLR.

рый можно просмотреть, а не некая невидимая рука за кулисами, как это было в Visual Basic 6.0. Кроме того, новые элементы управления добавляются в набор *Controls*, который форма унаследовала от базового класса. Он позволяет форме следить за всеми ее элементами управления.

Элементы управления запускают события, перехватываемые обработчиками в вашем коде.

1 При взаимодействии пользователя с элементами управления последние генерируют события, передаваемые своим контейнерам. Так, элемент управления «кнопка» генерирует событие, сообщаящее о ее нажатии. Нам потребуется написать функции обработки событий, вызываемые, когда элемент управления извещает о событии. Для этого в класс формы добавляется метод *<Имя\_элемента\_управления>\_<Имя\_события>*; здесь — метод *Button1\_Click* (рис. 5-6). При щелчке кнопки Visual Studio .NET добавляет нужный код автоматически; любой же человек, работающий с Блокнотом, может добавить метод *Button1\_Click* в код класса формы вручную. Когда элемент управления запустит событие, механизм события CLR ищет функцию обработки, получив от элемента точное название события, и, обнаружив подходящую функцию, запускает ее. Кроме того, функция Visual Basic под названием *AddHandler* позволяет вызывать обработчик события динамически, в процессе выполнения кода.

\* Пользователь щелкнул кнопку. Выводим сообщение  
\* с текущим содержимым текстового поля.

```
Private Sub Button1_Click(ByVal sender As Object, _
                        ByVal e As System.EventArgs)
    MessageBox.Show("You entered: " + textBox().Text)
End Sub
```

**Рис. 5-6.** Обработчик события «щелчок кнопки».

Формы поддерживают метод *Dispose* для детерминированного завершения.

1 Формы поддерживают детерминированное завершение (о нем см. главу 2, раздел, посвященный сборке мусора). Метод *Dispose* базового класса *System.Windows.Forms.Form* позволяет клиенту сразу ликвидировать форму и освободить занимаемые ей ресурсы, не ожидая и не вызывая полную сборку мусора. Это особенно важно в Windows Forms, так

как у каждой формы есть используемые ОС описатели окна для всех присутствующих на форме элементов управления и еще один описатель для самой формы. В Windows 95/98 (не заводите меня) ощущается дефицит таких описателей окна, а детерминированное завершение позволяет клиенту сразу по завершении работы с формой освободить занимаемые ей ресурсы. Изучив исходный код, вы увидите, что Visual Studio .NET автоматически переопределяет данный метод, добавляя во все элементы управления на форме код для вызова метода *Dispose*.

## Создание собственных элементов управления Windows Forms

Для разработки хороших приложений Windows Forms нужны хорошие элементы управления. CLR предоставляет множество неплохих и полезных элементов управления, однако этот набор — лишь малая толика дей-

ствий, которые, возможно, захотят выполнять пользователи. Нам бы действительно хотелось создавать собственные элементы управления Windows — как в целях безопасности, так и для дальнейшего использования. И мы хотели бы достичь такого уровня развития рынка, при котором сторонние поставщики стали бы разрабатывать элементы управления, а мы — покупать их. Прежде всего для этого потребуется, чтобы в CLR была хорошая встроенная поддержка написания новых элементов управления.

Идея распространения готовых элементов пользовательского интерфейса (с методами, свойствами и событиями) в удобном пакете, подключаемом к интеллектуальной среде для быстрой разработки программ, оказалось очень удачной на рынке ПО. Все началось с

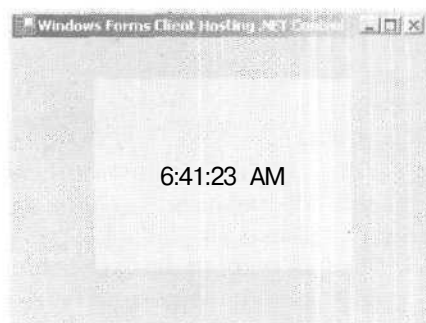
16-разрядных элементов управления VBX и затем, когда Windows стала 32-разрядной, перешло на 32-разрядные элементы управления ActiveX. Страницы *MSDN Magazine* и *Visual Basic Programmer's Journal* переполнены рекламой элементов управления ActiveX, реализующих всяческую функциональность — от электронных таблиц и программ проверки орфографии до ленточных самописцев и электрокардиограмм. Элементы управления раз-

Нам необходима возможность создавать собственные элементы управления .NET.

Рынок готовых элементов управления сторонних производителей растет фантастическими темпами,

множились, как кролики в Австралии, конкурирующие с сумчатыми. Если платформа .NET получит признание, она будет обеспечивать дальнейшее процветание рынка ПО.

Технология Windows Forms поддерживает создание пользовательских элементов управления Windows Forms. Саму тему поддержки в этой книге раскрыть невозможно, однако я написал небольшое приложение, показывающее, насколько просто создавать собственные элементы управления. Ниже показан клиент Windows Forms, содержащий элемент управления (рис. 5-7) и его код (рис. 5-8). Кое-кто, возможно, захочет раздуть эту тему до размеров целой книги, добавив побольше листингов и снимков экрана (молчу!).



**Рис. 5-7.** Элемент управления *WindowsForms* в клиенте *Windows Forms*.

```
Public Class UserControl1
    ' Элемент управления наследует свойства базового класса CLR
    ' с именем UserControl,
    Inherits System.Windows.Forms.UserControl
    ' У элемента управления есть открытое свойство ShowSeconds.
    Private myShowSeconds As Boolean

    Public Property ShowSeconds() As Boolean
        Get
            Return myShowSeconds
        End Get
    End Property
End Class
```

```

        Set
            myShowSeconds = Value
        End Set
    End Property

    ' Переопределяем обработчик OnPaint для использования
    ' собственного алгоритма,

    Protected Overrides Sub OnPaint(ByVal e As _
        System.Windows.Forms.PaintEventArgs)
        <код пропущен>
    End Sub
End Class

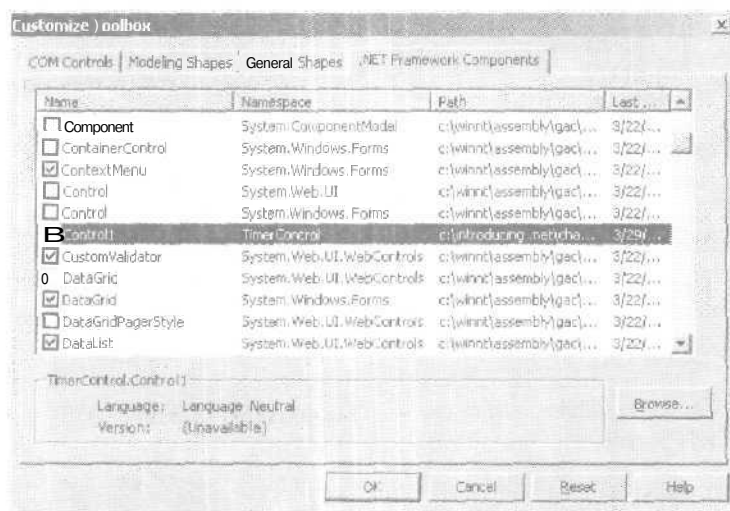
```

