Дан Эпплман

Переход на VB .NET

стратегии, концепции, код

БИБЛИОТЕКА ПРОГРАММИСТА

Дан Эпплман

Переход на VB .NET

стратегии, концепции, код

- основные характеристики архитектуры .NET
- критерии необходимости использования .NET
- концепции новых возможностей Visual Basic .NET
- изменения в языке Visual Basic в рамках среды .NET
- влияние новой модификации Visual Basic на программирование в новой архитектуре

Apress™

ПИТЕР®

СЕРИЯ

БИБЛИОТЕКА ПРОГРАММИСТА

ПИТЕР®

Dan Appleman

Moving to VB .NET

Strategies, Consepts, and Code

Apress™

Дан Эпплман

БИБЛИОТЕКА ПРОГРАММИСТА

Переход на VB .NET

стратегии, концепции, код

Санкт-Петербург
Москва • Харьков • Минск
2002

ПИТЕР®

Дан Эпплман

Переход на VB .NET: стратегии, концепции, код

Перевел с английского Е. Матвеев

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*А Васильев*
Научный редактор	*А. Черносвитов*
Литературный редактор	*Е. Саргаева*
Художник	*Н Биржаков*
Иллюстрации	*Ю. Дорохова*
Корректор	*В. Листова*
Верстка	*П. Быстров*

ББК 32.988.02-018
УДК 681.324

Эпплман Д.

Э72 Переход на VB .NET: стратегии, концепции, код. — СПб.: Питер, 2002. — 464 с.: ил.

ISBN 5-318-00746-5

Эта книга была задумана как одна из первых книг о .NET, которая ознакомит читателя с основными идеями новой архитектуры и подготовит его к знакомству с более детальной литературой, например документацией Microsoft и ее толкованиями, которая неизбежно появится на рынке. Она поможет вам взглянуть на эту технологию с позиций ваших собственных рабочих планов и быстро освоить те концепции, которые покажутся необычными для большинства программистов Visual Basic. Широко известный и уважаемый в компьютерном мире автор постарался излагать материал как можно более сжато и в то же время достаточно глубоко, чтобы сделать вас экспертами в области языка VB.NET и помочь в освоении тех областей архитектуры .NET, которые представляют наибольший интерес.

ISBN 5-318-00746-5

ISBN 1-893-115-976 (англ.)

ЗАО «Питер Бук». 196105, Санкт-Петербург, Благодатная ул., д. 67.
Лицензия ИД № 01940 от 05.06.00.
Налоговая льгота – общероссийский классификатор продукции ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 12.01.02. Формат 70×100$^{1}/_{16}$. Усл. п. л. 37,41. Тираж 4000 экз. Заказ № 2533.
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Содержание

Часть 2. Концепции

Введение

Вероятно, вы уже слышали об архитектуре .NET компании Microsoft и о новых возможностях Visual Basic .NET. Возможно, вы читали статьи в компьютерных журналах, знакомились с рекламными материалами Microsoft или даже успели поэкспериментировать с бета-версиями.

Но откуда бы ни была получена информация, на первых порах неизбежно чувствуешь легкую растерянность. Изменения в языке и в самом подходе к программированию выглядят настолько грандиозными, что становится трудно отделить реальность от маркетинга и решить, с чего же следует начинать освоение новой технологии.

Собственно, для этого и была написана эта книга. Из нее вы узнаете:

- на что следует в первую очередь обращать внимание при изучении .NET и какими критериями следует руководствоваться, принимая решение о том, когда и как использовать .NET;
- на каких концепциях основаны новые возможности Visual Basic .NET и как они влияют на процесс программирования в этой новой архитектуре;
- какие изменения произошли в самом языке Visual Basic.

Перед вами одна из первых книг о .NET, однако вы не найдете в ней подробного обзора этой архитектуры или сколько-нибудь полного справочника по Visual Basic .NET — несомненно, на эту тему еще будет написано немало книг. Впрочем, книга не является и банальным пересказом документации .NET. Архитектура .NET настолько грандиозна, а изменения в Visual Basic .NET настолько масштабны, что полное раскрытие темы заняло бы слишком много места и окончательно сбило бы читателя с толку, даже если бы всю информацию каким-то чудом удалось бы разместить в одной книге. Такие темы, как среда разработки, отладка и службы высокого уровня, вполне заслуживают отдельных книг и описываются в общих чертах (или не описываются вовсе).

Книга была задумана как одна из первых книг о .NET, которая подготовит читателя к знакомству с более серьезной литературой, документацией Microsoft и ее пересказами, которые неизбежно появятся на рынке. Она поможет вам взглянуть на эту новую технологию с позиций ваших собственных рабочих планов и быстро освоить те концепции, которые покажутся необычными для большинства

программистов Visual Basic. Я постарался излагать материал как можно более сжато и в то же время достаточно глубоко, чтобы сделать вас экспертами в области языка VB .NET и помочь в освоении тех областей архитектуры .NET, которые представляют наибольший интерес.

Для кого написана эта книга

Материал ориентирован на программистов Visual Basic среднего или высокого уровня. Книга определенно *не предназначена* для новичков в области программирования, хотя программисты, переходящие на VB .NET с других языков, найдут в ней немало полезного. Кроме того, книга может пригодиться руководителям групп и менеджерам, принимающим стратегические решения, — в первой части содержится немало информации общего и технологического плана.

В отличие от других книг такого рода, эта книга ставит перед собой только одну задачу — помочь программистам VB6 в освоении архитектуры .NET вообще и VB .NET в частности. Наверное, ее структура покажется несколько необычной. В частности, я не собираюсь попусту тратить время на элементарные вопросы типа «Что такое класс?», «Как работает цикл For...Next?» или «Что такое коллекция?» Места и так не хватает, и мы не будем отвлекаться на банальности, известные любому мало-мальски квалифицированному программисту VB6.

О Microsoft

Похоже, окружающий мир разделился на два лагеря: тех, кто любит и поддерживает компанию Microsoft, и тех, кто ее люто ненавидит. Лично я отношусь к той аполитичной группе, которая отказывается видеть в Microsoft верховное божество[1]. В результате противники Microsoft клеймят меня как «продажного наймита», а сторонники Microsoft — осуждают за «злобные нападки».

На самом деле я очень уважаю Microsoft[2]. В ее изделиях встречается немало хороших технологий, совсем немножко технологий замечательных и удручающе много рекламной шумихи на пустом месте. Маркетинговый отдел Microsoft работает исключительно эффективно, хотя его действия порой кажутся совершенно непостижимыми. Насколько мне известно, это весьма хаотичная и непредсказуемая компания, решения которой часто обусловлены не только техническими новшествами, но и внутренней политикой (собственно, это относится к любой организации, но по отношению к Microsoft почему-то воспринимается некоторыми как оскорбление).

Итак, я неплохо отношусь к Microsoft, но не упускаю возможности слегка поглумиться над ней — во-первых, жалко упускать такую великолепную мишень, а во-вторых, это вполне естественная человеческая реакция на любые непреодолимые[3] обстоятельства.

---

[1] Разве что божество второстепенное. На мой взгляд, дискуссии о роли Microsoft в выпуске клавиатур, разработке текстовых редакторов и операционных систем приобрели характер религиозных споров, а я в этой области считаю себя полным атеистом.

[2] Некоторые из моих лучших друзей работают в Microsoft — банально, но факт.

[3] Если не верите — подумайте, какое влияние .NET окажет на вашу личную карьеру.

Еще раз для протокола: я не являюсь ненавистником Microsoft и в целом хорошо отношусь к этой компании.

Впрочем, это вовсе не означает, что я слепо соглашаюсь с Microsoft или считаю, что эта компания всегда принимает верные технологические решения. Конечно, я верю далеко не всему, что написано о «единственно правильном подходе к программированию», да и вам не советую. Все мы хорошо знаем, как часто изменяется правильный подход к программированию, причем эти изменения часто застают Microsoft врасплох.

В этой книге я делюсь своими мнениями об архитектуре .NET и об изменениях в языке Visual Basic. Некоторые мнения отличаются от позиции Microsoft, другие ей прямо противоречат, а третьи окажутся ошибочными. В этом нет ничего страшного. Моя цель — не изрекать истину в последней инстанции, а представить новую технологию в практическом контексте, чтобы вы могли самостоятельно оценить ее. Надеюсь, мои старания расширят ваши представления о Visual Basic .NET и в конечном счете помогут приспособить эту технологию для ваших целей.

О бета-версиях

Если вам доводилось читать мои предыдущие книги и статьи, вероятно, вы знаете, что я особенно ненавижу два типа книг: пересказы документации и поделки, сляпанные на скорую руку на основе бета-версий и потому содержащие неточную информацию.

А теперь я сам пишу книгу, основанную на предварительной версии программы. Напрашиваются три объяснения.

1. Я — отъявленный лицемер.
2. Я позарился на деньги, потому что ранние книги хорошо продаются, даже если это сплошная туфта.
3. На то были веские причины. Возможно, для вас они неочевидны, но вы хотя бы захотите выслушать мои объяснения.

Знаю, о чем вы подумали. Правильный ответ — третий, не так ли?

Конечно. Но и без второго тоже не обошлось[1] — правда, я искренне надеюсь и верю, что перед вами вовсе не туфта.

Хотя архитектура .NET еще не получила «официального» воплощения, все общие принципы и концепции уже устоялись, как и подавляющее большинство пространств имен. Определение языка Visual Basic .NET пришло к окончательному виду, и все работает так, как написано в документации. Если верить Microsoft, от бета-2 до окончательной версии язык практически не изменится.

Чтобы заняться изучением языка, не обязательно дожидаться окончательной версии продукта. На данный момент все основные положения сформулированы достаточно четко, и о них уже можно писать.

Впрочем, важнее другое. Меня очень беспокоит, что неопытные пользователи могут неправильно отнестись к этой технологии.

- Одни будут несправедливо осуждать изменения в языке VB .NET, несовместимые с готовым кодом.

[1] Неприятно, но честно.

- Другие будут ругать Common Language Runtime только за большой объем среды и за то, что из VB .NET исчезла поддержка native-кода.
- Третьи начнут с энтузиазмом применять наследование и многопоточность так, что лишь наживут себе неприятности: это затруднит масштабирование, отладку и сопровождение.

С этой технологией, как ни с одной из тех, что мне вспоминаются, важно правильно начать — вы должны понять, почему в Microsoft приняли те или иные решения и почему эти решения большей частью оказались удачными. Вы должны понять, что наследование применяется лишь в крайнем случае и только после основательных размышлений. Вы должны понять, что неверные концептуальные решения в многопоточной среде порой обходятся очень дорого из-за сложностей, связанных с поиском нетривиальных ошибок синхронизации.

Именно поэтому я и решил написать эту книгу пораньше — мне хотелось помочь программистам VB правильно взяться за дело. Надеюсь, она вам действительно пригодится.

Примеры

В книгу включены многочисленные примеры программ, которые можно загрузить с сайта издательства «Питер» по адресу http://www.piter.com. Структура каталогов в архиве отражает структуру книги. Таким образом, встретив ссылку на проект с именем TwoInterfaces в главе 5, следует заглянуть в каталог Source\CH5\TwoInterfaces.

Предполагается, что читатель будет работать с исходными текстами с web-сайта вместо того, чтобы вводить весь программный код вручную. Это объясняется двумя причинами. Во-первых, во многих случаях я привожу лишь часть кода, необходимого для работы проекта, — только для того, чтобы пояснить обсуждаемую тему и избавить читателя от блуждания в огромных листингах. Во-вторых, данное издание все-таки основано на предварительной версии продукта, и в окончательной версии некоторые примеры могут измениться. Я постараюсь почаще обновлять примеры. Список обновлений и модификаций также будет вестись на web-сайте http://www.piter.com.

Desaware

Я всегда упоминаю в своих книгах компанию Desaware. Дело в том, что написание книг не окупает затраченного времени, так пусть оно хотя бы поможет продвижению моей компании. Сейчас я еще не знаю, какие продукты мы будем предлагать для VB .NET, но такие продукты наверняка появятся. Заходите почаще на сайт http://www.desaware.com и следите за новостями.

Итак, наше знакомство с архитектурой .NET и Visual Basic .NET начинается.

Дан Эпплман

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

На web-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Важное замечание о примерах программ

Книга, которую вы держите в руках, написана **для версии бета-2**. Между версиями бета-1 и бета-2 Visual Studio .NET существует немало принципиальных различий.

Обновления и исправления примеров и содержимого книги как для будущих бета-версий, так и для окончательной версии Visual Studio будут распространяться через Web. Следовательно, с выходом окончательной версии изложенный материал не устареет.

Примеры книги тестировались только в Windows 2000. Обновления и исправления для других операционных систем будут распространяться на web-сайте. На момент издания книги Microsoft рекомендует вести все разработки .NET только в Windows 2000.

Другое важное замечание о примерах программ

Во всех проектах этой книги используется режим жесткой проверки типов Option Strict On (устанавливается на вкладке Build диалогового окна Project Properties).

Я *настоятельно* рекомендую включать режим Option Strict On во *всех* проектах VB .NET.

Это действительно очень важно. Во всей книге я только один раз упоминаю о возможности сброса этого флажка. В главе 8 я постараюсь убедительно показать, что работа над любой программой должна начинаться с установки флажка жесткой проверки типов и что сброс этого флажка по умолчанию можно считать самой серьезной ошибкой Microsoft в отношении VB .NET.

Благодарности

В работе над книгой у меня было столько помощников, что я даже не знаю, с кого начать. Вся команда Apress работала просто замечательно. Грейс Вонг (Grace Wong) творила чудеса в качестве руководителя проекта и подобрала великолепную рабочую группу. Кари Брукс (Kary Brooks) умело справлялась со всеми изменениями графика. Кирстен Берк (Kiersten Burke) является идеальным редактором, по моим представлениям — она правит авторский текст деликатно, но проявляет ту последовательность и логичность в изложении материала, которых мне так не хватает. Спасибо Кэрол Бурбо (Carol Burbo), помогавшей мне с форматированием (я очень рад, что мы снова работаем вместе). Я также благодарен Сьюзен Глинерт (Susan Glinert), Джули Кавабата (Julie Kawabata) и Карлу Миядзиме (Karl Miyajima) за их вклад в верстку, составление алфавитного указателя и графическое оформление книги. Спасибо Стефани Родригес (Stephanie Rodriguez), которая делала все возможное и невозможное для рекламного продвижения книги на всех стадиях работы; Мэйсону Смиту (Mason Smith) и другим сотрудникам Springer, отвечавшим за маркетинг; Черрил Швингес (Cherryl Schwinges) за планирование и административную работу. Спасибо Карен Уоттерсон (Karen Watterson), которая, несмотря на крайнюю занятость, как-то ухитрялась выкроить время на просмотр многих глав и предлагала советы гуру. Я благодарен Джейсону Гилмору (Jason Gilmore), с которым мне было приятно работать вместе, хотя к этой книге он особого отношения не имеет. Конечно, не будем забывать и Гэри Корнелла (Gary Cornell) — человека, благодаря которому существует издательство Apress.

Я не смогу даже отдаленно выразить, насколько я благодарен Скотту Стабберту (Scott Stabbert), одному из лучших технических редакторов, с которыми я имел удовольствие работать. Даже если не упоминать о его очевидном вкладе в точность излагаемого материала (особенно учитывая многочисленные изменения в VB .NET во время написания книги), он проявлял безграничную терпимость в тех случаях, когда я считал нужным критиковать Microsoft, — большую, чем можно было бы ожидать от работника этой компании. Кроме того, он предоставлял дополнительные сведениях о мотивах и логике разработчиков. Спасибо, Скотт, — надеюсь, скоро мы продолжим совместную работу. Спасибо и другим сотрудни-

кам Microsoft, в том числе Эрику Андре (Eric Andre), Дрю Флетчеру (Drew Fletcher), Дейву Мендлену (Dave Mendlen), Ари Биксхорну (Ari Bixhorne) и Майку Айэму (Mike Iem) — они знают, за что, а вам я не скажу, чтобы у них не было неприятностей.

Спасибо Крису Мак-Аскиллу (Chris MacAskill), Мэгги Кэннон (Maggie Cannon) и Джуди Киркпатрик (Jusy Kirkpatrick) из MightyWoods. Их содействие в публикации моей электронной книги со сравнительным анализом VB .NET и C# также помогло распространению информации об этой книге.

Как обычно, эта книга никогда бы не вышла в свет без помощи моей «семьи» из компании Desaware. Кэрин Данкан (Karyn Duncan), Степан Пежич (Stjepan Pejic), Мариан Киклайтер (Marian Kicklighter), Фрэнки Вонг (Franky Wong) и другие: Ариэль, Джил, Игорь, Одед и Люк — спасибо всем!

Ну и конечно, моя семья за пределами Desaware — общество любителей поэзии доктора Сюсса (Dr. Seuss), которые не ворчали, когда мое присутствие на мероприятиях сводилось к сидению в уголке за портативным компьютером. Я благодарен Роану, Майе и Кендре, которые иногда силой вытаскивали меня за город. Наконец, спасибо маме за то, что она порой заставляла меня поесть, и папе за все — в том числе и за то, что он внимательно читал каждую главу перед сдачей.

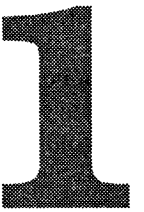
Часть 1

Стратегии

Технология — странная штука. Одной рукой она преподносит вам великие дары, а другой вонзает нож в спину.

Чарльз Перси Сноу

С чего начать?



Книга «Дюна» Фрэнка Херберта начинается с фразы: «С самого начала надо тщательно взвесить все обстоятельства, чтобы не уподобиться маятнику». Я бы трактовал эту фразу так: «Начало — это время, когда нужно проявить особую заботу о непредвзятом отношении».

По-моему, это утверждение относится и к книгам, и к изучению новых технологий.

Первые впечатления, общий подход и последовательность изложения очень сильно влияют на то, насколько быстро и глубоко вы осваиваете новый материал. Заблуждения, возникающие на начальной стадии, позднее оборачиваются серьезными препятствиями на пути к вершинам мастерства. Стоит упустить всего одну ключевую концепцию, и перед вами разверзнется пропасть, которая помешает в изучении нетривиальных тем.

Что ни говори, начало — ответственный и опасный момент.

Я тщательно обдумывал, как начать эту книгу.

Можно было выбрать тот путь, по которому идет Microsoft, представляя .NET на разных презентациях и семинарах. Я ясно вижу происходящее...

Вы сидите в громадной аудитории. Лектор на отдаленном подиуме едва виден на фоне монитора — картинка дублируется на нескольких экранах, которые своими размерами оказали бы честь среднему кинотеатру. Лектор демонстрирует Visual Basic .NET, размещает на форме несколько элементов и запускает программу. Бац! В браузере возникает готовая web-страница! Однако лектор идет еще дальше и создает на другом сервере нечто похожее на класс Visual Basic. Затем он возвращается в исходную программу — и вот уже методы и свойства нового класса, словно по волшебству, оказываются доступными через Web. Вся аудитория поражена простотой разработки и новаторским подходом, в соответствии с которым web-сайт предоставляет не только визуальную информацию или данные на базе XML, но и целые прикладные библиотеки, с которыми так легко работать в Visual Basic.

Возможно, вы даже мельком увидите кусочек программного кода, творящего все эти чудеса, и он вам покажется отдаленно знакомым...

Можно было пойти и по другому пути, который используется во многих учебниках.

Вся первая глава заполняется подробными инструкциями, фрагментами кода, которые вам предлагается ввести, и многочисленными снимками экранов. Все описание программы сводится к нескольким невразумительным комментариям и обещаниям все подробно объяснить «как-нибудь потом». Собственно, в таком стиле можно написать не только первую главу, а целую книгу; более того, вы даже сможете самостоятельно расширить примеры приложений, и они будут работать — если, конечно, все ваши потребности будут ограничиваться теми примерами, которые я подобрал. А если ваша конкретная задача не встречается в моем учебнике... что ж, всегда найдутся другие книги.

Наконец, можно выбрать подход, который лично я считаю правильным, хотя он обычно не встречается в подобных книгах. В начальных главах я нарушаю сразу несколько общепринятых правил.

Я не привожу ни одного примера кода.

Более того, я почти ничего не рассказываю о среде .NET и Visual Basic .NET.

Вместо этого речь пойдет о контексте. О маркетинге. О подходе к решению проблемы. Я постараюсь представить .NET с такой точки зрения, которая поможет обойти все начальные затруднения и как можно быстрее приспособиться к тому, что нас всех ожидает.

Конечно, это весьма непростой путь — программировать куда интереснее, чем рассуждать на общие темы. Поэтому я советую прерваться, запустить свою копию Visual Studio .NET и поэкспериментировать с простыми примерами, включенными в документацию. Когда вы более или менее освоитесь, возвращайтесь к чтению.

Что, уже вернулись? Хорошо, продолжим.

О важности контекста

Обожаю MSDN. Наверное, это самое лучшее, что когда-либо сделали в Microsoft — по крайней мере, для профессионального программиста. Каждый программист должен сделать постоянной закладку http://msdn.microsoft.com, а возможно — и стать постоянным подписчиком MSDN.

Но честно говоря, иногда мне кажется, что MSDN страдает близорукостью. Каждая новая технология объявляется лучшим (а как же!) решением. Каждую новую технологию необходимо как можно быстрее изучать и применять на практике. Каждая новая технология проще и удобнее всего, что было раньше (особенно благодаря непрерывно растущему числу курсов и учебников по сдаче сертификационных экзаменов).

Приведу конкретный пример.

Однажды мне понадобился сервер электронной почты для моей компании. Компания невелика, и в любой момент использовалось не более десятка-другого учетных записей электронной почты. Конечно, я прежде всего подумал о Exchange Server. Поскольку я пишу о технологиях Microsoft и занимаюсь разработками под них, было вполне логично как можно шире использовать их на практике, чтобы узнать о них побольше.

Итак, я установил Exchange Server и попытался понять, как он работает. Потом купил книгу — большую, толстую книгу — и попытался еще раз. Результат оказался печальным; на компьютере с NT начал хронически возникать синий «предсмертный экран». Наконец, я где-то прочитал, что Exchange Server нормально работает только на выделенном сервере с гораздо большим объемом памяти, чем у меня.

Проблема решилась форматированием жесткого диска и переустановкой NT — но уже без Exchange Server.

Я отправился в ближайший магазин и купил сервер Eudora Worldmail за $99, не считая цены нескольких дополнительных клиентских лицензий.

Установка заняла не более десяти минут.

Процесс конфигурации был элементарным и интуитивно понятным.

Наконец, система безукоризненно работала на сервере, который в это время занимался массой других полезных вещей.

Только поймите меня правильно — я ничего не имею против Exchange Server[1]; несомненно, это замечательный продукт. Я лично знаком с несколькими людьми, с большим успехом использующих его в крупных компаниях. Но истина заключается в том, что для маленькой компании вроде моей это решение оказалось неподходящим. Однако в материалах Microsoft никто не написал большими жирными буквами: «Если вы работаете в маленькой компании, в одном географическом месте и вам нужна только Интернет-почта — НЕ ИСПОЛЬЗУЙТЕ ЭТОТ ПРОДУКТ!!!»

Итак, я подошел к тому, о чем хотел сказать.

Технология — *любая* технология — обладает ценностью только в контексте той проблемы, которую она призвана решать. Другими словами, технология хороша лишь в том случае, если она подходит для вашей конкретной задачи. Если вы согласны с этой предпосылкой[2], вы увидите, что из нее следует ряд неизбежных затруднений.

В частности, это усложняет написание книг о любой конкретной технологии. Автор не знает, какие проблемы стоят перед его читателями. Следовательно, перед ним открываются два пути. Первый — предлагать общие решения и надеяться, что они вам подойдут (или еще хуже: объявить свои решения «единственно правильными» и не предупредить о возможных последствиях, если вдруг что-нибудь пойдет не так). Второй путь — рассказать о новой технологии столько, чтобы вы могли самостоятельно оценить ее в контексте своей ситуации.

Второй вариант труднее, но я все равно попробую. А пока запомните первое правило, которым необходимо руководствоваться на протяжении всей книги.

Все сказанное мной (или любым другим автором) следует воспринимать скептически и оценивать в контексте конкретной ситуации.

Вторая проблема еще важнее. Предлагая свои технологические решения, Microsoft стремится определить потребности и потенциальный рынок, удовлетворить эти потребности и продать свой товар. Я искренне верю, что большинство программистов Microsoft стремится написать качественный продукт и осчаст

[1] Потерянного времени, конечно, жалко, но это не так уж важно.

[2] А если не согласны — немедленно верните книгу в магазин; вам вряд ли понравится то, что последует дальше.

ливить своих коллег-программистов... но я верю и в то, что Microsoft стремится продавать программные продукты и зарабатывать на этом деньги[1]. Хотя по статистике продукты Microsoft удовлетворяют большую часть этих потребностей, это не значит, что любой предлагаемый продукт наилучшим образом подойдет для вашей конкретной задачи. Следовательно, вы должны запомнить и другое правило.

Все сказанное Microsoft (или любой другой коммерческой компанией) следует воспринимать скептически и оценивать в контексте конкретной ситуации.

Не правда ли, звучит несколько кощунственно?

Руководствуясь этими правилами, давайте познакомимся с .NET, но не с самой технологией — к этому вы еще не готовы, а скорее с контекстом ее применения.

.NET — реальность или маркетинг?

Когда-то я прослушал общий курс решения задач (problem solving). Как ни странно, приступая к решению любой проблемы, необходимо прежде всего выяснить, существует ли она. Просто удивительно, как много из них решается простым и здоровым сном. Похожий вопрос относится и к новым технологиям Microsoft — а существуют ли они?

На мой взгляд, Microsoft выпускает «продукты» двух типов.

Первый тип — это действительно продукты без кавычек; то, что можно купить и использовать. Visual Basic — продукт; Windows 2000 — продукт; Microsoft Transaction Server — тоже продукт, как и Microsoft Message Queue[2].

Если бы Microsoft продавала только такие продукты, мне бы не пришлось писать этот раздел.

Но существует и другая категория «продуктов» от Microsoft, которые не создаются руками программистов, а выходят из отдела маркетинга[3].

Вспомните, что произошло несколько лет назад, когда Microsoft «открыла» Интернет. В один прекрасный день выяснилось, что Microsoft начинает отставать от Netscape, Sun и других компаний, занимающихся интернет-технологиями. Немедленно под фанфары появилась новая технология ActiveX.

Что такое ActiveX?

Это OLE2 с новым названием.

Вот так. Вся новая «интернет-стратегия», все принципиальные технологические достижения обернулись новым названием для того, что существовало и раньше.

Я вас еще не убедил?

---

[1] Как ни странно, кое-кто воспринимает эти слова как клевету на Microsoft. Однако это коммерческая организация, и если бы ее целью не было зарабатывание денег, у держателей акций возникли бы обоснованные претензии.

[2] MTS и MSMQ изначально были продуктами. Помните об этом, когда будете читать дальше.

[3] Реорганизации Microsoft происходят так часто и отличаются такой сложностью, что человеку постороннему практически невозможно разобраться, что, когда и зачем делается и кто занимается программированием (и занимается ли им кто-нибудь вообще). Иначе говоря, мои комментарии с четким разделением программирования и маркетинга упрощены ради ясности изложения и не отражают реальных событий, происходящих в Microsoft.

Microsoft представила ряд важных компонентов, упрощающих создание масштабируемых COM-объектов в многоуровневых приложениях. Microsoft Transaction Server обеспечивал обработку транзакций и совместный доступ к данным, а Microsoft Message Queue — надежный асинхронный обмен сообщениями. В один прекрасный день появилась технология COM+, которая делала то же самое (и еще кое-что).

А что собой представляла технология Windows DNA?

Вы заметили, что все упоминания о Windows DNA исчезли буквально за считанные дни, а на смену им мгновенно пришло нечто, называемое .NET Enterprise Services? Более того, Microsoft даже опубликовала список соответствия терминов DNA и .NET.

Подозреваю, что все эти названия изобретают специалисты по маркетингу, чтобы связать разные технологии и представить их как часть единой стратегии, подходящей на все случаи жизни, и в этом нет ничего страшного. Но сюжет и термины изменяются так часто, что бывает трудно определить, чем является «принципиально новая стратегическая инициатива»: реальным технологическим достижением, новой оберткой для существующих компонентов или очередным маркетинговым ухищрением?

Значит, любые исследования в области .NET следует начинать с вопроса: что здесь реально, а что следует отнести на счет рекламной шумихи?

Когда вы будете читать эти строки, вероятно, информация из других источников уже убедит вас в том, что .NET действительно является новой технологией, а не простым упражнением в маркетинговой риторике.

Но даже сейчас вы еще не представляете, какие крупные изменения сулит пришествие .NET. Одно из последствий такой маркетинговой политики напоминает старую притчу о мальчике, который слишком часто кричал: «Волк! Волк!» — в Microsoft давно кончились все хвалебные эпитеты для действительно новых и оригинальных технологий.

Попробую восполнить этот пробел.

- Все, что вы читали о .NET, не дает даже отдаленного представления о «реальности» этой технологии, о ее принципиальной новизне.
- Пришествие .NET приведет к крупным изменениям на уровне парадигмы Windows-программирования.
- Для большинства программистов Visual Basic .NET означает радикальные изменения в стратегии и архитектуре программ.
- Всем программистам Visual Basic придется основательно потратиться на переобучение.

Список можно продолжить (и я сделаю это ниже), но вы, вероятно, уже поняли мою мысль. Для программистов Visual Basic внедрение .NET приведет к более глобальным изменениям, чем переход от 16-разрядных версий Windows к 32-разрядным. Масштаб изменений сравним разве что с переходом с DOS Basic на Visual Basic.

Однако .NET не только несет серьезные испытания для программистов Visual Basic, но и открывает замечательные возможности.

- .NET ставит программистов Visual Basic в равные условия с программистами C++ в отношении широты возможностей языка[1].
- .NET превращает программистов Visual Basic в разработчиков интернет-приложений, не требуя от них практически никаких усилий. Научившись создавать в VB .NET приложения Windows, вы окажетесь всего в одном шаге от создания Интернет-приложений.
- .NET предоставляет в распоряжение программиста Visual Basic огромную библиотеку объектов, ускоряющую процесс разработки и расширяющую его возможности.

Признаюсь, когда я наконец-то начал понимать, что же делает Microsoft, я был ошарашен. Потрясение не прошло до сих пор. Таким образом, прежде чем переходить к непосредственному описанию технологии, необходимо ответить на ряд общих вопросов.

[1] А по эффективности программирования программисты Visual Basic всегда опережали программистов C++.

VB .NET: взгляд без страха и паники

На последнем году учебы в университете я прошел обязательный курс, который назывался «Индивидуальные и организационные аспекты информационных технологий» (или что-то вроде этого). Тогда я не понимал, кому он нужен. Теперь мне ясно, что это был один из самых важных предметов за все время учебы.

Нельзя говорить об изучении новых технологий и забывать о том, как эти технологии влияют на людей.

В конце предыдущей главы я указал на то, что архитектура .NET представляет собой радикальное изменение парадигмы — это путь программирования, который окажется новым для большинства программистов Visual Basic. Хотя я еще не готов к подробному обсуждению технологии, позвольте мне кратко представить те изменения, о которых пойдет речь.

Господи, они сломали VB!

В основу архитектуры .NET заложена новая исполнительная среда Common Language Runtime (CLR). Тема исполнительной среды вообще является «больной» для многих программистов VB. Сначала казалось, что Visual Basic является «неполноценным» языком, поскольку он использовал огромную исполнительную среду, которая вроде бы не нужна для программ C++. Потом выяснилось, что многие программисты C++ тоже работают с большими исполнительными средами (например, библиотеками поддержки MFC[1]). Только программисты, работающие на уровне API (Application Programming Interface), и те, кто способен освоить все тонкости ATL (Active Template Library), создают компоненты и приложения, не требующие исполнительных сред. По своим размерам исполнительная среда CLR значительно превосходит старые среды VB и MFC[2]. Если верить Microsoft, при разработке CLR были поставлены следующие цели:

---

[1] Microsoft Foundation Classes.

[2] Сравнение усложняется тем, что в этих языках используется не только базовый модуль среды (такой, как 1,3-мегабайтная исполнительная среда VB6 или 1-мегабайтная среда MFC), но и разные вспомогательные DLL поддержки OLE или C. Точный размер .NET мне не известен, но предполагается, что он будет существенно больше и счет пойдет на десятки мегабайт.

- упростить создание объектов, которые могли бы использоваться в разных языках;
- упростить создание масштабируемых, защищенных от ошибок компонентов и приложений (в частности, это означает отсутствие проблем с многопоточностью и утечкой ресурсов и памяти);
- сделать возможным создание хорошо защищенных компонентов и приложений;
- упростить создание как web-компонентов и web-приложений, так и традиционных приложений Windows.

Мы еще вернемся к этим и другим целям[1], а пока давайте посмотрим, как они повлияли на синтаксис языка.

Язык .NET-программирования должен поддерживать следующие возможности (наряду с другими):

- наследование;
- свободная потоковая модель;
- поддержка всех типов переменных, определенных в CLR;
- атрибуты и метаданные (не беспокойтесь, вскоре вы узнаете, что это такое).

Эти возможности входят в спецификацию Common Language (CLS, Common Language Specification) и должны поддерживаться всеми .NET-языками.

Visual Basic не поддерживает ни одну из этих возможностей.

Перед разработчиками Microsoft встала интересная задача: как «подправить» существующие языки для работы и обеспечить поддержку новой среды CLR, на которой строится вся архитектура .NET. Перед ними открывалось несколько возможных путей.

- Дополнение языка специальными расширениями. Для Visual C++ был выбран именно такой путь[2].
- Создание нового языка, ориентированного на CLR. Так появился язык C#[3], о котором мы еще поговорим.
- Переработка языка в соответствии с требованиями CLR, невзирая на возможные нарушения совместимости с предыдущими версиями языка. Именно этот вариант был выбран для Visual Basic .NET.

Затем разработчики Microsoft приняли смелое, но весьма спорное решение. Если уж обратную совместимость сохранить все равно не удается, так почему бы не привести язык в порядок? Под «приведением в порядок» я имею в виду чистку синтаксиса и реализацию новых возможностей (например, жесткой проверки типов), которых так долго не хватало профессиональным программистам. Даже предвзятый поклонник VB не рискнет утверждать, что синтаксис команды `Line` выглядит логично. Часть 3 этой книги полностью посвящена специфике измене-

---

[1] Microsoft также провозгласила, что приложения, написанные для CLR, будут работать на любой платформе с поддержкой этой исполнительной среды — слышались туманные намеки, что CLR будет существовать в Apple, Linux и даже на карманных компьютерах Palm. Насчет Apple я готов поверить, а что касается остального — там посмотрим.

[2] Я лишь в общих чертах посмотрел, как эти расширения были реализованы в VC++. Признаюсь, первое впечатление не слишком благоприятное. Выглядит громоздко и неудобно.

[3] Произносится «си-шарп».

ний языка и адаптации готовых программ. А пока я перечислю самое важное, о чем необходимо помнить.

- VB .NET — это Visual Basic, но это *не тот* Visual Basic, который развивался от VB1 к VB6. Это другой язык.
- Код VB6 невозможно загрузить в VB .NET без предварительного преобразования.
- Microsoft предлагает использовать мастера Migration Wizard, который при загрузке проектов VB6 в Visual Studio .NET автоматически преобразует код в VB .NET. Насколько хорошо будет работать этот мастер, пока неизвестно[1].
- В VB .NET используется новый механизм форм. Кроме несомненных изменений в программировании, также следует ожидать неочевидных изменений в поведении форм.
- Новые возможности VB .NET открывают путь к новой, улучшенной архитектуре программ.

Итак, мы вплотную подошли к теме следующего раздела...

Все, что вы знали, устарело

Я начал эту главу с упоминания человеческого фактора технологии. Вероятно, типичный читатель этой книги программирует на Visual Basic 5 или 6, а его стаж составляет от нескольких месяцев до такого же срока, как и у меня[2]. Как же отреагирует программист на известие о том, что язык, в котором он достиг определенного мастерства, претерпел столь радикальные изменения?

Возникает немало вопросов.

- Как же сертификационные экзамены, которые мы сдавали? Неужели все они стали бесполезными в контексте VB .NET?
- Сохранятся ли наши рабочие места, когда наши компании или клиенты перейдут на .NET?
- Как изучить принципиально изменившуюся технологию за ограниченный промежуток времени, если мы по-прежнему проводим долгие часы за работой по старой технологии?

Я посетил немало конференций, говорил со многими программистами и сам размышлял над этими вопросами, чтобы понять реакцию программистов на подобные технологические сдвиги.

- Паника: «Мы не сможем уследить за всеми изменениями, и у нас не хватит времени на то, чтобы все узнать».
- Страх: «Мы рискуем своими карьерами, поскольку мы не сможем изучить все необходимое. Наверняка поблизости найдется кто-нибудь, кто знает больше».

---

[1] В части 3 я уделяю Migration Wizard значительно больше внимания. То, что я видел, выглядит неплохо, но вряд ли мастер окажется настолько хорош, чтобы преобразование больших или использующих нетривиальные возможности приложений обошлось без существенной дополнительной работы и тестирования.

[2] На всякий случай уточню, что я программирую на VB с момента выхода бета-версии VB1.

- Любопытство: «Большинство из нас занимается программированием, прежде всего, потому, что мы любим это занятие, нам нравится изучать и применять новые технологии».

Наверное, многим читателям даже не хочется думать о таких деликатных вещах. А кому-то покажется, что меня заносит — ну какое отношение эта психологическая дребедень имеет к изучению VB .NET?

Самое непосредственное.

Ваше личное отношение к новой технологии сильно влияет на то, насколько быстро и хорошо вы ее изучите.

Если смотреть на VB .NET с затаенным страхом, вам будет нелегко избавиться от балласта устаревших приемов и подходов. Многие нововведения покажутся ошибочными или бессмысленными. Возможно, вы присоединитесь к тем, кто осудит изменения и потребует, чтобы Microsoft вернула язык в прежнее состояние... или даже к противникам всей платформы .NET.

На протяжении всей книги я буду критиковать некоторые решения Microsoft и даже укажу на пару изменений, которые мне кажутся неудачными. Но в целом я положительно отношусь к новой технологии и ее возможностям. Давайте рассеем все опасения и попробуем взглянуть на .NET с позитивной точки зрения. Пусть эта точка зрения будет критической и не лишенной обоснованного скептицизма — главное, чтобы страх не заставлял нас попусту критиковать то, чего мы еще не понимаем.

А теперь я постараюсь объяснить, почему вы не должны бояться. Пока поверьте мне на слово — обоснования будут приводиться по мере изложения материала.

1. У вас есть время: внедрение .NET сопряжено с глобальными изменениями парадигмы программирования. Освоить такую технологию за день-другой невозможно. В этом заключается одна из причин, по которой Microsoft так широко распространяет сведения о новой технологии и бета-версии программ. Понятно, что освоение .NET потребует немалых усилий. Все остальные находятся в такой же ситуации, как и вы.

2. VB .NET в конечном счете помогает лучше программировать. Эта технология упрощает тестирование и отладку, а также долгосрочное сопровождение приложений. Ваши задачи будут решаться проще и быстрее, чем сейчас.

3. .NET ставит всех в равные условия. Для новичков и программистов средней квалификации, которые не представляли, как им угнаться за экспертами, это бесценный подарок — теперь все начинают примерно с одного уровня. Конечно, кое-кто обладает преимуществами. Например, образование в области компьютерных технологий поможет быстрее усвоить некоторые концепции, но в целом перед всеми открывается замечательная возможность приобщиться к передовым технологиям[1].

[1] Конечно, этот аспект .NET не радует многих сегодняшних «экспертов» и «гуру», — ведь им придется прилежно работать, чтобы достичь мастерства в новой области, не говоря уже о возросшей конкуренции. Я добровольно сложил с себя звание «гуру» в тот момент, когда впервые увидел .NET, но постараюсь снова стать достойным этого титула.

Программирование как бизнес

Помимо человеческого фактора при анализе .NET необходимо учитывать деловые факторы. Я уже несколько раз упоминал о том, что любая технология должна оцениваться в контексте конкретных практических задач. Если задуматься, это скорее деловой, нежели технический критерий. Надеюсь, многие читатели согласны со мной, но кто-то наверняка удивится: при чем здесь деловые решения, если речь идет об изучении языка?

Раскрою небольшой секрет... я получил высшее образование в области компьютерных технологий, но за время учебы в Калифорнийском университете я также получил степень бакалавра в области электроники. Одним из важнейших предметов была инженерная экономика, обязательный курс для всех студентов-электронщиков. Из этого курса я узнал, что профессиональный инженер всегда должен осознавать и учитывать экономические последствия своих решений — то есть видеть за технологией бизнес. Полагаю, это правило в равной степени относится и ко всем программистам[1].

История Visual Basic доказывает этот тезис. Почему Visual Basic пользуется такой популярностью, хотя в течение многих лет программисты (особенно программисты C++) называли его «игрушечным языком»? Ответ ясен: потому что на Visual Basic разработка приложений обходится дешевле, чем на любом другом языке. Visual Basic стал самым распространенным языком Windows-программирования по чисто экономическим причинам. Многие программисты C++ (в том числе и я) переходили на Visual Basic, потому что, несмотря на отсутствие некоторых возможностей, которые мы считали важными (например, жесткую проверку типов), написание, отладка и тестирование программного кода происходили на порядок быстрее, чем в C++. Изобилие общедоступных и недорогих программных компонентов VBX[2], а затем и ActiveX (OCX) позволяло наделять приложения Visual Basic широкими возможностями при очень малых затратах.

Взгляните на любой продукт или технологию от OS/2 до Windows, от ASP (Active Server Pages) до WebClasses. Признание и успех технологии почти всегда определяется человеческими, политическими и экономическими факторами, а вовсе не технологическими!

Если вы так и не поняли, почему эта книга не начинается с примеров кода или описания возможностей VB .NET, я подведу краткий итог своих рассуждений. Успех VB .NET, ваш успех в освоении этой технологии и ваши решения о том, стоит ли изучать и принимать эту технологию, в основном определяются человеческими, политическими и экономическими факторами. В большинстве технических книг автор обращает все внимание только на технологию, а с остальными проблемами читателю приходится разбираться самостоятельно. В этой книге я хочу с самого начала обсудить эти важные вопросы, еще до того, как вы получите хотя бы первое представление о технологии, чтобы потом ссылаться на них в последующих главах.

Учитывая все сказанное, мы переходим к следующей теме — стратегическим решениям, которые вам придется принимать в отношении VB .NET и архитектуры .NET.

---

[1] Кроме людей, программирующих для собственного удовольствия — у них другие критерии и приоритеты.

[2] VBX = Visual Basic Controls — предшественник современных элементов ActiveX.

Стратегии перехода

Вероятно, вы уже встречали информацию о .NET в журнальных статьях или на сайте Microsoft. Может быть, вы принимали участие в семинарах «Дней программиста» или других конференциях. Но даже если вы впервые столкнулись с .NET в этой книге, несомненно, у вас уже возникли важные вопросы.

- Сколько времени уйдет на изучение этой технологии?
- Следует ли нашей компании немедленно переходить на .NET или лучше подождать?
- По какому сценарию должно происходить внедрение этой технологии?
- Сколько будет стоить обучение персонала?
- Как это улучшит работу наших приложений (и улучшит ли вообще)?
- Нужно ли адаптировать существующие приложения?

Не рассчитывайте, что я отвечу на эти вопросы — они слишком сильно зависят от вашей конкретной ситуации. Вместо этого мы поговорим о тех факторах, которые необходимо учитывать при поиске ответов. По мере знакомства с технологическими аспектами .NET вы сможете воспользоваться своими новыми знаниями и найти ответы самостоятельно.

Сроки

Когда я приступил к работе над этой главой, официальная бета-версия Visual Studio .NET еще не распространялась. Microsoft уже успела опубликовать немало маркетинговых сведений о .NET, выпустила несколько статей с описанием технологии и архитектуры, а также распространила технологический обзор на своей Конференции профессиональных разработчиков. Но даже сейчас[1] еще неизвестно, когда выйдет окончательная версия Visual Studio .NET. Вероятно, это произойдет во второй половине 2001 года. Следовательно, вся эта новая технология вполне реальна, но говорить о ее массовом применении пока рановато.

---

[1] На стадии бета-2.

У ранней рекламной кампании Microsoft есть и оборотная сторона: программисты могут решить, будто VB6 уже устарел, поэтому нужно побыстрее изучать VB .NET и разбираться, как адаптировать существующие приложения для него. Конечно, это полная ерунда.

Даже после выхода окончательной версии программистам понадобится немало времени на освоение новой технологии и ее практическое внедрение. Масштаб изменений настолько велик, что на переход к VB .NET потребуются годы, а не месяцы — а для некоторых приложений это вообще бессмысленно.

Что же это означает лично для вас?

1. Не выбрасывайте свою версию Visual Basic. Вероятно, она вам еще понадобится в течение некоторого времени.
2. Не паникуйте! У вас будет время на то, чтобы освоить эту новую технологию и привыкнуть к ней.
3. Читайте книги, учитесь и экспериментируйте перед тем, как приступать к проектированию приложений и непосредственному программированию. Это увеличит вероятность того, что ваши приложения окажутся правильно спроектированными и будут в полной мере использовать новые возможности .NET.

Адаптация кода

Один из самых важных вопросов, на который вам предстоит ответить, — нужно ли адаптировать код существующих приложений для VB .NET. В предыдущих версиях Visual Basic принять решение было несложно. Переходы от VB1 к VB4 (16-разрядная версия) у большинства программистов происходили легко; лишь немногим из них приходилось откладывать переход на новую версию из-за проблем совместимости. Переход на 32-разрядный Visual Basic вызвал больше затруднений, однако основные задержки были связаны с ожиданием выхода компонентов ActiveX в новых OCX-версиях (вместо 16-разрядных VBX-версий). Переходы от 32-разрядного VB4 к VB6 также проходили относительно гладко. Хотя Microsoft приложила огромные усилия к тому, чтобы обеспечить обратную совместимость, решить эту задачу полностью так и не удалось, а в процессе обновления нередко появлялись хитроумные ошибки. Из-за этого даже в наши дни некоторые программисты продолжают использовать VB5.

А теперь представьте, как будет проходить адаптация кода на язык, в котором совместимость была нарушена намеренно?

Еще раз подчеркну, что я не критикую решение Microsoft о нарушении совместимости — на это у них были веские причины. Но это означает, что адаптация существующего кода в VB .NET потребует основательных затрат.

Во сколько обойдется адаптация готовых программ? Все зависит от качества вашего кода и от того, насколько хорош будет Migration Wizard.

Даже если Migration Wizard сможет правильно преобразовать 95 % программы из 10 000 строк, по крайней мере 500 строк придется просматривать, анализировать и обновлять вручную, не говоря уже о возможности появления нетривиальных ошибок, обусловленных изменением архитектуры или самим процессом

адаптации. По крайней мере, все адаптированные приложения и компоненты придется тщательно тестировать.

Что это значит для вас как для разработчика? То, что вы должны сопоставить затраты на адаптацию с преимуществами, которые она дает. Неизбежно выяснится, что в некоторых ситуациях адаптация существующего кода в VB .NET экономически невыгодна, поэтому Microsoft следовало бы в течение некоторого времени продавать и поддерживать Visual Basic 6[1].

Аргументы «за» и «против»

Причины перехода на .NET могут сильно различаться в зависимости от того, какие приложения и компоненты вы разрабатываете. Ниже перечислены некоторые ключевые факторы, влияющие на внедрение новой технологии.

- Архитектура .NET очень хорошо подходит для создания новых web-приложений. По возможностям масштабирования и управления ресурсами и памятью она заметно превосходит предыдущие технологии Visual Basic 6 и ASP.
- Архитектура .NET не требует установки исполнительных модулей .NET на всех компьютерах, использующих компоненты или приложения, написанные с использованием Common Language Runtime (это относится и ко всем компонентам и приложениям VB .NET).
- VB .NET достаточно сильно отличается от Visual Basic 6, и его изучение потребует от программистов немалых усилий.
- VB .NET обеспечивает потенциальное улучшение быстродействия в том случае, если свободная потоковая модель благоприятно сказывается на работе компонентов. Ускоренное исполнение кода в языках .NET (по сравнению со сценарными языками) также вносит свой вклад в повышение быстродействия приложений.
- Вероятно, VB .NET не приведет к повышению быстродействия, а то и понизит его в тех случаях, когда архитектурные изменения (свободная потоковая модель или переход от сценарного языка к компилируемому) не влияют на работу программы.
- Приложения и компоненты VB .NET могут пользоваться всеми многочисленными ресурсами Common Language Runtime, в том числе библиотеками и компонентами.

Все эти факторы позволяют высказать обоснованные предположения относительно того, в каких областях и насколько быстро будет задействована технология VB .NET.

---

[1] Я не слышал от Microsoft никаких официальных заявлений относительно будущего VB6, однако мое общение с сотрудниками Microsoft показывает, что они понимают суть проблемы, и некоторые высказываются за продолжение поддержки VB6 даже после выхода VB .NET.

Новые серверные приложения

Здесь и думать не о чем. Если вы работаете над новым серверным приложением, будь то web-приложение ASP.NET, код принятия решений на стороне сервера или бизнес-логика среднего уровня (middle tier), переход на VB .NET вскоре после выхода окончательной версии (или даже на стадии разработки поздней бета-версии) выглядит вполне логично. Все затраты сводятся к времени и расходам на обучение программистов. Правда, это может обойтись недешево, однако затраты быстро оправдаются дополнительными преимуществами — такими, как повышение надежности и доступ к новым возможностям .NET. Поскольку приложение работает только на серверах, затратами на установку и внедрение исполнительной системы можно пренебречь.

Старые серверные приложения

Аргументы примерно те же, что и в предыдущем случае, однако появляются новые затраты на адаптацию кода. Как упоминалось выше, эти затраты могут быть весьма существенными. К счастью, архитектура .NET спроектирована таким образом, что она хорошо работает с существующей технологией COM и компонентами (хотя, как вы вскоре узнаете, сама технология .NET не основана на COM). Это означает, что вы сможете избирательно адаптировать некоторые компоненты своего приложения для .NET, чтобы воспользоваться новыми возможностями, улучшенным быстродействием или масштабируемостью, и продолжить применение существующих компонентов COM в тех аспектах вашего решения, которые не выиграют от адаптации, или же если она обойдется слишком дорого.

В этом случае я бы порекомендовал пораньше начать обучение персонала, проанализировать различные компоненты системы и определить, какие из них стоит адаптировать для новой технологии. Начинайте с небольших, простых компонентов, чтобы накопить практический опыт. Еще лучше начинать с компонентов, которые добавляются для реализации новых возможностей в вашем приложении.

Сейчас еще рано что-либо уверенно утверждать, но я рекомендую сосредоточить внимание на компонентах, не обладающих пользовательским интерфейсом. Конечно, ничто не мешает объединять COM с компонентами .NET с пользовательским интерфейсом, однако это повышает риск возникновения всевозможных проблем и нетривиальных проявлений несовместимости в процессе адаптации. Таким образом, основное внимание следует уделять серверным приложениям, web-серверам и среднему уровню. Еще раз подчеркну, что мои опасения относительно совмещения COM с компонентами .NET, обладающими пользовательским интерфейсом, относятся только к адаптации кода, когда приходится учитывать весьма тонкие изменения в поведении программы. Использование существующих компонентов COM в новых проектах .NET (и наоборот) проблем не вызывает. Microsoft приложила немало усилий к тому, чтобы компоненты COM и .NET хорошо работали друг с другом.

Клиентские приложения

Главным фактором при принятии решения об использовании VB .NET для создания компонентов клиентской стороны (будь то автономное приложение или браузерный компонент) должна быть ваша уверенность в том, что клиенты уже установили исполнительную среду .NET, собираются установить ее или уже перешли на версию Windows, в которую она входит. Окончательный размер дистрибутива исполнительной части мне пока неизвестен, но скорее всего он будет настолько большим, что минимальным требованием для пересылки файлов клиенту будет наличие Zip-накопителя, дисковода CD-ROM или модема DSL. Передача данных на гибких дисках и по обычному модему практически исключается.

Если на компьютере отсутствует исполнительная среда .NET, приложения и компоненты VB .NET просто не будут работать.

Однако наличие исполнительной среды не только открывает доступ к новым возможностям, но и позволяет организовать безопасную установку вашего приложения и его компонентов. Вам не придется беспокоиться о проблемах «кошмара DLL» (DLL Hell), так часто встречающихся при современной технологии[1]. Кроме того, вы сможете создавать защищенные, проверяемые компоненты и приложения — это особенно важно, поскольку распространение компьютерных вирусов заставляет конечного пользователя уделять все большее внимание безопасности системы.

После того как Common Language Runtime войдет в поставку операционной системы (а это практически неизбежно), VB .NET станет весьма привлекательной платформой для клиентских приложений и браузерных компонентов[2]. В частности, меня совсем не удивит, если .NET во многих ситуациях затормозит распространение клиентских приложений с расширенной функциональностью, поскольку в .NET надежная установка такого приложения ничуть не сложнее просмотра web-страницы.

Переход на .NET не должен проходить так драматично, как переход с 16- на 32-разрядные приложения, поскольку среда CLR должна работать на всех существующих 32-разрядных версиях Windows. Тем не менее, если учесть разнородный состав пользователей и систем, в которых они работают, становится ясно, что на первых порах VB .NET не будет хорошей платформой для большинства клиентских приложений.

Вероятно, внедрение .NET в клиентской области начнется с малых компаний, в которых можно проконтролировать состояние рабочего стола любого пользователя, при необходимости быстро свернуть исполнительную среду и разобраться с любыми возникшими проблемами. Разработчики коммерческих программ, скорее всего, будут избегать .NET до тех пор, пока накопившаяся информация не позволит обоснованно судить о применимости Common Language Runtime в широком спектре используемых систем.

---

[1] Для тех, кто еще не знает этот термин, «кошмаром DLL» называется ситуация, при которой установка в системе обновленного компонента приводит к сбоям в работе других приложений. Ниже мы подробно рассмотрим эту проблему и процесс установки вообще.

[2] Существует и такая эффектная возможность, как создание приложений, работающих на web-сервере или в локальной системе с практически одинаковым пользовательским интерфейсом.

А как же C#?

Многие программисты Visual Basic страдали от легкого комплекса неполноценности (а программисты C++ называли Visual Basic «игрушечным языком» и развивали этот комплекс). Одни не выдержали и ушли изучать C++, другие переключились на Java[1].

С приходом VB .NET неизбежно возникает вопрос, не следует ли программистам VB перейти на C# или C++.

Если вам решительно не хочется иметь дела с .NET, появляется дополнительный аргумент в пользу перехода на C++. Этот язык, как и прежде, обеспечивает превосходную поддержку создания Windows-приложений и компонентов, использующих библиотечные модули MFC или (с несколько большими усилиями) вообще обходящиеся без поддержки исполнительной системы. C++ остается общепринятым фаворитом для разработки специализированных приложений, таких как приложения ISAPI, использующие ATL Server. Впрочем, опытный программист VB, вероятно, все же предпочтет остаться с VB6.

Если вы решите, что платформа .NET в полной мере отражает стремление Microsoft создать платформу программирования нового поколения, что она не только просуществует в течение долгого времени, но и продолжит свое развитие и будет поддерживаться на уровне текущего Win32 API, я настоятельно рекомендую остановить выбор на C# или VB .NET вместо C++ с управляемыми расширениями (managed extensions).

C# разрабатывался вместе с Common Language Runtime как «идеальный» язык для среды .NET. Возможность создания высокоэффективного проверяемого кода была заложена в него изначально.

А как же VB .NET? Когда стало ясно, что адаптация Visual Basic к .NET потребует нарушения языковой совместимости с VB6, в Microsoft решили пойти до конца и сделать VB .NET полностью .NET-совместимым языком. Также было решено «почистить» язык, видоизменить или исключить из него те самые факторы, которые обычно вызывали основную критику у программистов C++.

Другими словами, если вы собираетесь использовать .NET, между C# и VB .NET нет практически никаких различий ни по быстродействию, ни по набору поддерживаемых возможностей. C# дает большую свободу в использовании «ненадежного» кода — впрочем, этой низкоуровневой возможностью следует пользоваться лишь при крайней необходимости (более подробно эта тема рассматривается ниже). Но даже это не является достаточной причиной для того, чтобы отдать предпочтение C# перед VB .NET, поскольку в приложении можно легко смешивать разные языки[2].

---

[1] Далее я не собираюсь обсуждать Java. Мой опыт знакомства с этим языком не позволяет обоснованно оценить достоинства и недостатки Java как альтернативы VB6, VB .NET или C#.

[2] Собственно, простота использования компонентов и программного кода других языков и была одной из главных причин для создания Common Language Runtime. Например, класс VB .NET не только «видит» объекты C#, но и может создавать от них производные объекты путем наследования. Насколько мне известно, в настоящее время разрабатываются восемнадцать .NET-совместимых языков!

В некоторых областях синтаксис VB .NET кажется более четким и наглядным, чем аналогичный синтаксис C#. Впрочем, это не означает, что из-за этого программисты должны переходить на VB .NET вместо C#.

В конечном счете вы можете придерживаться того синтаксиса, к которому привыкли[1]. Если вы программируете на Visual Basic, выбирайте VB .NET. Для программистов C++ и Java более привычным покажется C#.

Альтернативы .NET

Некоторые читатели посмотрят на грандиозные изменения, связанные с переходом на .NET, и задумаются — а не лучше ли решать проблемы при помощи технологий и языков, не имеющих отношения к Microsoft? Я бы охотно помог советом в этой области, но, по правде говоря, я недостаточно хорошо ориентируюсь в этих альтернативах, чтобы их сравнивать. Более того, существующие платформы и языки изменяются так быстро, что любые сравнения просто не имеют смысла. Позвольте мне привести пару примеров.

Многие программисты VB поглядывают на Java и громкие обещания межплатформенной совместимости. К сожалению, большая часть совместимости так и остается на уровне обещаний, изрядно подпорченных корпоративной политикой, юридическими тонкостями и не лучшим быстродействием практических реализаций. Кстати говоря, ничто не мешает реализовать Java для .NET, но мне кажется, что при использовании .NET бессмысленно отдавать предпочтение Java перед C# (если только .NET-версия Java не будет реализована лучше других языков).

Microsoft особо подчеркивает как поддержку в .NET стандартов SOAP (Simple Object Access Protocol) и XML (Extensible Markup Language), так и то, что язык C# тоже должен быть оформлен в качестве стандарта. Впрочем, пока это всего лишь обещания, а переход от сегодняшнего состояния к окончательной стандартизации — дело политическое и неоднозначное.

Я не уверен, что для конкретного приложения можно подобрать «оптимальную» платформу, поскольку никто толком не знает, как будут развиваться события. Думаю, можно предположить, что в течение некоторого времени[2] платформа .NET будет вполне жизнеспособна, но никто не запрещает вам исследовать другие возможности, даже если золотые крупинки стоящей информации придется выискивать в горах рекламного порожняка.

В этой книге предполагается, что вы, по какой бы причине это ни произошло, остановили свой выбор на технологии Microsoft. Свою задачу я вижу в том, чтобы по возможности облегчить ваш переход от сегодняшней технологии к .NET[3].

---

[1] Снова экономика. Зачем попусту тратить драгоценное время на изучение нового синтаксиса?

[2] Когда-то я работал в компании, которая одна из первых перешла на работу в графической среде. Мне пришлось выбирать между платформой GEM (Digital Research) и Microsoft Windows 1. Как вы, наверное, догадались, я сделал правильный выбор, но в тот момент это было далеко не бесспорное решение, в чем другие компании убедились на собственном печальном опыте.

[3] После чтения этой книги ваших познаний в области .NET будет достаточно для того, чтобы самостоятельно исследовать другие альтернативы.

Я бы порекомендовал вам исследовать и оценить другие варианты в контексте ваших собственных потребностей.

Следующий шаг

Надеюсь, в первых трех главах вы нашли достаточно информации для размышления. Также хочется верить, что эти главы помогут вам сосредоточиться на своих практических потребностях при изучении дальнейшего материала. Итак, мы переходим к части 2, посвященной фундаментальным концепциям, с которыми должен ознакомиться каждый программист перед тем, как создавать свой первый компонент или приложение VB .NET.

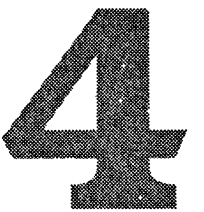
Часть 2

Концепции

Мастерство достигается не запоминанием всех мельчайших подробностей темы, а доскональным пониманием тех базовых концепций, на которой она основана.

Дан Эпплман, самоуверенный автор

.NET в практическом контексте



Наверное, вам не терпится увидеть хотя бы строчку программного кода. Потерпите еще немного — вскоре мы доберемся и до программ.

Как я уже говорил, если бы эта книга была типичным учебником для начинающих программистов, изучающих VB .NET, она бы имела совершенно иную структуру. Я бы начал с описания основных концепций и синтаксиса языка.

Но эта книга предназначена для программистов VB6 среднего и высокого уровня. Честно говоря, я нисколько не сомневаюсь в том, что вы легко усвоите синтаксис VB .NET. Вы уже знаете, что такое переменные и как работают циклы `For...Next`. Вы уже умеете работать с библиотеками объектов по прошлому опыту использования ActiveX DLL и элементов ActiveX. Вы уже знаете, как использовать методы и свойства объектов.

Я не стану обижать читателя, ставя под сомнение его квалификацию[1], и описывать те аспекты Visual Basic, которые либо совпадают в обоих вариантах языка, либо обходятся тривиальными различиями. В части 3 этой книги мы рассмотрим большое количество конкретных изменений, что поможет вам быстрее привыкнуть к новому синтаксису.

Нет, такие изменения меня совершенно не беспокоят.

Меня беспокоит другое: изменения, связанные со сменой парадигмы; изменения, влияющие на архитектуру программ .NET, связанные с теми концепциями, которые покажутся новыми для большинства программистов VB .NET. Что конкретно имеется в виду?

- Некоторые программисты VB не успеют в полной мере понять необходимость синхронизации перед тем, как проектировать свое первое многопоточное приложение или компонент, поскольку раньше VB6 скрывал от них эти сложности.
- Некоторые программисты VB испытают такой энтузиазм при виде полноценного наследования, что захотят использовать его на практике[2].

---

[1] Я вообще никоим образом не собираюсь обижать читателя, но бывает всякое. Если вы все же на что-нибудь обидитесь — честное слово, я этого не хотел.

[2] По мнению моего технического редактора Скотта Стабберта, эта фраза создает впечатление, будто я в принципе против наследования. Это не совсем так. Как будет показано в главе 5, наследование — серьезная штука, которую нельзя ни легко отвергнуть, ни легко принять.

- Некоторые программисты VB настолько привыкли использовать встроенные функции и функции API, что не станут тщательно исследовать все средства .NET.
- Некоторые программисты VB испытывают такое предубеждение против исполнительной среды и P-кода, что ограничатся скороспелыми суждениями о промежуточном языке .NET (IL, Intermediate Language) вместо объективных оценок.

В этой части книги вы познакомитесь с базовыми концепциями, которые должны быть известны каждому программисту VB .NET. Впрочем, насчет «каждого» сказано слишком сильно — начинающие смогут использовать язык и писать простые программы и без полного понимания этих концепций. В этой части книги рассматриваются концепции, представляющие интерес для всех *квалифицированных* программистов, переходящих с VB6 на VB .NET. Мы должны изучить их, прежде чем перейдем к рассмотрению непосредственных изменений в языке.

Виртуальная машина

С точки зрения программиста, термин «виртуальная машина» описывает платформу, для которой пишется программный код[1]. На компьютере могут быть установлены разные процессоры (Pentium III, Athlon, Pentium II и т. д.), он может быть оснащен разными объемом памяти — программиста не интересует физическая конфигурация компьютера. Он пишет программу для виртуальной машины, зависящей от операционной системы и программной среды.

На рис. 4.1 показана виртуальная машина, использовавшаяся в дни MS-DOS.

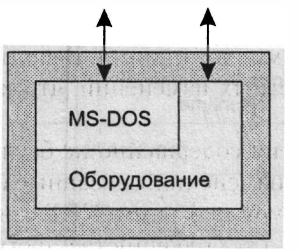


Рис. 4.1. Виртуальная машина с точки зрения DOS-программиста

В те дни программист пользовался относительно малым набором встроенных функций операционной системы, а также средствами, входящими в язык. Иногда приходилось пользоваться прямым доступом к оборудованию, поскольку в таких областях, как графика и коммуникации, возможности MS-DOS оставляли желать лучшего.

[1] Не путайте понятия «виртуальная машина» и «исполнительная среда». Например, виртуальная машина Java реализована на базе исполнительной среды и классов. Именно виртуальная машина определяет среду программирования, однако этот термин относится к *любой* среде. Даже при работе на ассемблере программист пишет код для виртуальной машины, определяемой набором инструкций процессора (которые, в свою очередь, реализуются микрокодом самого процессора).

В наше время программисты VB6 привыкли к более сложной виртуальной машине, изображенной на рис. 4.2. В нее входит исполнительная среда Visual Basic, Win32 API, подсистема COM и различные подсистемы, использующие COM. Прямой доступ к оборудованию практически невозможен.

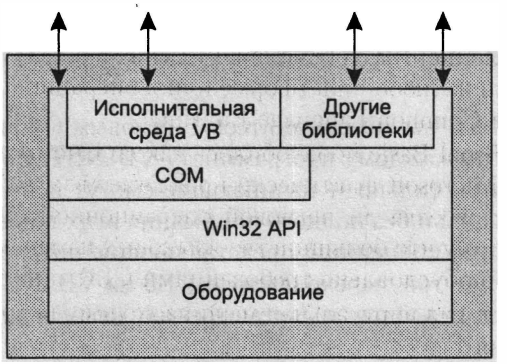


Рис. 4.2. Виртуальная машина с точки зрения программиста VB6

Главное, что необходимо понять из сравнительного анализа этих двух рисунков, — то, что наши программы работают на том же компьютере, но виртуальная машина принципиально отличается от той, что использовалась в MS-DOS и даже (хотя из рисунка этого не видно) во времена VB3.

А теперь взгляните на рис. 4.3, изображающий виртуальную машину с точки зрения программиста VB .NET.

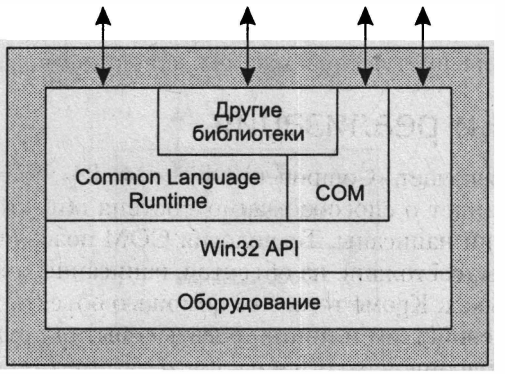


Рис. 4.3. Виртуальная машина для программистов VB .NET

Что бросается в глаза на этом рисунке?

- Из виртуальной машины исчезла исполнительная среда, специфическая для VB: Visual Basic использует .NET Common Language Runtime (CLR[1]).
- Visual Basic 6 и его исполнительная среда построены на базе COM. CLR не базируется на COM, хотя приложения .NET могут использовать компоненты COM, и наоборот.

[1] Если уж придираться к словам, то CLR, конечно, включает пространства имен, специфические для Visual Basic.

Короче говоря, программист VB .NET пишет код для виртуальной машины, которая принципиально отличается от виртуальной машины, используемой программистом VB6. Изучение VB .NET сопоставимо с изучением программирования в совершенно другой операционной системе. Теоретически приложение Common Language (то есть соответствующее спецификации Common Language Specification, определенной на уровне .NET) сможет без изменений и без перекомпиляции работать на любой платформе или в операционной системе, в которой имеются модули Common Language Runtime!

Почему же для Visual Basic понадобилось так сильно изменять виртуальную машину? И почему Microsoft фактически предлагает вам то, что с точки зрения программиста является едва ли не новой операционной системой? Это очень важные вопросы, потому что большинство языковых изменений при переходе от VB6 к VB .NET было обусловлено требованиями CLR, а не наоборот.

Чем была плоха старая виртуальная машина, почему потребовались столь радикальные изменения?

Ответ на этот вопрос дает сравнение рис. 4.1–4.3. В Windows не было CLR.

COM умер. Да здравствует COM?

Честно говоря, я крайне неохотно приступаю к дальнейшим объяснениям. То, что специалисты по маркетингу из Microsoft натворили с терминами COM, OLE и COM+ (едва не добравшись до COM+ 2.0), было непростительно. Любые попытки хоть как-то исправить положение — дело по меньшей мере рискованное, но я все же попытаюсь.

COM — идеи и реализация

Сокращение COM означает «Component Object Model», то есть «модель составного объекта». Речь идет о способе взаимодействия объектов, не зависящем от языка, на котором они написаны. Технология COM позволяет создавать приложения и компоненты, состоящие из объектов, написанных разными программистами на разных языках. Кроме того, с ее помощью объекты могут вызывать методы, обращаться к свойствам и инициировать события других объектов даже в том случае, если последние находятся на другом компьютере в сети.

COM основывается на нескольких относительно простых идеях.

- Интерфейс определяет «контракт» — список методов и свойств, поддерживаемых объектом. После определения интерфейс уже не изменяется.
- С каждым объектом и интерфейсом COM связывается GUID — глобально-уникальный идентификатор, — ассоциированный с данным объектом/интерфейсом и ни с каким другим.
- Объект COM может предоставлять пользователю несколько интерфейсов. При этом должны существовать средства для переключения между интерфейсами и однозначной идентификации объекта независимо от того, какой интерфейс при этом используется.

- На двоичном уровне должен существовать стандартный механизм, посредством которого объекты обращаются друг к другу. Этот механизм не должен зависеть от используемого языка.
- Должен существовать единый механизм инициирования ошибок, чтобы ошибка, возникшая в одном объекте, правильно обнаруживалась и идентифицировалась другими объектами.

Практическая реализация, заложенная в основу COM, решает все перечисленные задачи.

- Интерфейсы определяются в библиотеках типов — блоках метаданных, в которых перечисляются свойства и методы интерфейса, параметры методов, типы параметров и возвращаемых значений. Интерфейсы создаются автоматически при включении методов и свойств в класс VB или при помощи языка IDL (Interface Description Language) в C++.
- Для работы с объектом COM его GUID должен храниться в системном реестре.
- Каждый объект COM поддерживает интерфейс с именем `IUnknown`, состоящий из трех методов: `AddRef`, `Release` и `QueryInterface`. Методы `AddRef` и `Release` реализуют механизм подсчета ссылок, при помощи которого система следит за количеством ссылок на некоторый объект — когда количество ссылок уменьшается до нуля, объект можно удалять. Метод `QueryInterface` используется для перехода между интерфейсами, поддерживаемыми объектом.
- Во внутренней реализации объектов COM для вызова методов интерфейса объекта используются таблицы виртуальных функций (массивы указателей на функции). Механизм вызова скрыт от программистов VB6, если не считать неразумных личностей, предлагающих модифицировать таблицы в VB оператором `ObjPtr`[1].
- Объекты COM сообщают об ошибках при помощи кодов `HRESULT` — 32-разрядных величин стандартного формата. VB6 преобразует коды ошибок COM в типы инициируемых ошибок.

Говоря о недостатках COM, я скорее имею в виду реализацию, а не саму идею, причем к недостаткам реализации относятся не ее теоретические основы и даже не конкретный программный код, а практические последствия ее применения. Иначе говоря, недостаток технологии COM состоит не в том, что она не работает, а в том, что она плохо защищает нас от человеческой тупости — как нашей собственной, так и тупости конечных пользователей[2].

---

[1] Я выдержал немало жарких споров с людьми, которые применяли или выступали за этот подход. Упоминания о нем часто встречались в журналах и даже в книгах; к счастью, многие авторы предупреждали, что эту методику следует рассматривать как академическое упражнение, нежели практическое решение. Помнится, я утверждал, что это плохая практика программирования, которая крайне затрудняет сопровождение программы и вряд ли сохранится в следующей версии Visual Basic. Я и не думал, что окажусь настолько прав...

[2] В своей книге «The Dilbert Principle» Скотт Адамс (Scott Adams) говорит, что все мы время от времени оказываемся идиотами. Именно это обстоятельство и подвело COM — к сожалению, эта технология не оставляет места для идиотизма. Мы снова возвращаемся к тому, о чем я говорил в главе 2: «Признание и успех технологии почти всегда определяется человеческими, политическими и экономическими факторами, а вовсе не технологическими!»

Проблемы с интерфейсами

Интерфейсы COM подчиняются одному простому правилу: после определения и распространения интерфейса вы никогда, НИКОГДА не должны изменять его. Более того, постоянным должен оставаться не только интерфейс, но и реализуемые им функциональные возможности.

Рассмотрим следующую ситуацию. Вы создаете объект с методом:

```
Public Function Verify(CreditCardInfo As String) As Boolean
```

...и выпускаете версию 1 компонента, в котором этот метод используется для проверки кредитных карт. Также выходит версия 1 коммерческого приложения, использующего версию 1 компонента.

Затем происходит одно из следующих событий (или оба сразу).

- Ретивый программист решает, что функции нужен новый параметр (тип кредитной карты), и игнорирует предупреждение о двоичной совместимости в процессе компиляции компонента.

Или:

- При повторной компиляции компонента программист забывает правильно настроить двоичную совместимость.

Итак, выпускается версия 2 компонента с версией 1 системы проверки кредитных карт.

К сожалению, версия 2 компонента несовместима с версией 1 коммерческого приложения, что немедленно проявляется во всех системах, где была установлена программа (и обновленный компонент).

После катастрофического падения репутации и колоссальных затрат на техническую поддержку вы торопитесь создать версию 2 коммерческого приложения, работающую с версией 2 компонента. Более того, вы нашли способ существенно повысить быстродействие программы и поэтому переходите прямо к версии 2 компонента. На этот раз двоичная совместимость с компонентом версии 2 находится под самым тщательным контролем. К сожалению, один из программистов в процессе оптимизации компонента случайно изменяет одну из таблиц с описанием типа кредитной карты, поэтому счета по картам Discover теперь направляются в American Express.

Когда такой компонент устанавливается в системе, программа будет работать, поскольку интерфейс компонента при этом соблюдается. Однако с функциональной точки зрения новый компонент не обеспечивает обратной совместимости, поэтому на вас снова обрушится шквал жалоб со стороны клиентов.

Короче говоря, возникает ситуация, которую обычно называют «Кошмаром DLL».

Это происходит из-за того, что компоненты COM в DLL предназначены для внешнего использования.

Теоретически в COM «Кошмар DLL» просто исключен. Правила COM вполне однозначны: однажды определенный интерфейс никогда не изменяется, а обратная совместимость не нарушается.

К сожалению, программисты не идеальны. Более того, программисты Microsoft тоже не идеальны — вероятно, они создали больше проблем с «Кошмаром DLL»,

чем любая другая компания (что вполне объяснимо, поскольку они вообще создают больше DLL).

Проблемы с реестром

Самый очевидный выход из «Кошмара DLL» — разрешить каждому приложению загружать и запускать свои собственные версии компонентов. Ситуация, при которой разные версии одного компонента могут одновременно выполняться разными приложениями, называется «параллельным выполнением» (side-by-side execution). Хотя в Windows 2000 такая возможность существует, пользоваться ей неудобно — COM требует, чтобы для каждой версии регистрировался только один компонент.

Еще хуже то, что в COM используются только предварительно зарегистрированные компоненты. Это означает, что приложение будет работать лишь в том случае, если все его компоненты и их взаимосвязи были зарегистрированы в системе. Если другое приложение установит другую версию DLL в свой собственный каталог, оно благополучно уничтожит информацию, записанную в реестр первым приложением.

Наконец, при повреждении реестра вам придется заново переустанавливать и регистрировать все компоненты.

Проблемы с IUnknown

В COM каждое сохранение ссылки на объект должно сопровождаться вызовом метода `AddRef` интерфейса `IUnknown` объекта. При освобождении ссылок должен вызываться метод `Release`. Для объекта ведется внутренний счетчик текущих ссылок; когда его значение уменьшается до нуля, объект удаляет себя.

Возникают две проблемы.

Первая проблема в большей степени относится к C++, нежели к Visual Basic — программист может случайно забыть об удалении объекта. Пропущенные вызовы `AddRef` обнаруживаются очень легко, поскольку при обращении к удаленному объекту обычно инициируется исключение. С другой стороны, забытый вызов `Release` приводит к тому, что объект остается в памяти до завершения программы. Это может привести к утечке памяти, в результате которой приложение постепенно захватит все ресурсы системы — вполне реальная проблема для приложений, работающих по схеме 24/7[1].

Вторая проблема знакома программистам VB и продемонстрирована в следующем фрагменте.

Создайте класс `Class1` со следующим кодом:

```
Public HoldingCollection As Collection
Private Sub Class_Terminate()
  Debug.Print "Object Freed"
End Sub
```

Затем создайте форму с кнопкой, обработчик которой выглядит так:

```
Private Sub cmdExecute_Click()
  Dim col As New Collection
  Dim myobject As New Class1
```

[1] То есть 24 часа в сутки, 7 дней в неделю. — *Примеч. перев.*

```
  Dim counter As Long
  For counter = 1 To 1000
    Set myobject.HoldingCollection = col
    col.Add myobject
  Next counter
End Sub
```

Запустите программу и нажмите кнопку несколько раз. Понаблюдайте за сообщениями в окне отладки, свидетельствующими об удалении объекта. Ни одного такого сообщения вы не увидите. Попробуйте вызвать диспетчер задач и найдите приложение (или VB6, если код работает в среде программирования) в списке процессов. Запустите программу и нажимайте кнопку, наблюдая за столбцом **Mem Usage**.

Объем используемой памяти растет.

Перед нами типичная проблема циклических ссылок. Ссылка на каждый объект `Class1` хранится в коллекции, но каждый объект также содержит ссылку на коллекцию (вероятно, для того, чтобы иметь возможность обратиться к другим объектам `Class1`). При выходе из функции ссылка на коллекцию в переменной `col` освобождается, однако коллекция не удаляется, поскольку ее счетчик ссылок не равен нулю (кстати говоря, его значение совпадает с количеством объектов `Class1` в коллекции).

Подобные утечки памяти отрицательно сказываются как на долгосрочной надежности работы приложения, так и на его масштабируемости.

Обработка ошибок

Проблемы с обработкой ошибок в наши дни обусловлены не столько недостатками COM, как тем, что программисты никак не могут договориться о том, как именно должна быть организована обработка ошибок. Должны ли ошибки инициироваться при создании компонентов VB? А может, коды ошибок следует возвращать в качестве результатов вызова функций? Даже беглый взгляд на функции API обнаруживает множество несоответствий в способах возврата информации об ошибках.

И давайте честно признаем, что обработка ошибок в Visual Basic оставляет желать лучшего. `On Error Goto`? Никто не использует `Goto` — нас десятилетиями учили тому, что это нехорошо. Синтаксис обработки ошибок в VB является пережитком доисторической эпохи BASIC.

COM+

Забавно — хотя я и раньше сталкивался со всеми описанными проблемами, лишь при написании этого раздела понял, сколькими недостатками обладает текущая реализация COM. Поэтому прежде чем показывать, как эти проблемы решаются в .NET, я должен хотя бы в общих чертах упомянуть о COM+.

В целом COM+ относится к числу технологий, порожденных в основном маркетинговыми соображениями (см. главу 1). Не поймите меня превратно — в COM+ появилось немало технологических новшеств (контексты, транзакции, асинхронные операции и т. д.), но все они в той или иной степени уже существовали под другими названиями (главным образом в Microsoft Transaction Server и

Microsoft Message Queue). Появление термина COM+ не принесло ничего нового — просто появился новый маркетинговый ярлык для раскрутки[1].

В том, что касается описанных выше идей и их реализации, COM+ ничем не отличается от COM.

COM+ 2.0

В какой-то момент показалось, что Microsoft собирается предпринять шаг, который, на мой взгляд, не имел никаких логических объяснений. Совершенно серьезно рассматривался вопрос о присвоении библиотеке .NET Framework, реализованной на Common Runtime Language, названия «COM+ 2.0».

.NET Framework не использует COM. Ее работа не основана на COM. Да, классы .NET Framework могут взаимодействовать с COM, использовать компоненты COM и использоваться в них, но на этом все и кончается.

Я рад сообщить, что здравый смысл все же победил и Microsoft отказалась от идеи назвать .NET Framework «COM+ 2.0». Текущие ссылки на COM+ 2.0 в MSDN всего лишь перенаправляют читателя к .NET Framework.

Значит ли это, что технология COM мертва?

И да, и нет.

Да — если Microsoft удастся превратить .NET в доминирующую платформу разработки в мире Windows (а может, и не только?). В этом случае COM будет играть все меньшую роль. Если .NET интегрируется в будущих операционных системах, а основные приложения будут перестроены для .NET Frameworks, COM превратится в пережиток прошлого, в исторический казус, который по каким-то причинам продолжает существовать.

Но можно не сомневаться, существовать он будет. Слишком много всего построено на базе COM, чтобы эта технология могла умереть. Ведь Windows до сих пор поддерживает DDE, хотя в наши дни многие программисты даже не знают, что это такое[2].

Common Language Runtime

Любая технология должна изучаться в практическом контексте. Россыпи новых возможностей и преувеличенные обещания хорошо подходят для презентаций, но при оценке новой технологии нет ничего важнее, чем понимание контекста, в котором она существует, и проблем, которые она призвана решать. Это особенно важно при решении вопроса, подходит ли та или иная технология для выполнения ваших конкретных задач.

Итак, теперь вы знаете основные недостатки COM, и мы можем посмотреть, как же эти проблемы решаются в .NET.

---

[1] И не ждите, что я сейчас скажу что-нибудь о DNA. Эта технология умерла и уже не вернется.

[2] DDE (Dynamic Data Exchange) — старый протокол Windows для обмена информацией между приложениями. Ваш скромный слуга, которому досталось «удовольствие» программировать работу с DDE до появления стандартных библиотек, вспоминает об этом самыми сокровенными, задушевными словами...