**Рис. 5-8.** Сокращенный листинг кода элемента управления Windows forms.

Чтобы написать элемент управления Windows Forms, создайте класс на основе базового класса *System.Windows.Forms.UserControl*. Для этого просто откройте в Visual Studio .NET проект Windows Control Library. Программистам на C++ , создававшим элементы управления ActiveX на основе базового класса MFC под именем *COleControl*, такой подход покажется знакомым. Данный базовый класс содержит общую для всех элементов Windows Forms функциональность, начиная с простых свойств типа цвета фона и кончая сложными взаимодействиями с контейнером, нужными для перемещения и стыковки. Ваш элемент управления переопределяет обработчики событий Windows Forms, поведение которых требуется изменить. Здесь это обработчик уведомления *OnPaint*, в котором я создал вызовы CLR, закрашивающие прямоугольник моего элемента управления фоновым цветом и отображающие текущее время с использованием шрифта по умолчанию элемента управления. Делать что-либо еще для создания данного элемента управления не потребовалось. Чтобы добавить элемент управления в панель инструментов Visual Studio .NET, щелкните панель правой кнопкой, выберите в контекстном меню команду *Customize Toolbox* и затем выберите нужный элемент управления (рис. 5-9).

Здесь начинается пример с элементом управления Windows Forms.

Создание элемента управления Windows Forms — это лишь простое наследование функциональности готового базового класса CLR.



**Рис. 5-9.** Добавление элемента управления Windows Forms в панель инструментов VisualStudio .NET.

Элемент управления .NET может использоваться там же, где элемент управления ActiveX, например, .. в Visual Basic 6.0.

Но если я создам элемент управления Windows Forms, не будет ли это означать, что с ним смогут работать лишь приложения платформы .NET? Таких приложений пока мало, и не подождать ли мне вкладывать свои денежки? Чтобы убедить вас в обратном, скажу, что базовый класс *UserControl* содержит всю

функциональность, которая позволяет ему быть доступным из любых контейнеров элементов управления ActiveX (Visual Basic 6.0 и др.). Это позволяет разработчику элемента управления .NET задействовать преимущества огромной существующей базы и зарабатывать много денег.

Чтобы элемент управления .NET выглядел, как ActiveX-элемент, нужно написать один относительно небольшой фрагмент кода.

Чтобы сделать элемент управления .NET доступным контейнерам ActiveX, нужно написать один относительно небольшой фрагмент кода. В отличие от стандартных COM-серверов, элементы управления ActiveX добавляют в реестр несколько записей, и поэтому вам потребуется реализовать аналогичную функ-



циональность в элементе управления .NET. CLR включает готовые функции для добавления и удаления таких записей — *Control.ActiveXRegister* и *Control.ActiveXUnregister* соответственно. В классе элемента управления должно быть две функции с атрибутами, указывающими утилите регистрации COM-серверов платформы .NET вызвать их в процессе регистрации.

Эти функции должны передавать управление функциям *Control.ActiveXRegister* и *Control.ActiveXUnregister* (рис. 5-10). Это единственный дополнительный фрагмент кода, который вам потребуется написать сегодня, и я не удивлюсь, если в будущем он переместится в базовый класс.

```
Public Shared Sub _
    <System.Runtime.InteropServices.ComRegisterFunction()>
    AxRegister(ByVal regKey As String)

    Dim foo As New UserControl1()
    ActiveXRegister(foo.GetType)
End Sub

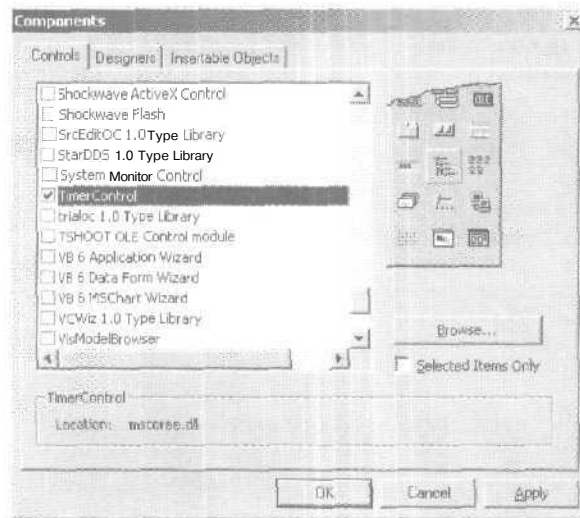
Public Shared Sub _
    <System.Runtime.InteropServices.ComUnregisterFunction()>
    AxUnregister(ByVal regkey As String)

    Dim foo As New UserControl1()
    ActiveXUnregister(foo.GetType)
End Sub
```

**Рис. 5-Ю.** Вспомогательные функции регистрации в элементе управления *WindowsForms*. Они необходимы, только если вам требуется, чтобы элемент был доступен контейнерам *ActiveX* как элемент управления *ActiveX*.

После этого для использования элемента управления .NET в качестве элемента управления *ActiveX* его потребуется лишь зарегистрировать — точно так же, как и любой другой класс .NET, который должен быть COM-сервером (см. главу 2). Атрибут *ComVisible* элемента управления .NET должен быть установлен в *True*; в противном случае COM-клиенты элемент не увидят. На клиентском компьютере должна быть установлена CLR. Создайте библиотеку типов и зарегистрируйте ее с помощью утилиты *RegAsm.exe*. Затем поместите *DLL-файл* элемента управления

туда, где его увидит клиентское приложение; в нашем случае Visual Basic — в каталоги CAC или VB98, в которых располагаются другие двоичные файлы. Ваш элемент управления .NET будет отображаться в списке доступных VB-компонентов (рис. 5-11). Его можно будет установить и запустить, как в прилагающемся к главе примере.