Для начала зададим себе следующие вопросы[1].

- Как добиться, чтобы программа, использующая компонент, продолжала работать даже в том случае, если кто-то установит более новую и притом несовместимую версию компонента?
- Как добиться, чтобы программа обнаруживала сам факт установки новой версии компонента, несовместимой с рабочей версией?
- Как устранить необходимость в регистрации компонентов?
- Как решить проблему циклических ссылок и утечки памяти даже в тех случаях, когда программист забывает освободить объект?

Во-первых, потребуется, чтобы в системе могли одновременно работать несколько версий одного компонента. В этом случае присутствие новой или старой копии компонента где-то в системе не повлияет на работу «правильной» копии, находящейся в каталоге приложения.

Во-вторых, в операционную систему придется встроить механизм, который бы позволял сравнивать методы и свойства, фактически поддерживаемые компонентом, с теми, которые ожидаются программой, и при возникновении проблем запрещал бы работу приложения. Проверка должна выполняться перед запуском программы, чтобы пользователю не приходилось дожидаться ошибки времени выполнения или исключения защиты памяти.

Операционная система должна автоматически находить и «регистрировать» компоненты без реестра — как правило, поиск должен происходить в каталоге приложения или в других заранее определенных каталогах, где могут находиться общие компоненты.

Наконец, операционная система должна отслеживать все объекты, используемые приложением в процессе его работы, и автоматически освобождать те из них, надобность в которых отпала.

Все эти требования невозможно удовлетворить в системе Windows в том виде, в котором она существует сейчас. Несовместимы они и с текущей реализацией COM. Понадобится совершенно другая архитектура, совершенно новая виртуальная машина. Такая виртуальная машина поддерживается исполнительной средой Common Language Runtime.

Visual Basic DLL или EXE-файл, созданные в VB .NET, сильно отличаются от тех, что использовались раньше. Да, в них используется тот же формат PE (Portable Executable), но при попытке выполнить исполняемый файл VB .NET в системе без установленной среды CLR на вас обрушатся многочисленные ошибки «DLL not found»[2]. Дело в том, что среда CLR нужна Windows для интерпретации новых типов записей, хранящихся в исполняемом файле.

В мире .NET вместо терминов «DLL» и «EXE-файлы» чаще используется термин «сборка» (assembly). Мы еще поговорим о сборках в этой главе, а пока просто считайте, что существует однозначное соответствие — каждый созданный вами

---

[1] Интересно, не составляли ли разработчики .NET подобный список и в каком порядке они расположили бы эти вопросы? К сожалению, об этом можно лишь гадать.

[2] Во всяком случае, так было в предварительных версиях. Будем надеяться, что в окончательной версии Microsoft придумает более удобный способ проверки и оповещения пользователей о том, что для работы программы необходима среда .NET.

DLL- или EXE-файл содержит одну сборку, и каждая сборка состоит из одного DLL- или EXE-файла[1].

Среди новых типов записей в исполняемых файлах .NET хранится так называемый «манифест» (manifest). Манифест содержит подробные сведения о сборке:

- список всех компонентов, используемых сборкой;
- номера версий этих компонентов и хэш-коды, по которым исполнительная среда узнает об изменении этих компонентов;
- список всех объектов, поддерживаемых сборкой, их методов, свойств, типов параметров и возвращаемых значений;
- список всех объектов, необходимых для работы сборки, их методов, свойств, типов параметров и возвращаемых значений и т. д.

Существует и другой новый тип записей — записи промежуточного языка (Intermediate Language, сокращенно IL). Вскоре мы поговорим на эту тему более подробно.

Манифест

Манифест призван решить проблемы с контролем версии и установкой компонентов, существовавшие в COM. Давайте последовательно рассмотрим его основные возможности.

- Программа, использующая некоторый компонент, сможет продолжить работу даже в том случае, если в системе будет установлена обновленная версия этого компонента, несовместимая с предыдущей.

В CLR поддерживается так называемое параллельное выполнение. Это означает, что если компонент существует в каталоге приложения, CLR загружает его даже при существовании того же компонента (в той же или другой версии) в других каталогах системы, в том числе и в каталоге общих компонентов.

Означает ли это, что разработчики начнут устанавливать компоненты в свои собственные каталоги вместо System32 или других общих каталогов, чтобы избавиться от проблем с распространением своих компонентов?

Да, именно так. Вы по-прежнему сможете создавать общие компоненты, но, несомненно, они будут встречаться реже.

Означает ли это, что система будет загромождаться разными версиями одного и того же компонента? Разве это не приведет к напрасным затратам дискового пространства, не говоря уже о затратах памяти на одновременную загрузку компонентов, когда это совершенно не нужно? Разве библиотеки динамической компоновки создавались не для того, чтобы обеспечить совместное использование памяти и уменьшить затраты места на диске?

Да, такой подход потенциально расточителен по отношению к использованию памяти и дискового пространства. Да, DLL создавались для решения именно этих проблем. Но все это было во времена, когда на обычном компьютере стояло

[1] На самом деле сборка может состоять из нескольких DLL- или EXE-файлов. Тем не менее в текущей бета-версии VB .NET эта возможность не поддерживается и ничто не говорит о том, что в окончательной версии VB .NET ситуация изменится.

640 Кбайт памяти, на особо мощных машинах объем памяти достигал нескольких мегабайт, а дисковое пространство стоило от $10 за мегабайт[1]. В наши дни даже на слабых компьютерах устанавливается от 64 Мбайт памяти, а мегабайт дискового пространства редко стоит больше 1 цента. При таких характеристиках затратами на дублирование файлов DLL в системе и даже в памяти можно пренебречь. В наши дни разработчики обращают основное внимание на борьбу с утечкой памяти, возникающей во время работы программ. Если ваше приложение работает целыми днями и даже неделями, такая утечка способна полностью израсходовать даже эти огромные ресурсы. Кроме того, разработчики стараются свести к минимуму влияние компонентов одного приложения на работу других приложений.

- Манифест позволяет программе обнаружить факт установки несовместимой версии компонента поверх его рабочей версии.

Что произойдет, если существующий компонент будет физически заменен новым, несовместимым с вашим приложением?

В этом случае результат отчасти зависит от конфигурации приложения. Например, вы можете потребовать, чтобы приложение всегда использовало определенную версию сборки. Если правильная версия не будет найдена, приложение не запустится. Манифест даже содержит хэшированные сигнатуры зависимых сборок, что позволит обнаружить изменения в них и в том случае, если разработчик забудет обновить номер версии!

Кроме того, можно разрешить приложению использовать обновленные версии компонента. В этом случае CLR по данным манифеста проверяет новую версию и убеждается в том, что она поддерживает все нужные объекты, а все методы, свойства и типы совпадают с теми, которые нужны для работы вашего приложения. В случае несовпадения выполнение программы не разрешается.

- Манифест снимает необходимость в регистрации компонентов.

CLR получает данные манифеста на стадии загрузки от приложения и зависимых компонентов, а не из реестра. Компоненты загружаются из каталога приложения или из глобального кэша (в зависимости от конфигурации приложения). Любопытно заметить, что благодаря этому обстоятельству (а также некоторым другим) становится возможным принципиально новый подход к установке. Теперь приложение можно установить простым копированием файлов или дерева каталогов[2]!

На всякий случай стоит отметить, что все эти замечательные возможности не имеют обратной силы. Иначе говоря, если ваше .NET-приложение использует традиционные компоненты COM, то все старые правила остаются в силе. Вам придется регистрировать эти компоненты и решать обычные проблемы совместимости — однако это относится лишь к традиционным компонентам COM.

Кроме того, следует учитывать, что долгосрочный успех этого подхода в немалой степени зависит от того, удастся ли Microsoft обеспечить обратную совместимость самой среды CLR по мере ее усовершенствования[3].

---

[1] Помню свой восторг от покупки 500-мегабайтного жесткого диска — и притом всего за $1000!

[2] Думаю, это многих позабавит. После долгих лет развития Windows-технологий мы наконец-то пришли к тому, что в DOS было вполне заурядным явлением!

[3] До меня доносились слухи о том, что в системе можно будет установить сразу несколько версий CLR. Возможно, это поможет избежать некоторых проблем с совместимостью.

Промежуточный язык (IL)

Как же CLR выполняет все эти чудеса с манифестом? Откомпилированная программа содержит большое количество вызовов функций. Работа COM основана на том, что при компиляции приложения, использующего компоненты COM, для каждого объекта создается таблица виртуальных функций — массив указателей на функции, вызываемые приложением. Впрочем, так работает механизм раннего связывания. При позднем связывании возможен динамический вызов функций по именам, однако это сопряжено с существенными затратами.

И конечно, остается последнее требование из нашего списка: среда CLR должна решать проблемы циклических ссылок и утечки памяти даже в тех случаях, когда программист забывает освободить объект.

Каким образом исполнительная среда может проанализировать откомпилированное приложение и выяснить, где объект используется, а где его можно освободить? И не отразится ли эта проверка на быстродействии приложения?

В традиционных моделях код вызова методов, обращения к свойствам, создания и освобождения ссылок на объекты генерируется компилятором, поскольку компилятор располагает полной информацией об именах и типах методов и их параметрах. Компилятор знает, где в программе создаются и освобождаются объекты. Но после завершения компиляции эта дополнительная информация пропадает, и отслеживать ссылки на объекты уже поздно.

В .NET эта проблема решается наиболее очевидным способом: часть компиляции откладывается до момента загрузки приложения.

Компиляция сборки .NET в DLL-библиотеку или EXE-файл не доводится до конца. Большая часть информации, обычно используемой компилятором, хранится в манифесте. Вместо машинного кода компилятор генерирует так называемый IL-код.

При первой загрузке приложения .NET запускается JIT-компилятор, который и генерирует машинный код, необходимый для работы приложения. Но этим функции JIT-компилятора не ограничиваются.

- JIT-компилятор анализирует программу и убеждается в том, что все обращения к памяти осуществляются через переменные правильного типа (то есть проверяет, что ваше приложение не выходит за пределы тех объектов и структур данных, которые в нем определяются[1]).
- JIT-компилятор генерирует код и таблицы, по которым CLR находит корневые переменные вашей программы. К их числу относятся как глобальные переменные, так и локальные переменные, созданные в стеке или временно хранящиеся в регистрах процессора.
- Программа компилируется только в случае необходимости.

Не путайте IL-код с P-кодом, знакомым многим программистам Visual Basic. Да, P-код является разновидностью промежуточного языка, однако он интерпретируется исполнительной средой, а IL-код компилируется в машинный код, ко-

[1] В документации .NET особо подчеркивается, что в приложениях .NET отсутствуют проблемы с неинициализированными указателями, стирающими содержимое произвольных участков памяти, и с выходом за границы объектов из-за ошибок указателей, часто приводящих к ошибкам защиты памяти. Впрочем, это замечательное усовершенствование не произведет впечатления на программистов VB, которые с указателями вообще не работали.

торый затем сохраняется. В результате время загрузки увеличивается, но зато при каждом последующем выполнении этого блока обеспечивается превосходное быстродействие машинного кода. Сборку также можно обработать JIT-компилятором во время установки и сохранить полученный машинный код; это позволит ускорить первый запуск приложения[1].

Под термином «управляемый код» (managed code) понимается IL-код, который всегда обращается к памяти через определенные типы данных. В управляемом коде не разрешено использовать указатели, поскольку им могут быть присвоены значения, соответствующие недопустимым областям памяти. VB .NET создает управляемый код. Язык C# тоже ориентирован на управляемый код, однако в нем предусмотрен режим создания обычного кода. В C++ были включены специальные расширения, позволяющие создавать управляемый код.

Использование управляемого IL-кода приводит к интересному побочному следствию: поскольку машинный код генерируется лишь в момент обработки IL-кода JIT-компилятором на рабочем компьютере, теоретически возможно, что приложения .NET будут работать на всех платформах или операционных системах с поддержкой CLR. Интересно, появится ли CLR в каких-нибудь операционных системах, кроме Windows? А пока будем надеяться, что приложения .NET по крайней мере будут легко совмещаться с всевозможными разновидностями Windows, которых становится все больше.

Прощание с циклическими ссылками

Благодаря двухшаговой схеме компиляции (IL + JIT-компиляция) CLR может получить список корневых переменных приложения или компонента во время работы программы.

- CLR знает, где находятся глобальные переменные приложения.
- CLR знает, как перебрать содержимое стека и взять из каждого кадра его локальные переменные.
- CLR знает, в каких регистрах хранятся временные переменные.
- CLR знает, в каких переменных хранятся ссылки на объекты.
- CLR знает, какие члены объектов содержат ссылки на другие объекты.
- Все объекты создаются на стадии выполнения по данным сборок, поэтому CLR известно местонахождение всех объектов в куче и стеке.

В начале работы приложения .NET создается куча (heap), состоящая из одного большого блока памяти. Как правило, когда сборка запрашивает у CLR объект (а CLR рассматривает любые элементы данных как объекты), память выделяется из свободного блока в верхней части кучи, при этом среда не пытается заполнять пустые места, появившиеся на месте ранее освобожденных объектов[2]. Когда весь

---

[1] При предварительной компиляции во время установки необходимо сохранить исходный IL-код и манифест — это объясняется тем, что при модификации зависимых сборок CLR, возможно, придется откомпилировать программу заново.

[2] Как будет показано ниже, объекты структурных типов тоже могут создаваться в стеке, но пока мы говорим о куче.

свободный блок будет исчерпан, CLR приступает к «сборке мусора» (garbage collection). Ниже приведено упрощенное описание того, что при этом происходит.

- CLR помечает все объекты в куче как неиспользуемые.
- CLR проверяет все глобальные переменные вашего приложения и помечает объекты, на которые они ссылаются, как используемые. Далее производится рекурсивный поиск всех объектов, на которые ссылаются эти объекты «первого уровня», и они также помечаются как используемые. Если в процессе поиска оказывается, что найденный объект уже помечен как используемый, дальнейший поиск для этого объекта не производится — известно, что все объекты уже были найдены.
- CLR перебирает содержимое стека и проверяет все объекты, на которые ссылаются локальные переменные в каждом кадре стека. Производится аналогичный рекурсивный поиск, и все найденные объекты помечаются как используемые.
- CLR проверяет все объекты, ссылки на которые хранятся в регистрах процессора, и помечает эти объекты как используемые (тоже с рекурсивным поиском).
- После завершения этого процесса все объекты кучи, оставшиеся непомеченными, удаляются. Все остальные объекты перемещаются в нижнюю часть кучи, а все ссылки на них обновляются.

Этот процесс схематически изображен на рис. 4.4–4.7.

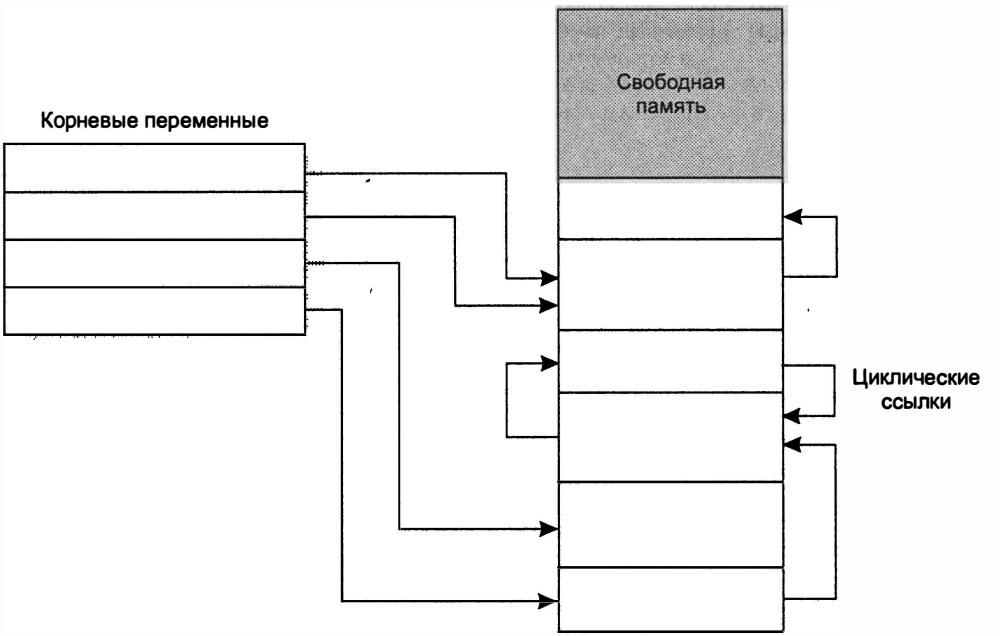


Рис. 4.4. Глобальные переменные и переменные, найденные в процессе трассировки стека, считаются «корневыми». Ссылки объектов на кучу представлены указателями. Объекты в куче тоже могут ссылаться на другие объекты. Обратите внимание на циклические ссылки, которые не будут удалены в COM

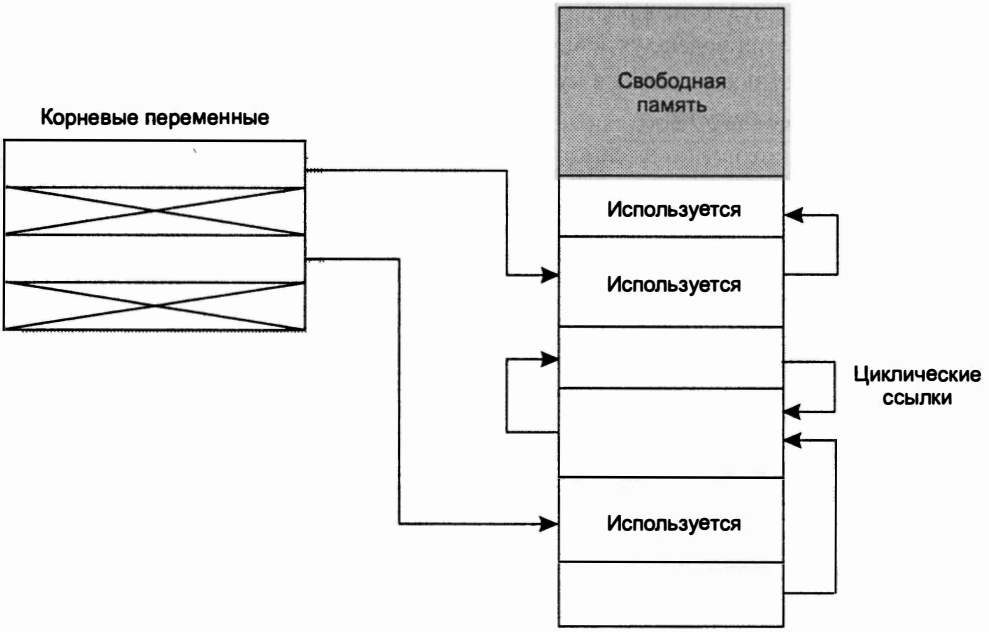


Рис. 4.5. В корневых переменных освобождаются две ссылки. Объекты, на которые ссылаются остальные переменные, помечаются как используемые — как и объекты, на которые они, в свою очередь, ссылаются

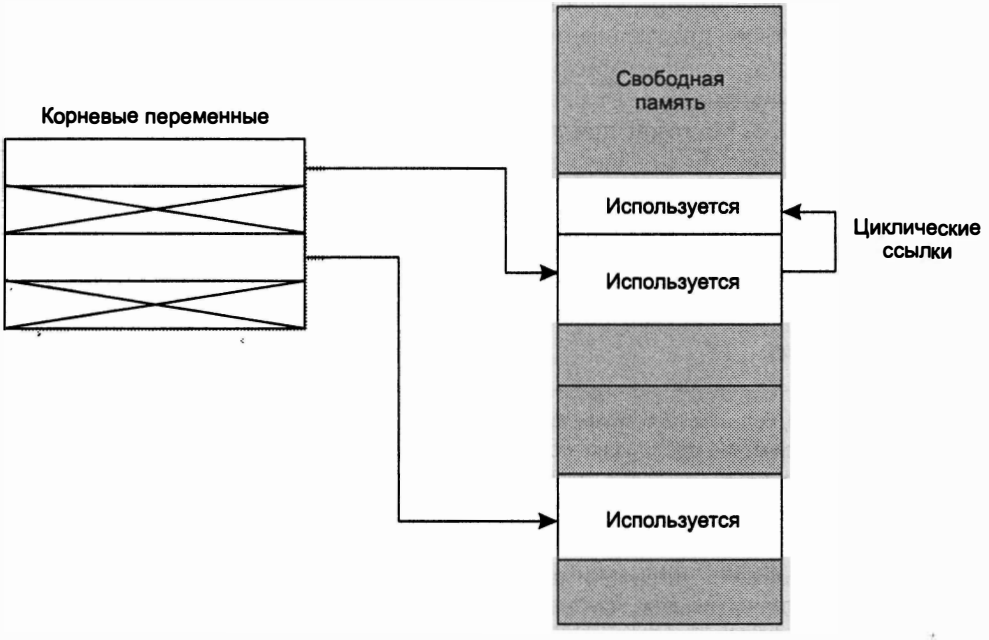


Рис. 4.6. Блоки памяти, занятые непомеченными объектами, освобождаются

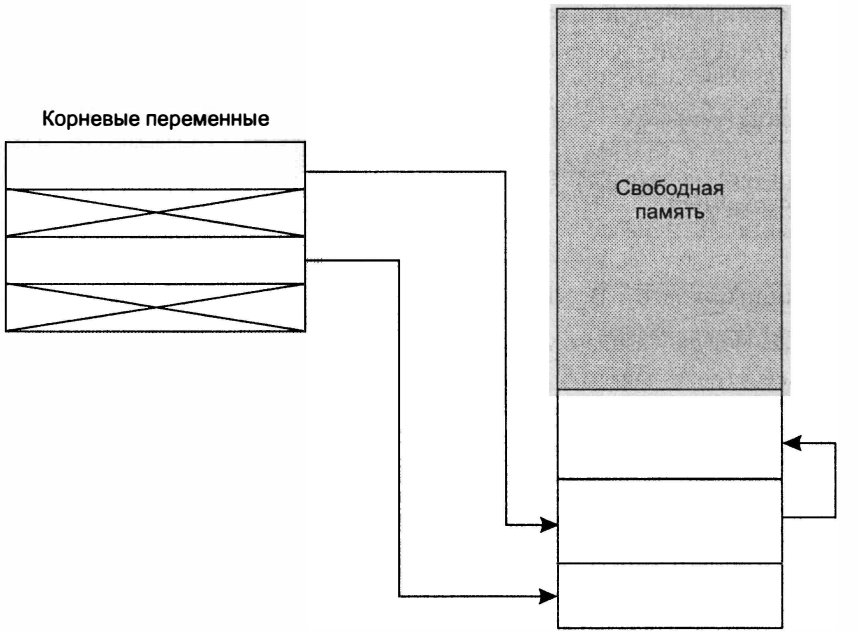


Рис. 4.7. Объекты перемещаются в нижнюю часть кучи с соответствующим изменением ссылок

В главе 6 вы узнаете, что в приведенном описании не упомянуты такие важные аспекты сборки мусора, как поколения и списки завершения, но, по крайней мере, оно дает общее представление о происходящем.

Конечно, за все хорошее приходится платить. При нехватке памяти для выполнения очередного запроса ваше приложение приостанавливается для сборки мусора. К счастью, Microsoft приложила огромные усилия по оптимизации процесса сборки мусора. Если учесть все преимущества, затраты вполне оправдываются.

Первая программа

Проект MemoryLeakNot наглядно показывает, что в CLR решена проблема циклических ссылок.

ВНИМАНИЕ

Вероятно, большая часть приведенного кода покажется вам непонятной. Многие концепции и синтаксические конструкции этого примера будут описаны значительно позднее. Я кратко прокомментирую некоторые незнакомые места, но в целом эту программу следует рассматривать как общий образец, а не пример, в котором вы должны досконально разобраться.

Исходный вариант приложения VB6 включает следующий класс:

```
Public HoldingCollection As Collection

Private Sub Class_Terminate()
   Debug.Print "Object freed"
End Sub
```

...и форму с кнопкой, при нажатии которой вызывается следующая процедура:

```
Private Sub cmdExecute_Click()
   Dim col As New Collection
   Dim myobject As New Class1
   Dim counter As Long
   For counter = 1 To 1000
    Set myobject.HoldingCollection = col
    col.Add myobject
   Next counter
End Sub
```

Аналогичный класс VB .NET приведен в листинге 4.1.

Листинг 4.1. Модуль TestClass.vb проекта MemoryLeakNot

```
' Отсутствие утечки памяти
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Public Class TestClassObject
  Shared m_classcount As Integer
  Dim m_ClassId As Integer
  Dim m_Collection As Collection
  Public Sub New(ByVal mycontainer As Collection)
    MyBase.New()
    mycontainer.Add (Me)
    m_Collection = mycontainer
    m_ClassId = m_classcount
    m_classcount = m_classcount + 1
  End Sub

  Protected Overrides Sub Finalize()
    System.Diagnostics.Debug.WriteLine ("Destructed " + _
    m_ClassId.ToString())
  End Sub

End Class
```

Код формы приведен в листинге 4.2.

Листинг 4.2. Форма TestForm.vb проекта MemoryLeakNot

```
' Отсутствие утечки памяти
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Public Class Form1
  Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

  Public Sub New()
    MyBase.New()

    ' Следующий вызов необходим
    ' для дизайнера форм Windows.
    InitializeComponent()

    ' Дальнейшая инициализация выполняется после вызова
    ' InitializeComponent()
  End Sub

  ' Форма переопределяет Dispose для очистки списка компонентов.
  Public Overloads Overrides Sub Dispose()
```

продолжение

Листинг 4.2 (продолжение)

```
    MyBase.Dispose()
    If Not (components Is Nothing) Then
      components.Dispose()
    End If
  End Sub
  Private WithEvents button1 As System.Windows.Forms.Button

  ' Необходимо для дизайнера форм Windows.
  Private components As System.ComponentModel.Container

  ' ВНИМАНИЕ: следующий фрагмент необходим
  ' для дизайнера форм Windows.
  ' Для его модификации следует использовать дизайнер форм.
  ' Не изменяйте его в редакторе!
  <System.Diagnostics.DebuggerStepThrough()> Private Sub _
    InitializeComponent()
    Me.button1 = New System.Windows.Forms.Button()
    Me.SuspendLayout()
    '
    'button1
    '
    Me.button1.Location = New System.Drawing.Point(104, 48)
    Me.button1.Name = "button1"
    Me.button1.TabIndex = 0
    Me.button1.Text = "Test"
    '
    'Form1
    '
    Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
    Me.ClientSize = New System.Drawing.Size(275, 144)
    Me.Controls.AddRange(New System.Windows.Forms.Control() _
      {Me.button1})
    Me.Name = "Form1"
    Me.Text = "Memory Leak - Not"
    Me.ResumeLayout (False)

  End Sub

  Private Sub button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button1.Click
    Dim x As Integer
    Dim col As New Collection()
    Dim obj As TestClassObject
    For x = 1 To 100
      obj = New TestClassObject(col)
    Next x
    Debug.WriteLine("Collection contains " + _
    col.Count.ToString() + " objects")

    ' Объекты все равно будут освобождены при выходе из функции,
    ' здесь это делается лишь в демонстрационных целях.
    col = Nothing
    obj = Nothing

    ' В обычных программах этот фрагмент не нужен.
    ' В данном примере он просто доказывает,
    ' что проблема циклических ссылок в VB .NET решена.
```

```
        gc.Collect()
        gc.WaitForPendingFinalizers()

    End Sub

#End Region

End Class
```

Не паникуйте. Все не так плохо, как кажется с первого взгляда.

В этом примере используется несколько иное архитектурное решение. В VB6 объект содержал ссылку на коллекцию, но эту ссылку приходилось задавать на уровне формы через открытое свойство объекта. В приложении MemoryNotLeak используется новая возможность VB .NET — параметризованные конструкторы, позволяющие передать параметр при инициализации объекта. В нашем примере передается ссылка на коллекцию.

Давайте еще раз взглянем на класс, приведенный в листинге 4.1.

В классе объявлены три локальные переменные. Впрочем, действительно локальными для каждого объекта являются лишь две из них. Переменная `m_classcount` объявлена общей (`Shared`). Это означает, что все объекты, созданные классом в домене приложения, будут работать с одним экземпляром этой переменной. В нашем примере это позволяет следить за количеством созданных объектов данного типа и присваивать им номера, сохраняемые в переменной `m_ClassId`.

Не беспокойтесь по поводу того, что у вас быстро кончатся целые номера. В VB .NET используется 32-разрядный тип `Integer` (аналог типа `Long` в VB6).

В переменной `m_Collection` хранится ссылка на коллекцию объектов.

Метод `New` является конструктором. Он получает ссылку на коллекцию в виде параметра и сохраняет ее в переменной. При этом нам не приходится использовать надоевшее ключевое слово `Set` из VB6. Первая команда `MyBase.New()` вызывает конструктор базового класса. Что такое базовый класс? Я отвечу на этот вопрос при описании наследования в главе 5. Далее конструктор увеличивает переменную `m_classscount`; новое значение будет использовано для следующего объекта этого типа.

Метод `Finalize` вызывается при подготовке объекта к окончательному уничтожению[1]. В нашем примере этот метод просто выводит служебную информацию в окне отладки. На смену знакомому объекту `Debug`, реализованному средствами COM, пришел значительно более мощный, хотя и менее знакомый объект `Debug` из пространства имен `System.Diagnostics`.

Параметр функции выглядит довольно странно. Для вывода идентификатора объекта в VB6 обычно применялись конструкции вида:

```
"Destructed " & m_Classid
```

[1] С технической точки зрения это утверждение правильно, однако оно не вносит полной ясности. Во-первых, непонятно, *когда именно* уничтожается объект. Во-вторых, подготовка к уничтожению еще не гарантирует, что объект *действительно* будет уничтожен. В-третьих, объекты могут уничтожаться и без вызова этого метода. В двух словах этого не объяснить — за подробными комментариями обращайтесь к главе 6.

VB6 видит, что вы собираетесь присоединить целое число к строке, и автоматически преобразует его в строку. В данном случае это удобно, но подобные преобразования типов часто приводят к всевозможным ошибкам. Следующая команда VB .NET осуществляет явное преобразование целого числа в строку методом `ToString`:

```
"Destructed " + m_Classid.ToString()
```

Стоп... Откуда взялся метод у типа `Integer`? Ведь это обычное целое число?

И да, и нет. Переменные типа `Integer` содержат целые числа, но в VB .NET они могут интерпретироваться как объекты. Все типы данных VB .NET являются производными от класса `Object`. А поскольку в классе `Object` определен метод `ToString` для создания текстового представления объекта, все объекты производных классов, включая `Integer`, тоже содержат метод `ToString`. Любопытно, правда? Не огорчайтесь, если вам что-то покажется непонятным — наследование рассматривается в следующей главе.

В листинге 4.3 приведена процедура `button1_Click` модуля формы.

Листинг 4.3. Метод button1_Click модуля TestForm.vb

```
Private Sub button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
  Dim x As Integer
  Dim col As New Collection()
  Dim obj As TestClassObject
  For x = 1 To 100
    obj = New TestClassObject(col)
  Next x
  Debug.WriteLine("Collection contains " + _
  col.Count.ToString() + " objects")

  ' Объекты все равно будут освобождены при выходе из функции,
  ' здесь это делается лишь в демонстрационных целях.
  col = Nothing
  obj = Nothing

  ' В обычных программах этот фрагмент не нужен.
  ' В данном примере он просто доказывает,
  ' что проблема циклических ссылок в VB .NET решена.
  GC.Collect()
  GC.WaitForPendingFinalizers()

End Sub
```

Большинство программистов без особого труда разберется в этом коде. Возможно, вас несколько удивят параметры события `Button_Click` — ведь вы привыкли к тому, что это событие вызывается без параметров.

Процедура события создает объект коллекции, затем в цикле создает 100 объектов `TestClassObject` и включает их в коллекцию. Помните, что в нашем примере объект включается в коллекцию передачей ссылки при вызове конструктора (вызываемого при создании объекта командой `New`). Заполнение коллекции проходит успешно, на что указывает свойство `Count`, выводимое в окне отладки методом `Debug.WriteLine`.

В конце этого метода приведены два примера «плохого» кода VB .NET. В первом фрагменте мы присваиваем переменным `col` и `obj` значение `Nothing`. Делать

этого не нужно, поскольку обе переменные автоматически освобождаются при выходе объекта из области видимости. Иногда значение `Nothing` приходится присваивать объектам внутри процедуры, но присваивание `Nothing` локальным переменным перед выходом из процедуры абсолютно излишне.

В следующем фрагменте запускается сборщик мусора. Скорее всего, запускать его вручную вам практически никогда не придется[1]. В нашем примере это делается только потому, что я хочу продемонстрировать уничтожение объектов в CLR. Не забывайте, что перед вами один из примеров циклических ссылок — коллекция содержит ссылку на каждый объект, который, в свою очередь, содержит ссылку на коллекцию. В COM коллекция и объекты будут освобождены лишь при выходе из приложения. В VB .NET они освобождаются, поскольку на них не ссылается ни одна корневая переменная приложения. Тем не менее освобождение происходит лишь во время работы сборщика мусора, но нельзя заранее предсказать, когда это произойдет. В нашей программе я запускаю сборщик мусора специальной командой, чтобы вы могли проследить за уничтожением объектов, наблюдая за сообщениями в окне отладки.

Как бы эффектно ни выглядело решение проблемы циклических ссылок, у предусмотрительного читателя наверняка возникает вопрос: как планировать действия программы при завершении объекта, если вы не знаете, когда именно оно произойдет? Более того, положение дополнительно усложняется тем, что в некоторых ситуациях метод `Finalize` вообще не вызывается.

Впрочем, мы только начинаем рассматривать управление памятью в Common Language Runtime. В главе 6 эта тема будет рассмотрена более подробно.

Новый подход

Обращает на себя внимание и новый код в модуле формы. Присмотритесь к нему. Не огорчайтесь, если вы чего-то не поймете — многие команды выглядят вполне знакомо. В частности, мы видим команды для создания формы и элемента-кнопки. Также бросается в глаза интересный комментарий:

```
' ВНИМАНИЕ: следующий фрагмент необходим
' для дизайнера форм Windows.
' Для его модификации следует использовать дизайнер форм.
' Не изменяйте его в редакторе!
```

Означает ли это, что вся среда программирования VB .NET генерирует весь код, необходимый для реализации приложений VB? Что произошло с инкапсуляцией кода на уровне языка, благодаря которой все технические подробности скрывались от пользователя? Не было ли это одной из тех особенностей, благодаря которым Visual Basic стал таким простым и доступным для многих программистов?

Перед нами один из первых спорных моментов, связанных с VB .NET. Раньше Visual Basic и Visual C++ номинально входили в пакет Visual Studio, но в действительности это были разные среды программирования.

---

[1] Возможно, в какой-нибудь экзотической ситуации такая необходимость и возникнет, но сейчас я не берусь привести конкретный пример.

Visual Basic скрывал от программиста все внутреннее взаимодействие компонентов приложения, которое фактически инкапсулировалось на уровне языка и исполнительной среды. Программисту не приходилось думать о том, откуда берутся формы и кнопки — они существовали и ими можно было пользоваться. При двойном щелчке на элементе открывался обработчик события, и никто не знал, что на самом деле происходило «за кулисами».

В Visual C++ обеспечивался аналогичный уровень автоматизации: чтобы создать обработчик события, было достаточно дважды щелкнуть на элементе в диалоговом окне, а свойства и методы добавлялись в программе Class Wizard. Однако в каждом из этих случаев вы в действительности взаимодействовали с большой и сложной программой. Эта программа генерировала код, который можно было видеть, но обычно не стоило модифицировать.

Что же произошло в VB .NET? Ответ на этот вопрос зависит от точки зрения.

Одни говорят, что Visual Basic наконец-то стал единой частью интегрированной среды разработки, что заметно упростило программирование приложений, написанных на нескольких языках (в сущности, выбор языка теперь определяется личным вкусом, нежели функциональными различиями).

Другие говорят, что разработчики Visual Studio ограбили Visual Basic; что новая среда, основанная на использовании автоматически сгенерированного кода, заметно уступает по эффективности традиционной среде VB и воплощает в себе все, чего многие программисты VB намеренно избегали.

Сторонники обеих точек зрения высказываются страстно и красноречиво.

Вероятно, вы заметили, что я тоже бываю страстным и красноречивым — но боюсь, моя точка зрения никого не устроит.

Дело в том, что я просто не берусь оценивать правильность подхода, избранного Microsoft, и реакцию на него программистов VB. Хотя я часто использую VB в работе над приложениями и компонентами, я также занимаюсь COM-программированием в Visual C++ для ATL[1], поэтому к Visual Studio я уже привык.

Думаю, большинство программистов Visual Basic без особого труда привыкнет к новому подходу, особенно если учесть, что редактор Visual Studio скроет от них все сложности.

И все же... Мне бы очень хотелось послушать споры, которыми сопровождалось принятие этого решения в Microsoft.

Итоги

Microsoft .NET несет программистам такие грандиозные изменения, что даже трудно начать с чего-то определенного. В этой главе я попытался представить .NET, не ограничиваясь простым перечислением новых возможностей. Я хотел, чтобы вы поняли, почему существуют эти возможности и для решения каких проблем они предназначались.

[1] Не помню, когда я в последний раз создавал приложение Visual C++ с пользовательским интерфейсом. К тому же с появлением ATL я полностью забросил MFC, хотя ATL гораздо сложнее. Впрочем, это совсем другая история.

Глава начинается с описания виртуальных машин и эволюции Windows-программирования. Вы узнали, что технология COM завела развитие Windows в эволюционный тупик, причем это связано не с какими-то теоретическими дефектами COM, а с тем, что эта технология не прощает ошибок программистам.

В .NET для решения этих проблем используется новая архитектура, основанная на Common Language Runtime. Приложения и компоненты .NET состоят из сборок, реализованных в виде одного или нескольких DLL и EXE-файлов. В каждой сборке имеется манифест с описанием всех компонентов, необходимых для ее работы, а также IL-код, компилируемый JIT-компилятором в машинный код при первой загрузке сборки или при ее установке в системе.

Архитектура .NET исправляет два основных недостатка COM: проблемы контроля версии и размещения, а также циклических ссылок и утечки памяти в тех случаях, когда программист забывает своевременно освобождать объекты в своей программе.

Кроме того, эта глава дает представление о коде VB .NET. Вы познакомились с параметризованными конструкторами, вызываемыми при создании объекта. Заметные изменения в синтаксисе языка оправдывают мое утверждение о том, что VB .NET — это совсем не тот Visual Basic, к которому вы привыкли. Наконец, мы рассмотрели наследование, пусть в самом элементарном варианте, а также выяснили, что в VB .NET абсолютно все переменные, включая числовые, являются объектными.

В этой главе были представлены те концепции, с которых, на мой взгляд, стоило начать изучение нового материала. Впрочем, это был всего лишь первый шаг.

В ближайших главах будут описаны многие новые возможности VB .NET:

- фактическое объединение механизмов наследования и включения;
- применение многопоточности для увеличения числа одновременно обслуживаемых клиентов, особенно в web-приложениях (требует величайшей осторожности при проектировании!);
- огромная библиотека функций Common Language Runtime;
- структурная обработка ошибок на базе исключений (в настоящей главе эта тема была лишь кратко упомянута, но мы непременно вернемся к ней в будущем);
- создание защищенного кода (вы можете управлять тем, какие полномочия в системе предоставляются коду из разных источников, а также создавать код с корректным сокращением функциональности в зависимости от уровня предоставленных привилегий.

Следующая глава посвящена широко разрекламированной теме — наследованию в VB .NET. Наследование играет ключевую роль в архитектуре CLR, однако на практике оно иногда преподносит неприятные сюрпризы.

Наследование 5

Когда информация о .NET только начинала распространяться, я часто видел, как докладчики из Microsoft стояли перед программистами Visual Basic и перечисляли новые возможности того, что когда-то называлось VB7. Одной из таких возможностей, неизменно вызывавшей громкое одобрение аудитории, было наследование.

Я никогда не мог понять, почему это происходит.

Впрочем, у меня есть предположение. Думаю, программисты Visual Basic комплексовали перед своими коллегами, работавшими на C++, которые часто задирали носы и говорили: «Visual Basic — *не настоящий* объектно-ориентированный язык. В *настоящем* объектно-ориентированном языке есть полноценное наследование». Услышав это, бедный программист VB удалялся в свою каморку и за час выдавал больше кода, чем программист C++ мог написать за целую неделю. Но никто не обращал на это внимания, потому что C++ был «современным, элегантным, профессиональным объектно-ориентированным языком», а VB считался «пережитком прошлого, игрушкой, языком новичков» для тех, кто не мог освоить C++.

Так почему бы программистам VB не порадоваться?

Поделюсь с вами маленьким секретом.

Я программировал на C++ в течение большего срока, чем на Visual Basic, — и продолжаю активно программировать на обоих языках. Я стал убежденным сторонником объектно-ориентированного программирования в тот самый момент, когда впервые осознал концепцию класса в 1977 году. Я программировал в таких библиотеках, как ATL, в которых широко и успешно использовалось наследование.

Но за все эти годы я припоминаю всего пять или шесть случаев, когда применение наследования было правильным решением в моих собственных приложениях и компонентах.

Да, .NET использует наследование — оно встроено в саму архитектуру. Да, на базе наследования создаются иерархии классов, при помощи которых вы строите собственные программы.

Но если вы действительно понимаете суть наследования, возможно, вам удастся успешно прожить, не создав ни одного класса или компонента, который будет использоваться в механизме наследования.

В этой главе я постараюсь обосновать свою точку зрения.

«Повторное использование кода» — мантра программиста

Несмотря на всю сопровождающую болтовню, наследование решает всего одну задачу: оно позволяет повторно использовать готовый программный код. В двух словах идея заключается в том, что когда класс объявляется производным от другого класса («наследует» от него), он мгновенно реализует всю функциональность этого класса, которую можно дополнить новыми возможностями.

Здорово, не правда ли?

Однако наследование — не единственный способ повторного использования кода.

Существуют библиотеки программ, при использовании которых фрагменты кода буквально копируется из одного проекта в другой. Существуют компоненты, объекты и функциональные возможности которых могут многократно использоваться в программах. Наконец, существуют объекты и программы, которые создают экземпляры других объектов для предоставления доступа к их функциональности.

У каждого из этих подходов есть свои достоинства и недостатки.

Чтобы вы лучше поняли все «плюсы» и «минусы», для начала я покажу пример плохого кода. В этой главе плохой код встречается сплошь и рядом.

На первый взгляд это смотрится довольно странно: читатель обычно рассчитывает найти в книге примеры хорошего кода и научиться тому, как правильно решить ту или иную задачу. Думаю, не менее важно взглянуть и на оборотную сторону медали; примеры плохого кода помогут вам избегать неверных решений[1].

В примере, который мы рассмотрим, используется простой связанный список — чрезвычайно полезная структура данных для работы со списками объектов. В общем случае связанный список эффективнее массива, поскольку операции вставки и удаления объектов выполняются без сдвига элементов массива. Связанные списки бывают удобнее коллекций, поскольку в них проще управлять порядком объектов и переставлять их по мере необходимости.

Общая идея связанного списка проста: в каждом объекте хранится указатель (ссылка) на следующий объект в списке. Корневая переменная указывает на первый объект.

Связанный список в VB6

Начнем с построения класса VB6 для работы со связанным списком. Да, я помню, что эта книга посвящена VB .NET, но поверьте — самый простой способ действительно изучить и понять наследование основан на сравнении с включением и наследованием интерфейсов, реализованным в VB6. В проекте LinkListVB6 пред-

[1] Я специально подчеркиваю это обстоятельство, поскольку читатели, знакомые с темой, могут пропустить объяснения, сразу просмотреть примеры и решить, что я окончательно выжил из ума. Пожалуйста, не торопитесь с поспешными выводами и изучайте примеры лишь после того, как я опишу их и объясню, в чем же заключаются их недостатки.

ставлен класс, упрощающий реализацию функциональности связанных списков в другом классе. Другими словами, наша цель — сделать так, чтобы этот код можно было использовать заново с минимальными хлопотами.

Объект LinkList обладает следующими свойствами и методами:

- свойство `NextItem` (используется для получения ссылки на следующий объект в списке);
- свойство `PreviousItem` (используется для получения ссылки на предыдущий объект в списке);
- метод `Remove` (удаляет текущий объект из списка);
- метод `Append` (включает текущий объект в список).

В Visual Basic 6 повторное использование кода обеспечивалось посредством включения (containment). Класс, объекты которого связывались в список, содержал экземпляр класса `LinkList` и вызывал его методы для выполнения необходимых операций.

Но к какому типу объекта относятся свойства `NextItem` и `PreviousItem`? К объекту `LinkList` или к тому объекту, который содержит объект `LinkList`?

На рис. 5.1 показан первый вариант, при котором объекты `LinkList` ссылаются на другой объект `LinkList`. В этом случае необходимы средства для получения ссылки на объект-контейнер (в нашем примере это гипотетический объект `Customer`).

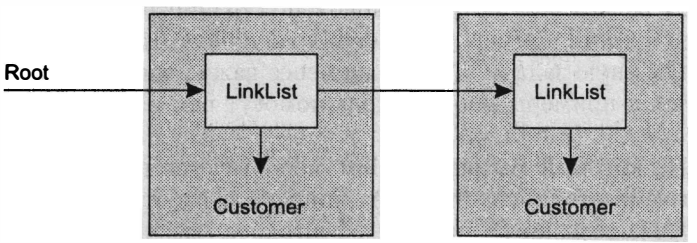


Рис. 5.1. Объекты LinkList ссылаются на другие объекты LinkList. Для получения ссылки на контейнер используется свойство Container

В другом варианте, показанном на рис. 5.2, объект `LinkList` ссылается на объект-контейнер, содержащий внутренний объект `LinkList`.

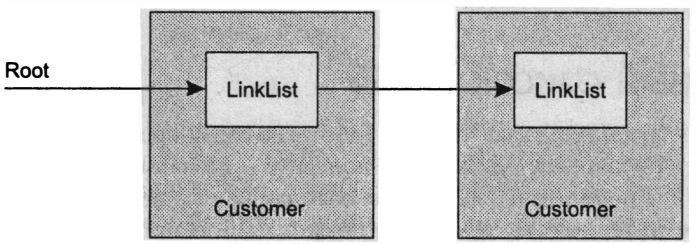


Рис. 5.2. Объект LinkList ссылается на объект-контейнер Customer

Начнем со второго подхода, реализованного в проекте LinkListVB6. Класс `LinkList` не только рассчитан на включение в контейнер, но и обеспечивает ин-

терфейс, через который контейнер выполняет операции со связанным списком. В контейнере используется команда `Implements`, обеспечивающая наследование интерфейсов. Это позволяет быстро реализовать в нужном объекте поддержку связанных списков.

Объект `LinkList` содержит две закрытые (`Private`) переменные. В переменной `m_Next` хранится ссылка на следующий объект списка. Объект `LinkList` указывает на контейнер, но поскольку контейнер реализует интерфейс `LinkList`, переменная `m_Next` может относиться к типу `LinkList`. В переменной `m_Container` хранится ссылка на объект-контейнер (вскоре вы поймете, зачем это нужно). Определение объекта `LinkList` выглядит так:

```
' LinkList VB6 пример #1
' Copyright ©2000 by Desaware Inc.  All Rights Reserved

Option Explicit

' Данная версия демонстрирует внедрение объектов,
' поэтому для связи используется тип LinkList
Private m_Next As LinkList
' Ссылка на контейнер необходима
' для включения нового элемента в список
Private m_Container As Object

' Контейнер задается в процессе инициализации
' Если вы захотите оформить класс в виде компонента,
' потребуется объявление с атрибутом Public
Friend Property Set Container(ByVal ContainerObject As Object)
  Set m_Container = ContainerObject
End Property
```

Свойство `NextItem` просто возвращает ссылку на переменную `m_Next`, которая содержит ссылку на следующий объект в списке:

```
' Просто вернуть ссылку на следующий объект в списке.
Public Property Get NextItem() As LinkList
  Set NextItem = m_Next
End Property

Public Property Set NextItem(ByVal nextptr As LinkList)
  Set m_Next = nextptr
End Property
```

Со свойством `PreviousItem` дело обстоит сложнее. Поскольку мы реализуем односвязный список, при запросе этого свойства необходимо перейти в начало списка и, последовательно перебирая все элементы, найти в нем текущий объект (точнее, объект, у которого свойство `NextItem` совпадает с текущим объектом). Сложность заключается в том, как задать объект для сравнения. Сравнивать с `Me` нельзя, поскольку `Me` относится к внутреннему объекту `LinkList`, а не к тому объекту, на который ссылается переменная `m_Next`. Однако в сравнении должны участвовать объекты-контейнеры, поэтому в каждый объект `LinkList` приходится включать ссылку на контейнер:

```
' В односвязном списке метод свойства Previous
' должен провести поиск от начала списка
Public Property Get PreviousItem(Root As LinkList) As LinkList
  Dim currentitem As LinkList
```

```
  ' Помните: все ссылки относятся к объекту-контейнеру!
  If (Root Is m_Container) Or (Root Is Nothing) Then
    Exit Property
  End If
  Set currentitem = Root
  Do
    If currentitem.NextItem Is m_Container Then
      Set PreviousItem = currentitem
      Exit Property
    Else
      Set currentitem = currentitem.NextItem
    End If
  Loop While Not currentitem Is Nothing
End Property
```

Метод `Remove` при помощи свойства `PreviousItem` находит в списке элемент, предшествующий удаляемому. Если такой элемент существует, его переменной `m_Next` присваивается значение `m_Next` текущего объекта, что фактически приводит к исключению текущего объекта из списка. Если удаляемый объект находится в первой позиции списка, то ссылка на следующий объект в списке присваивается переменной `Root` (передаваемой по ссылке). Пусть вас не смущает то обстоятельство, что в присваивании участвует переменная типа `LinkList`. В действительности `Root` присваивается ссылка на контейнер, реализующий интерфейс `LinkList`, а не на внутренний объект:

```
' Метод Remove находит предыдущий элемент списка
' последовательным перебором.
' Переменная Root должна передаваться по ссылке: это позволяет
' присвоить ей новое значение, если удаляемый объект
' находится в начале списка.
Public Sub Remove(Root As LinkList)
  Dim previtem As LinkList
  Set previtem = PreviousItem(Root)
  If previtem Is Nothing Then
    Set Root = m_Next
  Else
    Set previtem.NextItem = m_Next
  End If
End Sub
```

Метод `Append` также использует свойство `m_Container` для правильного включения объекта в список. Он находит последний объект в списке и присваивает его переменной `m_Next` ссылку на контейнер текущего объекта:

```
' Метод Append последовательно перебирает элементы
' от Root до конца списка.
Public Sub Append(Root As LinkList)
  Dim currentitem As LinkList
  Set currentitem = Root
  If Root Is Nothing Then
    Set Root = m_Container
  Else
    While Not currentitem.NextItem Is Nothing
      Set currentitem = currentitem.NextItem
    Wend
    Set currentitem.NextItem = m_Container
  End If
End Sub
```

Класс контейнера определяется в файле Customer.cls. В нем имеется открытое (Public) свойство CustomerName, предназначенное для получения имени контейнера. Класс реализует LinkList, тем самым наследуя интерфейс для работы со связанными списками. Он создает закрытый объект LinkList с именем m_MyLinkList, инициализируемый в обработчике события Class_Initialize. Как видно из листинга 5.1, все реализованные функции просто вызывают соответствующие функции объекта LinkList.

Листинг 5.1. Класс Customer.cls проекта LinkListVB6[1]

```
' LinkList VB6 пример #1
' Copyright ©2000 by Desaware Inc.  All Rights Reserved

Option Explicit

' Эта версия "реализует" LinkList, чтобы работать с его методами
' через объекты LinkList
Implements LinkList

Public CustomerName As String

' Функциональность обеспечивается внутренним объектом LinkList
Private m_MyLinkList As LinkList

Private Sub Class_Initialize()
  Set m_MyLinkList = New LinkList
  Set m_MyLinkList.Container = Me
End Sub

' Не вызывается никогда. О том, как решить эту проблему,
' будет рассказано ниже.
Private Sub Class_Terminate()
  Debug.Print "Terminating customer " & CustomerName
End Sub

' Методы и свойства просто отображаются на соответствующие
' методы и свойства LinkList
Private Sub LinkList_Append(Root As LinkList)
  m_MyLinkList.Append Root
End Sub

Private Property Set LinkList_NextItem(ByVal nextobject As LinkList)
  Set m_MyLinkList.NextItem = nextobject
End Property

Private Property Get LinkList_NextItem() As LinkList
  Set LinkList_NextItem = m_MyLinkList.NextItem
End Property

Private Property Get LinkList_PreviousItem(Root As LinkList) As LinkList
  Set LinkList_PreviousItem = m_MyLinkList.PreviousItem(Root)
End Property

Private Sub LinkList_Remove(Root As LinkList)
  m_MyLinkList.Remove Root
End Sub
```

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

На форме TestForm.frm находятся: текстовое поле для ввода имени клиента, кнопка для включения нового клиента в перечень, список с именами клиентов и кнопка для удаления текущей выбранной строки из списка. Модуль содержит переменную m_List, в которой хранится ссылка на первый объект в списке. Хотя переменная m_List относится к типу LinkList, она, как и прежде, ссылается на объект-контейнер, а не на внутренний объект LinkList:

```
' LinkList VB6 пример #1
' Copyright ©2000 by Desaware Inc.  All Rights Reserved

Option Explicit

Dim m_List As LinkList
```

VB6 позволяет работать со свойствами и методами объекта при помощи переменной, представляющей этот интерфейс. Для получения списка объектов нам понадобятся две переменные: объект LinkList для перебора элементов (свойство NextItem) и объект Customer для получения имени клиента (свойство CustomerName). Этот принцип — каждый интерфейс обладает собственным набором свойств и каждый объект может поддерживать несколько интерфейсов — играет очень важную роль в COM. Присваивание переменной одного типа значения другого типа (как, например, в приведенной ниже строке Set currentcustomer = currentlist) сопровождается вызовом метода QueryInterface, используемого COM для перехода между интерфейсами объекта. Вскоре вы поймете, почему я подчеркиваю это обстоятельство. Код формы Testfrm.frm выглядит следующим образом:

```
' Обновление списка
' Обратите внимание на использование разных переменных
' для работы с интерфейсами
Private Sub UpdateList()
  lstCustomers.Clear
  Dim currentlist As LinkList
  Dim currentcustomer As Customer
  Set currentlist = m_List
  Do While Not currentlist Is Nothing
    Set currentcustomer = currentlist
    lstCustomers.AddItem currentcustomer.CustomerName
    Set currentlist = currentlist.NextItem
  Loop
End Sub
```

Как показано в листинге 5.2, в функциях cmdAdd_Click и cmdRemove_Click для работы с объектами также используются переменные двух типов.

Листинг 5.2. Вставка и удаление элементов списка

```
  Dim newEntry As New Customer
  Dim newEntryll As LinkList
  newEntry.CustomerName = txtCustomerName.Text
  Set newEntryll = newEntry
  newEntryll.Append m_List
  UpdateList
End Sub

Private Sub cmdRemove_Click()
```

```
  Dim currentlist As LinkList
  Dim currentcustomer As Customer

  Set currentlist = m_List
  Do While Not currentlist Is Nothing
    Set currentcustomer = currentlist
    If currentcustomer.CustomerName = lstCustomers.Text Then
      currentlist.Remove m_List
      UpdateList
      Exit Sub
    End If
    Set currentlist = currentlist.NextItem
  Loop
  UpdateList
End Sub
```

Какие же выводы можно сделать из этого примера?

- Включение и наследование интерфейсов позволяет организовать повторное использование функциональности при минимальном объеме кода, необходимого для его реализации в объекте-контейнере (`Customer`).
- Наследование интерфейсов неудобно тем, что вам приходится явно переключаться между разными интерфейсами для вызова их методов. В нашей тестовой форме это проявилось в дополнительных переменных и операциях присваивания.

Кстати, вы заметили настоящий, убийственный недостаток этого примера? Если не заметили — запустите его, переключитесь в окно отладки (Immediate window) и посмотрите, что происходит при удалении объектов из списка. Ну как, поняли?

- Необходимость хранения ссылки на контейнер во внутреннем объекте и хранения ссылки на внутренний объект в контейнере означает, что вы автоматически создаете циклические ссылки!

Если бы вы действительно захотели реализовать этот путь на практике, вам пришлось бы каким-то образом разорвать циклические ссылки. В самом распространенном решении вложенный объект инициирует событие, параметром которого является переменная `Object`, передаваемая по ссылке. Получив это событие, контейнер должен присвоить параметру значение `Me`. Таким образом, объект `LinkList` получает ссылку на свой контейнер, после чего он может воспользоваться полученной ссылкой для сравнения или вернуть ее как значение свойства. Ссылку на контейнер необходимо немедленно освободить, чтобы избежать зацикливания. Основными недостатками подобного решения является его громоздкость и плохое быстродействие. Раннее связывание (early binding) на события не распространяется, и это может серьезно замедлить работу программы.

Связанные списки с применением включения в VB .NET

Начнем с прямой адаптации предыдущего примера из VB6 в VB .NET (проект LinkListNet2). Заодно вы увидите, как сильно изменился синтаксис в VB .NET. Несмотря на все различия, понять код не так уж трудно, но это лишний раз дока-

зывает, что адаптация крупных приложений из VB6 в VB .NET — задача весьма серьезная.

ПРИМЕЧАНИЕ

Во всех примерах программ, приведенных в этой книге, устанавливается флажок Option Strict (вкладка Build диалогового окна Project Properties). Я настоятельно рекомендую устанавливать этот флажок во всех проектах VB .NET. Обоснования будут приведены в главе 8.

Сразу бросается в глаза первое изменение: в VB .NET концепция интерфейса отделена от концепции класса (реализации интерфейса). В VB6 добавление новых методов в объект автоматически определяло его интерфейсе. Если вы захотели определить интерфейс для реализации или совместного использования в VB .NET, вам придется явно определить его как интерфейс. Ниже приведен интерфейс `ILinkList`[1], определяемый в файле `LinkList.vb`.

```
' LinkList .Net - пример с агрегированием
' Copyright © 2001 by Desaware Inc. All Rights Reserved

' Классы теперь не реализуются - только интерфейсы,
' поэтому связанный список определяется в виде интерфейса
Public Interface ILinkList
  WriteOnly Property Container() As Object

  Property NextItem() As ILinkList

  ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList

  Sub Remove(ByRef Root As ILinkList)

  Sub Append(ByRef Root As ILinkList)
End Interface
```

Итак, интерфейс у нас есть, но к нему еще нужна реализация (которая может использоваться объектом `Customer`). Объект `LinkList` реализует интерфейс `ILinkList`. На этот раз переменная `m_Next` указывает на интерфейс `ILinkList`, а не на объект `LinkList`. Почему? Потому что мы собираемся хранить в этой переменной ссылки на объект-контейнер, как в предыдущем примере для VB6. Контейнер будет реализовывать *интерфейс* `ILinkList`, а не *объект* `LinkList`:

```
Public Class LinkList
  Implements ILinkList

  ' Данная версия демонстрирует внедрение объектов,
  ' поэтому ссылка указывает на объект-контейнер

  Private m_Next As ILinkList
  ' Ссылка на контейнер необходима для включения
  ' нового элемента в список
  Private m_Container As Object
```

Синтаксис реализации функций интерфейса в VB .NET выглядит несколько иначе. Вместо имен методов вида `ILinkList_Container` в объявлении метода явно указывается, какой метод он реализует. Имя метода, используемое в классе,

[1] По общепринятым правилам имена интерфейсов всегда начинаются с буквы I.

может не иметь ничего общего с именем метода в интерфейсе. Более того, один метод может реализовывать несколько методов интерфейса. Вместо отдельных методов Get/Set/Let блоки Get и Set теперь входят в определение свойства.

Блок Get возвращает значение свойства присваиванием имени свойства (как и в VB6 или при использовании команды Return). Блок Set получает присваиваемое значение в виде встроенной переменной Value. Поскольку в VB .NET команда Set не может использоваться для присваивания объектов, задать значение свойства можно только одним способом. Пусть имя блока Set вас не смущает: разработчики взяли методы VB6 Set и Let, объединили их и выделили в блок Set.

Синтаксис свойств тоже изменился.

```
' Просто вернуть ссылку на следующий объект в списке.
' Обратите внимание на изменения в синтаксисе.
' См. описание команды Implements в тексте.
Public Property NextItem() As ILinkList Implements ILinkList.NextItem
  Get
    NextItem = m_Next
  End Get
  Set(ByVal Value As ILinkList)
    m_Next = Value
  End Set
End Property
```

В VB6 доступность свойства только для чтения или записи определялась только наличием или отсутствием методов Get/Set. В VB .NET доступ к свойству контролируется при помощи атрибутов WriteOnly и ReadOnly. При ограничении доступа в реализацию свойства включается только нужный блок[1].

```
' Контейнер задается в процессе инициализации
' Если вы захотите оформить класс в виде компонента,
' потребуется объявление с атрибутом Public
Friend WriteOnly Property Container() As Object Implements
ILinkList.Container
  Set(ByVal Value As Object)
    m_Container = Value
  End Set
End Property
```

Если не считать уже упоминавшихся изменений в синтаксисе, код свойства PreviousItem и методаRemove очень похож на код VB6.

```
' В односвязном списке метод свойства Previous
' должен провести поиск от начала списка
Public ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList
Implements ILinkList.PreviousItem
  Get
    Dim currentitem As ILinkList
    ' Помните: все ссылки относятся к объекту-контейнеру!
    If (Root Is m_Container) Or (Root Is Nothing) Then
      Exit Property
    End If
```

[1] Зачем задавать атрибуты ReadOnly и WriteOnly, если язык сам сможет определить уровень доступа по наличию или отсутствию методов Set/Get? Хороший вопрос. Вероятно, разработчики могли пойти по этому пути, но использование ReadOnly/WriteOnly лучше соответствует архитектуре .NET. Ключевые слова ReadOnly и WriteOnly в действительности являются *атрибутами*; эта тема подробно рассматривается в главе 11.

```
    currentitem = Root
    Do
      If currentitem.NextItem Is m_Container Then
        PreviousItem = currentitem
        Exit Property
      Else
        currentitem = currentitem.NextItem
      End If
    Loop While Not currentitem Is Nothing
  End Get
End Property

' Метод Remove находит предыдущий элемент списка
' последовательным перебором.
' Переменная Root должна передаваться по ссылке: это позволяет
' присвоить ей новое значение, если удаляемый объект
' находится в начале списка.
Public Sub Remove(ByRef Root As ILinkList) Implements ILinkList.Remove
  Dim previtem As ILinkList

  previtem = PreviousItem(Root)
  If previtem Is Nothing Then
    Root = m_Next
  Else
    previtem.NextItem = m_Next
  End If
End Sub
```

Метод `Append` очень похож на одноименный метод VB6 и отличается от него всего в двух местах: при присваивании `m_Container` переменной `Root` и свойству `NextItem`. В VB6 задача решалась простым присваиванием. В новой версии нам приходится явно преобразовывать объект `m_Container` в тип `LinkList`. Почему?

На первый взгляд может показаться, что перед нами пример более строгой проверки типов, воплощенной в VB .NET. До определенной степени это действительно так, но на самом деле здесь происходит нечто большее. Несколькими абзацами ниже я помогу вам найти ответ на этот вопрос, который, как я надеюсь, продемонстрирует некоторые фундаментальные различия между VB6 и VB .NET.

```
' Метод Append последовательно перебирает элементы
' от Root до конца списка.
Public Sub Append(ByRef Root As ILinkList) Implements ILinkList.Append
  Dim currentitem As ILinkList
  currentitem = Root
  If Root Is Nothing Then
    Root = CType(m_Container, ILinkList)
  Else
    While Not currentitem.NextItem Is Nothing
      currentitem = currentitem.NextItem
    End While
    currentitem.NextItem = CType(m_Container, ILinkList)
  End If
End Sub

End Class
```

Объект `Customer` определяется в файле Customer.vb (листинг 5.3). Он реализует определенный ранее интерфейс `ILinkList` и обладает свойством `CustomerName`, как

и одноименный объект из примера для VB6. Если не считать синтаксических изменений VB .NET, этот класс практически идентичен объекту VB6 `Customer`.

Листинг 5.3. Класс Customer из файла Customer.vb

```
' LinkList .Net - пример с агрегированием
' Copyright © 2001 by Desaware Inc. All Rights Reserved

Public Class Customer
  ' Эта версия реализует интерфейс ILinkList,
  ' методы которого вызываются через объекты LinkList
  Implements ILinklist

  Public CustomerName As String

  ' Функциональность обеспечивается внутренним объектом LinkList
  Private m_MyLinkList As LinkList

  Public Sub New()
    MyBase.New()
    m_MyLinkList = New LinkList()
    m_MyLinkList.Container = Me
  End Sub

  ' В VB .NET объекты деинициализируются.
  ' Подробности приведены в тексте.
  Protected Overrides Sub Finalize()
    System.diagnostics.Debug.WriteLine ("Terminating customer " + _
      CustomerName)
  End Sub

  ' Методы и свойства просто отображаются на соответствующие
  ' методы и свойства LinkList
  Public Sub Append(ByRef Root As ILinkList) Implements ILinkList.Append
    m_MyLinkList.Append (Root)
  End Sub

  Public Property NextItem() As ILinkList Implements ILinkList.NextItem
    Set(ByVal Value As ILinkList)
      m_MyLinkList.NextItem = Value
    End Set
    Get
      NextItem = m_MyLinkList.NextItem
    End Get
  End Property

  Friend WriteOnly Property Container() As Object _
  Implements ILinkList.Container
    Set(ByVal Value As Object)
      m_MyLinkList.Container = Value
    End Set
  End Property

  Public ReadOnly Property PreviousItem(ByVal Root As ILinkList) _
  As ILinkList Implements ILinkList.previousitem
    Get
      PreviousItem = m_MyLinkList.PreviousItem(Root)
    End Get
  End Property
```

продолжение ⇩

Листинг 5.3 (продолжение)

```
  Sub Remove(ByRef Root As ILinkList) Implements ILinkList.Remove
    m_MyLinkList.Remove (Root)
  End Sub

End Class
```

Перейдем к определению формы, находящемуся в файле TestForm.vb. Я не буду приводить вспомогательный код, сгенерированный средой программирования, поскольку он не относится к теме. Как и в примере для VB6, в программе объявляется переменная, указывающая на первый объект в списке:

```
Public Class Form1
  Private m_List As ILinkList
```

Функция `UpdateList` заполняет поле списка текущим перечнем клиентов. Между ее двумя версиями существует ряд важных различий.

Одно изменение несущественно: списки работают не так, как раньше. Вместо вызова метода `AddItem` новая строка добавляется включением нового элемента в коллекцию `Items`. На самом деле все элементы Windows Forms[1] обладают синтаксическими и функциональными различиями по сравнению со своими эквивалентами VB6.

Также стоит обратить внимание на явное приведение переменной `m_List` (тип `ILinkList`) к объекту `currentcustomer` (тип `Customer`).

Однако самое принципиальное изменение заключается в том, что эта функция уже не использует для работы с объектом две разные переменные — для интерфейсов `Customer` и `LinkList`. Как показано в листинге 5.4, объект `currentcustomer` позволяет напрямую обращаться к свойствам `CustomerName` и `NextItem`[2].

Листинг 5.4. Функция обновления в TestForm.vb

```
' Обновление списка
' Обратите внимание на то, что необходимость в использовании разных
' переменных отпала — интерфейс ILinkList интегрируется с объектом.
' Также обратите внимание на изменившийся синтаксис ListBox.
Private Sub UpdateList()
  lstCustomers.Items.Clear()
  Dim currentcustomer As Customer
  ' Тем не менее повышение ссылки на интерфейс до уровня объекта
  ' должно сопровождаться явным преобразованием типа.
  ' При неверном типе объекта произойдет ошибка стадии выполнения.
  currentcustomer = CType(m_List, Customer)
  Do While Not currentcustomer Is Nothing
    lstCustomers.Items.Add (currentcustomer.CustomerName)
    currentcustomer = CType(currentcustomer.NextItem, Customer)
  Loop
End Sub
```

Давайте еще раз вернемся к этим важным фактам.

- В VB6 переменная должна соответствовать интерфейсу, методы которого вызываются приложением. В VB .NET переменная `Object` позволяет напрямую работать со всеми методами всех интерфейсов.

---

[1] Этот термин .NET обозначает все элементы управления в архитектуре .NET. Считайте их .NET-аналогами встроенных элементов Visual Basic или элементов ActiveX.

[2] При желании можно при помощи атрибута `Private` скрыть методы, реализующие интерфейс, и разрешить доступ к методам интерфейса только через интерфейсные переменные.

- В VB6 переменной, ссылающейся на интерфейс объекта, можно напрямую присвоить ссылку на другой интерфейс объекта. В VB .NET необходимо выполнить явное преобразование типа от реализованного интерфейса к объекту-контейнеру или другому реализованному интерфейсу.

Речь идет вовсе не о малозначительных деталях. В них отражены фундаментальные изменения базовой архитектуры, которые очень важно правильно понять.

Механизм работы Visual Basic 6 продиктован технологией COM. В COM для ссылок на объекты используются интерфейсные указатели, и через каждый интерфейсный указатель вызываются методы соответствующей группы. Чтобы вызвать метод другого интерфейса, вам придется предварительно получить указатель на него. Интерфейсы соответствуют типам Visual Basic, поэтому перед вызовом методов переменной необходимо присвоить объект соответствующего типа.

VB .NET не базируется на COM, поэтому старые правила здесь не действуют.

При реализации интерфейса в VB .NET фактически происходит наследование. Интерфейс становится составной частью объекта, его подмножеством. Присваивание ссылки на объект переменной с типом унаследованного интерфейса может выполняться напрямую. Это объясняется тем, что VB .NET уже на стадии компиляции знает, что интерфейс реализуется объектом. Например, следующий фрагмент является правильным:

```
Dim il as ILinkList
Dim co as Customer
co = New Customer
il = co
```

Такое присваивание работает, поскольку VB .NET знает, что объект `Customer` всегда реализует интерфейс `ILinkList`.

Однако обратное неверно. Если вы попытаетесь присвоить объекту `Customer` ссылку на объект `ILinkList`, на стадии компиляции VB .NET никак не сможет определить, соответствует ли присваиваемый объект `ILinkList` объекту `Customer`. Переменная `ILinkList` может указывать на объект `LinkList` или на какой-то произвольный объект, реализующий интерфейс `ILinkList`. На стадии выполнения CLR может определить, на какой тип объекта ссылается данная переменная, поэтому присваивание выполняется во время выполнения программы; но поскольку его успешное выполнение не гарантировано, при попытке прямого присваивания (как в следующем фрагменте) компилятор выдает сообщение об ошибке.

```
Dim il as ILinkList
Dim co as Customer
co = New Customer
co = il
```

Вместо этого необходимо выполнить явное преобразование типа функцией `CType`:

```
co = CType(il, Customer)
```

Тем самым вы сообщаете компилятору, что он должен довериться вам и выполнить преобразование так, как приказано. Если `il` не содержит ссылки на объект `Customer`, CLR инициирует ошибку времени выполнения (CLR не допускает присваивание объектам при несовпадении типов). Возникает резонный вопрос:

если CLR все равно проверяет тип во время выполнения, зачем нужно явное преобразование? Ведь у VB6 хватает сообразительности для того, чтобы выполнить эти преобразования за вас. В действительности мы имеем дело с такой замечательной новой возможностью VB .NET, как *жесткая проверка типов* (Strict Type Checking). Если сбросить этот флажок, VB .NET выполнит преобразования автоматически и вам не придется заниматься явным преобразованием. Тем не менее на страницах этой книги я неоднократно рекомендую *не делать* этого. Жесткая проверка типов — одна из самых замечательных особенностей VB .NET. Она улучшает программу, сокращает количество ошибок и ускоряет их диагностику. Начиная работу над любым проектом VB .NET, непременно *включите* жесткую проверку типов[1].

Поскольку объект `Customer` реализует интерфейс `ILinkList`, все методы этого интерфейса напрямую доступны для объекта `Customer`. Старые правила COM здесь не действуют. Поскольку объект `Customer` является надмножеством методов, свойств и всех реализованных интерфейсов объекта `Object`, он может поддерживать все его методы, а также методы всех реализованных интерфейсов.

В остальных функциях формы вы разберетесь без труда. Они очень похожи на функции примера VB6, если не считать синтаксических изменений, о которых говорилось выше. Также на форме появилась новая кнопка `cmdGC`, при нажатии которой выполняется сборка мусора. Попробуйте щелкнуть на этой кнопке после исключения объекта из списка — вы увидите, что объект успешно удаляется. Это доказывает, что в VB .NET нет проблемы с циклическими ссылками.

```
Protected Sub cmdRemove_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdRemove.Click
  Dim currentcustomer As Customer

  currentcustomer = CType(m_List, Customer)

  Do While Not currentcustomer Is Nothing
    If currentcustomer.CustomerName = _
    CStr(lstCustomers().SelectedItem) Then
      currentcustomer.Remove (m_List)
      UpdateList()
      Exit Sub
    End If
    currentcustomer = CType(currentcustomer.NextItem, Customer)
  Loop
  UpdateList()

End Sub

Protected Sub cmdAdd_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdAdd.Click
  Dim newEntry As New Customer()
  newEntry.CustomerName = txtCustomerName().Text
  newEntry.Append (m_List)
  UpdateList()
End Sub
```

[1] Потому что Microsoft зачем-то оставила этот флажок сброшенным по умолчанию.

```
' При нажатии кнопки GC выполняется немедленная сборка мусора.
' Убедитесь в том, что исключенные объекты успешно уничтожаются,
' несмотря на наличие циклических ссылок.
Protected Sub cmdGC_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdGC.Click
  gc.Collect()
  gc.WaitForPendingFinalizers()
End Sub

End Class
```

Подведем итог.

- В VB .NET решена проблема с циклическими ссылками, присутствующая в решении для VB6.
- Обеспечивая прямой доступ ко всем методам и свойствам объекта и реализованных им интерфейсов, VB .NET радикально упрощает программирование при использовании методики включения.
- Объем программного кода, реализующего включение в объекте `Customer`, остается минимальным.

Короче говоря, даже без применения наследования VB .NET значительно упрощает повторное использование кода с применением включения.

Связанные списки с применением наследования в VB .NET

В определенном смысле вы уже встречались с наследованием. Команда `Implements` обеспечивает то, что называется «наследованием интерфейса» — такая возможность существует и в VB6. Под «наследованием интерфейса» понимается реализация объектом всех методов интерфейса, определенных где-то в другом месте (в определении интерфейса в VB .NET или в определении класса в VB6). При наследовании интерфейса объект доступен как с помощью переменной, имеющей тип интерфейса, так и переменной того же типа, что и сам объект.

Проект LinkListNetInh демонстрирует то наследование, о котором так много говорили, — полноценное наследование реализации[1]. При наследовании реализации объект наследует от базового класса не «правила поведения», а фактическую реализацию и при желании может переопределить ее. При этом объект наследует от базового класса реальную функциональность, которую при желании можно переопределить. В таком сценарии объект доступен с помощью переменной, имеющей тип либо базового, либо производного класса.

В листинге 5.5 изменился только объект `Customer` (файл Customer.vb).

---

[1] Возможно, вы также слышали о так называемом «визуальном наследовании» — этот термин обычно встречается в маркетинговой информации о .NET. Ничего подобного в природе не существует. То, что «пиарщики» называют визуальным наследованием, в действительности представляет собой обычное наследование, примененное к объекту (например, форме или элементу) с визуальными характеристиками (выводимому на экран или на принтер, обладающему пользовательским интерфейсом и т. д.).

Листинг 5.5. Объект Customer в проекте LinkListNetInh

```
' LinkList .Net - пример с агрегированием
' Copyright © 2001 by Desaware Inc. All Rights Reserved
Public Class Customer
  Inherits LinkList

  Public CustomerName As String

  Public Sub New()
    MyBase.New()
    Me.Container = Me
  End Sub

  Protected Overrides Sub Finalize()
    System.diagnostics.Debug.WriteLine ("Terminating customer " + _
      CustomerName)
  End Sub
  ' Создавать внутренний объект "LinkList" не нужно -
  ' объект Customer также является и объектом LinkList.
  ' Реализация интерфейса ILinkList тоже не нужна,
  ' его методы и свойства уже реализованы базовым классом.

End Class
```

Закрытый объект `LinkList` куда-то исчез. Вместе с ним исчезли и все реализованные функции. Когда объект `Customer` наследует от объекта `LinkList`, он «становится» объектом `LinkList`. Более того, мы могли бы пойти еще дальше и модифицировать объект `LinkList` так, чтобы в нем не использовалась переменная `Container` — объекты `LinkList` и `Customer` стали одним и тем же.

Здорово, не правда ли?

Вообще-то не очень.

Да, эта программа работает. Она экономит несколько строк кода и берет на себя все операции с внутренним объектом. Однако такое решение совершенно не изменяет клиентскую сторону, а работать с объектом `Customer` ничуть не проще.

Но самый страшный недостаток этого кода относится не столько к технической, сколько к архитектурной стороне дела. Его можно сформулировать следующим образом.

- Объект `Customer` может моделировать многие сущности, например отдельную личность, компанию и т. д. Но он ни при каких условиях не моделирует связанный список.

Наследование должно использоваться только при наличии явной логической связи, при которой *наследующий объект* является частным случаем *наследуемого объекта.*

Если подобная связь не существует, пользуйтесь наследованием интерфейсов и включением. Несколько дополнительных строк кода — совсем небольшая цена за четкость архитектуры (не говоря уже о сокращении затрат на долгосрочную поддержку).

Как ни странно, подобные связи возникают в приложениях не так уж часто. С другой стороны, они постоянно встречаются при построении библиотек классов, используемых разработчиками. В частности, как вы вскоре убедитесь, наследование широко применяется в архитектуре .NET. Да, в каждом написанном вами

приложении или компоненте будут создаваться объекты, производные от классов .NET. Но вам практически не придется создавать объекты, от которых будут порождаться другие объекты.

Впрочем, решение проблемы циклических ссылок в сочетании с простотой вызова методов разных унаследованных интерфейсов позволяет легко и эффективно организовать повторное использование программного кода.

Двойной связанный список

Помимо архитектурных соображений, против применения наследования в данном примере имеются и сугубо практические доводы. Связанные списки часто используются для размещения объектов в определенном порядке. А вдруг потребуется отсортировать объекты по двум критериям сразу, чтобы объект присутствовал сразу в двух списках? Наследовать один и тот же интерфейс дважды нельзя. Можно определить новый интерфейс для поддержки двух связанных списков, но обычно в таких ситуациях используется другой подход, продемонстрированный в проекте LinkListNetDual.

В листинге 5.6 приведена новая версия класса `LinkList`. Она похожа на предыдущую, однако ссылка на контейнер оформлена в виде открытого свойства. Кроме того, использованная архитектура также отличается от предыдущего примера и следует схеме, показанной на рис. 5.1: ссылка теперь указывает на объект `LinkList`, а не на объект-контейнер. Для получения ссылки на контейнер используется свойство `Container`.

Листинг 5.6. Новая версия LinkList: двойное связывание в одном объекте

```
' LinkList .Net - пример с двойным связыванием
' Copyright ©2001 by Desaware Inc.  All Rights Reserved
Public Interface ILinkList

  ' В этой версии связи устанавливаются между внутренними
  ' объектами LinkList.
  Property NextItem() As ilinklist

  ' В этой версии все элементы списка относятся к типу LinkList,
  ' поэтому для обращения к контейнеру используется
  ' открытое свойство Container.
  Property Container() As Object

  ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList

  Sub Remove(ByRef Root As ILinkList)

  Sub Append(ByRef Root As ILinkList)
End Interface

Public Class LinkList
  Implements ILinkList

  ' В этой версии связи устанавливаются
  ' между внутренними объектами LinkList.
  Private m_Next As ILinkList

  ' Получение ссылки на контейнер
  Private m_Container As Object
```

продолжение ⇩

Листинг 5.6 (продолжение)

```
  Friend Property Container() As Object Implements ILinkList.Container
    Get
      Container = m_Container
    End Get
    Set(ByVal Value As Object)
      m_Container = Value
    End Set
  End Property

  ' Просто вернуть ссылку на следующий объект в списке.
  Public Property NextItem() As ILinkList Implements ILinkList.NextItem
  Get
    NextItem = m_Next
    End Get
    Set(ByVal Value As ILinkList)
      m_Next = Value
    End Set
  End Property

  ' В односвязном списке метод свойства Previous
  ' должен провести поиск от начала списка.
  Public ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList
Implements ILinkList.PreviousItem
    Get
      Dim currentitem As ILinkList
      If (Root Is Me) Or (Root Is Nothing) Then
        Exit Property
      End If
      currentitem = Root
      Do
        If currentitem.NextItem Is Me Then
          PreviousItem = currentitem
          Exit Property
        Else
          currentitem = currentitem.NextItem
        End If
    Loop While Not currentitem Is Nothing
    End Get
  End Property

  ' Метод Remove находит предыдущий элемент списка
  ' последовательным перебором.
  ' Переменная Root должна передаваться по ссылке: это позволяет
  ' присвоить ей новое значение, если удаляемый объект
  ' находится в начале списка.
  Public Sub Remove(ByRef Root As ILinkList) Implements ILinkList.Remove
    Dim previtem As ILinkList

    previtem = PreviousItem(Root)
    If previtem Is Nothing Then
      Root = m_Next
    Else
      previtem.NextItem = m_Next
    End If
  End Sub
```

```
  ' Метод Append последовательно перебирает элементы
  ' от Root до конца списка.
  Public Sub Append(ByRef Root As ILinkList) Implements ILinkList.Append
    Dim currentitem As ILinkList
    currentitem = Root
    If Root Is Nothing Then
      Root = Me
    Else
      While Not currentitem.NextItem Is Nothing
        currentitem = currentitem.NextItem
      End While
      currentitem.NextItem = Me
    End If
  End Sub

End Class
```

Объект `Customer` (листинг 5.7) переработан весьма основательно, поскольку он должен поддерживать принадлежность объекта к двум спискам одновременно. Объект уже не наследует интерфейс `ILinkList`: при таком подходе он ограничивался бы всего одним связанным списком. Вместо этого объект поддерживает разные методы для разных списков (например, `NextItem1` и `NextItem2`).