**Рис. 5-11.** Элемент управления .NET в списке компонентов, доступных Visual Basic,

## Размещение элементов управления ActiveX в приложениях Windows Forms

В приложениях Windows Forms можно размещать элементы управления ActiveX.

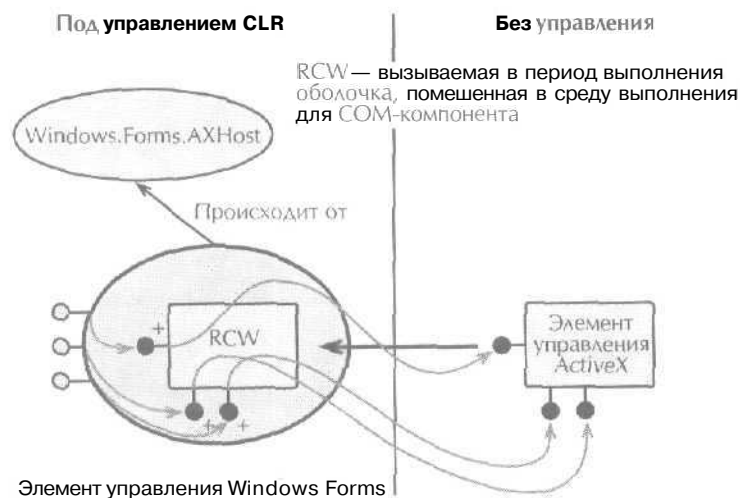
Я уже писал об огромной популярности элементов управления ActiveX. Если бы .NET не позволяла их использовать, многие программисты просто не стали бы работать с этой платформой, т. е. ActiveX-разработчики ли-

шились бы довольно большого рынка, для которого стоило переделывать элементы управления, и вся концепция оказалась бы мертворожденной. Именно поэтому авторы Windows Forms мудро решили реализовать поддержку элементов управления ActiveX.

Функциональность таких элементов основана на COM, и всем, кого интересует данная тема, я рекомендую изучить раздел главы 2, посвященный взаимодействию .NET и COM.

Приложение Windows Forms не знает, как использовать элементы управления ActiveX. Оно понимает лишь элементы управления, созданные в соответствии с архитектурой Windows Forms. Чтобы разместить элемент управления ActiveX в приложении Windows Forms, создайте класс-оболочку, который будет содержать собственно элемент, посредничать между его COM-парадигмой и .NET-парадигмой контейнера, представлять элемент управления как «родной» элемент Windows Forms. Вам действительно нужна мощная оболочка (см. главу 2), которая будет использовать все реализуемые элементом управления ActiveX COM-интерфейсы и предоставлять ему интерфейсы, требуемые им от сервера (рис. 5-12). Если вы думаете, что придется выполнить много работы, вы правы. Но не волнуйтесь — класс CLR по имени *System.Windows.Forms.AxHost* сделает всю работу за вас.

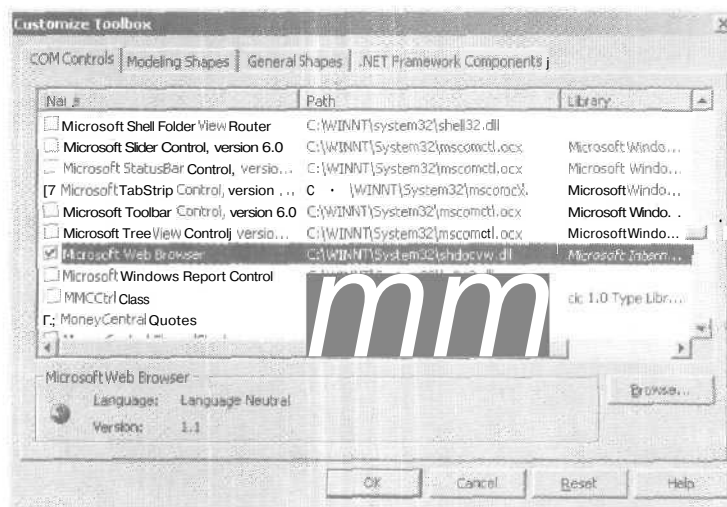
Для размещения элемента управления ActiveX приложение Windows Forms создает класс-оболочку, аналогичный оболочке, вызываемой в период выполнения и заключающей в себе COM-объект.



**Рис. 5-12.** Архитектура Windows Forms для размещения элементов управления ActiveX.

Класс-оболочка создается в Visual Studio.NET или при помощи утилиты командной строки.

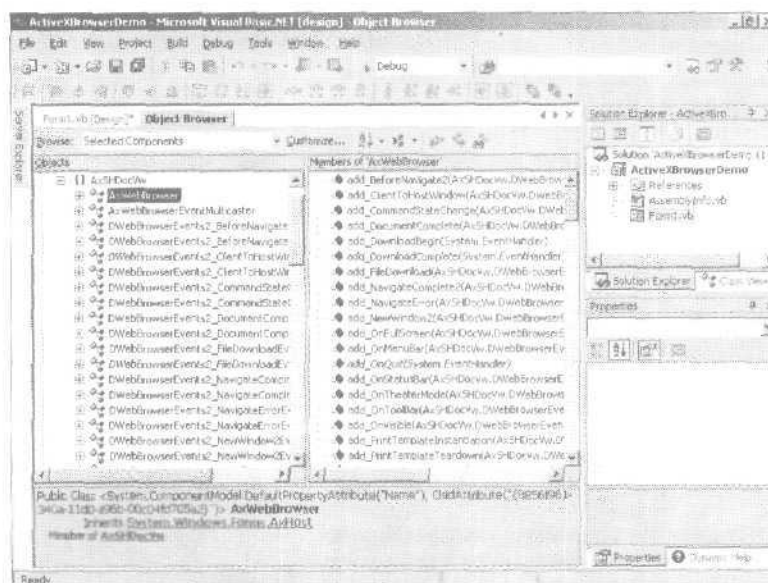
Для каждого класса элемента управления ActiveX, размещаемого в приложении, надо создать отдельный класс-оболочку на основе класса *AxHost*. Такой класс будет включать идентификатор класса или программы, использовавшейся для создания элемента управления ActiveX, и будет представлять свойства, методы и события внутреннего элемента ActiveX в родном формате платформы .NET. Это покажется знакомым всем, кто когда-либо импортировал элемент управления ActiveX в Visual C++ или Visual J++. Классы-оболочки создает утилита командной строки *AxImp.exe*, поставляемая с .NET SDK. При работе в Visual Studio .NET можно щелкнуть панель инструментов правой кнопкой и выбрать в контекстном меню команду *Customize Toolbox*. Откроется диалоговое окно (рис. 5-13) со списком компонентов, которые Microsoft называет *COM Controls* («Элементы управления COM»). (Прощай, «ActiveX», и скатертью дорога! Одним MINFU в мире меньше.)