Такой подход увеличивает объем программного кода. Это объясняется тем, что объект `LinkList` умеет объединять в список только объекты, реализующие интерфейс `ILinkList` (как и он сам).

Листинг 5.7. Объект Customer с поддержкой нескольких связанных списков

```
' LinkList .Net - пример с двойным связыванием
' Copyright ©2001 by Desaware Inc.  All Rights Reserved
Public Class Customer
  Public CustomerName As String

  ' Эта версия показывает, как включить один элемент сразу
  ' в два списка. Обратите внимание: объект НЕ реализует
  ' LinkList командой Implement. Хотя внутренние связи
  ' устанавливаются между объектами LinkList, окружающий мир
  ' видит только объекты Customer; таким образом,
  ' все параметры и возвращаемые значения методов
  ' относятся к типу Customer, а не LinkList.
  Private m_MyLinkList1 As ILinkList
  Private m_MyLinkList2 As ILinkList

  Public Sub New()
    MyBase.New()
    m_MyLinkList1 = New LinkList()
    m_MyLinkList1.Container = Me
    m_MyLinkList2 = New LinkList()
    m_MyLinkList2.Container = Me
  End Sub

  Protected Overrides Sub Finalize()
    System.diagnostics.Debug.WriteLine ("Terminating customer " + _
      CustomerName)
  End Sub

  ' Поскольку ссылка указывает на внутренний объект,
  ' необходимо организовать доступ к нему из других
  ' элементов списка.
```

продолжение ⇩

Листинг 5.7 (продолжение)

```
Friend ReadOnly Property LinkList1() As ILinkList
  Get
    LinkList1 = m_MyLinkList1
  End Get
End Property

Friend ReadOnly Property LinkList2() As ILinkList
  Get
    LinkList2 = m_MyLinkList2
  End Get
End Property

' Функции требуют некоторой дополнительной работы для проверки
' граничных условий (например, пустого списка).
Public Sub Append1(ByRef Root As Customer)
  If Root Is Nothing Then
    Root = Me
  Else
    ' Если Root = Nothing, произойдет ошибка.
    m_MyLinkList1.Append (Root.m_MyLinkList1)
  End If
End Sub

Public Sub Append2(ByRef Root As Customer)
  If Root Is Nothing Then
    Root = Me
  Else
    m_MyLinkList2.Append (Root.m_MyLinkList2)
  End If
End Sub

' Еще раз посмотрите, как реализация использует интерфейс
' ILinkList, а пользователи объекта Customer видят только
' ссылки на объекты Customer.
Public ReadOnly Property NextItem1() As Customer
  Get
    Dim nextref As ILinkList
    nextref = m_MyLinkList1.NextItem
    ' Необходима явная проверка nextref,
    ' в противном случае вызов enextref.Container
    ' завершится неудачей.
    If nextref Is Nothing Then
      nextitem1 = Nothing
    Else
      ' nextref.container относится к типу Object.
      ' Необходимо явное преобразование типа.
      nextitem1 = CType(nextref.Container, Customer)
    End If

  End Get
End Property

Public ReadOnly Property NextItem2() As Customer
  Get
    Dim nextref As ILinkList
    nextref = m_MyLinkList2.NextItem
    If nextref Is Nothing Then
```

```
      nextitem2 = Nothing
    Else
      nextitem2 = CType(nextref.Container, Customer)
    End If
  End Get
End Property


Public ReadOnly Property PreviousItem1(ByVal Root As Customer) As Customer
  Get
    PreviousItem1 = CType(m_MyLinkList1.PreviousItem(Root.LinkList1), _
                    Customer)
  End Get
End Property

Public ReadOnly Property PreviousItem2(ByVal Root As Customer) As Customer
  Get
    PreviousItem2 = CType(m_MyLinkList2.PreviousItem(Root.LinkList1), _
                    Customer)
  End Get
End Property


Sub Remove1(ByRef Root As Customer)
  Dim llroot As ILinkList
  llroot = Root.LinkList1
  ' Почему бы не воспользоваться командой
  ' m_MyLinkList.Remove Root.LinkList1?
  ' Потому что Root.LinkList1 копируется
  ' во временную переменную, которая затем
  ' передается по ссылке; изменения этой временной
  ' переменной никак не отразятся в Root.LinkList1.
  ' Следовательно, для проверки изменений необходимо
  ' использовать промежуточную переменную.
  m_MyLinkList1.Remove (llroot)
  If llroot Is Nothing Then
    Root = Nothing
  Else
    Root = CType(llroot.Container, Customer)
  End If
End Sub

Sub Remove2(ByRef Root As Customer)
  Dim llroot As ILinkList
  llroot = Root.LinkList2
  m_MyLinkList2.Remove (llroot)
  If llroot Is Nothing Then
    Root = Nothing
  Else
    Root = CType(llroot.Container, Customer)
  End If
End Sub

End Class
```

В листинге 5.8 приведен обновленный код формы TestForm.vb. На форме появился второй список для клиентов, имена которых начинаются с букв A–M. Обратите внимание: обе переменные со ссылками на списки (`m_List` и `m_ListAToM`)

теперь относятся к типу Customer. В сущности, форма и не подозревает о существовании интерфейса ILinkList, который теперь все равно не может использоваться для обращения к объекту Customer, поскольку он не участвует в наследовании. В остальном код формы напоминает уже виденное, если не считать дополнений для работы с двумя списками.

Листинг 5.8. Форма TestForm.vb с поддержкой двух связанных списков

```
' LinkList .Net - пример с двойным связыванием
' Copyright ©2001 by Desaware Inc.  All Rights Reserved

Public Class Form1
  Inherits System.Windows.Forms.Form

  ' Обратите внимание: корневые ссылки теперь относятся к типу
  ' Customer вместо типа LinkList.
  Private m_List As Customer
  Private m_ListAtoM As Customer

  Protected Sub cmdGC_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles cmdGC.Click
    gc.Collect()
    gc.WaitForPendingFinalizers()
  End Sub

  ' Удаление из обоих списков
  Protected Sub cmdRemove_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles cmdRemove.Click
    Dim currentcustomer As Customer

    currentcustomer = m_List
    Do While Not currentcustomer Is Nothing
      If currentcustomer.CustomerName = CStr(lstCustomers().SelectedItem) Then
        currentcustomer.Remove1 (m_List)
        Exit Do
      End If
      currentcustomer = currentcustomer.NextItem1
    Loop

    currentcustomer = m_ListAtoM
    Do While Not currentcustomer Is Nothing
      If currentcustomer.CustomerName = _
        CStr(lstCustomers().SelectedItem) Then
        currentcustomer.Remove2 (m_ListAtoM)
        Exit Do
      End If
      currentcustomer = currentcustomer.NextItem2
    Loop

    UpdateList()

  End Sub

  ' В этом простом примере все клиенты с именами >"M
  ' исключаются из второго списка.
  ' Также обратите внимание на то, что нам уже не приходится
  ' использовать разные переменные Customer и LinkList -
  ' мы всегда работаем только с Customer.
```

```
  Protected Sub cmdAdd_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles cmdAdd.Click
    Dim newEntry As New Customer()
    newEntry.CustomerName = txtCustomerName().Text
    newEntry.Append1 (m_List)
    If UCase(Strings.Left(newEntry.CustomerName, 1)) <= "M" Then
      newEntry.Append2 (m_ListAtoM)
    End If

    UpdateList()
  End Sub

  ' Вывести содержимое обоих связанных списков
  Private Sub UpdateList()
    lstCustomers().Items.Clear()
    lstAtoM().Items.Clear()

    Dim currentcustomer As Customer
    currentcustomer = m_List
    Do While Not currentcustomer Is Nothing
      lstCustomers().Items.Add (currentcustomer.CustomerName)
      currentcustomer = currentcustomer.NextItem1
    Loop

    currentcustomer = m_ListAtoM
    Do While Not currentcustomer Is Nothing
      lstAtoM().Items.Add (currentcustomer.CustomerName)
      currentcustomer = currentcustomer.NextItem2
    Loop

  End Sub

End Class
```

Проект LinkListVB6-2 показывает, как реализовать двойное связывание в VB6 (однако и в этом примере существует проблема циклических ссылок).

Конфликты имен

Итак, VB .NET предоставляет все методы и свойства наследуемых объектов и интерфейсов пользователям объекта верхнего уровня. А что произойдет, если в двух интерфейсах будут определены методы с одинаковыми именами?

На самом деле ответ уже встречался в примерах программ, хотя этот вопрос и не ставился напрямую. Секрет кроется в секции `Implements` объявления метода. В листинге 5.9 приведен класс Class1.vb из проекта TwoInterfaces. В пространстве имен определены два интерфейса, содержащие методы с одинаковыми именами `CommonFunction`.

Листинг 5.9. Реализация интерфейсов с одинаковыми именами методов

```
' Пример разрешения конфликтов имен в интерфейсах
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Interface MyFirstInterface
  Sub UniqueFunction()
  Sub CommonFunction()
End Interface
```

продолжение ☞

Листинг 5.9 (продолжение)

```
Interface MySecondInterface
  Sub SecondUniqueFunction()
  Sub CommonFunction()
End Interface

Public Class Class1
  Implements MyFirstInterface
  Implements MySecondInterface

  Sub UniqueFunction() Implements MyFirstInterface.UniqueFunction

  End Sub

  Sub SecondUniqueFunction() Implements _
    MySecondInterface.SecondUniqueFunction

  End Sub

  ' Эти функции следовало бы объявить закрытыми,
  ' чтобы избежать возможных недоразумений.
  Sub CommonFunction() Implements MyFirstInterface.CommonFunction
    Console.WriteLine ("Common Function on first interface")
  End Sub

  Sub CommonFunctionSecondInterface() _
      Implements MySecondInterface.CommonFunction
    Console.WriteLine ("Common function on second interface")
  End Sub

End Class
```

Как видите, проблема решается переименованием одного из методов. Синтаксис команды `Implements` позволяет назначать реализованным функциям разные имена внутри класса, благодаря чему вы можете (а вернее, обязаны) избавиться от конфликтов имен.

```
' Пример разрешения конфликтов имен в интерфейсах
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Module Module1

  Public Sub Main()
    Dim c As New class1()
    Dim i1 As MySecondInterface
    c.CommonFunction()
    i1 = c
    i1.CommonFunction()
    Console.ReadLine()
  End Sub

End Module
```

Проблемы с реализацией решены, но как насчет вызывающей стороны?

При вызове открытого метода `CommonFunction` (команда `c.CommonFunction`) выводится строка `Common Function on first interface`. Однако при вызове `CommonFunction` через переменную `il` (`il.CommonFunction`) будет выведена строка `Common Function on second interface`; следовательно, при этом вызывается метод `CommonFunctionSecondInterface`.

Подобных ситуаций следует избегать из-за высокой вероятности ошибок. Простейшее решение заключается в ограничении доступа к методу на уровне класса, для чего метод объявляется с атрибутом `Private`. Метод всегда можно вызвать через интерфейсную переменную, поскольку в интерфейсе метод объявлен открытым.

Конечно, в этом случае для вызова метода вам придется присвоить объект переменной соответствующего типа.

Наследование в .NET

Наследование широко используется в архитектуре .NET. Среда Common Language Runtime поддерживает только одиночное наследование. Если бы эта книга предназначалась для программистов C++, в этот момент я бы начал подробно расписывать преимущества одиночного наследования перед множественным. Поскольку книга написана для программистов Visual Basic, вряд ли стоит тратить время на разговоры о множественном наследовании. Впрочем, если вам кажется, будто вас обделили, я скажу следующее: как бы скептически я ни относился к наследованию, к множественному наследованию я отношусь в десять раз хуже. Оно создает в программе множество сложностей и дает слишком мало преимуществ. Я помню всего один случай эффективного применения множественного наследования — ATL. Кстати, это одна из причин, по которой ATL так трудно изучать. Я использовал множественное наследование в своей работе всего один раз, да и то по ошибке[1].

Объекты, сплошные объекты...

Любые разговоры о наследовании в .NET должны начинаться с объектов[2]. Как упоминалось выше, для борьбы с утечкой памяти CLR следит за объектами и освобождает их в тот момент, когда на них отсутствуют явные ссылки. Но что происходит с данными, которые не являются объектами? Как CLR управляет этими данными и обеспечивает их своевременное освобождение?

Весьма каверзный вопрос.

В CLR любой элемент данных является объектом. Каждая структура — это объект. Даже обычное целое число — тоже объект. Просмотрите модуль Module1.vb из проекта IntegerObject, приведенный в листинге 5.10.

Листинг 5.10. Консольное приложение IntegerObject

```
' Демонстрация объектной сущности всех данных.
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Module Module1

  Sub Main()
    console.WriteLine ("This is a test")
```

продолжение

[1] На всякий случай сообщаю, что в C# множественное наследование тоже не поддерживается.

[2] Постойте-ка! Разве мы не говорили о наследовании с самого начала главы? Да, но тогда речь шла о наследовании как концепции и возможности языка. Сейчас мы переходим к использованию наследования в .NET.

Листинг 5.10 (продолжение)

```
    Dim i As Integer = 15
    console.WriteLine (i.ToString())
    console.WriteLine ("Hash is: " + i.GetHashCode().ToString())
    console.WriteLine ("Type is: " + i.GetType().ToString)
    console.WriteLine ("Type full name is: " + i.GetType().FullName())
    console.WriteLine ("Type assembly qualified name is: " + _
      i.GetType().AssemblyQualifiedName)
    console.WriteLine ("Type assembly qualified name is: " + _
      i.GetType().Namespace)

    console.Write ("Press Enter to continue")
    console.ReadLine()

  End Sub

End Module
```

Да, перед вами консольное приложение. Эта категория приложений теперь очень хорошо поддерживается VB .NET. Консольные приложения удобны для тестирования и объяснения новых концепций, поскольку они не требуют лишних затрат энергии на создание формы.

В этом фрагменте заслуживает внимания целочисленная переменная `i`. Хотя она объявлена с типом `Integer`, для нее можно вызывать методы. Вы когда-нибудь слышали о целых числах с методами?

Это объясняется тем, что тип `Integer`, как и любой тип переменной, создается производным от типа `Object` и для него, как для любого объекта, определяются некоторые методы. Метод `ToString` возвращает строковое представление данных переменной. Метод `GetHashCode` возвращает хэш-код объекта, используемый при поиске в коллекциях объектов. Метод `Equals` (в листинге не приведен) предназначен для сравнения объектов; он позволяет сравнивать объекты на основании внутренних данных объекта вместо простого сравнения ссылок в двух переменных. Метод `GetType` возвращает описание типа объекта — метаданные, содержащие полную информацию о свойствах и методах объекта.

Выходные данные программы IntegerObject выглядят так[1]:

```
This is a test
15
Hash is: 15
Type is: Int32
Type full name is: System.Int32
Type assembly qualified name is: System.Int32, mscorlib, Version=1.0…..,Cul
ture=neutral,PublicKeyToken=b77a5c561934e089Type assembly qualified name is:
System
Press Enter to continue
```

Как правило, для типа `Integer` из всех базовых методов вызывается только метод `ToString`. Другие методы обычно используются в более сложных объектах, которые переопределяют встроенную реализацию и оптимизируют ее для своих целей. О переопределении функций будет рассказано ниже.

[1] Вероятно, в вашем случае значения Version и PublicKey будут другими. Полное имя зависит от того, используете ли вы бета-версию, промежуточную сборку или окончательную версию исполнительной среды .NET.

Преобразование стандартных типов данных в объекты обеспечивает логическое единство, необходимое CLR для управления данными в приложениях. Возникает резонный вопрос: а не слишком ли дорого за это приходится платить и не повлияет ли поддержка методов для простых целочисленных переменных на быстродействие приложения? Ведь целое число обычно представляет собой простую переменную, выделенную в стеке, а объект неизбежно приводит к лишним затратам ресурсов.

К счастью, CLR неплохо справляется с этой проблемой. Оказывается, в .NET объекты делятся на две категории. Ссылочные (reference) объекты вам хорошо знакомы — они реализуются при помощи классов, хранятся в куче, а для обращения к ним используются ссылки. К категории структурных (value) объектов относятся «облегченные» объекты, которые могут храниться в стеке. Пока структурные объекты используются как простые переменные (например, в математических операциях для целых чисел), компилятор генерирует точно такой же код, как и в текущей версии Visual Basic. Загрузка, сохранение и математические вычисления выполняются непосредственно с числовыми данными. Но при использовании структурного объекта в ссылочном контексте CLR выполняет операцию, называемую *упаковкой* (boxing) — данные преобразовываются во временный объект, с которым выполняется необходимая операция, после чего объект снова распаковывается (unboxing). Для объявления объектов структурного типа в Visual Basic используется ключевое слово `Structure` (см. главу 6).

Формы

До настоящего момента мы не обращали внимания на код формы, сгенерированный дизайнером форм. В листинге 5.11 приведен фрагмент приложения LinkListNet2. В пространстве имен `System.WinForms` собраны реализации различных Windows-компонентов. Новые формы наследуют от объекта `System.WinForms.Form`.

Полная иерархия объекта `Form` выглядит следующим образом:

```
Object
  MarshalByRefObject
    Component
      Control
        ScrollableControl
          ContainerControl
            Form
```

Каждый тип в этой иерархии наследует функциональность объекта предыдущего уровня. Я не стану углубляться в подробности, но вполне очевидно, что даже перед написанием первой строки программного кода ваша форма уже поддерживает определенный набор базовых возможностей. В VB6 эти возможности просто появлялись на пустом месте, как по волшебству. В VB .NET базовая функциональность форм сохранена, но вы можете точно определить, откуда берутся те или иные ее аспекты. Кроме того, при желании можно создавать объекты, производные от любых промежуточных объектов этой иерархии.

Просматривая листинг 5.11, помните, что все вызываемые методы наследуются от одного из базовых классов. Еще раз подчеркну, что этот код приведен не для подробного анализа, а лишь для демонстрации наследования в .NET.

Листинг 5.11. Файл TestForm.vb (проект LinkListNet2)

```
' LinkList .Net - пример с агрегированием
' Copyright © 2001 by Desaware Inc. All Rights Reserved

Public Class Form1
  Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

  Public Sub New()
    MyBase.New()

    ' Необходимо для дизайнера форм Windows.
    InitializeComponent()

    ' Дальнейшая инициализация выполняется
    ' после вызова InitializeComponent().
  End Sub

  ' Форма переопределяет Dispose для очистки списка компонентов.
  Public Overloads Overrides Sub Dispose()
    MyBase.Dispose()
    If Not (components Is Nothing) Then
      components.Dispose()
    End If
  End Sub
  Private WithEvents cmdAdd As System.Windows.Forms.Button
  Private WithEvents lstCustomers As System.Windows.Forms.ListBox
  Private WithEvents cmdGC As System.Windows.Forms.Button
  Private WithEvents txtCustomerName As System.Windows.Forms.TextBox
  Private WithEvents cmdRemove As System.Windows.Forms.Button
  Private WithEvents label1 As System.Windows.Forms.Label

  ' Необходимо для дизайнера форм Windows.
  Private components As System.ComponentModel.Container

  ' ВНИМАНИЕ: следующий фрагмент необходим для дизайнера
  ' форм Windows.
  ' Для его модификации следует использовать дизайнер форм.
  ' Не изменяйте его в редакторе!
  <System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
    Me.txtCustomerName = New System.Windows.Forms.TextBox()
    Me.cmdGC = New System.Windows.Forms.Button()
    Me.label1 = New System.Windows.Forms.Label()
    Me.cmdAdd = New System.Windows.Forms.Button()
    Me.lstCustomers = New System.Windows.Forms.ListBox()
    Me.cmdRemove = New System.Windows.Forms.Button()
    Me.SuspendLayout()
    '
    'txtCustomerName
    '
    Me.txtCustomerName.Location = New System.Drawing.Point(88, 24)
    Me.txtCustomerName.Name = "txtCustomerName"
    Me.txtCustomerName.Size = New System.Drawing.Size(136, 20)
    Me.txtCustomerName.TabIndex = 4
    Me.txtCustomerName.Text = ""
```

```
    '
    'cmdGC
    '
    Me.cmdGC.Location = New System.Drawing.Point(200, 152)
    Me.cmdGC.Name = "cmdGC"
    Me.cmdGC.Size = New System.Drawing.Size(64, 32)
    Me.cmdGC.TabIndex = 2
    Me.cmdGC.Text = "GC"
    '
    'label1
    '
    Me.label1.Location = New System.Drawing.Point(16, 24)
    Me.label1.Name = "label1"
    Me.label1.Size = New System.Drawing.Size(64, 16)
    Me.label1.TabIndex = 5
    Me.label1.Text = "Customer:"
    Me.label1.TextAlign = System.Drawing.ContentAlignment.MiddleRight
    '
    'cmdAdd
    '
    Me.cmdAdd.Location = New System.Drawing.Point(200, 72)
    Me.cmdAdd.Name = "cmdAdd"
    Me.cmdAdd.Size = New System.Drawing.Size(64, 32)
    Me.cmdAdd.TabIndex = 0
    Me.cmdAdd.Text = "Add"
    '
    'lstCustomers
    '
    Me.lstCustomers.Location = New System.Drawing.Point(24, 72)
    Me.lstCustomers.Name = "lstCustomers"
    Me.lstCustomers.Size = New System.Drawing.Size(160, 108)
    Me.lstCustomers.TabIndex = 3
    '
    'cmdRemove
    '
    Me.cmdRemove.Location = New System.Drawing.Point(200, 112)
    Me.cmdRemove.Name = "cmdRemove"
    Me.cmdRemove.Size = New System.Drawing.Size(64, 32)
    Me.cmdRemove.TabIndex = 1
    Me.cmdRemove.Text = "Remove"
    '
    'Form1
    '
    Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
    Me.ClientSize = New System.Drawing.Size(292, 216)
    Me.Controls.AddRange(New System.Windows.Forms.Control() _
    {Me.label1, Me.txtCustomerName, Me.lstCustomers, Me.cmdGC, _
    Me.cmdRemove, Me.cmdAdd})
    Me.Name = "Form1"
    Me.Text = "Link List Test 2"
    Me.ResumeLayout (False)
  End Sub
```

Наследование в VB .NET

Несомненно, вам придется часто использовать наследование, по крайней мере по отношению к объектам .NET. Пора поближе познакомиться с поддержкой наследования на уровне языка.

В приведенном ниже проекте Customers1 приведен классический пример наследования. Некая организация работает с несколькими категориями клиентов. Все классы клиентов являются производными от общего базового класса `Customer`, объявленного с атрибутом `MustInherit`. Это означает, что создавать объекты типа `Customer` в программе нельзя[1]; допускается лишь создание объектов классов, производных от `Customer`. Классы также могут объявляться с атрибутом `NotInheritable`, запрещающим создание производных классов.

```
' Приложение Customers
' Copyright ©2001 by Desaware inc. All Rights Reserved
Public MustInherit Class Customer
  Public Name As String
  Public MustOverride Function DefaultNet() As Integer
  Public Overridable Function DisplayTerms() As String
    DisplayName()
    Console.WriteLine ("... is net " + CStr(DefaultNet()))
  End Function
  Sub DisplayName()
    Console.WriteLine ("Customer is " + Name)
  End Sub

End Class
```

Методы и свойства класса могут помечаться ключевыми словами `Overridable` и `MustOverride`.

Если метод помечен ключевым словом `Overridable`, производный класс при желании может переопределить его и предоставить собственную реализацию. Если это ключевое слово отсутствует, попытки производного класса создать метод с тем же именем приведут к ошибкам на стадии компиляции. Ключевое слово `MustOverride` означает, что производный класс обязан реализовать данный метод. Естественно, поскольку такой метод всегда реализуется в производных классах, реализовывать его в базовом классе не обязательно.

Переопределение непомеченных свойств и методов по умолчанию запрещается. Для всех методов базового класса с ключевым словом `Overridable` возможность переопределения сохраняется в производном классе и во всех классах, производных от него. Чтобы запретить дальнейшее переопределение метода, достаточно переопределить его в одном из производных классов с ключевым словом `NotOverridable`.

В главе 10 вы также научитесь использовать ключевое слово `Overloads` для объявления методов с именами, совпадающими с именами методов базового класса, но с другими параметрами. Пока я не буду останавливаться на этой возможности, чтобы не усложнять ситуацию.

Рассмотрим три класса, производных от `Customer`. Класс `Commercial` переопределяет функцию `DefaultNet`, чтобы установить стандартный размер платежей для коммерческих организаций. Все остальные методы и свойства этого класса наследуются от базового класса.

```
Public Class Commercial
  Inherits Customer
  Public Overrides Function DefaultNet() As Integer
```

[1] В ООП такие классы принято называть «абстрактными». — *Примеч. перев.*

```
    Return (30)
  End Function
End Class
```

Класс `Individual` также переопределяет метод `DisplayTerms` таким образом, чтобы вместо описания условий платежа выводилось сообщение о необходимости немедленной оплаты.

```
Public Class Individual
  Inherits Customer
  Public Overrides Function DefaultNet() As Integer
    Return (0)
  End Function
  Public Overrides Function DisplayTerms() As String
    DisplayName()
    Console.WriteLine ("... must pay immediately")
  End Function

End Class
```

Класс `Government`, помимо переопределения метода `DisplayTerms`, определяет новый метод `BranchInfo`, который представляет отдел правительственного учреждения. Иначе говоря, объект расширяет функциональность базового класса и добавляет в него новые возможности.

```
Public Class Government
  Inherits Customer
  Public Overrides Function DefaultNet() As Integer
    Return (120)
  End Function
  Public Overrides Function DisplayTerms() As String
    DisplayName()
    Console.WriteLine ("... will pay someday we hope")
  End Function
  Sub BranchInfo()
    Console.WriteLine ("Legislative")
  End Sub

End Class
```

Модуль Module1.vb (листинг 5.12) содержит простое консольное приложение, демонстрирующее применение этих классов.

Листинг 5.12. Модуль Module1.vb из проекта Customer1

```
' Приложение Customers
' Copyright ©2001 by Desaware inc. All Rights Reserved
Module Module1

  Sub Main()
    Dim store As New Commerical()
    store.Name = "Worst buys"
    store.DisplayName()
    store.DisplayTerms()
    console.Write ("Press enter to continue:")
    console.ReadLine()
    Dim Person As New Individual()
    Person.Name = "Jim Smith"
    Person.DisplayName()
    Person.DisplayTerms()
```

продолжение ⇩

Листинг 5.12 (продолжение)

```
    console.Write ("Press enter to continue:")
    console.ReadLine()
    Dim USA As New Government()
    USA.Name = "U.S.A."
    USA.DisplayName()
    USA.DisplayTerms()
    USA.BranchInfo()
    console.Write ("Press enter to continue:")
    console.ReadLine()
    Dim BaseObject As Customer
    BaseObject = USA
    BaseObject.DisplayName()
    BaseObject.DisplayTerms()
    ' BaseObject.BranchInfo()
    console.ReadLine()

  End Sub

End Module
```

Результат выглядит следующим образом:

```
Customer is Worst buys
Customer is Worst buys
... is net 30
Press enter to continue:
Customer is Jim Smith
Customer is Jim Smith
... must pay immediately
Press enter to continue:
Customer is U.S.A.
Customer is U.S.A.
... will pay someday we hope
Legislative
Press enter to continue:
Customer is U.S.A.
Customer is U.S.A.
... will pay someday we hope
Legislative
Press enter to continue:
```

Многие результаты очевидны. При вызове метода для объекта производного класса вместо старого метода будет вызван новый.

Однако последний пример — когда мы создаем переменную типа `Customer` и присваиваем ей один из производных объектов — выглядит более интересно. Прежде всего, стоит заметить, что этот фрагмент подтверждает то, о чем я писал ранее: при переходе от производного класса к базовому классу (или базовому интерфейсу) явное преобразование типов не требуется. Компилятор знает, что объект поддерживает базовый класс или интерфейс, и выполняет соответствующее неявное преобразование.

Следующий интересный момент связан с функцией `DisplayTerms`. Мы используем ссылку на базовый класс `Customer`. Базовый класс имеет собственную реализацию `DisplayTerms`, однако при вызове `DisplayTerms` для объекта базового класса вызывается метод объекта `Government`. Как такое возможно?

CLR знает тип объекта, с которым вы работаете, даже если доступ к нему осуществляется через объект базового класса. Во время выполнения программы CLR обнаруживает, что метод был переопределен, и вызывает правильный метод. Этот механизм называется *полиморфизмом* и входит в число основных особенностей любого настоящего объектно-ориентированного языка. Впрочем, и полиморфный вызов методов можно изменить, используя ключевое слово `Shadows` вместо `Overrides`. Ключевое слово `Shadows` означает, что метод производного класса, хотя и имеет такое же имя, является совершенно другой функцией и вызовы через объект базового класса не должны передаваться новому методу.

В данном примере метод `BranchInfo` невозможно вызвать для переменной `BaseObject`. Хотя производный класс может использовать методы и свойства базового класса, обратное невозможно: базовый класс не может использовать методы и свойства, добавленные в производном классе.

В главе 10 более подробно рассматриваются различные атрибуты и ключевые слова, влияющие на наследование в .NET.

Решение проблемы неустойчивости базового класса

А сейчас я продемонстрирую одну из самых впечатляющих возможностей CLR и ее применение при решении одной проблемы, которая встречается довольно редко, но обычно вызывает серьезные затруднения.

Проект ClassLibrary2 (листинг 5.13) содержит определение и реализацию знакомого класса `Customer` из предыдущего примера. В проекте определяется уникальное пространство имен, поскольку мы собираемся использовать его в других программах.

Листинг 5.13. Класс Customer из проекта ClassLibrary2

```
' Пример ClassLibrary2 (глава 5)
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Namespace MigratingBook.Chapter5.ClassLibrary2
  Public MustInherit Class Customer
    Public Name As String
    Public MustOverride Function DefaultNet() As Integer
    Public Overridable Function DisplayTerms() As String
      DisplayName()
      Console.WriteLine ("... is net " + CStr(DefaultNet()))
    End Function
    Sub DisplayName()
      Console.WriteLine ("Customer is " + Name)
    End Sub
  End Class
End Namespace
```

Проект представляет собой библиотеку классов. В результате компиляции вы получаете DLL, на которую можно ссылаться из других сборок. Происходящее отчасти напоминает создание ActiveX DLL, совместно используемых разными приложениями, хотя в рассматриваемом случае не поддерживаются многие нетривиальные возможности компонентов, о которых будет рассказано ниже. Такой подход является простым и эффективным средством совместного использования программного кода.

Проект Customers2 идентичен проекту Customers1, не считая того, что класс `Customer` определяется не внутри класса, а во внешней библиотеке.

А теперь рассмотрим следующую ситуацию: класс `Government` имеет собственную реализацию `BranchInfo`. Что произойдет, если в один прекрасный день разработчик класса `Customer` добавит в него новую переопределяемую функцию `BranchInfo`, не подозревая о том, что в производном классе уже определен метод с таким именем?

Базовые принципы полиморфизма предполагают, что при вызове `BranchInfo` в коде базового класса должна вызываться функция производного класса. Это приводит к фатальным последствиям, поскольку разработчик базового класса никак не может предвидеть последствия от вызова метода производного класса. Несомненно, будет сделано совсем не то, что должна была сделать реализация базового класса.

Эта проблема называется «проблемой неустойчивости базового класса» и иногда приводит к весьма серьезным последствиями. В той или иной степени она проявляется и в C++ и в Java.

Проекты ClassLibrary3 и Customers3 показывают, как эта проблема решается в .NET. Чтобы увидеть, как возникает эта проблема и как она решается, вам придется воспроизвести ее вручную. Программный код примера соответствует лишь одному из моментов этого процесса.

Для начала приведите модуль Class1.vb из проекта ClassLibrary3 к виду, показанному в листинге 5.14.

Листинг 5.14. Объект Customer из проекта ClassLibrary3

```
' Пример ClassLibrary3 (глава 5)
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Namespace MigratingBook.Chapter5.ClassLibrary3
  Public MustInherit Class Customer
    Public Name As String
    Public MustOverride Function DefaultNet() As Integer
    Public Overridable Function DisplayTerms() As String
      DisplayName()
      Console.WriteLine ("... is net " + CStr(DefaultNet()))
      ' ConsoleWrite(" Branch is: ")
      ' BranchInfo()
    End Function
    Sub DisplayName()
      Console.WriteLine ("Customer is " + Name)
    End Sub
    ' Public Overridable Sub BranchInfo()
    ' Console.WriteLine("New BranchInfo Function")
    ' End Sub
  End Class
End Namespace
```

Перед вами исходное состояние, соответствующее предыдущему примеру. Откомпилируйте библиотеку классов и включите ссылку на нее в проект Customers3 (идентичный приведенному выше проекту Customers2). Теперь откомпилируйте проект Customers3 и убедитесь в том, что программа выдает те же результаты, что и выше.

Теперь восстановите закомментированные строки класса `Customer` в листинге 5.14. Откомпилируйте DLL и скопируйте ее в один каталог с только что построенным исполняемым файлом Customers3.

Метод BranchInfo вызывается в двух местах: в базовом и производном классах. В соответствии с принципом полиморфизма все вызовы метода (даже через ссылку, относящуюся к базовому классу) должны передаваться в производный класс. Тем не менее результат оказывается иным:

```
Customer is Worst buys
Customer is Worst buys
... is net 30
Branch is: New Branchinfo Function
Press enter to continue:
Customer is Jim Smith
Customer is Jim Smith
... must pay immediately
Press enter to continue:
Customer is U.S.A.
Customer is U.S.A.
... will pay someday we hope
Legislative
Press enter to continue:
```

Приведенный вывод соответствует последнему состоянию проекта.

Даже в конечном объекте вызовы функции BranchInfo из базового класса передаются реализации базового класса! Впрочем, при вызове BranchInfo в функции производного класса вызывается реализация производного класса.

Другими словами, если добавить новый метод или свойство в класс или компонент, у которого имеется производный класс с одноименным методом или свойством, а затем откомпилировать этот компонент и распространить его среди пользователей, и компонент, и производный класс будут работать как ни в чем не бывало.

Попробуйте открыть проект Customers3. Неожиданно появляется сообщение об ошибке! VB .NET обнаруживает, что в производном классе имеется метод с именем, совпадающим с именем метода базового класса, но не имеющий атрибута Overrides. Теперь вам предстоит решить, следует ли переопределить метод базового класса (это нужно делать осторожно, хорошо понимая, что именно вы переопределяете), скрыть реализацию базового класса от производного класса при помощи ключевого слова Shadows[1] или переименовать метод.

Видимость методов

Говоря о наследовании, остается рассмотреть последнюю тему — видимость и правила замещения. Большинство методов и свойств в приведенных выше примерах объявлялось с атрибутом Public. В производном классе такие методы и свойства полностью доступны.

Члены классов также могут объявляться с атрибутами Private, Friend и Protected.

Закрытые (Private) члены доступны только внутри класса. Производный класс не наследует их и не может обращаться к ним. Если бы в предыдущем примере функция BranchInfo объекта Customer была объявлена с атрибутом Private,

[1] Ключевое слово Shadows не было реализовано в версии бета-1.

любые потенциальные конфликты с методом `BranchInfo` класса `Government` были бы исключены.

Вероятно, атрибут `Friend` уже знаком вам по VB6. Дружественные члены классов наследуются производными классами, однако остаются видимыми только в пределах сборки.

Защищенные (`Protected`) члены классов относятся к числу нововведений VB .NET. К защищенным членам класса можно обращаться в производном классе, и они наследуются классами, производными от производных. Тем не менее к ним нельзя обращаться за пределами производного класса.

Допустим, у вас имеется класс `A` с защищенной функцией `MyFunc`:

```
Class A
  Protected Sub MyFunc()
  End Sub
End Class
```

и от него создается производный класс `B`:

```
Class B
  Inherits A
  Public Sub MyPublicFunc()
  End Sub
End Class
```

В классе `B` метод `MyFunc` может вызываться напрямую: он был унаследован от класса `A`. Тем не менее, если попытаться объявить переменную типа `B` в другом месте, попытка вызова `B.MyFunc` завершится неудачей. Метод `MyFunc` доступен только в производном классе `B`; для внешних функций он остается невидимым. Метод также можно определить с атрибутами `Protected Friend`; при таком объявлении объединяются характеристики обоих атрибутов.

Итоги

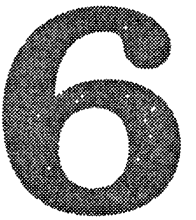
В этой главе были описаны некоторые важные концепции.

- Архитектура .NET в значительной степени основана на наследовании. Все переменные и классы, объявленные в программе, являются производными от одного из классов иерархии (даже если это класс `Object`).
- Хотя ваши классы всегда являются производными от других классов, необходимость в дальнейшем наследовании возникает редко. Я бы даже порекомендовал завести полезную привычку — помечать классы атрибутом `NotOverridable`, чтобы предотвратить их возможное применение способом, на который они не рассчитаны.
- В большинстве программ VB повторное использование кода лучше всего реализуется на базе включения. Благодаря решению проблем с циклическими ссылками и перебором объектов VB .NET избавляет программистов от большинства трудностей с включением, существовавших в VB6.

- Если вы захотите создать объект, предназначенный для дальнейшего наследования, к его проектированию следует подходить с величайшей осторожностью. Хотя проблема неустойчивости базовых классов учтена в VB .NET, ее решение направлено лишь на то, чтобы обновленные компоненты можно было устанавливать без нарушения работы существующего кода. Вам все равно придется перепрограммировать производные классы при последующей компиляции.

Окончательный вердикт: хотя из всех новых возможностей VB .NET именно наследование сопровождалось наибольшей рекламной шумихой, скорее всего, вам никогда не придется создавать классы, предназначенные для дальнейшего наследования.

Управление памятью в VB .NET



В главе 4 вы узнали, что в .NET проблема циклических ссылок решается отслеживанием всех объектов, используемых вашим приложением, и освобождением всех объектов, на которые отсутствуют ссылки, при помощи сборщика мусора.

В главе 5 было сказано, что все элементы данных вплоть до простейшей целой переменной представляют собой объекты.

Следовательно, все элементы данных, используемые в вашем приложении, подчиняются правилам управления памятью в .NET.

И это почти правда...

Ссылочные и структурные объекты

Разработчики .NET хорошо понимали, что было бы ужасно расточительно хранить в куче все данные до последней числовой переменной. Даже несмотря на то, что операции с кучей в .NET отличаются выдающейся эффективностью, в них неизбежно используется операция выделения памяти, которая выполняется значительно медленнее простого размещения данных в стеке. Не следует забывать и о затратах на отслеживание объектов и сборке мусора после освобождения.

Тем не менее разработчики .NET стремились к простоте и логической последовательности, обусловленной наследованием всех переменных от корневого типа `Object`.

Для разрешения этого конфликта им пришлось определить два типа объектов: ссылочные объекты и структурные объекты[1].

Ссылочные (reference) объекты похожи на объекты COM, хорошо знакомые нам по VB6. Присваивание одного ссылочного объекта другому не сопровождается копированием данных: вместо этого вы просто получаете новую ссылку на существующие данные. Ссылочные объекты всегда размещаются в куче. Ссылочные объекты могут наследовать от других объектов, и другие объекты, в свою очередь, могут наследовать от них.

---

[1] На всякий случай поясню, что термины «ссылочный тип» и «ссылочный объект» в данном контексте означают одно и то же (как и термины «структурный тип» и «структурный объект»).

Структурные объекты

Структурные (value) объекты размещаются в сегменте данных приложения или в стеке — не в куче. Структурные объекты VB .NET создаются практически так же, как и пользовательские типы в VB6, если не считать того, что для них используется ключевое слово `Structure`.

Пример простого структурного типа, определяемого в проекте ValueType:

```
Public Structure mystruct
  Public AString As String
  Public Sub SetString(ByVal newstring As String)
    AString = newstring
  End Sub
  Public Sub New(ByVal InitialString As String)
    AString = InitialString
  End Sub
End Structure
```

В этом объявлении следует обратить внимание на ряд интересных обстоятельств.

Прежде всего, структура содержит строковое поле. В VB6 можно было определять строки фиксированного размера, хранящиеся в структуре. В VB .NET тип данных `String` является ссылочным[1]. Таким образом, у вас имеется структурный объект, который содержит переменную ссылочного типа. Так зачем же в этом случае использовать структурный тип?

Абсолютно незачем. Это неудачное решение.

Структурные типы лучше всего подходят для создания небольших объектов, в основном для числовых типов данных, например комплексных чисел или координат точек на плоскости.

В отличие от пользовательских типов VB6, структуры VB .NET могут обладать методами и свойствами. В них также могут определяться конструкторы — методы, предназначенные для инициализации структур. Впрочем, есть одна тонкость: определяемый вами конструктор должен получать хотя бы один параметр. Конструктор по умолчанию, используемый CLR, инициализирует все поля структуры нулями.