**Рис. 5-13.** Диалоговое окно со списком доступных для импорта в проект элементов управления ActiveX.

После того, как вы выберете из списка элемент управления, среда Visual Studio .NET сама запускает приложение *AxImp.exe* и ге-

нерирует класс-оболочку. Класс встраивается в относящуюся к проекту отдельную DLL-библиотеку. Просмотреть исходный код напрямую нельзя, однако вы можете изучить его свойства и методы в Object Browser (рис. 5-14). Новый элемент управления отобразится на панели инструментов, и с ним можно будет работать, как обычно.



**Рис. 5-14.** Object Browser, отображающий методы и свойства класса-оболочки, созданного для ActiveX-элемента управления Web Browser.

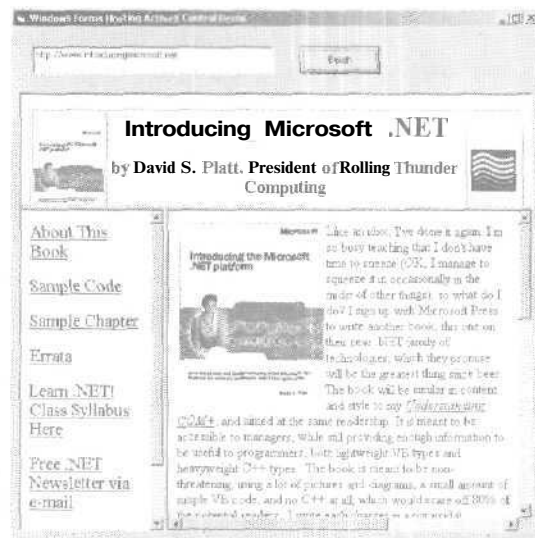
Я написал небольшую программу Windows Forms, использующую ActiveX-элемент управления Microsoft Web Browser (рис. 5-15).

В соответствии с приведенными инструкциями я импортировал элемент управления ActiveX в Visual Studio .NET. Затем я поместил элемент на форму и написал код (рис. 5-16).

При щелчке пользователем кнопки Fetch вызывается метод *Navigate* класса-оболочки, которому передается введенный пользователем URL. Класс-оболочка преобразует дан-

Вот ое приложе-  
ние демонстрирующее  
использование ActiveX-  
элемента с Windows  
Forms.

ный вызов в COM-вызов и передает его внутреннему элементу управления ActiveX.



**Рис. 5-15.** Пример приложения *Windows Forms*, использующего ActiveX-элемент управления *Web Browser*.

```
' Вызываем метод нашего класса-оболочки, как если бы это был
' родной элемент управления Windows Forms.
Protected Sub Button1_Click(ByVal sender As Object,
                             ByVal e As System.EventArgs)
    AxWebBrowser1.Navigate(TextBox1.Text, 0, "", "", "")
End Sub
```

**Рис. 5-16.** Пример кода ActiveX-элемента управления *Windows Forms*.

# Эпилог и благословение

*И Сборщик к делу преступил. Но сборщик сам — Творец!  
Механик и мастеровой, порой — простой трудяга,  
Он соберет, о Боже, твой Ковчег Добра и Блага.  
Не мне судить, хорош иль плох он будет на плаву,  
Но я — хвала Тебе — тружусь. Хвала Тебе — живу!  
Судить положено не мне. Судить — твоя затея.  
Судить, прощать... Эй вы, не спать! На задний ход скорее!*

Р. Киплинг о неотвратимости технического прогресса.  
(«Молитва МакЭндрю», 1894 г.)

Как и МакЭндрю, я сделал то, что должен был сделать: написал эту книгу. И теперь вам судить о ней. Является ли .NET совершенной ОС? Не глупите. Совершенна ли эта книга о ней? Не скажите еще большую глупость. Нет в мире совершенства. Но .NET позволит вам заработать денег больше, чем что-либо другое, и я надеюсь, эта книга помогла вам понять как.

Моя дочь, родившаяся в этом году, принадлежит к первому поколению, узнающему Интернет с колыбели, в отличие от нас с вами. Ее старшие братья и сестры принадлежат к первому поколению, выросшему с настольными ПК, ее родители — к первому поколению с телевидением, ее бабушки и дедушки — первое поколение, услышавшее радио. Вы можете сказать, как это повлияет на нее? Конечно, нет, и я тоже — никто не может. Вернее, множество людей имеет взаимоисключающие мнения на этот счет, но никому не известно, кто прав. Я пишу эти строки в 2001 году и вспоминаю вступление Артура Кларка к его книге «2001: космическая одиссея»: «Важно помнить, что это — лишь плод

фантазии. Действительность, как всегда, будет куда удивительней». Он один из тех парней, кому я верю.

Мы, разработчики программ, стали гораздо ответственней. Как недавно написал в факультетском журнале наш декан машиностроительного факультета, сейчас благосостояние и безопасность нашей нации гораздо больше зависят от битов и байтов, чем от пуль и золотых слитков. Если вылетит Пасьянс, кого это будет волновать, кроме игрока, который потерял записи о своих рекордах? Другое дело, если упадет система резервирования авиабилетов, и уж совсем другое, если больница потеряет истории болезней всех своих пациентов. Киплинг писал о пассажирах МакЭндрю:

*Я — Твой слуга. Их воля — плыть, хоть в раб, хоть в бездну ада, —  
Да ведь не мне людей судить; перевозить их надо.*

*Моя вина — но лишь одна — не будет прощена:  
Когда шесть тысяч здешних тонн поглотит глубина.*

Переварите это хорошенько.