Метод `Button1_Click` проекта ValueType выглядит следующим образом:

```
Private Sub Button1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles button1.Click
  Dim a As mystruct
  a.SetString ("Hello")
  Dim b As mystruct = a
  Debug.WriteLine("B's String: " + b.AString)
  Dim c As New mystruct("Another string")
  Debug.WriteLine("C's String: " + c.AString)
  Debug.WriteLine("C's ToString: " + c.ToString())

  Dim obj As Object
  obj = c
```

[1] Строки более подробно описаны в главах 8 и 9 — в VB .NET они заметно отличаются от строк VB6. А пока достаточно помнить, что строка хранится в структуре в виде указателя на блок памяти в куче. Непосредственное содержимое строки в структуре отсутствует.

```
  ' Упаковка в объект
  Debug.WriteLine(obj.ToString())

End Sub
```

В этом методе продемонстрированы три разных способа инициализации структуры `mystruct`.

Команда `Dim a As mystruct` создает структурную переменную `a`. Конструктор по умолчанию инициализирует все поля структуры нулями (строки инициализируются значением `Nothing`). Если бы тип `mystruct` в этой команде был ссылочным, переменная `a` была бы равна `Nothing`: ссылочные объекты должны создаваться перед использованием. Не существует такого понятия, как недействительная или пустая структурная переменная. Следовательно, объявления `Dim a As mystruct` и `Dim a As New mystruct` эквивалентны, если не считать того, что во втором варианте могут использоваться другие конструкторы.

VB .NET позволяет инициализировать переменные синтаксической конструкцией вида:

```
Dim myvaluetype = начальное_значение
```

Таким образом, команда

```
Dim I As Integer = 5
```

является вполне допустимым способом объявить целую переменную и присвоить ей начальное значение 5.

Структуры одного типа можно свободно присваивать друг другу, при этом VB .NET копирует данные структур на уровне полей. Таким образом, следующий фрагмент вполне допустим:

```
Dim a As mystruct
Dim b As mystruct
b = a
```

Такие конструкции можно сокращать до вида:

```
Dim a As mystruct
Dim b As mystruct = a
```

Последний объект `c` использует перегруженный[1] метод `New`, чтобы создание и инициализация строки выполнялись одной командой:

```
Dim c As New mystruct("Another string")
```

Как показывает следующий фрагмент, структурный объект может присваиваться ссылочной переменной:

```
Dim obj As Object
obj = c
```

Когда у CLR возникает необходимость интерпретировать структурный тип в ссылочном контексте, выполняется операция, называемая *упаковкой* (boxing): в куче создается временный объект, и в него копируются члены структурного типа. Обратный процесс называется *распаковкой* (unboxing).

[1] Термин «перегрузка» означает наличие нескольких одноименных методов в классе или структуре. Перегруженные методы различаются только параметрами. Перегрузка подробно рассматривается в главе 10.

При выполнении обработчика события Button1_Click в окне отладки выводится следующий результат:

```
B's String: Hello
C's String: Another string
C's ToString: ValueType.mystruct
ValueType.mystruct
```

Этот пример демонстрирует копирование и инициализацию структур, а также тот факт, что в упакованном ссылочном объекте сохраняется информация о его типе.

Обработчик Button2_Click (листинг 6.1) еще нагляднее поясняет различия между структурными и ссылочными типами.

Листинг 6.1. Копирование объектов ссылочных и структурных типов[1]

```
Private Class myclass1
  Public AString As String
  Public Sub SetString(ByVal newstring As String)
    AString = newstring
  End Sub
  Public Sub New(ByVal InitialString As String)
    AString = InitialString
  End Sub
End Class

Private Sub Button2_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles button2.Click
  Dim S1 As New mystruct("Hello")
  Dim S2 As mystruct
  Dim c1 As New myclass1("Hello")
  Dim c2 As myclass1
  S2 = S1
  c2 = c1
  S2.SetString ("Modified")
  c2.SetString ("Modified")
  Debug.WriteLine("Struct: " + S1.AString)
  Debug.WriteLine("Class: " + c1.AString)
End Sub
```

Класс myclass1 практически идентичен структуре mystruct, и инициализация, казалось бы, происходит одинаковым образом. Что произойдет, если создать копии структуры и объекта, а затем модифицировать их? При нажатии кнопки в окне отладки выводятся значения свойства AString исходной структуры и объекта:

```
Struct: Hello
Class: Modified
```

Как видите, при модификации копии структуры изменяется только копия, а оригинал остается в прежнем состоянии. Присваивание ссылочной переменной означает всего лишь создание новой ссылки на исходный объект — не важно, какая из переменных будет использоваться для обращения к объекту для чтения или записи.

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Ниже перечислены некоторые обстоятельства, которые необходимо учитывать при выборе типа создаваемого объекта.

- Структуры не могут объявляться производными от других классов/структур.
- От структур нельзя производить классы.
- Структурные переменные всегда содержат действительные данные; им нельзя присвоить `Nothing`.
- Присваивание структур сводится к копированию данных на уровне полей, что может привести к большим потенциальным затратам времени.
- Сравнение структур сводится к сравнению значений их полей.
- Структуры не могут содержать деструкторов (завершителей).
- Неконстантные поля структур не могут инициализироваться конкретными значениями. Для полей-массивов не допускается задание начального размера.
- В рекомендациях Microsoft говорится, что затраты на копирование данных в структурных типах превышают затраты на сборку мусора, когда размер структурного типа превышает 16 байт.

Ссылочные объекты

Наверное, вы уже догадались, что к структурам я отношусь без особого энтузиазма. На первый взгляд это может показаться странным, поскольку в VB6 пользовательские типы обладали рядом существенных преимуществ перед объектными типами. В частности, они могли содержать массивы фиксированного размера и фиксированные строковые данные, что позволяло организовать эффективную пересылку данных в виде больших блоков.

В структурах массивы и строки остаются объектами, создаваемыми в куче и находящимися под управлением CLR[1]. Куча устроена таким образом, что операции создания и освобождения объектов в ней выполняются очень быстро. При создании ссылочной переменной память просто выделяется из свободной области в верхней части кучи[2]. Освобождение ссылочной переменной сводится к простому присваиванию `Nothing` или ссылки на другой объект — куча в этом вообще не участвует. Конечно, замечательная скорость операций выделения и освобождения памяти не обходится бесплатно. Как вы узнали из главы 4, системе приходится периодически выполнять сборку мусора, чтобы освободить память от неиспользуемых объектов и перевести ее в свободную часть кучи.

Структурные типы лучше всего подходят для ситуаций, когда вы работаете с большим количеством мелких объектов, основанных на других структурных типах и не содержащих полей, относящихся к ссылочным типам.

---

[1] Существуют средства, позволяющие более точно управлять расположением полей в структурах и определять в структурах строки и массивы фиксированной длины — в первую очередь, при вызовах функций API и передаче данных средствами COM. Об этом речь пойдет в части 3 этой книги.

[2] Я немного упрощаю; в действительности CLR для повышения эффективности поддерживает раздельные пространства свободной памяти для малых и больших объектов.

В остальных случаях обычно лучше обходиться ссылочными типами. Иначе говоря, если вы привыкли создавать и использовать пользовательские типы вместо классов, пора менять привычки.

Снова о сборке мусора

Среда CLR оптимизирована для ускоренного создания, освобождения и уничтожения мелких объектов. В частности, для этого отслеживается срок существования объектов. Каждый раз, когда объект «переживает» сборку мусора, он «стареет». Оказывается, в типичной программе действует правило: чем дольше существует объект, тем больше вероятность того, что он продолжит существование вплоть до завершения программы. Новые объекты часто относятся к категории временных или существуют в течение вызова одной функции. Когда возникает необходимость в сборке мусора, .NET сначала пытается ограничиться «нулевым поколением», то есть только новыми объектами. По информации Microsoft, во многих случаях мелкие временные объекты уничтожаются настолько быстро и эффективно, что они вообще не покидают процессорного кэша и не записываются в основную память.

Давайте разберемся, что же в действительности происходит при освобождении объекта в процессе сборки мусора.

На рис. 6.1 показана куча с пятью объектами. С двумя объектами (помеченными буквой F) ассоциируются завершители (то есть в программный код этих объектов входит метод `Finalizer`). CLR в сочетании с компилятором JIT опознает объекты, имеющие завершители, и отслеживает их особо.

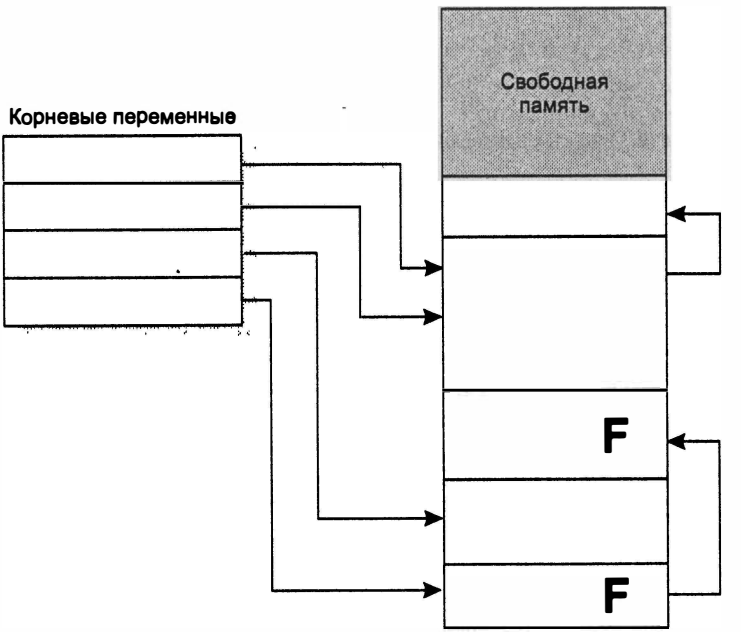


Рис. 6.1. Куча, содержащая два объекта с завершителями

Когда сборщик мусора видит, что на объект с завершителем больше не существует ни одной ссылки, он не удаляет этот объект немедленно. Вместо этого объект заносится в отдельный список объектов, которые должны пройти стадию завершения (рис. 6.2).

Объекты остаются в списке завершения, пока отдельный фоновый программный поток не запустит завершитель. После запуска завершителя объект удаляется из списка завершения и, наконец, освобождается в следующем цикле работы сборщика мусора[1].

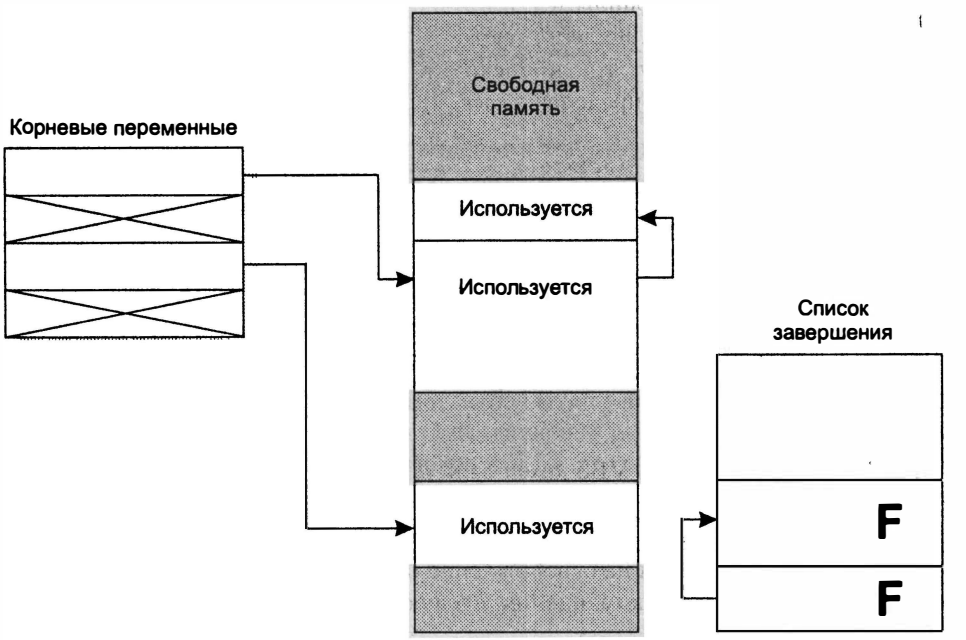


Рис. 6.2. Объекты с завершителями не уничтожаются при сборке мусора

Сколько времени проходит от исчезновения последней ссылки на объект до выполнения завершителя?

Неизвестно.

Более того, текущая документация .NET даже не гарантирует, что завершители всегда будут выполнены в конце работы приложения, хотя на практике это все же происходит.

Завершители

Не много найдется областей, которые бы лучше демонстрировали различия между VB .NET и VB6, чем проблемы завершения.

Программисты VB6, изучавшие теорию объектно-ориентированного программирования, знают «правильный» способ инициализации и деинициализации ком-

[1] Да, все верно — объекты с завершителями проходят сборку мусора дважды: при включении в список завершения и при фактическом освобождении памяти.

понентов. В обработчике события Initialize инициализируются переменные класса и даже выделяются необходимые системные ресурсы. В обработчике события Terminate выполняются необходимые завершающие действия. Событие Terminate происходит сразу же после освобождения последней ссылки на объект, поэтому в общем случае можно считать, что в обработчике Terminate все объектные переменные сохраняют действительные значения[1]. Вы можете твердо рассчитывать на то, что событие Terminate наступит и даст вам возможность «прибрать» за приложением в момент прекращения его работы (освободить системные ресурсы, закрыть файлы и т. д.).

В VB .NET эти правила не действуют[2].

Чтобы понять, почему это происходит, достаточно рассмотреть пример Undead1. В этом проекте определен класс Undying:

```
Public Class Undying
  Protected Overrides Sub Finalize()
    Debug.WriteLine("I have been finalized")
  End Sub
End Class
```

На форме находится кнопка cmdCreate, которая создает и удаляет объект:

```
Private Sub button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
  ' Внимание: в отличие от VB6 следующая команда создает объект.
  Dim obj As New Undying()
End Sub
```

В VB .NET в отличие от VB6 команда New немедленно создает объект. В VB6 команда New просто сообщает VB о том, что объект должен быть автоматически создан при первом обращении к его свойствам или методам.

Это изменение влияет и на освобождение объектных переменных. В VB6 при объявлении переменной в команде New следующего вида:

```
Dim myObject As New SomeObjectType
```

объект myObject создавался заново при каждой ссылке на него в программе. Если присвоить переменной myObject значение Nothing, а затем снова сослаться на него, будет использован новый экземпляр объекта SomeObjectType.

В VB .NET объект SomeObjectType создается непосредственно при выполнении команды Dim. Если затем присвоить myObject значение Nothing и попытаться сослаться на него, возникает ошибка времени выполнения.

Теперь попробуйте запустить приложение и нажмите кнопку. После некоторого количества нажатий будет выполнена сборка мусора с вызовом завершителей (правда, для этого может потребоваться тысяча нажатий, а то и больше). Закройте приложение. При окончательной сборке мусора будут выполнены все завершители.

---

[1] При выходе из приложений VB6 объекты освобождаются в произвольном порядке. Следовательно, во время обработки события класса Terminate ссылки на объекты могут стать недействительными, что иногда даже приводит к ошибкам защиты.

[2] И не обманывайте себя мыслью, будто вы избежите всех затруднений переходом на C#. В C# имеются деструкторы, похожие на деструкторы C++, однако работают они точно так же, как и завершители VB .NET.

Промежуток между освобождением и завершением кажется незначительным, но в действительности он играет довольно важную роль. Если ваши объекты захватывают ограниченные ресурсы (например, подключения к базе данных или сокеты), то временная недоступность ресурсов для этих освобожденных, но еще не уничтоженных объектов может отрицательно сказаться на быстродействии и масштабируемости приложения.

Недетерминированное завершение: трудный выбор

Visual Basic 6 (и объекты COM вообще) обеспечивает детерминированное завершение: вы всегда знаете, что завершители *будут* вызваны и когда именно это произойдет. Как видно из предыдущего обсуждения, в CLR детерминированное завершение *не поддерживается.* Нельзя точно предсказать, когда будет вызван завершитель, а в некоторых случаях вызов завершителя вообще не гарантирован.

Некоторые программисты VB осуждают эти изменения и ругают Microsoft за то, что в VB .NET не поддерживается детерминированное завершение.

Честно говоря, я совершенно не представляю, как Microsoft могла сохранить эту возможность. Вспомните: одним из главных аргументов в пользу сборки мусора был отказ от подсчета ссылок, часто приводящего к утечке памяти и возникновению циклических ссылок. Все, что мне приходит в голову, — либо выполнять сборку мусора после каждой модификации объектной переменной, либо вернуться к подсчету ссылок. Первый вариант совершенно неприемлем из-за катастрофического снижения быстродействия, а при возврате к подсчету ссылок снова возникнут проблемы с утечкой памяти и циклическими ссылками.

Лично я считаю, что преимущества сборки мусора компенсируют отказ от детерминированного завершения.

С другой стороны, эти изменения неизбежно отразятся на процессе программирования объектов в VB .NET.

Инициализация в целом почти не изменилась. Если уж и говорить о каких-то изменениях, то это изменения к лучшему, потому что теперь при конструировании объекта можно передавать параметры. Впрочем, отсутствие детерминированного завершения наложило свой отпечаток — выделение ресурсов при инициализации следует свести к минимуму и выделять ресурсы лишь перед непосредственным использованием.

Вместо того чтобы дожидаться вызова завершителя для освобождения ресурсов, в .NET рекомендуется использовать другой подход: когда надобность в объекте отпадает, работающий с ним клиент требует выполнить все необходимые действия по деинициализации и освобождению ресурсов. По общепринятым правилам для этой цели обычно используется метод `Dispose`. Деинициализация в методе `Dispose` позволяет отказаться от вызова завершителей (в результате из процесса сборки мусора исключается лишняя стадия, что улучшает быстродействие программы). Также часто встречается комбинированное решение: метод `Dispose` освобождает ресурсы по запросу клиента, а завершитель выполняет окончательную деинициализацию в том случае, если метод не вызывался.

Одна из возможных реализаций использована в приложении InitAndDestruct. В этом примере класс `Component1` объявляется производным от класса `System.ComponentModel.Component`. Класс `Component` может использовать дизайнер компонентов, обеспечивающий включение других компонентов. Класс `System.ComponentModel.Component` также наследует интерфейс `IDisposable`, в котором определяется метод `Dispose`. Реализация этого метода по умолчанию вызывает методы `Dispose` внутренних компонентов, что помогает автоматизировать процесс деинициализации.

В классе `Component1` продемонстрированы следующие возможности:

- конструктор класса (вызывается при первом создании объекта класса);
- конструктор объекта (вызывается при создании объекта);
- завершитель объекта (обычно вызывается в какой-то момент после освобождения объекта);
- метод `Dispose` (вызывается — по общепринятым правилам — клиентом, использующим объект).

Определение компонента приведено в листинге 6.2 (код конструктора частично опущен).

Листинг 6.2. Пример InitAndDestruct

```
' Инициализация и уничтожение
' Copyright © 2001 by Desaware Inc.
Public Class Component1
    Inherits System.ComponentModel.Component

    Shared Sub New()
      MsgBox ("Shared component initializer called")
    End Sub

    Public Sub New()
      MyBase.New()

      ' Следующий вызов необходим для дизайнера компонентов
      ' InitializeComponent()

      ' Дальнейшая инициализация выполняется после
      ' InitializeComponent()
      MsgBox ("My component instance created")

    End Sub

    Protected Overrides Sub Finalize()
      MsgBox ("My component finalizer called")
    End Sub

    Public Overloads Overrides Sub Dispose()
      MyBase.Dispose()
      MsgBox ("I've been disposed")
    End Sub

End Class
```

Для тестирования компонента используется обработчик события Click, который создает и удаляет объект:

```
Private Sub button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button1.Click
  Dim obj As New Component1()
  ' Завершив работу с объектом, вызовите метод Dispose (см. текст)
  obj.Dispose()
End Sub
```

При первом нажатии кнопки сначала вызывается конструктор класса, затем конструктор объекта, а затем метод Dispose (который должен явно вызываться клиентом). При следующих нажатиях кнопки вызывается только конструктор объекта и метод Dispose. Завершитель вызывается при освобождении объекта.

Клиент должен помнить о необходимости вызова Dispose. К счастью, во многих случаях .NET решает эту задачу за вас. Например, элемент, размещаемый на форме, включается в коллекцию Controls формы. Когда форма выполняет собственную деинициализацию (это происходит в тот момент, когда среда вызывает метод Dispose для формы), она, в свою очередь, вызывает методы Dispose всех элементов в коллекции Controls. Тем не менее среда ничего не знает о компонентах, создаваемых в программе (таких, как компонент Component1 в рассмотренном примере). Для таких объектов вы должны вызвать Dispose самостоятельно.

Заставлять контейнер явно вызывать методы Dispose его объектов — далеко не идеальное решение. Впрочем, с учетом архитектуры .NET вряд ли можно придумать что-нибудь лучшее. Мне кажется, что преимущества от деинициализации объектов средой CLR перевешивают недостатки — но, признаюсь, мне все же не хватает детерминированного завершения.

Ниже перечислены некоторые рекомендации, относящиеся к завершению объектов.

- По возможности избегайте решений, требующих деинициализации объектов. Не определяйте завершитель, если без него можно обойтись.
- Если ваш объект требует деинициализации, реализуйте метод Dispose.
- Рассмотрите возможность создания объекта-контейнера (производного от System.ComponentModel.Container) и включения в него всех используемых компонентов. Вызов Dispose для объекта-контейнера приводит к автоматическому вызову методов Dispose всех объектов, содержащихся в нем. Помните, что программа-мастер (wizard) часто автоматически генерирует соответствующий код для компонентов, находящихся под ее управлением.
- Даже если вы реализовали метод Dispose, подумайте, не стоит ли реализовать завершитель для проведения необходимой деинициализации на случай, если метод Dispose не вызывался в программе. Для некоторых типов компонентов (например, элементов управления) в вызове Dispose можно практически не сомневаться, поскольку элементы должны входить в коллекцию Controls и для них метод Dispose вызывается средой автоматически[1]. Но если вы со-

[1] Теоретически элемент управления можно создать в программе и не использовать его. Для таких элементов автоматический вызов метода Dispose не поддерживается, и вам придется вызывать Dispose вручную.

здаете компонент, который должен использоваться внешними клиентами, программисты могут забыть о вызове `Dispose`, сколько бы вы ни твердили об этом в документации.

- В Windows NT/2000/XP большинство системных ресурсов освобождается автоматически при выходе из приложения. В Windows 95/98/ME освобождение ресурсов не столь надежно.

Воскрешение объектов

Сценарий, о котором я собираюсь поведать, выглядит совершенно фантастически. Несомненно, он полностью противоречит опыту практически любого программиста VB.

Рассмотрим следующую последовательность событий.

- Когда на объект не остается ни одной ссылки из переменных корневого уровня, объект может быть уничтожен сборщиком мусора.
- Если у неиспользуемого объекта имеется завершитель, этот объект не уничтожается немедленно, а включается в список объектов, подлежащих завершению. Вплоть до выполнения завершителя объект остается несуществующим.
- Предположим, во время выполнения завершителя создается переменная корневого уровня, ссылающаяся на объект. Таким образом, на объект появляется ссылка, и он уже не подходит для сборки мусора.

Вот так сюрприз! Объект, который уже был освобожден и подготовлен к уничтожению, снова стал действительным! Иначе говоря, «мертвый» объект вернулся к жизни.

Рассмотрим класс `Resurrected` из проекта Resurrection.

```
Public Class Resurrected

  Public mycontainer As Form1

  Public Sub AreYouThere()
    MsgBox ("I am here")
  End Sub

  Protected Overrides Sub Finalize()
    MsgBox ("I'm being finalized")
    mycontainer.PriorObject = Me
  End Sub

End Class
```

Открытому свойству `mycontainer` присваивается ссылка на форму-контейнер. Также имеется метод с именем `AreYouThere`, который просто доказывает, что объект еще существует. У формы имеется открытое свойство с именем `PriorObject`, содержащее ссылку на объект `Resurrected`. Циклическая ссылка (форма ссылается на объект, а объект ссылается на форму) в VB .NET не вызывает проблем. Как только форма перестает ссылаться на объект, он будет передан сборщику мусора.

Помимо свойства PriorObject в форме определены четыре события для четырех кнопок: Create & Delete, ForceGC, Check Prior Object, Turn Finalizer Back On.

- Кнопка Create & Delete инициирует событие cmdCreate_Click. Обработчик этого события создает объект и присваивает ссылку на форму свойству mycontainer. Это позволяет объекту обращаться к свойству PriorObject формы.
- Кнопка ForceGC инициирует событие cmdGC_Click. Обработчик этого события форсирует немедленную сборку мусора и ожидает выполнения всех завершителей.
- Кнопка Check Prior Object инициирует событие cmdPrior_Click. Обработчик этого события проверяет действительность свойства PriorObject формы. Если свойство действительно, вызывается метод AreYouThere.
- Кнопка Turn Finalizer Back On инициирует событие cmdPrior_Click. Обработчик этого события активизирует завершитель объекта (см. ниже).

Содержимое модуля формы приведено в листинге 6.3.

Листинг 6.3. Модуль формы приложения Resurrection

```
Public PriorObject As Resurrected

Private Sub cmdRefinalize_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles cmdRefinalize.Click
  If Not PriorObject Is Nothing Then
    GC.ReRegisterForFinalize (PriorObject)
  End If
  PriorObject = Nothing
End Sub

Private Sub cmdPrior_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles cmdPrior.Click
  If Not PriorObject Is Nothing Then
    ' Доказательство того, что завершенный объект
    ' все еще существует!
    PriorObject.AreYouThere()
  End If
End Sub

Private Sub cmdCreate_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles cmdCreate.Click
  Dim obj As New Resurrected()

  ' Объект получает ссылку на форму.
  obj.mycontainer = Me
End Sub

Private Sub cmdGC_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles cmdGC.Click
  ' Выполнить принудительную сборку мусора.
  GC.Collect()
  GC.WaitForPendingFinalizers()
End Sub

End Class
```

За «воскрешением» объектов можно понаблюдать на примере проекта Resurrection (см. листинг 6.3).

Выполните следующие действия.

1. Нажмите кнопку Create & Destroy — приложение создает объект, присваивает его свойству `mycontainer` ссылку на форму и освобождает объект (при выходе из функции форма уже не содержит переменную со ссылкой на объект).
2. Нажмите кнопку Force GC, форсирующую немедленную сборку мусора и завершение объектов. В процессе завершения объект использует свое свойство `mycontainer` и создает в свойстве `PriorObject` формы ссылку на себя. Другими словами, на объект снова существует ссылка в переменной `Form` (корневого уровня).
3. Нажмите кнопку Check Prior Object — как видите, объект, для которого был вызван завершитель, продолжает существовать!
4. Закройте приложение. Хотя завершители должны вызываться при выходе из приложения, вы увидите, что завершитель нашего объекта не вызывается.

Как такое возможно? Выполненный завершитель не выполняется снова, даже если объект «воскрес»!

Снова запустите программу Resurrection, повторите действия 1–3 и нажмите кнопку Turn Finalizer Back On; обработчик этой кнопки тоже присваивает переменной `PriorObject` значение `Nothing`. Снова нажмите кнопку Force GC — вы увидите, что завершитель выполняется повторно. Метод `GC.ReRegisterForFinalize` сообщает сборщику мусора о том, что объект вернулся к жизни и его завершитель должен быть снова выполнен при повторной сборке мусора[1].

На первый взгляд кажется, будто я трачу много времени на описание проблемы, которая никогда не возникнет ни у одного здравомыслящего программиста. В каком-то смысле это верно: подобные ситуации гораздо чаще возникают в результате ошибок, нежели в результате сознательных решений проектировщика.

Однако понимание этого процесса очень важно для овладения одной важной методикой, которой можно и нужно пользоваться.

Наряду с методом `ReRegisterForFinalize` существует метод `SuppressFinalize`, при помощи которого вы сообщаете сборщику мусора о том, что объект не должен проходить завершение, даже если у него имеется завершитель. На этом приеме основана одна важная оптимизация. Выше я советовал деинициализировать объекты в методе `Dispose` и использовать метод-завершитель на тот случай, если клиент не вызовет `Dispose`. Вызывая `SuppressFinalize` в методе `Dispose`, вы предотвратите ненужную деинициализацию.

Итоги

Управление памятью в .NET не только имеет далеко идущие последствия для языка VB .NET, но и значительно влияет на общий подход к проектированию приложений.

[1] При закрытии приложения окно сообщения не выводится: умный VB .NET понимает, что приложение завершает работу, и не выводит окна сообщений в это время.

В начале этой главы вы узнали о различиях между структурными и ссылочными типами и о том, что преимущества пользовательских типов VB6 не распространяются на структурный тип VB .NET. Другими словами, в VB .NET классам следует отдавать предпочтение перед структурами практически во всех случаях, кроме небольших структур, состоящих только из других структурных типов.

Кроме того, вы узнали, что за дополнительные преимущества в надежности и масштабируемости (устранение утечек памяти и циклических ссылок) приходится расплачиваться отсутствием в CLR детерминированного завершения.

Из этого следует, что программисты VB .NET должны избегать освобождения ресурсов в методе `Finalize`. Вместо этого следует унаследовать интерфейс `IDisposable`[1] и реализовать метод `Dispose`. Метод-завершитель должен использоваться как последнее средство для выполнения деинициализации в ситуациях, когда метод `Dispose` не вызывался. При вызове `Dispose` для повышения быстродействия можно запретить вызов завершителя методом `GC.SupressFinalize`.

Наконец, вы узнали, что «мертвые» объекты могут вернуться к жизни, если из-за ошибок объектной модели на стадии завершения на объект будет создана ссылка.

Темы глав 5 и 6 — наследование и управление памятью — представляют два из трех крупных концептуальных изменений, с которыми сталкиваются программисты при переходе с VB6 на VB .NET. Именно эти новшества вызовут немало проблем у программистов VB6, не желающих расстаться со старыми навыками программирования или слепо хватающихся за новые разрекламированные возможности языка. В главе 7 рассматривается третье из этих изменений — многопоточность.

[1] Или объявить класс производным от класса, наследующего интерфейс IDisposable.

Многопоточность в VB .NET

В классическом научно-фантастическом романе Альфреда Бестера «Звезды — цель моя» описывается психокинетическое взрывчатое вещество ПирЕ, всего одна крупинка которого может взорвать целый дом. Для взрыва требуется совсем немного — чьего-нибудь мысленного пожелания. Герою романа приходится решать, сохранить ли тайну или же открыть ее людям, чтобы судьба мира находилась в руках и мыслях каждого человека на планете.

Нечто похожее можно сказать и о многопоточности.

С одной стороны, это полезная технология, способная улучшить быстродействие (реальное или кажущееся) ваших приложений. Но с другой стороны, при неправильном использовании этот «атомный реактор от программирования» рванет и уничтожит вашу программу. Нет, еще хуже — он уничтожит вашу репутацию и ваш бизнес, поскольку многопоточные ошибки нередко поднимают затраты на тестирование и отладку до астрономических высот.

Многопоточность в VB .NET страшит меня больше, чем все остальные новшества, причем, как и во многих новых технологиях .NET, это объясняется скорее человеческими, нежели технологическими факторами.

За несколько месяцев до презентации .NET я принимал участие в конференции VBits. Я спросил свою аудиторию, состоявшую из довольно опытных программистов Visual Basic, хотят ли они видеть свободную многопоточность в следующей версии Visual Basic. Практически все подняли руки. Затем я спросил, кто из присутствующих знает, на что идет. На этот раз руки подняли всего несколько человек, и на их лицах были понимающие улыбки.

Я боюсь многопоточности в VB .NET, потому что программисты Visual Basic обычно не обладают опытом проектирования и отладки многопоточных приложений[1]. В реализации многопоточности для VB6 действуют повышенные меры защиты (наряду с довольно жесткими ограничениями). Существует только один

[1] Чтобы вы не думали, что я безнадежно зазнался, поясню: я программирую многопоточные приложения в течение многих лет и до сих пор сталкиваюсь с нетривиальными ошибками. Многопоточность используется во внутренней работе пакета Desaware NT Service Toolkit, и половина всего времени была потрачена на проектирование, тестирование и отладку управления программными потоками.

путь к безопасному использованию многопоточности — вы должны хорошо разобраться в ней и правильно проектировать свои приложения.

Еще раз подчеркиваю — *правильно проектировать свои приложения*. При неправильном подходе к проектированию исправить недостатки позднее практически невозможно, а потенциальные затраты на решение проблем многопоточности могут оказаться сколь угодно большими.

Я всегда считал, что автор должен не только описывать новую технологию, но и представлять ее в контексте практического применения и помогать читателям в правильном выборе технологии для их конкретных задач. Поскольку многопоточность является исключительно серьезной темой, в этой главе я пойду несколько необычным путем. Вместо того чтобы рассказывать о преимуществах многопоточности и приводить доводы в пользу ее применения, в начале этой главы я постараюсь дать общее представление об этой технологии и проблемах, с которыми вам предстоит столкнуться[1]. Только ближе к концу главы, когда вы поймете, как работают многопоточные приложения, мы рассмотрим некоторые ситуации, в которых ее стоит использовать[2].

Первое знакомство с многопоточностью

Вероятно, вы как программист Visual Basic среднего или высокого уровня хотя бы в общих чертах представляете себе, что такое многопоточность. Упрощенно говоря, Windows позволяет одновременно выполнять несколько фрагментов программного кода, быстро переключаясь между ними. Но что это означает на практике?

Процессор содержит регистры, используемые при выполнении программ. В регистре указателя инструкций хранится адрес выполняемой инструкции; в регистре указателя стека хранится адрес программного стека, в котором находятся локальные переменные и адреса возврата функций. Другие регистры используются для хранения временных данных. Когда ОС принимает решение о переключении на другой фрагмент программного кода, она прерывает нормальную последовательность выполнения инструкций, сохраняет содержимое регистров, загружает текущие значения регистров для другого фрагмента и приступает к его выполнению[3].

Итак, любой код, работающий в системе, выполняется в программном потоке (thread).

Тогда что такое «процесс»?

Процесс состоит из одного или нескольких программных потоков, работающих в отдельном пространстве памяти. При запуске приложения Windows с точки зрения процесса все выглядит так, словно он один распоряжается всей систем-

---

[1] Проще говоря, я постараюсь вас до смерти напугать.

[2] А если вы собираетесь пролистать эту главу, потому что не намерены использовать многопоточность, позвольте напомнить, что по умолчанию все завершители объектов работают в отдельном потоке завершения! Следовательно, проблемы многопоточности могут возникнуть и в том случае, если вы не создаете потоков в своих приложениях.

[3] Строго говоря, в эпоху конвейерной обработки и многопроцессорных систем описанная схема кажется чрезмерно упрощенной, но для наших целей она вполне адекватна.

ной памятью. На самом деле процесс не может записывать данные в адресное пространство других процессов, а другие процессы не могут записывать в его адресное пространство. Разделение адресных пространств обеспечивает повышенную степень защиты и является главной причиной того, что 32-разрядные версии Windows не «зависают» так часто, как прежние 16-разрядные версии[1].

Из предыдущего абзаца непосредственно следует важнейший фактор, который должен учитываться при многопоточном программировании:

Все потоки многопоточного приложения работают в одном и том же пространстве памяти.

Ну и что?

Рассмотрим следующий сценарий.

Фиаско в магазине

Мистер и миссис Купилл — типичная счастливая супружеская пара, живущая в пригороде. Однажды утром мистер Купилл решает приобрести своей супруге новейшую модель электроутюга с подключением к Интернету. Будучи человеком осторожным и несколько стесненным в средствах (после того, как сбережения всей жизни были потрачены на покупку домика в Пало Альто), он проверяет свою кредитку, убеждается в том, что у него хватит денег, после чего отправляется в магазин за покупкой.

В это время миссис Купилл обнаруживает, что в web-магазине проводится распродажа 125-гигабайтных жестких дисков, о которых так мечтал мистер Купилл (чтобы наконец-то установить Office 2005). Проверив состояние кредитной карты, она видит, что у нее как раз хватает денег на покупку, и оформляет заказ.

Тем временем мистер Купилл приезжает в магазин и после недолгого 45-минутного ожидания добирается до кассы. Представьте его потрясение, когда кассир вдруг сообщает, что вся сумма на счету израсходована и кредитка подлежит уничтожению.

Мистер Купилл с позором покидает магазин и пытается понять, что же произошло. От огорчения он не смотрит по сторонам, и его сметает толпа из пяти тысяч обезумевших подростков, спешащих в магазин за только что вышедшей приставкой Playstation 4 X-Box (последняя разработка MS-Sony).

Какое отношение это имеет к многопоточности, спросите вы?

Самое прямое.

Мистер и миссис Купилл действовали независимо друг от друга и в любой момент времени располагали полным доступом к кредитной карте. Мистер Купилл прочитал данные с карты, но в промежутке времени между чтением информации и ее использованием эти данные были изменены без его ведома — с самыми катастрофическими последствиями.

А теперь заменим супружескую чету программными потоками, обладающими доступом к общей памяти.

Что произойдет, если эти два потока обратятся к одному объекту COM?

---

[1] Ситуация также описана несколько упрощенно. В Windows NT, 2000 и XP пространства памяти изолируются более надежно, чем в Windows 95/98/ME. Помимо разделения пространств памяти, процессы обладают и другими возможностями.

Напомню, что при работе с объектами COM количество ссылок на объект определяется при помощи счетчика ссылок. Если два потока попытаются одновременно изменить счетчик ссылок, возникнет точно такая же проблема, как описано выше. Может, при освобождении объекта счетчик ссылок не уменьшится, и в памяти появятся не уничтоженные объекты. А может, неправильное увеличение счетчика приведет к тому, что объект будет уничтожен до освобождения последней ссылки на него.

Теоретически ошибки такого рода могут возникать всюду, где используются общие ресурсы или переменные.

Почему же эта проблема не существовала в Visual Basic 6? Потому что VB6 создает отдельную копию всех глобальных переменных для каждого потока в многопоточном приложении или DLL[1]. Разделение осуществлялось на уровне Visual Basic 6, а не на уровне ОС. В VB .NET глобальные и общие переменные совместно используются всеми потоками приложения. Таким образом, все проблемы многопоточности лежат исключительно на вашей совести.

Подробнее о многопоточности

Поскольку я сейчас не обучаю вас премудростям многопоточности, а лишь пытаюсь привить здоровые опасения, самое время рассмотреть практический пример. А чтобы было интереснее, я сразу скажу, что программа содержит недостатки, на которые я укажу позднее. Посмотрим, удастся ли вам самостоятельно обнаружить потенциальные ошибки.

Приведенная ниже программа имитирует взаимодействие «клиент-сервер». В качестве модели выбрана обычная семья. В роли сервера — глава семьи, получающий зарплату, а клиентами являются его дети, постоянно требующие денег на карманные расходы. Невзирая на упрощенность, этот сценарий применим во многих реальных ситуациях.

Перед вами одно из самых длинных приложений, описанных в книге, но я рекомендую не жалеть времени и тщательно разобраться в нем. Происходящее не так уж сложно, а чтобы понять недостатки проектирования, необходимо хорошо знать код.

Структуры данных

Взявшись за новое приложение, я обычно прежде всего думаю о том, какие объекты будут в нем использоваться. Я уверен, что вы как опытный объектно-ориентированный программист делаете то же самое.

В этом приложении у отца и детей имеются банковские счета. Базовые механизмы зачисления и снятия денег со счета и проверки баланса для этих двух случаев очень похожи, если не одинаковы. Поскольку с логической точки зрения и счет ребенка, и счет отца являются банковскими счетами, между ними и абстрактным банковским счетом существует несомненная связь:

- счет ребенка *является* банковским счетом;
- счет отца *является* банковским счетом.

[1] Точнее, глобальные переменные VB6 хранятся в локальной памяти потока.

Иначе говоря, иерархия объектов данного примера наводит на мысли о применении наследования.

Ниже приведен класс банковского счета `Account` приложения Threading1. В переменной `m_Account` хранится текущая сумма на счету. Кроме того, в переменных объекта хранятся суммарный расход и суммарные поступления за время существования счета. Также в классе `Account` хранится объект типа `Random` — объект CLR, предназначенный для работы со случайными числами и заменяющий функцию VB6 `Rnd`. Функция `GetRandomAmount` создает значение в интервале от 0 до 1 доллара и имитирует расходуемые суммы.

```
Public Class Account
  Protected m_Account As Double
  Protected m_Spent As Double
  Protected m_Deposited As Double
  Private Shared m_Random As New Random()

  ' Возвращает случайную сумму от $0 до $1.00.
  Protected Shared Function GetRandomAmount() As Double
    Dim amount As Double
    amount = Int(m_Random.NextDouble * 100)
    GetRandomAmount = amount / 100
  End Function
```

В листинге 7.1 приведена серия свойств, обеспечивающих доступ к переменным класса.