Собратья! Наш вид делает следующий шаг на пути эволюции — ни больше, ни меньше: человечество создает собственный образ. Топорный, ограниченный, глупый (это ли не человеческие черты?), но свой собственный образ. Вот почему программирование приводит в такой неповторимый трепет. Некоторые сравнивают это с сексуальными переживаниями, и, учитывая созидательную природу обоих явлений, с этим можно согласиться. Вот что чувствовал МакЭндрю 100 лет назад:

*Я — пьян?.. Когда Ты создал мир, в начале было Слово, —  
И не оно ль внушало нам, что создал образцово?*

Вот почему мы ввязались в это безумное дело и почему с ним остаемся. Вот почему очень немногие вешают свою мышь и идут учиться на юристов, даже если у них нет ничего за душой. Прочитайте, что писал Киплинг 100 лет назад. Замените «лошадиные силы» на «мегафлопсы». И скажите мне после этого, что вы чувствуете не то же самое, зная, что ваша система работает:

*И пусть им жизнь дает не мать, а — плавка,ковка,сварка, —  
Их невозможно не понять, когда вздыхают жарко!  
Но никто не вразумил их голос никогда.  
Семь тысяч лошадиных сил...О Господи, о да!*



## Предметный указатель

- .NET
  - ActiveX 202
  - COM70.68
    - RCW 68
    - Web-служба 160, 161
    - HTTP GET 169, 170, 171
    - HTTP Post 172
    - HTTP POST 169
    - Soap 173
    - SOAP 169
    - WSDL 168, 170 см. также ЛЕТ:
  - Web-служба: контракт
    - базовая страница 166
    - доступ 169
    - клиент 165, 168, 171
    - клиентская форма 173
    - контракт 169 см. также .NET:
  - Web-служба:WSDL
    - логистика 166
    - метод 174
    - обращение 165
    - прокси-класс 175–176, 177
    - развертывание объектов 166
    - функция 165
    - Windows Forms 191
    - индексации элементов массивов 27
    - клиент 27, 68, 5, 70
    - межязыковое наследование 57
    - наследование 53, 57
    - объект 74, 79
    - объект-сервер 27
    - пространство имен 34
    - сборка компонентов 36
    - структурная обработка исключений 80
    - транзакции 76
    - управление памятью 59–61
    - элемент управления 204
  - .NET CLR
    - mscorlib.dll 40
    - безопасность доступа к коду 88
    - объект 35
    - функция 35
  - .NET Framework 8, 12, 22, 24, 68
    - безопасность доступа к коду 25
    - бесшовное взаимодействие с COM 25
    - сборка 42
    - функциональность ОС 25
  - .NET SDK 27, 30, 70, 162, 163
  - .NET Web Forms 9
  - .NET Windows Forms 10
- A**
  - Active Server Pages CM. ASP
  - ActiveX
    - Windows Forms 205, 208
    - контейнер 202
    - элемент управления 203–204, 205
  - .NET 202
  - ADO.NET 10
  - API 104
  - ASP (Active Server Pages) 101
  - ASP.NET 8, 73, 101
    - cookie-аутентификация 131
    - passport-аутентификация 135–137
    - Web Forms 103
    - Web-служба
    - ASMX-файл 183

- Global.asax 183
- управление состоянием 185
- Windows-аутентификация 130, 142
- авторизация 140, 141
- аутентификация 129
- доверенный пользователь 144
- защита 104
- идентификационные данные [44]
- инфраструктура 168
- приветствие 163
- создание 105
- управление сеансом 126
- файл
  - machine.config 118, 141
  - web.config 118
- фоновый код 102
- элемент управления 110
- assembly *см.* сборка
- Assembly Cache Viewer 42
- Assembly Generation Utility *см.* утилита:
- AL.exe
- Authenticode 87

## C

- CCW (COM callable wrapper) 74, 76
- CLR (Common Language Runtime) 22
- CLR.NET 80
- code access security *см.* .NET CLR: безопасность доступа к коду
- code groups *см.* сборка: группы программ
- code-behind *см.* ASP.NET: фоновый код
- COM 18, 22
  - вызываемая оболочка 74
- COM callable wrapper *см.* CCW
- Common Language Runtime *см.* CLR

## D

- DCOM 157
- deterministic finalization *см.* детерминированное завершение

- DHTML 116
- DLL 29, 30, 36
  - замена 46
  - класс 30
  - метод 30

## E

- exception handler block *см.* блок обработки исключений

## F

- finalizer *см.* сборщик мусора; завершитель
- fully qualified name *см.* полностью квалифицированное имя

## G

- CAC (global assembly cache) 42, 43, 49
- garbage *см.* объект: мусор
- garbage collection *см.* сбор мусора

## H

- HTML 101
- HTML-страница 4
- HTTP (Hypertext Transfer Protocol) 9, 130, 159
- HTTP GET 169
- HTTP POST 169

## I

- IIS (Internet Information Server) 9, 101
- IIS (Internet Information Services) 162
- ILCM. MSIL
- ILDASM 48, 53
- ILDASM.exe *см.* сборка: дизассемблер IL
- impersonation-delegation *см.* заимствование прав и делегирования
- informational version *см.* сборка: декларация: информационная версия

**J**

JIT (just-in-time) 23  
 JItter (just-in-time compiler) 23, 32  
 JIT-компиляция 33

**K**

Kerberos 131

**M**

managed code см. управляемый код  
 managed heap см. управляемая куча  
 manifest см. сборка: декларация  
 membership conditions см. сборка: условия членства  
 metadata см. метаданные  
 MFC [Microsoft Foundation Classes] 189  
 Microsoft .NET 7  
 Microsoft Foundation Classes см. MFC  
 Microsoft Intermediate Language см. язык программирования: MSIL  
 Microsoft Management Console см. MMC  
 Microsoft Transaction Server 77  
 MINFU 182  
 MMC (Microsoft Management Console) 183  
 Mscoree.dll 75  
 MSMQ 157

**N**

namespace см. пространство имен  
 Napster 9  
 native image generator см. утилита: Ngen.exe  
 NTLM 131

**P**

permission set см. набор прав доступа  
 postback data см. регистрация ответных данных  
 private assembly см. закрытая сборка

**Q**

q-name см. q-имя  
 qualified name см. квалифицированное имя  
 qualifier см. квалификатор  
 Quicken j  
 q-имя 36 см. также квалифицированное имя