Листинг 7.1. Свойства класса Account[1]

```
Property Withdrawn() As Double
  Get
    Withdrawn = m_Spent
  End Get
  Set(ByVal Value As Double)
    m_Spent = Value
  End Set
End Property

Property Balance() As Double
  Get
    Balance = m_Account
  End Get
  Set(ByVal Value As Double)
    m_Account = Value
  End Set
End Property

Property Deposited() As Double
  Get
    Deposited = m_Deposited
  End Get
  Set(ByVal Value As Double)
    m_Deposited = Value
  End Set
End Property
```

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Переменные класса можно было бы объявить открытыми или защищенными, но обычно рекомендуется обеспечить доступ к ним через свойства. Это обеспечивает гибкость, необходимую для расширения функциональности приложения в будущем.

Метод `Withdraw` (листинг 7.2) получает в качестве параметра сумму, снимаемую со счета, и возвращает фактически снятую сумму (которая при нехватке денег на счету может быть меньше запрашиваемой). Кроме того, снятая сумма прибавляется к суммарному расходу (`m_Spent`). Метод `Deposit` заносит указанную сумму на текущий счет и прибавляет ее к суммарным поступлениям (`m_Deposited`). Метод `Clear` сбрасывает переменные класса в исходное состояние.

Листинг 7.2. Методы Withdraw и Depost класса Account

```
' Попытаться снять со счета запрашиваемую сумму,
' вернуть фактически снятую сумму.
Protected Function Withdraw(ByVal amount As Double) As Double
  If amount > m_Account Then
    amount = m_Account
  End If
  m_Account = m_Account - amount
  m_Spent = m_Spent + amount
  Return amount
End Function

Protected Sub Deposit(ByVal amount As Double)
  m_Account = m_Account + amount
  m_Deposited = m_Deposited + amount
End Sub

Public Overridable Sub Clear()
  m_Account = 0
  m_Deposited = 0
  m_Spent = 0
End Sub

End Class
```

Счет ребенка представлен классом `KidAccount`, производным от класса `Account`. В этом классе добавлена переменная `m_FailedRequests`, в которой хранится количество случаев, когда ребенок хотел потратить деньги, но не располагал нужной суммой на счету. Как нетрудно предположить, эта переменная будет быстро увеличиваться.

Для зачисления денег на счет используется метод `GetAllowance`. Можно ли было воспользоваться методом `Deposit`? Напрямую — нельзя. Поскольку метод `Deposit` базового класса является защищенным, он не может напрямую вызываться извне. Мы также могли создать новый метод `Deposit` и воспользоваться ключевым словом `Shadows`, чтобы скрыть унаследованный метод базового класса (а также вызвать метод базового класса через объект `MyBase`, но об этом речь пойдет в главе 10).

```
Public Class KidAccount
  Inherits Account
  Private m_FailedRequests As Double
  ReadOnly Property FailedRequests() As Double
    Get
      FailedRequests = m_FailedRequests
```

```
    End Get
  End Property

  ' Получение карманных денег от родителя
  Public Sub GetAllowance(ByVal amount As Double)
    deposit (amount)
  End Sub
```

Деньги расходуются методом Spend. Метод выбирает случайную сумму до $1 и пытается снять ее со счета. Если при снятии баланс падает до нуля, попытка считается неудачной, а переменная m_FailedRequests увеличивается. Метод Clear сбрасывает как переменную m_FailedRequests, так и методы базового класса (листинг 7.3).

Листинг 7.3. Методы Spend и Clear класса KidAccount

```
  ' Попытка потратить случайную сумму
  Public Sub Spend()
    Dim amount As Double
    amount = GetRandomAmount()
    If amount > m_Account Then amount = m_Account
    If amount = 0 Then
      m_FailedRequests = m_FailedRequests + 1
    Else
      Withdraw (amount)
    End If
  End Sub

  ' Обнуление переменных объекта и базового класса
  Overrides Sub Clear()
    m_FailedRequests = 0
    MyBase.Clear()
  End Sub
End Class
```

Класс ParentAccount (листинг 7.4), также объявленный производным от Account, моделирует родительский счет. Метод GiveAllowance выбирает случайную сумму, которая снимается со счета отца. Возвращаемое значение определяет сумму, фактически зачисляемую на счет ребенка методом GetAllowance. Метод DepositPayroll зачисляет на родительский счет ежемесячную зарплату.

Листинг 7.4. Класс ParentAccount

```
Public Class ParentAccount
  Inherits Account

  ' Метод выбирает случайную сумму карманных расходов
  ' и снимает ее со счета.
  Public Function GiveAllowance() As Double
    Dim amount As Double
    amount = GetRandomAmount()
    amount = Withdraw(amount)
    ' Возвращение фактически снятой суммы (может быть равна 0)
    Return (amount)
  End Function

  Public Sub DepositPayroll(ByVal amount As Double)
    deposit (amount)
  End Sub
End Class
```

Имитация

Класс `FamilyOperation` имитирует финансовые отношения в семье. В него включена переменная `Kids()`, содержащая объекты `KidAccount`, по одному для каждого ребенка в семье. Переменная `Parent` содержит объект `ParentAccount`, представляющий родительский счет.

Имитация рассчитана на работу в одном или нескольких фоновых потоках. Это соответствует применению пула потоков для обработки клиентских запросов или (с небольшими изменениями) наличию отдельного независимого потока для каждого клиента.

Для управления потоками используется переменная `Threads`, содержащая ссылку на массив объектов `System.Threading.Thread` — эти объекты управления потоками поддерживаются CLR.

Переменная `m_NumberOfKids` содержит количество детских счетов. Присваивая переменной `m_Stopping` значение `True`, вы сигнализируете фоновым потокам о необходимости завершения. Переменная `m_Random` относится к типу `Random()` и используется имитатором для определения суммы, переводимой на счета детей.

```
Public Class FamilyOperation
  Private Kids() As KidAccount
  Private Parent As ParentAccount

  Private Threads() As System.Threading.Thread

  Private m_NumberOfKids As Integer
  Private m_Stopping As Boolean

  Private m_Random As New Random()
```

Объект `FamilyOperation` спроектирован таким образом, что значение свойства `NumberOfKids` задается перед началом имитации, причем изменить его уже не удастся. Метод `Get` тривиален: он просто возвращает значение внутренней переменной `m_NumberOfKids`. Метод `Set` сначала проверяет присваиваемую величину: допустимыми считаются значения в интервале от 1 до 50. Кроме того, метод проверяет, было ли значение `m_NumberOfKids` присвоено ранее; если проверка дает положительный результат, инициируется ошибка (листинг 7.5).

Метод `Throw` является новым и рекомендуемым способом инициирования ошибок в VB .NET. В CLR определено довольно большое количество документированных исключений. В нашем примере используются исключения `ArgumentOutOfRangeException` (недопустимое значение параметра или свойства) и `InvalidOperationException` (попытка выполнения недопустимой операции). Исключения и новый механизм обработки ошибок рассматриваются в главе 8.

Если проверка проходит успешно, метод переходит к инициализации родительского счета и счетов детей.

Листинг 7.5. Свойство NumberOfKids класса FamilyOperation

```
Property NumberOfKids() As Integer
  Get
    NumberOfKids = m_NumberOfKids
  End Get
  Set(ByVal Value As Integer)
```

```
    If Value < 1 Or Value > 50 Then
      Throw New ArgumentOutOfRangeException(_
        "Property must be between 1 and 50")
    End If
    If m_NumberOfKids <> 0 Then
      Throw New InvalidOperationException(_
        "NumberOfKids may only be set once")
    End If
    Dim Kid As Integer
    m_NumberOfKids = Value
    ReDim Kids(m_NumberOfKids - 1)
    For Kid = 0 To m_NumberOfKids - 1
    Kids(Kid) = New KidAccount()
    Next
    Parent = New ParentAccount()
  End Set
End Property
```

Метод `KillFamily`[1] просто присваивает переменной `m_Stopping` значение `True`, сигнализируя потокам о завершении работы. О том, как это происходит, будет рассказано ниже. Средства зачисляются на родительский счет методом `ParentPayday`.

```
Private Sub KillFamily()
  m_Stopping = True
End Sub

Public Sub ParentPayday(ByVal Amount As Double)
  Parent.DepositPayroll (Amount)
End Sub
```

Свойства `TotalDepositToParent`, `TotalAllocatedByParent` и `ParentBalance` предназначены для получения статистики о родительском счете.

Всегда должно соблюдаться следующее условие:

```
TotalDepositToParent - TotalAllocatedByParent = ParentBalance
```

Код этих свойств приведен в листинге 7.6.

Листинг 7.6. Свойства TotalDepositedToParent, TotalAllocatedByParent и ParentBalance

```
Public ReadOnly Property TotalDepositedToParent() As Double
  Get
    If m_NumberOfKids = 0 Then Return 0
    Return Parent.Deposited
  End Get
End Property

Public ReadOnly Property TotalAllocatedByParent() As Double
  Get
    If m_NumberOfKids = 0 Then Return 0
    Return Parent.Withdrawn
  End Get
End Property

Public ReadOnly Property ParentBalance() As Double
  Get
```

продолжение ☞

[1] При написании этого метода ни одна семья не пострадала.

Листинг 7.6 (продолжение)

```
    If m_NumberOfKids = 0 Then Return 0
    Return Parent.Balance
  End Get
End Property
```

Свойства TotalGivenToKids, TotalSpentByKids, TotalKidsBalances и TotalFailedRequests используются для получения сводной статистики по всем детским счетам. Следующее условие всегда должно выполняться:

```
TotalGivenToKids - TotalSpentByKids = TotalKidsBalance
```

Программный код этих свойств приведен в листинге 7.7.

Листинг 7.7. Свойства для получения сводной информации о счетах детей

```
Public ReadOnly Property TotalGivenToKids() As Double
  Get
    If m_NumberOfKids = 0 Then Return 0
    Dim Idx As Integer
    Dim Total As Double
    For Idx = 0 To m_NumberOfKids - 1
      Total = Total + Kids(Idx).Deposited
    Next
    Return Total
  End Get
End Property

Public ReadOnly Property TotalSpentByKids() As Double
  Get
    If m_NumberOfKids = 0 Then Return 0
    Dim Idx As Integer
    Dim Total As Double
    For Idx = 0 To m_NumberOfKids - 1
      Total = Total + Kids(Idx).Withdrawn
    Next
    Return Total
  End Get
End Property

Public ReadOnly Property TotalKidsBalances() As Double
  Get
    If m_NumberOfKids = 0 Then Return 0
    Dim Idx As Integer
    Dim Total As Double
    For Idx = 0 To m_NumberOfKids - 1
      Total = Total + Kids(Idx).Balance
    Next
    Return Total
  End Get
End Property

Public ReadOnly Property TotalFailedRequests() As Double
  Get
    If m_NumberOfKids = 0 Then Return 0
    Dim Idx As Integer
    Dim Total As Double
    For Idx = 0 To m_NumberOfKids - 1
      Total = Total + Kids(Idx).FailedRequests
```

```
    Next
    Return Total
  End Get
End Property
```

Вся настоящая работа выполняется методом `KidsSpending`. Он состоит из цикла, выполняемого до тех пор, пока переменной `m_Stopping` не будет присвоено значение `True` (листинг 7.8). В цикле последовательно выполняются следующие действия:

- выбор случайного ребенка;
- запрос «карманных денег» с родительского счета — случайной суммы не более $1;
- зачисление возвращаемой суммы на счет ребенка;
- ребенку предоставляется возможность истратить случайную сумму денег.

Листинг 7.8. Функция KidsSpending

```
Public Sub KidsSpending()
  Dim ChildIndex As Integer
  Dim Allowance As Double
  Dim thiskid As KidAccount
  Do
    ' Случайно выбранный ребенок тратит некоторую сумму.
    ChildIndex = CInt(Int(m_Random.NextDouble() * CDbl(m_NumberOfKids)))
    thiskid = Kids(ChildIndex)

    Allowance = Parent.GiveAllowance()
    thiskid.GetAllowance (Allowance)
    thiskid.Spend()
  Loop Until m_Stopping

End Sub
```

Конечно, вызов этого метода в основной программе привел бы к зацикливанию. Выход из функции происходит лишь в тот момент, когда переменная `m_Stopping` становится равной `True`, а в самой функции значение этой переменной не изменяется. Впрочем, метод `KidsSpending` предназначен для вызова из независимого потока. При возврате из метода поток прекращает свою работу.

Метод `StartThread` создает заданное количество новых потоков и запускает их. Он динамически переобъявляет размер массива `Threads` и создает новые объекты `Thread`, каждому из которых передается делегат для метода `KidsSpending`.

Делегат?

Вероятно, вы знакомы с оператором VB6 `AddressOf`, возвращающим адрес (указатель) функции в модуле. В VB .NET оператор `AddressOf` возвращает делегата, которого лучше всего представлять себе как указатель на метод конкретного объекта. В данном примере возвращается указатель на метод `KidsSpending` текущего объекта[1].

Каждый запущенный поток вызывает метод `KidsSpending`.

---

[1] Напрашивается вопрос: можно ли получить делегата для метода другого объекта и вызвать его? Ответ: да, можно. Более того, как будет показано ниже, именно так работают события VB.NET.

В настоящем примере каждому потоку назначается более низкий приоритет по сравнению с главным потоком пользовательского интерфейса. Если бы мы оставили их приоритеты равными приоритету главного потока и запустили несколько интенсивно работающих потоков (а метод `KidsSpending` постоянно использует процессор), это привело бы к заметному ухудшению быстродействия главного потока пользовательского интерфейса.

Задавая свойству `IsBackground` значение `True`, мы сообщаем CLR, что создаваемый поток является фоновым и должен уничтожаться автоматически при прекращении работы всех не фоновых потоков[1].

```
Public Sub StartThreads(ByVal ThreadCount As Integer)
  If ThreadCount < 1 Then ThreadCount = 1
  ReDim Threads(ThreadCount - 1)
  Dim Idx As Integer
  For Idx = 0 To ThreadCount - 1
    Threads(Idx) = New Threading.Thread(AddressOf Me.KidsSpending)
    Threads(Idx).Priority = System.Threading.ThreadPriority.BelowNormal
    Threads(Idx).IsBackground = True
    Threads(Idx).Start()
  Next
End Sub
```

В методе `StopThreads` продемонстрирован новый механизм обработки ошибок в VB .NET. Базовый принцип уничтожения потоков прост. Сначала вызывается метод `KillFamily`, который присваивает переменной `m_Stopping` значение `True`. После этого все потоки должны завершиться при достижении конца цикла `Do`. Метод `Join` объекта `Thread` заставляет текущий поток дождаться фактического завершения заданного потока.

В приведенном примере метод `Join` всегда работает нормально, поскольку все потоки начинают работу при создании. Но если метод будет вызван перед инициализацией массива `Threads`, метод `GetUpperBounds` инициирует исключение. В результате управление передается блоку, следующему сразу же за командой `Catch`. В листинге 7.9 ошибки просто игнорируются.

Листинг 7.9. Метод StopThreads

```
Public Sub StopThreads()
  Dim Idx As Integer
  Try
    KillFamily() ' Все потоки должны остановиться
    For Idx = 0 To Threads.GetUpperBound(0)
      ' Дождаться завершения потока
      Threads(Idx).Join()
    Next
  Catch
    ' Игнорировать все ошибки
  End Try

End Sub
```

Описанная архитектура не отличается особой устойчивостью к ошибкам. Предполагается, что после остановки потоков будет создан новый объект

[1] Конечно, полагаться на это нельзя. Вы всегда должны самостоятельно завершать свои потоки, как показано в этом примере. Свойство IsBackground просто обеспечивает дополнительную страховку.

FamilyOperation, вместо того чтобы перезапускать уже существующий объект. Обработка ошибок также реализована на минимальном уровне, однако для демонстрации фоновых операций в многопоточных приложениях этого вполне достаточно.

На форме находятся два текстовых поля, в которых вводится количество банковских счетов у детей и количество потоков. Также на ней имеются три кнопки: для занесения денег на счет, для запуска и для остановки имитации. Результаты выводятся в виде списка. С формой ассоциируется единственная переменная myFamily, содержащая ссылку на объект FamilyOperation. Перед выгрузкой формы вызывается метод StopThreads, поэтому фоновые потоки останавливаются даже в том случае, если пользователь не нажал кнопку Stop.

```
Public Class Form1
  Dim myFamily As FamilyOperation

  ' Форма переопределяет Dispose для очистки списка компонентов.
  Public Overloads Overrides Sub Dispose()
    MyBase.Dispose()
    If Not (components Is Nothing) Then
      components.Dispose()
    End If
    ' Остановить потоки
    If Not myFamily Is Nothing Then
      myFamily.StopThreads()
    End If

  End Sub
```

Метод UpdateResults (листинг 7.10) заполняет список сводной статистической информацией. По этим данным можно убедиться в том, что разность между зачисленной и снятой со счета суммой равна текущему балансу (как для отца, так и для детей).

Листинг 7.10. Формы метода UpdateResults

```
Private Sub UpdateResults()
  lstResults.Items.Clear()
  lstResults.Items.Add ("Parent:")
  lstResults.Items.Add ("- Total Deposited: " + _
    Format(myFamily.TotalDepositedToParent, "0.00"))
  lstResults.Items.Add ("- Total Withdrawn: " + _
    Format(myFamily.TotalAllocatedByParent, "0.00"))
  lstResults.Items.Add ("- Expected Balance: " + _
    Format(myFamily.TotalDepositedToParent - _
    myFamily.TotalAllocatedByParent, "0.00"))
  lstResults.Items.Add ("- Actual Balance: " + _
    Format(myFamily.ParentBalance, "0.00"))
  lstResults.Items.Add ("Kids:")
  lstResults.Items.Add ("- Total Deposited: " + _
    Format(myFamily.TotalGivenToKids, "0.00"))
  lstResults.Items.Add ("- Total Withdrawn: " + _
    Format(myFamily.TotalSpentByKids, "0.00"))
  lstResults.Items.Add ("- Expected Balance: " + _
    Format(myFamily.TotalGivenToKids - _
      myFamily.TotalSpentByKids, "0.00"))
  lstResults.Items.Add ("- Actual Balance: " + _
    Format(myFamily.TotalKidsBalances, "0.00"))
End Sub
```

Содержимое списка обновляется раз в одну-две секунды по событиям таймера. Кнопка Deposit заносит заданную сумму на родительский счет при помощи метода FamilyOperation.ParentPayday. Кнопка Start создает новый объект FamilyOperation, задает значение свойства NumberOfKids и затем вызывает метод StartThreads с указанным количеством потоков. Метод StopThreads сначала останавливает потоки, а затем выводит окончательный результат в списке. Программный код, выполняющий все эти операции, приведен в листинге 7.11.

Листинг 7.11. Код формы приложения Threading1

```
Protected Sub Timer1_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles timer1.Tick
  UpdateResults()
End Sub

Protected Sub cmdDeposit_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdDeposit.Click
  Dim Amount As Double
  Amount = Val(txtDeposit().Text)
  myFamily.ParentPayday (Amount)
End Sub

Protected Sub cmdStart_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdStart.Click
  myFamily = New FamilyOperation()
  Dim Kids As Integer
  Dim Threads As Integer

  Kids = CInt(Val(txtKids().Text))
  Threads = CInt(Val(txtThreads().Text))
  myFamily.NumberOfKids = Kids
  myFamily.StartThreads (Threads)
  lstResults.Items.Clear()
  timer1.Enabled = True
  cmdStart.Enabled = False
  cmdStop.Enabled = True
  cmdDeposit.Enabled = True
End Sub

Protected Sub cmdStop_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdStop.Click
  myFamily.StopThreads()
  cmdStop.Enabled = False
  cmdStart.Enabled = True
  cmdDeposit.Enabled = False
  UpdateResults()
End Sub
```

Тестирование

Чтобы поближе познакомиться с работой нашего приложения, попробуем запустить его со стандартными параметрами: для десяти детей и одного потока. Нажмите кнопку Deposit 20 или 30 раз, чтобы почувствовать, как деньги переходят от отца к детям и как дети их расходуют. Типичная ситуация показана на рис. 7.1.

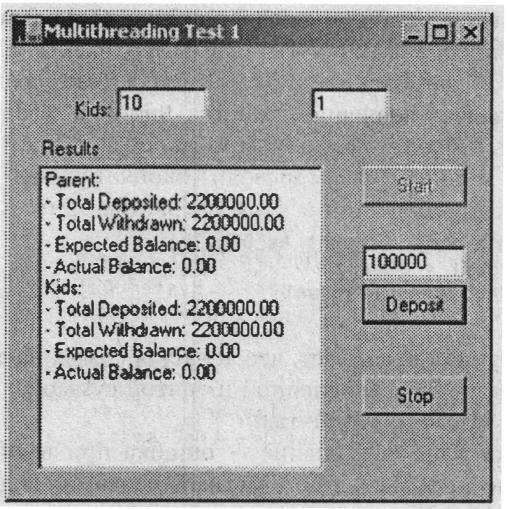


Рис. 7.1. Форма приложения с одним потоком

Итак, мы подошли к моменту истины. Еще раз взгляните на программу и убедитесь в том, что вы понимаете, как она работает.

Вам удалось найти какие-нибудь недочеты?

Попробуйте запустить приложение для 10 детей с 10 фоновыми потоками. Нажмите кнопку **Deposit** несколько раз (чтобы получить результат, показанный на рис. 7.2, потребуется несколько сот нажатий).

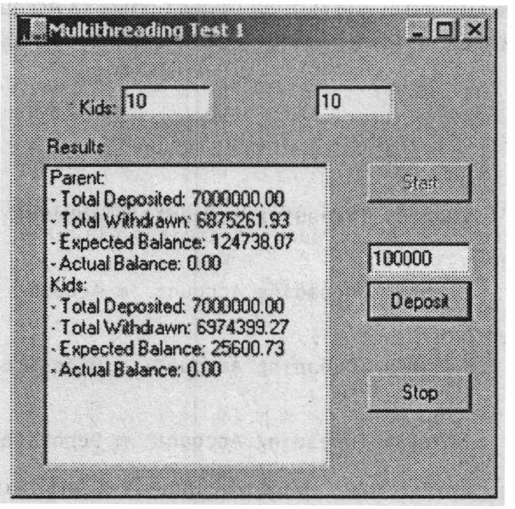


Рис. 7.2. Форма приложения с несколькими потоками

В чем проблема?

Тестируемое приложение устроено чрезвычайно просто. В объекте `Account` определены два метода:

```
Protected Function Withdraw(ByVal amount As Double) As Double
  If amount > m_Account Then
    amount = m_Account
  End If
  m_Account = m_Account - amount
  m_Spent = m_Spent + amount
  Return amount
End Function

Protected Sub Deposit(ByVal amount As Double)
  m_Account = m_Account + amount
  m_Deposited = m_Deposited + amount
End Sub
```

Простая арифметика показывает, что текущая сумма на счету всегда должна быть равна разности между зачисленной и снятой суммой.

Однако в нашем примере это не так.

Возможно только одно объяснение — ошибка при выполнении этих очень простых математических операций. Другими словами, подвох кроется в этих строках:

```
m_Account = m_Account - amount
m_Spent = m_Spent + amount
m_Account = m_Account + amount
m_Deposited = m_Deposited + amount
```

Как это возможно?

Давайте еще раз проанализируем метод `Deposit`, но на этот раз воспользуемся листингом на промежуточном языке (IL), сгенерированном для этого метода (листинг 7.12[1]).

Листинг 7.12. Промежуточный код метода Deposit объекта Account

```
.method family instance void Deposit(float64 amount) il managed
{
  // Code size       31 (0x1f)
  .maxstack 8
  IL_0000: nop
  IL_0001: ldarg.0
  IL_0002: ldarg.0
  IL_0003: ldfld     float64 Threading.Account::m_Account
  IL_0008: ldarg.1
  IL_0009: add
  IL_000a: stfld     float64 Threading.Account::m_Account
  IL_000f: ldarg.0
  IL_0010: ldarg.0
  IL_0011: ldfld     float64 Threading.Account::m_Deposited
  IL_0016: ldarg.1
  IL_0017: add
  IL_0018: stfld     float64 Threading.Account::m_Deposited
  IL_001d: nop
  IL_001e: ret
} // end of method Account::Deposit
```

[1] Листинг получен при помощи дизассемблера .NET. Проблемы, связанные с относительной простотой дизассемблирования приложений .NET (VB или C#), рассматриваются в главе 16.

Я не собираюсь учить вас языку IL. Во-первых, я его сам не знаю, а во-вторых, он вам не нужен. Однако для того, чтобы разобраться в происходящем, достаточно обычной логики. Похоже, команда

```
m_Account = m_Account + amount
```

преобразуется в фрагмент

```
IL_0001: ldarg.0
IL_0002: ldarg.0
IL_0003: ldfld     float64 Threading.Account::m_Account
IL_0008: ldarg.1
IL_0009: add
IL_000a: stfld     float64 Threading.Account::m_Account
```

Как можно предположить, в начале этого фрагмента `m_Account` дважды заносится в стек, после чего верхняя величина в стеке загружается в регистр. Затем загружается указатель на аргумент `amount` и две величины суммируются. Результат присваивается переменной `m_Account`, находящейся в стеке (поскольку она была загружен дважды). Вероятно, происходящее будет понятно читателям, имеющим опыт работы с обратной польской записью[1] (например, владельцам инженерных калькуляторов HP). Остальные пусть не огорчаются; не так уж важно, что именно здесь происходит. Вам лишь необходимо понять, что действия выполняются в следующей последовательности.

1. Переменная `m_Account` загружается в регистр.
2. Параметр `amount` прибавляется к значению `m_Account`.
3. Результат сохраняется в переменной `m_Account`.

Что произойдет, если два потока попытаются одновременно выполнить одну и ту же операцию? Например, возможна следующая последовательность событий.

1. Поток 1 загружает `m_Account` в регистр.
2. Операционная система прерывает поток 1.
3. Поток 2 загружает `m_Account` в регистр.
4. Поток 2 прибавляет параметр `amount` к значению в регистре.
5. Поток 2 сохраняет результат в переменной `m_Account`.
6. Поток 2 прерывается.
7. Поток 1 прибавляет параметр `amount` к содержимому регистра (в котором восстанавливается значение, содержавшееся до прерывания потока на шаге 2, однако текущее содержимое регистра не учитывает изменения, внесенные потоком 2!).
8. Поток 1 сохраняет результат сложения в переменной `m_Account`, фактически стирая значение, сохраненное потоком 2.

Результат — значение `amount` прибавлялось к переменной `m_Account` дважды, но в переменной отражена лишь одна из этих операций!

---

[1] Информацию об обратной польской записи можно найти по адресу http://www.hpmuseum.org/rpn.htm.

Другими словами, поскольку операционная система выполняет переключение задач на ассемблерном уровне (более низком, чем уровень IL), вы должны учитывать возможность переключения внутри отдельных команд VB .NET.

В текущей реализации члены объекта `Account` совместно используются всеми потоками приложения. Это достаточно плохо, но я хочу обратить ваше внимание еще на один факт.

Описанная проблема возникает лишь при очень специфической комбинации операций и переключений. Какова вероятность возникновения этой ошибки, если учесть, что среди тысяч инструкций IL существует лишь несколько мест, в которых переключение потоков может привести к нежелательным последствиям?

Как вы помните, в нашем примере переводимые суммы составляли от 0 до 1 доллара. Если предположить, что средняя сумма равна 50 центам, а общие зачисления на счет равны 7 миллионам долларов (см. рис. 7.2), значит, вероятность ошибки равна примерно 1/14 000 000.

Спрашивается, как отлаживать код, в котором ошибка возникает один раз из 14 000 000 выполнений? А ведь последствия ошибок могут быть самыми разными: от пропажи нескольких центов со счета в банковской программе до смерти пациента в больнице (при выборке сведений о дозировках лекарств из медицинской базы данных).

Надеюсь, вы убедились, что мое первоначальное заявление вовсе не является преувеличением. Затраты на тестирование и отладку многопоточных приложений действительно могут достигать астрономических величин.

Означает ли это, что вы не должны использовать многопоточность?

Нет.

Это означает лишь то, что вы должны хорошо разобраться в многопоточности, прежде чем использовать ее в своих программах. Кроме того, прежде чем переходить к написанию кода, необходимо правильно спроектировать многопоточное приложение. Вы должны безжалостно расправиться со всеми случаями одновременного доступа к переменным и решить, чем их заменить.

А если вы еще недостаточно напуганы, учтите еще одно обстоятельство.

Многие классы CLR *не являются* безопасными по отношению к потокам. Иначе говоря, обращения к объектам CLR из нескольких потоков приводят к таким же фатальным последствиям, как и обращения к общим переменным. Об этом мы поговорим позднее в этой главе.

Первый уровень защиты: проектирование

Пример Threading2 не решает всех проблем приложения Threading1, а лишь демонстрирует некие общие принципы, которые необходимо усвоить. Многопоточные проблемы возникают при обращении к общим переменным из нескольких потоков.

Существует несколько разных подходов к устранению этих проблем.

- Исключить глобальные переменные. Если в программе не будет общих данных, не будет и конфликтов.
- Оставить глобальные переменные, но разрешить доступ к любой отдельной переменной со стороны только одного потока.

- Воспользоваться средствами синхронизации, чтобы с общими данными в любой момент времени работал только один поток.

В примере Threading2 основное внимание уделяется первым двум подходам. Вместо того чтобы разрешить каждому потоку обращения к любым счетам детей, мы связываем каждый поток с одним счетом. Хотя объекты детских счетов остаются общими, с каждым из них работает только один поток, что исключает возможность конфликта.

Чтобы эта схема работала, необходимо предусмотреть механизм идентификации потоков. К сожалению, не существует простого механизма передачи параметров новому потоку при запуске. Следовательно, нам придется воспользоваться общей переменной. В нашем примере используется общая переменная `ThreadCounter`, значение которой увеличивается при каждом создании потока.

Каждый поток должен сохранить свой индекс. Для сохранения можно было бы воспользоваться стековой переменной (например, одной из переменных метода `KidsSpending`), однако в приложении Threading2 продемонстрирован другой подход. Переменная `ThisThreadIndex` объявляется общей для всех экземпляров класса, что эквивалентно ее объявлению как глобальной. Но еще важнее тот факт, что с этой переменной ассоциируется атрибут `ThreadStatic()`, означающий, что для каждого потока приложения создается отдельная копия этой переменной[1]. Вам это ничего не напоминает? Вспомните, о чем говорилось выше: именно так VB6 поступает со всеми глобальными переменными. Для поддержки хранения отдельных копий переменных для каждого потока операционная система использует локальную память потока (как при указании атрибута `ThreadStatic`, так и при хранении глобальных переменных в VB6).

Поскольку каждый поток создает и выполняет свой метод `KidsSpending`, копия переменной `ThreadIndex` загружается текущим значением счетчика `ThreadCounter`, который после этого увеличивается.

Впрочем, на самом деле индекс потока сохраняется несколько иначе: сначала переменная `ThreadCounter` увеличивается, а затем `ThisThreadIndex` присваивается новое значение, уменьшенное на 1. Существует очень малая вероятность того, что сама переменная `ThreadCounter` приведет к проблемам многопоточности, поскольку она совместно используется всеми потоками. Чтобы ликвидировать даже малейшую вероятность конфликта между потоками за эту переменную (например, получения одинаковых индексов двумя потоками), для увеличения переменной `ThreadCounter` применяется метод `Threading.Interlocked.Increment`. Этот общий метод[2] класса `Threading.Interlocked` выполняет атомарное увеличение переменной. Другими словами, процесс увеличения не может быть прерван операционной системой[3].

---

[1] Как будет показано в главе 11, атрибуты .NET — весьма обширная тема. Пока можете рассматривать атрибуты как способ передачи данных CLR — в нашем примере это информация о том, как должна компилироваться переменная.

[2] Напоминаю: общие методы принадлежат всему классу, а не его конкретным объектам и могут вызываться без указания объекта.

[3] Возможно, вы подумали, нельзя ли воспользоваться методом `Interlocked.Increment` для увеличения переменной `m_Account` класса `Account`? Нет, нельзя (метод позволяет увеличивать переменные только на 1), но вы мыслите в верном направлении, и позднее я покажу один из вариантов применения этой методики.

Переменная ThisThreadIndex содержит уникальный номер потока, который используется при индексации массива Kids (листинг 7.13) и гарантирует, что с каждым детским счетом работает один и только один поток.

Листинг 7.13. Метод KidsSpending приложения Threading2

```
Private ThreadCounter As Integer
<ThreadStatic()> Private Shared ThisThreadIndex As Integer

Public Sub KidsSpending()
  Dim Allowance As Double
  Dim thiskid As KidAccount
  ThisThreadIndex = Threading.Interlocked.Increment(ThreadCounter) - 1
  Do
    ' Каждый детский счет обслуживается одним потоком
    ChildIndex = CInt(Int(m_Random.NextDouble() * _
      CDbl(m_NumberOfKids)))
    thiskid = Kids(ThisThreadIndex)

    Allowance = Parent.GiveAllowance()
    thiskid.GetAllowance (Allowance)

    thiskid.Spend()
  Loop Until m_Stopping

End Sub
```

Результат показан на рис. 7.3. После многочисленных нажатий кнопки Deposit становится видно, что родительский объект по-прежнему подвержен многопоточным ошибкам, потому что он совместно используется разными потоками. Тем не менее объекты Kids работают правильно: в результате изменений в архитектуре приложения они перестали быть общими.

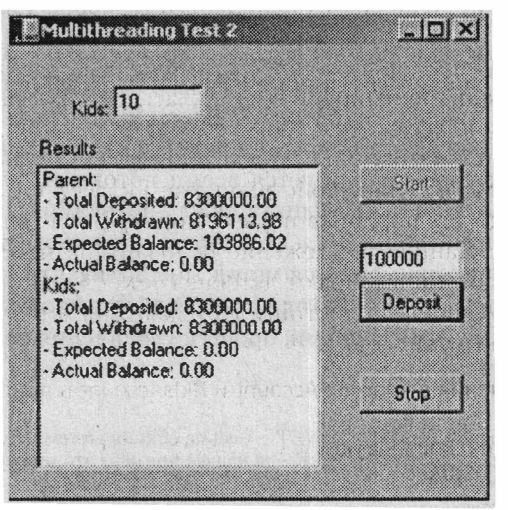


Рис. 7.3. Приложение Threading2

Почему я представил вам пример, ограничивающийся частичным решением проблемы?

Потому что он демонстрирует одну из основных концепций COM, которая может сыграть важную роль при выполнении нетривиальных операций VB .NET, включая операции, связанные со взаимодействием с объектами COM.

Из-за применения механизма подсчета ссылок объекты COM также подвержены многопоточным проблемам: внутренние счетчики ссылок являются общими для всех потоков, использующих объект. В COM определяются три разных потоковых модели: однопоточная (single thread), совместная модель (STA, Single Threaded Apartment) и свободная (MTA, Multi-Threaded Apartment). Решения для однопоточной и совместной модели основаны на описанных выше принципах. Ограничивая доступ к объекту потоком, создавшим объект, эти модели фактически ликвидируют проблемы многопоточности, поскольку с заданным объектом (и всеми его методами/свойствами) может работать только один поток.

Как видите, VB6 выполняет немалую работу для обеспечения безопасности многопоточных приложений. Все глобальные переменные размещаются в локальной памяти потока. Для всех объектов используется совместная потоковая модель, поэтому программисту не приходится беспокоиться о доступе к методам и свойствам со стороны нескольких потоков.

В VB .NET эти меры безопасности не принимаются по умолчанию. Следовательно, перед тем, как наделять свои приложения многопоточными возможностями, вы должны хорошо разобраться в том, как сделать их безопасными. Пример Threading2 — всего лишь первый шаг на этом пути. Давайте посмотрим, какими еще возможностями вы располагаете.

Второй уровень защиты: синхронизация

Итак, борьба с многопоточными проблемами должна начинаться с правильного проектирования. Следующим шагом является применение средств синхронизации, предотвращающих одновременный доступ к общим данным со стороны нескольких потоков.

В проекте Threading3 продемонстрирован подход, основанный на «грубой силе», при котором синхронизируется весь класс. Это означает, что со всеми методами и свойствами класса потоки могут работать лишь поочередно. Если некоторый поток работает с методом или свойством класса, никакие другие потоки не могут работать с методами или свойствами этого класса. CLR переводит их в состояние ожидания до тех пор, пока первый поток не завершит операции с методом или свойством. Данное приложение основано на приложении Threading1 (другими словами, потоки выбирают детские счета случайным образом).

Фрагменты, приведенные в листинге 7.14, дают представление об изменениях в классах `Account` и `KidsAccount`.

Листинг 7.14. Изменения в классах Account и KidsAccount в приложении Threading3

```
Imports System.Runtime.Remoting
' Наследует от ContextBoundObject
' для синхронизации KidAccount.
Public Class Account
  Inherits ContextBoundObject
      .
      .
      .
End Class
```

продолжение ⇩

Листинг 7.14 (продолжение)

```
' Синхронизировать KidAccount,
' чтобы избавиться от проблем с многопоточным доступом.
<Contexts.Synchronization()> Public Class KidAccount
  Inherits Account
      .
      .
      .
End Class
```

Прежде всего бросается в глаза то, что класс `Account` объявляется производным от `ContextBoundObject`. В первоначальном варианте он был производным только от `Object` (поскольку в .NET все типы являются производными от `Object`). Класс, производный от `ContextBoundObject` (который, в свою очередь, является производным от `MarshalByRefObject` и `Object`), наследует дополнительную функциональность, позволяющую CLR ассоциировать объекты с контекстом. В частности, контекст позволяет управлять внешним доступом. Объявление класса производным от `ContextBoundObject` не влияет на внутреннее устройство самого класса `Account`. Однако при работе с классом `KidAccount` CLR видит установленный атрибут `Synchronization`, указывающий на то, что одновременный доступ к классу должен ограничиваться одним потоком.

На рис. 7.4 показан результат выполнения примера Threading3 после многих нажатий кнопки **Deposit**.

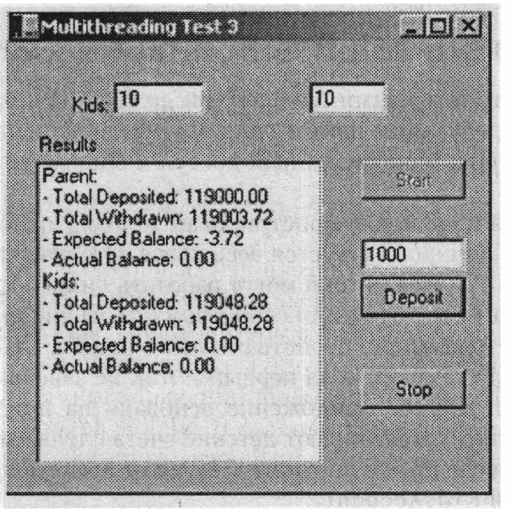


Рис. 7.4. Приложение Threading3

Как и в приложении Threading2, детские счета работают правильно, поскольку доступ к объекту `KidAccount` синхронизирован. Родительский счет остается несинхронизированным, и это, как видно из рисунка, приводит к многопоточным ошибкам. Экспериментируя с этим приложением, нетрудно заметить, что оно работает значительно медленнее предыдущих версий. Общие вопросы быстродействия многопоточных приложений будут рассматриваться ниже. В нашем примере причина замедления очевидна: каждый раз, когда два потока пытаются

обратиться к одному классу, один из потоков блокируется, а операции блокировки и перезапуска потока выполняются относительно долго.

Мы уже выяснили, что в данном случае синхронизация критична только для двух методов: `Deposit` и `Withdraw`. Следовательно, синхронизация всех методов и свойств класса — явный перебор, замедляющий работу программы.

Решения с позиций «грубой силы» (вроде описанного в этом разделе) нежелательны, хотя у них есть свои преимущества — они очень легко реализуются. Вы просто синхронизируете все свои классы и можете быть уверены в полном отсутствии проблем синхронизации. Правда, при этом теряется часть преимуществ многопоточности, но это другой вопрос.

Помимо атрибута `Synchronization` классу можно назначить атрибут `ThreadAffinity`. Атрибут `Synchronization` сообщает CLR о том, что с объектом в любой момент времени может работать только один поток. Атрибут `ThreadAffinity` сообщает CLR, что объект доступен только для того потока, которым он был создан, и в случае необходимости CLR следует организовать передачу данных между потоками, чтобы один поток мог получить доступ к методам и свойствам объекта, существующего в контексте другого потока. Аналогичная концепция используется COM для реализации потоковой модели STA. Учтите, что эти атрибуты не обеспечивают синхронизации общих методов и свойств, относящихся ко всем экземплярам класса.

Ручная синхронизация

Атрибуты `Synchronization` и `ThreadAffinity` обеспечивают простейшую возможность синхронизации доступа ко всем методам и свойствам класса. Как оценить разумность этого подхода? Очень просто: заглянуть в бета-документацию, где под заголовком «Принципы синхронизации» сказано: «...Годится для наивных пользователей».

Надеюсь, после этого вы будете относиться к этому способу синхронизации с таким же «энтузиазмом», как и я.

При всей простоте он практически неизбежно приводит к синхронизации доступа и к тем членам класса, которые *не связываются* с общими данными и могут вполне безопасно использоваться в многопоточных приложениях. В таких случаях рекомендуется использовать ручную синхронизацию.