**R**

RCW (runtime callable wrapper) 68  
 Read The Funny Manual см. RTFM  
 Regsvcs.exe 78  
 Resource Managers 77  
 RPC 157  
 RTFM (Read The Funny Manual) 54  
 runtime callable wrapper см. RCW

**S**

Secure Socket Layer см. SSL  
 security policy см. политика безопасности  
 SEH 82  
 shared name см. совместно используемое имя  
 SOAP (Simple Object Access Protocol) 169, 173  
 SSL (Secure Socket Layer) 130  
 strong name см. строгое имя  
 structured exception handling см. структурная обработка исключений

**T**

Tlbimp.exe 70

**V**

view state см. Web: элемент управления: состояние отображения  
 Visual Basic.NET 163  
 — клиент 31

- компилятор 30
- Visual Studio .NET
  - Windows Forms 192–193, 195, 202
  - панель инструментов 196
- Visual Studio.NET 12, 28, 69
- Web-служба 181
- тестирование 183
- строгое имя 44

## W

- Web 2, 19
  - элемент управления 109
  - состояние отображения 115
- Web Forms Server Control см. серверный элемент управления Web Forms
- Web Service см. Web-служба
- Web Service Descriptor Language см. WSDL
- Web-служба 159
  - Service1.aspx 183
  - создание 161, 162, 182
- Windows Forms 191
  - ActiveX 205, 208
  - клиент 200
  - пример приложения 192
  - форма 193
  - элемент управления 199–200, 201
- WSDL (Web Service Descriptor Language) 168

## X

- XML 9

## A

- автономный ПК 3
- атрибут
  - <WebMethod /> 165
  - AutoComplete 79
  - Class 163
  - Codebehind 183
  - ComVisible 203

- Const 165
- impersonation 148
- Language 163
- loginUrl 133
- memoryLimit 150
- mode 126, 127
- Private 165
- Public 165
- requestLimit 150
- SqlConnectionString 127
- stateConnectionString 126
- style 101
- System.Runtime.InteropServices.ComVisible 76
- timeout 150
- verb 140
- аутентификация 128
  - cookie 131
  - passport 135
  - Windows 1"iO

## Б

- безопасность 3
  - код 4
- библиотека ActiveX 27
- блок обработки исключения 81

## В

- вызываемая оболочка периода выполнения см. RCW

## Г

- глобальный кэш сборки см. CAC

## Д

- детерминированное завершение 65-66
- директива
  - Imports 28, 163
  - Namespace 38
  - Namespace TimeComponentNS 30

- *using* 31
- *WebService* 163
- диспетчер кучи 26

### З

заимствование прав и делегирования 147

### И

- Интернет
  - безопасность 3
- Интернет-приложение 3
- требования к инфраструктуре 7
- интерфейс
  - *IDispatch* 72
  - *IDisposable* 65
  - *IIdentity* 145
  - *IPrincipal* 142
  - *ISupportErrorInfo* 80
  - *IErrorInfo* 80
- исключение 82

### К

- квалификатор 36
- квалифицированное имя 36, 37 см. также q-имя
- класс 20, 50, 165
  - *AxHost* 206
  - *GenericPrincipal* 142-143
  - *SimplestHelloWorld* 194
  - *System.Exception* 84
  - *System.Object* 53, 57, 63, 65
  - *System.Runtime.InteropServices.TypeLibConverter* 70
  - *System.Web.Services.Protocols.SoapHttpClientProtocol* 175
  - *System.Windows* 198
  - *System.Windows.Forms.AxHost* 205
  - *System.Windows.Forms.DataGrid* 65
  - *System.Windows.Forms.Form* 193

- *System.Windows.Forms.UserControl* 201
- *TimeComponent* 31, 33
- *TimeService* 164
- *UserControl* 202
- *WebService* 164, 186
- базовый 52, 164
- деструктор 57-58
- конструктор 57
- межязыковое наследование 57
- наследование 52
- объявление 29
- подмена метода 56
- производный (derived) 52, 164
- функциональность 57
- ключевое слово
  - *Catch* 81
  - *Imports* 37, 164
  - *Inherits* 53, 164
  - *Overridable* 57
  - *override* 56
  - *Overrides* 56
  - *Return* 30
  - *Throw* 83
  - *Try* 81
  - *using* 37
  - *virtual* 57

код

- безопасность 87
- компилятор по требованию см. JITer
- компиляция по требованию см. JIT
- конструктор 57-58, 59
- криптография 43

### М

- метаданные 30
- метод
  - *Activator.CreateInstance* 73
  - *Begin* 179
  - *BeginGetTime* 179

- *Button1\_Click* 198
- *Class\_Terminate* 62, 63
- *Console.Write* 31
- *DisableCommit* 79
- *Dispose* 65, 84, 198
- *EnableCommit* 79
- *End* 180
- *EndGetTime* 179
- *Equals* 55
- *File.Open* 82
- *Finalize* 63
- *GetHashCode* 55
- *GetTime* 27, 31, 162, 172, 174
- *GetType* 55
- *InitializeComponent* 197
- *invoke* 176
- *IsCompleted* 181
- *IsInRole* 142, 145
- *Main* 31
- *Navigate* 207
- *New* 58
- *OnAuthenticate* 143
- *Session.Abandon* 124
- *SetAbort* 79
- *SetComplete* 79
- *System.Object.ToString* 56
- *ToString* 55, 56
- *Type.GetTypeFromCLSID* 72
- *Type.GetTypeFromProgID* 72
- детерминированного завершения 65
- подмена 56

## Н

- набор прав доступа 90
- наследование 20, 31, 51, 164
  - дерево 53
  - явное объявление 54
- настольное ПО 188

## О

- обработчик исключений 81
  - Try-Finally 84
- объект 51
  - *Application* 127
  - *base* 57
  - *IAsyncResult* 179, 181
  - *MessageBox* 36
  - *MyBase* 57
  - *Session* 122, 123
  - *String* 76
  - *System.EnterpriseServices.ContextUtil* 78, 79
  - *System.Exception* 82
  - *TimeService* 172
  - *User* 145
- детерминированное завершение 65
- клиент 32
- конструктор 57–58
- конфликт имен 25
- мусор 61
- удаление 60, 63
- сервер 29
- ООП (объектно-ориентированное программирование) 20, 50
- С++ 51
- COBOL 51
- Java 51
- Visual Basic 51
- наследование 53
- оператор
  - *imports* 70
  - *new* 32, 58, 70
- операция *get* 140

## П

- параметр *ShowSeconds* 171
- подпространство имен 36

позднее связывание 72  
 политика безопасности 90  
 полностью квалифицированное имя 36  
 право доступа 91  
 пространство имен 28, 163  
 — System 25, 31, 35  
 — System.Console 36  
 — System.Windows.Forms 36  
 — Visual Basic 28-29  
 — импорт 29  
 — импорт 31, 37  
 — имя 38  
 — объявление 29—30

## Р

регистрация ответных данных 114

## С

C++ 10  
 сбор мусора 24, 26, 62, 66  
 сборка 38, 40  
 — AssemblyInfo.vb 47  
 — CAC 42, 49  
 — версия 40  
 — группы программ 93  
 — декларация 38, 40  
 — — информационная версия 48  
 — — хэш 45  
 — дизассемблер IL 40  
 — закрытая 33, 41  
 — закрытый ключ 45  
 — клиент 41  
 — младшая версия 47  
 — многофайловая 39  
 — номер версии 47  
 — номер компоновки 47  
 — однофайловая 39  
 — открытый ключ 41  
 — развертывание 41  
 — ревизия 47

— совместно используемая 42  
 — — подпись 45  
 — старшая версия 47  
 — управление версиями 46, 48—49  
 — условия членства 93  
 — файл 39  
 — — добавление/удаление 40  
 сборка мусора 8  
 сборщик мусора 26  
 — завершитель 63  
 — удаление объекта 63  
 свойство  
 — AutoPostBack 106  
 — CopyLocal 45  
 — DeactivateOnReturn 79  
 — Identity 145  
 — MaintainState 116  
 — MyTransactionVote 79  
 — Now 30  
 — Text 194

серверный элемент управления Web Forms 103

служба

— Passport 135

— доступ 8

событие

— Class\_Initialize 58

— SelectedIndexChanged 107

совместно используемое имя 43

спецификатор Shared 194

способ указания версии 24

строгое имя 43, 44

структурная обработка исключений 80

## У

управление сеансом 123

управляемая куча 61

управляемый код 22

утечка памяти 19

утилита

- AL.exe 40
- AxImp.exe 206
- COM+ Explorer 119
- GACUTIL.exe 42
- Ngen.exe (native image generator) 33
- RegAsm.exe 75, 203
- SN.exe 44
- Wsdl.exe 175

## Ф

файл конфигурации 49

функция

- AddHandler 198
- CoCreateInstance 70
- CoCreateInstance 75
- Console.Write 31, 37
- Control.ActiveXRegister 203
- Control.ActiveXUnregister 203
- CreateFile 80
- CreateWindow 80
- Finalize 63
- GetObjectContext 78
- GetTime 165
- LoadLibrary 80
- Main 194
- Now 28, 40
- System.GC.Collect 62, 72
- System.GC.SuppressFinalize 65

- System.Runtime.InteropServices.Marshal.ReleaseComObject 37, 72
- System.Security.Principal.WindowsIdentity.GetCurrent() Impersonate 148
- System.Web.Security.FormsAuthentication.RedirectFromLoginPage 134
- System.Windows.Forms.Application.Run 195
- Type.InvokeMember 73
- Write 37
- конфликт имен 25
- объявление 29

## Ц

цифровой сертификат 21

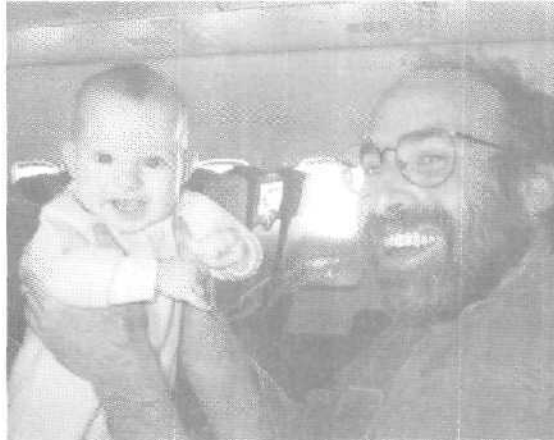
## Я

язык программирования

- C# 11
- COBOL 51
- Java 18
- JScript 23
- MSIL [Microsoft Intermediate Language, ID 22
- Visual Basic 19
- взаимодействие 21
- выбор 17
- C++ 19
- совместимый с CLR 23



## Дэвид С. Платт



Дэвид С. Платт, основатель и президент Rolling Thunder Computing, преподает программирование для .NET в компаниях по всему миру. Он автор еще пяти книг по программированию для Windows, последняя из которых, Understanding COM+ (Microsoft Press, 1999), по уровню продаж на Amazon.com не уступала книге Тома Кланси «Каждый мужчина — тигр» (что говорит об интеллектуальном уровне его читателей). Он также часто публикует статьи в MSDN Magazine.

Дэвид получил высшее инженерное образование в Аартмутском колледже. Бросив работу, он стал работать еще больше. Ему интересно, нужно ли связать два пальца у своей дочери, чтобы она научилась считать в восьмеричной системе. Он живет в Ипсвиче, штат Массачусетс, и связаться с ним можно по адресу [www.rollthunder.com](http://www.rollthunder.com).

Платт Дэвид С.

# Знакомство с Microsoft .NET

Перевод с английского под общей редакцией **В. Г. Вшивцева**

Технический редактор **С. В. Дергачев**

Компьютерная верстка **В. Б. Хильченко**

Дизайнер обложки **Е. В. Козлова**

Оригинал-макет выполнен с использованием  
издательской системы Adobe PageMaker 6.0

**TypeMarketFontLibrary**  
легальный пользователь

ПОЛЬЗОВАТЕЛЬ  
**Para(-)Type**  
IN LEGAL USE

Главный редактор **А. И. Козлов**

Подготовлено к печати издательско-торговым домом «Русская Редакция»

 РУССКАЯ РЕДАКЦИЯ •

Лицензия АР № 066422 от 19.03.99 г.

Подписано в печать 20.07.01 г. Тираж 3 000 экз.

Формат 70х100/16. Физ. л. л. 15

Отпечатано с готовых диапозитивов в ОАО «Типография «Новости»  
107005, Москва, ул. Фр. Энгельса, 46.

# Гарантия Вашей квалификации!



Издательство «Русская Редакция» — партнер Microsoft Press в России — предлагает широкий выбор литературы по современным информационным технологиям.

Мы переводим на русский язык бестселлеры ведущих издательств мира, а также сотрудничаем с компетентными российскими авторами.

## Наши книги Вы можете приобрести

### • в Москве:

«Библио-Глобус» ул. Мясницкая, 6, тел.: (095) 928-3567  
«Московский дом книги» ул. Новый Арбат, 8, тел.: (095) 290-4507  
«Дом технической книги» Ленинский пр-т, 40, тел.: (095) 137-6019  
«Молодая гвардия» ул. Большая Полянка, 28, тел.: (095) 238-5001  
«Дом книги на Соколе» Ленинградский пр-т, 78, тел.: (095) 152-4511  
«Мир печати» ул. 2-я Тверская-Ямская, 54, тел.: (095) 978-5047  
Торговый дом книги «Москва» ул. Тверская, В, тел.: (095) 229-6483  
«Алекс к К» Магазин «Книги» г. Зеленоград, Панфиловский пр-т, 11065, тел.: (095) 532-9669

### Фирменный магазин «Компьютерная и деловая книга»

Москва, Ленинский проспект, строение 38, тел.: (095) 778-7269

- в Санкт-Петербурге:  
ЗАО «Диалект», тел.: (812) 247-1483
- в Новосибирске:  
ООО «Топ-книга», тел.: (3832) 36-1026
- в Набережных Челнах:  
ООО «Аспект-С», тел.: (8552) 58-8013
- в Алма-Ате (Казахстан):  
ЧП Болат Амреев, тел.: 8-327-290-191-25, (3272) 26-1404
- в Киеве (Украина):  
ООО Издательство «Ирина», тел.: (044) 269-0423  
«Техническая книга на Петровке», тел.: (044) 464-6895
- в Минске (Белоруссия):  
ООО «Полурри», тел.: 8-10-375-17-2225726  
ООО «Аргфинанс», тел.: 8-10-375-17-2366716

Интернет-магазин <http://www.itbook.ru>

# И Р У С С К А Я   Р Е Д А К Ц И Я

тел.: (095) 142-0571; тел./факс: (095) 145-4519  
e-mail: [info@rusedit.ru](mailto:info@rusedit.ru); <http://www.rusedit.ru>



# Microsoft®

## для профессионалов

Издательство «Русская Редакция»  
представляет новую серию книг  
Microsoft Press

### Ресурсы Microsoft Windows 2000 Server

официальный источник основных  
технических сведений  
по операционной системе  
Windows 2000 Server.

В книгах серии содержится  
подробная информация  
для администраторов сети,  
Web-мастеров и опытных  
пользователей, занимающихся  
настройкой, администрированием,  
оптимизацией и устранением  
неполадок в Windows 2000 Server,  
а также сетей и Web-узлов  
на основе Windows 2000 Server.

издательство компьютерной литературы  
**И РУССКАЯ РЕДАКЦИЯ**

#### ПРОДАЖА КНИГ

оптом тел.: (095) 142-0571. e-mail: alexg@rusedit.ru;  
интернет-магазин <http://www.ITbook.ru>; тел.: (095) 145-4519;  
в розницу магазин «КОМПЬЮТЕРНАЯ И ДЕЛОВАЯ КНИГА»  
Москва, Ленинский пр-т, стр. 38. тел.: (095) 778-7269



## Официальные учебные пособия *Microsoft* —

**г а р а н т и я**  
**Вашей квалификации/**



### Книги серии «Учебный курс» для подготовки к экзаменам **MCSE-2000**

Книги этой серии помогут Вам приобрести фундаментальные знания и подготовиться к сдаче экзаменов по программам сертификации Microsoft.

Каждая книга — это серьезное изложение темы, полная справочная информация от первоисточника; занятия, упражнения для самостоятельной работы, видеоролики и вопросы для самопроверки и закрепления знаний.

издательство компьютерной литературы  
**И Р У С С К А Я Р Е Д А К Ц И Я**

#### ПРОДАЖА КНИГ

оптом Тел.: (095) 142-0571, e-mail: alexg@rusedit.ru;  
интернет-магазин <http://www.ITbook.ru>; тел.: (095) 145-4519;  
в розницу магазин «КОМПЬЮТЕРНАЯ И ДЕЛОВАЯ КНИГА»  
Москва, Ленинский пр-т, стр. 3В, тел.: (095) 778-7269

# Где

книги по современным  
информационным  
технологиям

# купить

новый  
интернет-магазин

<http://ITbook.ru>



магазин  
«Компьютерная  
и деловая книга»

г. Москва, Ленинский пр-т, стр. 38  
тел.: (095) 778-7269

**БОГАТЫЙ  
ВЫБОР**

**НИЗКИЕ  
ЦЕНЫ**



# Знакомство с **MICROSOFT** **.NET**

**Примеры программ  
доступны в любую минуту!**

Примеры программ на *Visual Basic.NET* можно загрузить  
с Web-сайта книги:

**[www.introducingmicrosoft.net](http://www.introducingmicrosoft.net)**

## **Подлинная история .NET — кратко и увлекательно!**

Какие проблемы решает .NET? Как работать с .NET и как извлечь из этого выгоду? Ответы на эти и другие вопросы вы получите из этой книги.

Дэвид С. Платт — известный автор и консультант — последовательно раскрывает тайны новой платформы Microsoft. Минимум технического жаргона и максимум остроумия, масса подробных иллюстраций, изобилие выразительных аналогий и четких разъяснений.

Прочитав эту увлекательную книгу, вы будете достаточно знать о передовой платформе Microsoft .NET, чтобы спланировать свое будущее в программной индустрии.

### **В книге рассматриваются:**

- объекты .NET;
- ASP.NET и Web Forms;
- Web-службы.NET;
- « Microsoft Windows Forms.

### **Об авторе:**

Дэвид С. Платт преподает компьютерные науки в Гарвардском университете. Он основал учебную и консультационную компанию **Rolling Thunder Computing** ([www.rollthunder.com](http://www.rollthunder.com)).

Платт — автор еще пяти книг, включая *Understanding COM+*, а также частый автор **Microsoft Developer Network**.

ISBN 5-7502-0186-4



9 785750 201860

Web-узел издательства: [www.rusedit.ru](http://www.rusedit.ru)  
Интернет-магазин: [www.ITbOok.ru](http://www.ITbOok.ru)