В программе Threading4 продемонстрирован один из способов решения этой задачи с применением новой команды VB .NET `SyncLock`. Во внутренней реализации этой команды синхронизация программных блоков осуществляется при помощи специального объекта, называемого *монитором*. В листинге 7.15 приведены изменения исходной программы Threading1.

Листинг 7.15. Измененные фрагменты приложения Threading4 (относительно Threading1)

```
Imports System.Runtime.Remoting
Public Class Account
  Protected m_Account As Double
  Protected m_Spent As Double
  Protected m_Deposited As Double
  Private Shared m_Random As New Random()
  Protected LockingObject As String = "HoldTheLock"
```

продолжение

Листинг 7.15 (продолжение)

```
  ' Попытаться снять со счета запрашиваемую сумму,
  ' вернуть фактически снятую сумму.
  Protected Function Withdraw(ByVal amount As Double) As Double
    SyncLock LockingObject
      If amount > m_Account Then
        amount = m_Account
      End If
      m_Account = m_Account - amount
      m_Spent = m_Spent + amount
    End SyncLock
    Return amount
  End Function

  Protected Sub Deposit(ByVal amount As Double)
    SyncLock LockingObject
      m_Account = m_Account + amount
      m_Deposited = m_Deposited + amount
    End SyncLock
  End Sub
      .
      .
      .
End Class
```

Команде `SyncLock` в качестве параметра передается любой объект ссылочного типа (поэтому целые числа могут использоваться лишь после упаковки). Вы можете просто создать пустой класс и воспользоваться им, но я предпочитаю передавать строку, упрощающую идентификацию объекта.

В начале блока `SyncLock` CLR проверяет, не установил ли какой-нибудь другой поток блокировку переменной `LockingObject`. Если блокировка отсутствует, то переменная `LockingObject` блокируется и программе разрешается дальнейшее выполнение. При выходе из блока `SyncLock` блокировка переменной `LockingObject` снимается. Если при достижении команды `SyncLock` переменная `LockingObject` оказывается заблокированной, текущий поток приостанавливается и продолжает работу лишь после того, как поток, установивший блокировку, снимет ее с объекта.

Достоинствами такого решения являются его простота реализации, эффективность и ограничение синхронизации только тем кодом, который обращается к общим переменным класса `Account`.

Как видно из рис. 7.5, синхронизация базового класса `Account` (вместо производных классов `KidAccount` и `ParentAccount`) обеспечивает правильную синхронизацию как родительских, так и детских счетов.

Мы рассмотрели очень простой пример, но давайте проанализируем ситуацию, возникающую в более сложных случаях.

Допустим, у вас имеется два потока, A и B.

Поток A входит в блок `SyncLock` и блокирует переменную `LockingObject`.

Поток B пытается войти в блок `SyncLock`, но ему это не удается из-за блокировки переменной `LockingObject`.

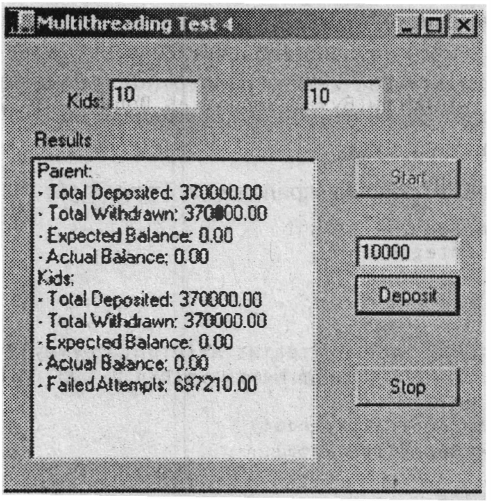


Рис. 7.5. Приложение Threading4

Тем временем поток A, все еще находящийся в блоке `SyncLock`, переходит в ожидание некоторой операции, выполняемой потоком B[1].

Поток A ожидает действий со стороны потока B, но работа потока B приостановлена до тех пор, пока объект A не снимет блокировку с объекта `LockingObject`. В результате оба потока стоят на месте.

Подобная ситуация называется взаимной блокировкой (deadlock). Она возникает каждый раз, когда работа двух или более потоков приостанавливается в ожидании действий со стороны других потоков[2].

Также необходимо ответить на другой вопрос: должна ли переменная `LockingObject` быть общей или нет. Если переменная объявляется общей, все экземпляры классов, производных от `Account`, будут синхронизироваться друг с другом. В нашем случае это перебор, поскольку недостатки приложения обусловлены одновременным доступом к конкретному экземпляру со стороны разных потоков. Тем не менее, если бы объект содержал общие переменные или статические методы, синхронизация по общей объектной переменной была бы неизбежной.

Синхронизация и ожидание

При внимательном взгляде на рис. 7.5 можно заметить в конце списка новую строку. В ней выводится количество неудачных попыток — сумма переменных `m_FailedRequests` по всем объектам детских счетов. Другими словами, значение

[1] Существуют разные ситуации, при которых потоку приходится ожидать выполнения операций другим потоком. Может быть, поток B отвечает за некоторую фоновую операцию или содержит класс с установленным атрибутом `ThreadAffinity`, обращения к которому должны происходить из этого потока.

[2] Один из принципов, известных многим программистам, гласит: «Если взаимная блокировка может произойти, она непременно произойдет».

в этой строке увеличивается каждый раз, когда ребенок хочет потратить деньги, но оказывается, что у него на счету нет достаточной суммы[1].

При внимательном анализе метода KidsSpending (листинг 7.16) можно заметить, что в нем работает бесконечный цикл, который пытается тратить деньги даже в том случае, если счет пуст.

Листинг 7.16. Метод KidsSpending (приложение Threading4)

```
Public Sub KidsSpending()
  Dim ChildIndex As Integer
  Dim Allowance As Double
  Dim thiskid As KidAccount
  Do
    ' Случайно выбранный ребенок тратит некоторую сумму.
    ChildIndex = CInt(Int(m_Random.NextDouble() * CDbl(m_NumberOfKids)))
    thiskid = Kids(ChildIndex)
    Allowance = Parent.GiveAllowance()
    thiskid.GetAllowance (Allowance)
    thiskid.Spend()
  Loop Until m_Stopping

End Sub
```

Чтобы ознакомиться с побочными эффектами такого решения, откройте окно диспетчера задач во время работы этой программы.

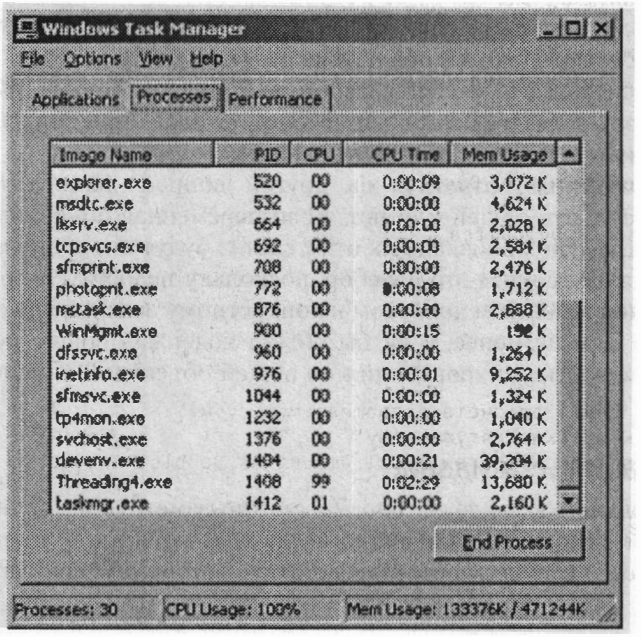


Рис. 7.6. Влияние приложения Threading4 на быстродействие системы

[1] Большая величина обусловлена громадным количеством операций, выполняемых нашим приложением для выявления проблем синхронизации (впрочем, с точки зрения родителя это число наверняка выглядит вполне разумно).

Если вы увидите, что загрузка процессора составляет 99 % (см. рис. 7.6), значит, что-то определенно не так. И действительно, наша программа поглощает все свободное процессорное время и существенно снижает быстродействие системы. Короче говоря, это приложение спроектировано просто ужасно.

На самом деле наше приложение должно быть достаточно «умным» и дожидаться появления денег на счету ребенка перед тем, как их тратить. Одна из приятных особенностей многопоточных приложений заключается в том, что вы можете приостанавливать отдельные потоки без остановки всей программы. Например, можно приостановить поток, выполняющий фоновую операцию, и это никак не отразится на пользовательском интерфейсе. Более того, приостановленный поток в режиме ожидания (например, ожидающий объекта, заблокированного вызовом SyncLock) практически не расходует системных ресурсов.

Проблема решается в приложении Threading5. Начнем с рассмотрения листинга 7.17, в котором для синхронизации доступа вместо блока SyncLock используется объект Mutex.

Листинг 7.17. Класс Account приложения Threading5

```
Public Class Account
  Protected m_Account As Double
  Protected m_Spent As Double
  Protected m_Deposited As Double
  Private Shared m_Random As New Random()

  Protected myMutex As New Threading.Mutex(False)
  Protected Shared MoneyAvailable As New Threading.ManualResetEvent(False)
  .
  .
  .
  Property Deposited() As Double
    Get
      Deposited = m_Deposited
    End Get
    Set
      m_Deposited = Value
    End Set
  End Property

  ' Попытаться снять со счета запрашиваемую сумму,
  ' вернуть фактически снятую сумму.
  Protected Function Withdraw(ByVal amount As Double) As Double
    Try
      myMutex.WaitOne()
    Catch e As Threading.ThreadInterruptedException
      Return 0
    End Try

    If amount > m_Account Then
      amount = m_Account
    End If
    m_Account = m_Account - amount
    m_Spent = m_Spent + amount
    myMutex.ReleaseMutex()
    Return amount
  End Function
```

продолжение

Листинг 7.17 (продолжение)

```
  Protected Sub Deposit(ByVal amount As Double)
    Try
      myMutex.WaitOne()
    Catch e As Threading.ThreadInterruptedException
      Return
    End Try
    m_Account = m_Account + amount
    m_Deposited = m_Deposited + amount
    myMutex.ReleaseMutex()
  End Sub

End Class
```

Чем же применение объекта `Mutex` принципиально отличается от решения с `SyncLock`? В данном случае — ничем. В некоторых ситуациях объект `Mutex` обеспечивает большую гибкость, поскольку ожидание может выполняться сразу для нескольких объектов `Mutex`. Я привел этот пример только для того, чтобы показать, что существуют и другие объекты синхронизации. Обработчик ошибок предотвращает ошибку времени выполнения при прерывании потока (как вы вскоре убедитесь, это существенно). Прежде чем переходить к написанию многопоточных приложений, непременно прочитайте справочную документацию пространства имен `System.Threading` и познакомьтесь с разными объектами синхронизации[1].

В классе `Account` также определяется общий объект `ManualResetEvent`, использующий события синхронизации Win32 (термин «event» не имеет ничего общего с привычными событиями Visual Basic). Объект объявлен общим, поскольку признак наличия денег в нашей имитации является общим для всех объектов детских счетов[2]:

```
Protected Shared MoneyAvailable As New Threading.ManualResetEvent(False)
```

Метод `Spend` класса `KidAccount` проверяет, равна ли доступная сумма нулю. При отсутствии денег вызывается метод `MoneyAvailable.WaitOne()`, который приостанавливает выполнение потока и ожидает установки объекта `ManualResetEvent` другим потоком.

Команда `Wait` находится внутри блока `Try`. Это связано с тем, что в некоторых ситуациях ожидание прерывается самим приложением, точнее говоря, в конце своей работы приложение должно прервать все ожидающие потоки, чтобы обеспечить корректное завершение[3]. В листинге 7.18 показано, как мьютекс `MoneyAvailable` используется классом `KidAccount`.

---

[1] Почему я не рассматриваю их в книге? Потому что моя цель — представить основные концепции, лежащие в основе многопоточности, и помочь вам научиться программировать многопоточные приложения в VB.NET с приемлемым уровнем надежности. От пересказа сведений, содержащихся в документации, никакого проку не будет. Небольшой совет: не ограничивайтесь описаниями объектов. Загляните в документацию Win32 Platform SDK и ознакомьтесь с синхронизационными функциями API; это даст вам более глубокое представление о работе различных объектов.

[2] В более строгой иерархии объект `ManualResetEvent` следовало бы ассоциировать с объектом `Parent`, а объекты `KidAccount` — с конкретным объектом `ParentAccount`. В нашем простом примере это несущественно, однако такие обстоятельства должны учитываться при проектировании иерархий классов, предназначенных для повторного использования.

[3] В нашем приложении используются потоки со свойством `IsBackground = True`, поэтому при прекращении работы основного потока приложение будет успешно завершено. С другой стороны, хороший стиль программирования требует останавливать все потоки перед выходом из приложения, не полагаясь на капризы CLR.

Листинг 7.18. Класс KidAccount приложения Threading5

```
' Синхронизировать KidAccount,
' чтобы избавиться от проблем с многопоточным доступом.
Public Class KidAccount
  Inherits Account

  Private m_FailedRequests As Double
  ReadOnly Property FailedRequests() As Double
    Get
      FailedRequests = m_FailedRequests
    End Get
  End Property

  ' Получить карманные деньги от родителя
  Public Sub GetAllowance(ByVal amount As Double)
    deposit (amount)
  End Sub

  ' Попытаться потратить случайную сумму
  Public Sub Spend()
    Dim amount As Double

        ' Дождаться поступления денег
    Try
      If m_Account = 0 Then MoneyAvailable.WaitOne()
    Catch
      ' Ожидание прерывается при выходе из приложения.
      Exit Sub
    End Try

    amount = GetRandomAmount()
    If amount > m_Account Then amount = m_Account
    If amount = 0 Then
      m_FailedRequests = m_FailedRequests + 1
    Else
      Withdraw (amount)
    End If
  End Sub

    ' Обнуление переменных объекта и базового класса
  Overrides Sub Clear()
    m_FailedRequests = 0
    MyBase.Clear()
  End Sub
End Class
```

Объект `MoneyAvailable` класса `ManualResetEvent` находится под управлением родительского счета. Если при попытке перевода денег на детские счета выясняется, что текущий баланс равен 0, объект `MoneyAvailable` сбрасывается. Когда объект `ManualResetEvent` находится в установленном состоянии, все потоки, ожидающие этого объекта, могут продолжать работу. Когда объект сбрасывается, как в листинге 7.19, все потоки, ожидающие объекта `ManualResetEvent`, приостанавливаются до его установки. Метод `DepositPayroll` устанавливает объект `ManualResetEvent`, тем самым оповещая ожидающие потоки о наличии денег на счету.

Листинг 7.19. Класс ParentAccount приложения Threading5

```
Public Class ParentAccount
  Inherits Account

  ' Метод выбирает случайную сумму карманных расходов
  ' и снимает ее со счета.
  Public Function GiveAllowance() As Double
    Dim amount As Double
    amount = GetRandomAmount()
    amount = Withdraw(amount)
    ' Вернуть фактически снятую сумму (может быть равна 0).
    ' Если денег не осталось, остановить процесс.
    ' Внимание: здесь присутствует
    ' нетривиальная ошибка синхронизации.
    ' Удастся ли вам ее найти?
    If m_Account = 0 Then MoneyAvailable.Reset()
    Return (amount)
  End Function

  Public Sub DepositPayroll(ByVal amount As Double)
    deposit (amount)
    ' Установить объект ManualResetEvent -
    ' сообщить детям о наличии денег.
    MoneyAvailable.Set()
  End Sub
End Class
```

Для корректного завершения приложения также необходимо модифицировать метод StopThreads. При обнаружении ожидающего потока (листинг 7.20) ожидание прерывается методом Threads.Interrupt. В результате операция ожидания инициирует исключение, которое в нашем примере перехватывается и игнорируется.

Листинг 7.20. Метод StopThreads приложения Threading5

```
Public Sub StopThreads()
  Dim Idx As Integer
  Try
    KillFamily() ' Остановить все потоки
    For Idx = 0 To Threads.GetUpperBound(0)
      ' Ожидать завершения потока
      ' Теоретически не исключена редкая ошибка синхронизации -
      ' что, если поток перейдет в состояние ожидания
      ' после этого сравнения?
      If (Threads(Idx).ThreadState And _
        System.Threading.ThreadState.WaitSleepJoin) <> 0 Then
          Threads(Idx).Interrupt()
      End If
      Threads(Idx).Join()
    Next
  Catch
    ' Игнорировать все ошибки
  End Try

End Sub
```

Результат показан на рис. 7.7.

Как видите, количество неудачных попыток заметно уменьшилось. В процессе работы приложения неудачные попытки по-прежнему случаются, но после того,

как все родительские деньги будут потрачены и все потоки детских счетов перейдут в состояние ожидания, дальнейшие попытки прекращаются. Просмотр статистики в диспетчере задач во время работы этой программы показывает, что в это время загрузка процессора равна 0.

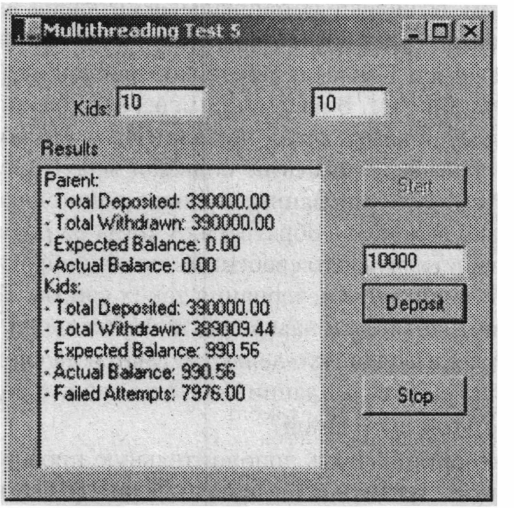


Рис. 7.7. Приложение Threading5

Некоторые тонкости синхронизации

В начале этой главы я говорил, что пойду по несколько необычному пути и буду демонстрировать примеры плохого кода, чтобы вы лучше поняли суть возникающих проблем. Пока все наше внимание было обращено на класс `Account` и использование средств синхронизации для ожидания. Тем не менее приложение Threading5 блестяще демонстрирует некоторые неочевидные проблемы, часто возникающие при синхронизации потоков в приложении.

Какой поток остановлен?

Пример Threading5 построен на основе примера Threading1. В листинге 7.21 приведен основной код метода Threading5.

Листинг 7.21. Метод KidsSpending приложения Threading5

```
Public Sub KidsSpending()
  Dim ChildIndex As Integer
  Dim Allowance As Double
  Dim thiskid As KidAccount
  Do
    ' Случайно выбранный ребенок тратит некоторую сумму.
    ChildIndex = CInt(Int(m_Random.NextDouble() * _
      CDbl(m_NumberOfKids)))
    thiskid = Kids(ChildIndex)
    Allowance = Parent.GiveAllowance()
    if (Not m_Stopping) Then thiskid.GetAllowance (Allowance)
    thiskid.Spend()
  Loop Until m_Stopping

End Sub
```

Одно из изменений — проверка флага `m_Stopping` перед вызовом `GetAllowance`. Необходимость проверки вызвана тем, что существует вероятность выхода из функции `GiveAllowance` из-за прерывания потока. В этом случае безусловный вызов `GetAllowance` приведет к повторному блокированию потока, которое уже не прервется. Таким образом, возникает потенциальная ситуация взаимной блокировки, поскольку главный поток будет бесконечно ждать завершения потока.

Впрочем, это еще не все. Как вы помните, при каждой итерации поток выбирает произвольный детский счет. В нашем примере, если баланс равен нулю, поток, обращающийся к этому объекту, приостанавливается. Но поскольку счета выбираются случайным образом, существует большая вероятность того, что два или более потока окажутся заблокированными в ожидании одного объекта, тогда как к другим счетам никто вообще не обратится. А когда на родительском счету кончатся деньги, вероятность того, что свободные потоки обратятся к непустым счетам детей, уменьшается по мере исчерпания таких счетов. Таким образом, после приостановки всех потоков почти наверняка найдутся непустые счета. Это видно из рис. 7.7, который показывает, что деньги остаются на счетах детей даже после приостановки всех потоков в ожидании перевода денег с родительского счета.

Как справиться с этой проблемой?

Например, можно организовать дополнительную проверку в функции `KidsSpending` и не пытаться тратить деньги, если баланс равен нулю.

Неожиданности с общими переменными

Вероятно, вы уже поняли, что проблемы синхронизации возникают из-за доступа к общим данным из разных потоков. Внимательно просмотрите листинг 7.22 с фрагментом класса `ParentAccount`: в нем содержится вполне недвусмысленный намек на возникающую проблему.

Листинг 7.22. Методы GiveAllowance и DepositPayroll приложения Threading5 (повторение)

```
Public Function GiveAllowance() As Double
  Dim amount As Double
  amount = GetRandomAmount()
  amount = Withdraw(amount)
  ' Вернуть фактически снятую сумму (может быть равна 0).
  ' Если денег не осталось, остановить процесс.
  ' Внимание: здесь присутствует
  ' нетривиальная ошибка синхронизации.
  ' Удастся ли вам ее найти?
  If m_Account = 0 Then MoneyAvailable.Reset()
  Return (amount)
End Function

Public Sub DepositPayroll(ByVal amount As Double)
  Deposit (amount)
  ' Установить объект ManualResetEvent -
  ' сообщить детям о наличии денег.
  MoneyAvailable.Set()
End Sub
```

Ну как, увидели?

Рассмотрим следующую последовательность событий.

- Поток вызывает метод `DepositPayroll`.
- Второй поток продолжает работать. (Даже если родительский счет опустел, другие потоки продолжают работать в течение некоторого промежутка времени, поскольку у детей еще есть деньги.) Этот поток вызывает метод `GiveAllowance`.
- Второй поток прерывается непосредственно после проверки условия `m_Account = 0`.
- Первый поток продолжает выполнять метод `DepositPayroll`, вызывает метод `Deposit` и устанавливает объект `MoneyAvailable` класса `ManualResetEvent`.
- Второй поток продолжает работу и сбрасывает объект `MoneyAvailable`.

Результат — излишнее ограничение доступа к объектам детских счетов.

Вероятность такого совпадения очень мала — настолько, что в приложении вы практически никогда ее не встретите (даже если вам каким-то образом удастся ее обнаружить). Но теоретически это все же возможно.

Почему возникла эта проблема?

Потому что сам объект `MoneyAvailable` класса `ManualResetEvent` является общей переменной, к которой могут одновременно обращаться несколько потоков!

Оказывается, использование служебной переменной для синхронизации не защищает ее от собственных проблем одновременного доступа! Если это общая переменная (как большинство объектов синхронизации), возможно, вам придется воспользоваться дополнительным механизмом синхронизации (таким, как блок `SyncLock`) для ограничения доступа к объекту и тому коду, которым он управляет, чтобы он не создавал проблемы синхронизации следующего уровня.

В нашем примере эта потенциальная проблема решается включением проверки условия в метод `GiveAllowance` и оба вызова `DepositPayroll` в блоке `SyncLock` (с той же переменной).

Окончание работы приложения

Давайте вернемся к новому варианту функции `StopThreads` (листинг 7.23).

Листинг 7.23. Функция StopThreads приложения Threading5

```
Public Sub StopThreads()
  Dim Idx As Integer
  Try
    KillFamily() ' Остановить все потоки
    For Idx = 0 To Threads.GetUpperBound(0)
      ' Ожидать завершения потока.
      ' Теоретически не исключена редкая ошибка синхронизации -
      ' что, если поток перейдет в состояние ожидания
      ' после этого сравнения?
      If (Threads(Idx).ThreadState And _
        System.Threading.ThreadState.WaitSleepJoin) <> 0 Then
        Threads(Idx).Interrupt()
      End If
      Threads(Idx).Join()
    Next
  Catch
    ' Игнорировать все ошибки
  End Try

End Sub
```

Также рассмотрим функцию Spend:

```
Public Sub Spend()
  Dim amount As Double

  ' Дождаться поступления денег.
  Try
    If m_Account = 0 Then MoneyAvailable.WaitOne()
  Catch
    ' Ожидание прерывается при выходе из приложения.
    Exit Sub
  End Try
  .
  .
  .
```

Рассмотрим следующую ситуацию.

- Метод StopThreads вызывается в тот момент, когда поток выполняет метод Spend, а переменная m_Account равна нулю (у ребенка кончились деньги). У родителя тоже нет денег, поэтому объект MoneyAvailable сбрасывается.
- Система переключается с этого потока на другой, в котором выполняется StopThreads.
- Метод StopThreads проверяет условие, обнаруживает, что поток не находится в состоянии ожидания, переходит к методу Join и ждет завершения потока.
- Система снова переключается на поток с методом Spend, который обнаруживает, что счет пуст, и переводит поток в состояние ожидания вызовом метода WaitOne.
- Поток StopThreads приостанавливается методом Join в ожидании завершения потока Spend, однако последний приостановлен в ожидании наличия денег. В результате возникает взаимная блокировка, а приложение не завершится.

Как и прежде, это очень редкая ситуация, и вероятность ее чрезвычайно мала. Она тоже обусловлена тем, что объекты синхронизации (как MoneyAvailable, так и сам объект Thread) являются общими.

Однако на этот раз простым решением уже не обойтись.

На первый взгляд хочется заключить вызов Spend в функции KidsSpending в блок SyncLock и включить проверку m_Stopping перед вызовом метода Spend, как показано в следующем фрагменте:

```
Public Sub KidsSpending()
  .
  .
  .
  SyncLock simeobject
    If Not m_Stopping Then thiskid.Spend()
  End SyncLock
  Loop Until m_Stopping
```

Вызов функции KillFamily тоже заключается в блок SyncLock.

Тем самым мы предотвращаем присваивание True переменной m_Stopping во время выполнения метода Spend. После успешного вызова KillFamily вызов thiskid.Spend уже не состоится, поскольку проверка m_Stopping и вызов Spend находятся в одном блоке SyncLock.

К сожалению, такое решение приводит к взаимной блокировке, поскольку потоки, ожидающие объекта `MoneyAvailable`, удерживают блокировку и предотвращают выполнение не только `KillFamily`, но и любых других потоков, пытающихся вызвать `Spend`.

Честно говоря, я не вижу сколько-нибудь изящного решения для наших классов `Account`[1]. Один из возможных вариантов — установка тайм-аута для вызова `Join`. При обработке исключения следует найти приостановленный поток, прервать его и снова войти в `Join`. Это хлопотное и неуклюжее решение, но оно работает.

Наиболее правильный подход — заново спроектировать классы `Account` таким образом, чтобы в них были определены собственные методы `Interrupt`. При вызове из потока объект не только выходит из текущего ожидания, но и устанавливает флаг (конечно, должным образом синхронизируемый), чтобы в будущем он ни при каких условиях не вошел в состояние ожидания заново.

Короче говоря, многопоточность затрудняет частые исправления и модификации кода. Если вы сталкиваетесь с нетривиальными ошибками синхронизации и не знаете, что с ними делать, лучше вернуться и пересмотреть архитектуру приложения.

Многопоточность также сильно затрудняет тестирование. Все проблемы, описанные в этом разделе (кроме первой), носят в основном теоретический характер и на практике возникают крайне редко. Я знаю о них только потому, что внимательно проанализировал программу и спросил себя: «А что, если...?» Если вы собираетесь писать многопоточные приложения, привыкните к мысли, что вам придется тщательно анализировать каждую строку программы, в которой потоки могут взаимодействовать посредством общих переменных или методов. Для поиска и решения многопоточных проблем следует использовать лучший отладчик из всех существующих — тот, что находится у вас в голове.

Помните о завершителях

В нашем случае это несущественно, поскольку ни в одном из представленных классов завершители не используются, однако вы должны помнить о том, что завершители работают в отдельном потоке. Впрочем, можно смело предположить, что другие методы конкретного экземпляра никогда не выполняются одновременно с завершителем — если бы это было возможно, завершитель объекта попросту бы не вызывался. Следовательно, особое внимание следует обращать на глобальные и общие переменные, а не на переменные конкретного объекта.

Странности Random

В этом разделе мы рассмотрим одну потенциальную проблему, о который вы, вероятно, и не догадывались. Оказывается, в нашем приложении есть еще одна общая переменная. В классе `Account` присутствует следующая строка:

```
Private Shared m_Random As New Random()
```

Для получения всех случайных чисел использовался один объект-генератор, который используется всеми экземплярами класса из любых потоков.

Но кто сказал, что этот объект безопасен по отношению к потокам?

Я не говорил. Более того, в документации об этом тоже ничего не сказано.

Запомните раз и навсегда: многие классы CLR *не являются* безопасными по отношению к потокам.

---

[1] Принимаются предложения.

На данный момент трудно сказать, какие классы безопасны по отношению к потокам, а какие — нет. Надеюсь, в будущем Microsoft включит эти сведения в документацию для всех классов. Пока я точно знаю, что класс `Console` безопасен, и почти уверен в том, что класс `Random` не безопасен. Почему? Потому что фрагмент

```
Dim amount As Double
amount = int(m_Random.NextDouble() * 100)
```

в очень редких случаях вызывает ошибку переполнения.

Как может возникнуть переполнение, если `m_Random.NextDouble` всегда возвращает значение из интервала от 0 до 1? Никак, если только `m_Random.NextDouble` по каким-то причинам не нарушает границы этого интервала. Это может объясняться либо ошибкой в программной реализации генератора случайных чисел (теоретически возможно), либо порчей данных, обусловленной тем, что объект небезопасен по отношению к потокам (гораздо более вероятно). За долгие часы тестирования эта ошибка возникла всего два раза; это в очередной раз доказывает, что с многопоточными проблемами следует бороться на уровне проектирования. Обнаружить их в процессе тестирования очень трудно.

Формы и элементы тоже небезопасны по отношению к потокам. В главе 13 вы узнаете, как организовать безопасное обращение к членам объекта формы из другого потока приложения.

Преимущества многопоточности

Я нисколько не удивлюсь, если после описания всевозможных трудностей и проблем, связанных с многопоточностью, многие читатели готовы сдаться и никогда не связываться с программированием многопоточных приложений. Хотя это и не входило в мои намерения, по-моему, честный и объективный взгляд на ситуацию все же лучше слепого энтузиазма, который часто преподносит многопоточность как панацею, сулящую громадный рост быстродействия.

Теперь, когда вы познакомились со всеми опасностями (особенно в приложениях со свободной потоковой моделью), необходимо разобраться, какие же положительные стороны имеет многопоточность. Это позволит вам понять, способна ли многопоточность поднять быстродействие в конкретной ситуации и оправдает ли выигрыш возросшие затраты на разработку и тестирование.

Чтобы оценить преимущества многопоточности, необходимо хорошо понимать условия, в которых должен работать программный продукт. У приложений действуют одни факторы, у компонентов — другие, причем в последнем случае многое зависит от приложения, управляющего работой компонента. Иногда потоковую безопасность приходится обеспечивать даже для компонентов, которые сами не создают и не используют потоков, если эти компоненты должны использоваться клиентами со свободной потоковой моделью.

Подробное рассмотрение даже стандартных ситуаций в Windows выходит за рамки этой книги. К счастью, все проблемы, возникающие в разных условиях, неизменно сводятся к нескольким фундаментальным преимуществам многопоточности, которые будут рассмотрены ниже.

Эффективное ожидание

Приложение Threading5 демонстрирует одну из самых простых и притом полезных особенностей многопоточности — возможность перевода фоновых потоков в состояние ожидания с высокой эффективностью. Windows поддерживает фоновую реализацию для многих операций (например, пересылки данных между файлами или сетевыми сокетами) и позволяет потокам ожидать завершения этих операций. Возможны и другие условия ожидания, например истечение некоторого интервала времени или прекращение работы потока/процесса. Однопоточные приложения и компоненты не могут использовать эти возможности оптимальным образом, поскольку приостановка основного потока заметно повлияла бы на работу приложения/компонента. В частности, это привело бы к полной блокировке всего пользовательского интерфейса приложения. В таких ситуациях приходится использовать таймер и проводить периодический опрос объекта с проверкой завершения операции — такое решение работает, но крайне неэффективно.

В VB6 проблема эффективного ожидания решается плохо. Хотя EXE-приложения ActiveX могут создавать новые потоки, вызовы методов этих объектов обычно производятся синхронно. Следовательно, даже в случае приостановки потока EXE-приложения ActiveX, он не сможет вернуть управление вызывающему методу, что приведет к фактической блокировке работы клиента (а нередко и к тайм-ауту OLE Automation). Компоненты, оформленные в виде ActiveX DLL, в Visual Basic 6 не могут надежно создавать потоки, поэтому операция ожидания приводит к приостановке потока клиентского приложения[1].

В VB .NET вы просто порождаете новый объект потока, приказываете ему выполнить операцию ожидания (и таким образом войти в высокоэффективное состояние ожидания) и инициировать событие при ее завершении.

Преимущество такого подхода заключается в том, что создаваемые для этой цели потоки обычно относительно просты и не вызывают проблем синхронизации. Главный поток готовит нужную операцию и запускает новый поток. Подчиненный поток выполняет заданную операцию и ожидает результата. После завершения операции подчиненный поток инициирует событие или устанавливает флаг, сообщающий о завершении операции, после чего прекращает работу, а главный поток получает и использует информацию тогда, когда сочтет нужным. Вообще говоря, главный поток должен задавать значения всех общих данных перед запуском подчиненного потока — в этом случае проблемы синхронизации маловероятны. Главный поток не обращается к общим данным до тех пор, пока фоновый поток не установит признак готовности. Если действовать достаточно внимательно, при подобных операциях проблемы синхронизации не возникают. Главное — не забывайте о том, чтобы прервать ожидающие потоки и дать им возможность нормально завершиться при окончании работы компонента.

Фоновые операции

С концептуальной точки зрения идея фоновых операций проста. Конечно, все приложения этой главы в той или иной степени демонстрируют фоновые опера-

[1] Замечу, что в распоряжении пользователей пакета Desaware SpyWorks уже давно находится компонент для работы с фоновыми потоками, упрощающий ожидание и общие фоновые операции в Visual Basic 6.

ции в многопоточном приложении. Программа Threading5 показывает, как использовать многопоточность для организации ожидания; фоновые операции всего лишь расширяют этот принцип. Если ваше приложение или компонент выполняет продолжительные операции (например, пересылку данных), ожидание конца этих операций обычно приводит к неприемлемому снижению быстродействия. Для примера возьмем текстовый редактор. Отдав команду сохранения файла, пользователь вполне может подождать ее завершения, но автоматическое сохранение резервной копии, периодически «замораживающее» приложение, вызвало бы сильное раздражение. Средства фонового выполнения операций — от фоновой печати до пересчета формул в электронной таблице, от построения сложных изображений до передачи данных по сети — играют важную роль во многих современных приложениях.

Трудность безопасной реализации многопоточности такого рода определяется сложностью приложения/компонента и фоновой операции.

Эффективный доступ со стороны клиента

Типичный web-сервер может получать запросы от сотен разных клиентов. Если бы в любой момент времени обрабатывался запрос лишь от одного клиента, это могло бы существенно снизить скорость обработки запросов. Обычно такие приложения поддерживают пул потоков; очередной запрос обрабатывается свободным потоком, что предотвращает блокировку сервера одним клиентом. Обратите внимание: я говорю «могло бы», а не «снизило бы». Как вы вскоре узнаете, проблема многопоточности даже в серверных приложениях весьма непроста и неочевидна.

Одна из главных причин, по которой программисты Visual Basic давно стремились к свободной многопоточности, заключается в том, что программы типа Internet Information Server (IIS) лучше всего работают с компонентами, использующими свободную потоковую модель. Сейчас я объясню, с чем это связано.

Каждый web-запрос, поступающий к IIS, полностью изолирован от всех остальных. Чтобы функции web-сайта выходили за рамки простого просмотра статических страниц, сервер должен располагать средствами для сохранения информации (состояния) между запросами от одного пользователя. Эту задачу можно решить разными способами, которые мы не рассматриваем, — достаточно просто сказать, что IIS позволяет web-приложениям сохранять информацию между запросами. Например, это позволяет создать web-приложение, которое читает информацию из базы данных, генерирует форму и отправляет ее пользователю. При поступлении следующего запроса от того же пользователя приложение продолжает работу и обрабатывает введенные данные. IIS обладает необходимыми средствами, чтобы написанное вами web-приложение могло сохранять информацию состояния между отправкой формы и получением заполненной формы от пользователя.

Для обеспечения максимальной эффективности IIS передает новые запросы любым свободным потокам.

Что произойдет, если ваше web-приложение использует объект Visual Basic 6 и потребует у IIS сохранить его?

Объект использует совместную потоковую модель. Это означает, что все обращения к методам и свойствам должны осуществляться из потока, создавшего объект. Если IIS сохраняет такой объект, он должен проследить за тем, чтобы все

запросы текущего сеанса (для конкретного пользователя и конкретного приложения) передавались тому же самому потоку. А что, если поток занят и выполняет какую-то другую продолжительную операцию? В этом случае входящему запросу придется ждать его освобождения.

Впрочем, эту проблему ни в коем случае нельзя назвать фатальной, просто по сравнению со свободной потоковой моделью быстродействие несколько снижается. IIS приходится предпринимать дополнительные усилия, чтобы отслеживать потоки для каждого объекта и выполнять необходимое переключение потоков.

Теоретически эта ситуация относится ко всем приложениям и службам, обрабатывающим запросы от нескольких клиентов, включая бизнес-службы и компоненты, работающие на сервере.

Сложность программирования многопоточных приложений такого рода зависит от того, что именно вы создаете: сервер или компонент, используемый сервером.

Если создается сервер с собственным пулом потоков, необходимо учитывать ряд факторов.

- Вы должны правильно организовать управление пулом потоков, обращая особое внимание на синхронизацию и распределение потоков между клиентами.
- Особое внимание необходимо уделить общим функциональным возможностям или общим данным, совместно используемым всеми потоками пула.
- Вы должны обеспечить изоляцию потоков и синхронизацию доступа в тех областях, в которых потоки взаимодействуют друг с другом.

Если речь идет о компонентах, используемых этими серверами, вы должны внимательно изучить все требования и ограничения, устанавливаемые сервером. Если известно, что сервер является многопоточным и ожидает, что все его компоненты используют свободную потоковую модель, вы должны проследить за тем, чтобы все открытые методы и свойства компонента могли безопасно вызываться в любой момент времени любым потоком. Если ваш компонент может инициировать события сервера, необходимо выяснить, могут ли события инициироваться любыми потоками или же существуют какие-либо ограничения.

К сожалению, многие программисты полагают, что многопоточность является обязательным атрибутом любых приложений «клиент-сервер». Об этом стоит поговорить особо.

Оценка быстродействия в многопоточных приложениях

Один из моих любимых афоризмов звучит так:

«Существует ложь, наглая ложь и эталонные тесты»[1].

Я уверен, что это правда. Правильно провести эталонное тестирование очень трудно, а манипулировать результатами легче легкого. Если бы результаты президентских выборов 2000 года были основаны на опросах общественного мнения вместо относительно точного подсчета бюллетеней в Палм-Бич, мы бы до сих пор ждали результата в суде.

[1] Редактор потребовал, чтобы я дал ссылку на источник. Марк Твен однажды сказал: «Существует ложь, наглая ложь и статистика» — но я уверен, что он не упомянул эталонные тесты только потому, что в его время не существовало компьютеров.

На одной конференции я сделал доклад, посвященный многопоточности и масштабируемости. Используя теоретические аргументы, я показал, что многопоточность не всегда является оптимальным решением для серверного приложения, обслуживающего большое количество клиентов. Иногда она приводит к снижению быстродействия.

Поскольку VB .NET поддерживает многопоточность, я решил подкрепить этот факт практическим примером. Ниже приведены вполне реальные результаты. Впрочем, вряд ли вам удастся точно воспроизвести их: эксперимент зависит слишком от многих факторов. Вы работаете на другом компьютере и в другой ОС; возможно, при проведении хронометража в системе работают другие приложения, влияющие на полученный результат. И все же эти числа достаточно наглядно доказывают то, что я хочу сказать.

Приложение ThreadPerformance

В приложении ThreadPerformance определяется класс WorkerThread, выполняющий различные операции по запросу клиента. Этот класс также позволяет измерять продолжительность этих операций. В CLR определяется объект `System.TimeSpan`, представляющий промежуток времени и хорошо подходящий для хронометража. Свойство `ElapsedTimeForCall` возвращает ссылку на текущий объект `TimeSpan`.

Чтобы увеличить продолжительность выполняемой операции, следует присвоить переменной `LongDuration` значение `True`. В программе этот факт используется для проведения экспериментов с операциями разной продолжительности.

```
Imports System.Threading

Public Class WorkerThread
  Private myTimeSpan As TimeSpan
  Public ReadOnly Property ElapsedTimeForCall() As TimeSpan
    Get
      Return myTimeSpan
    End Get
  End Property

  Public LongDuration As Boolean
```

Метод `WorkingOperation` имитирует операцию с интенсивной загрузкой процессора посредством выполнения очень длинного цикла. Объект `TimeSpan` инициализируется текущим временем, которое затем будет вычтено из времени окончания операции. Вызывая этот метод из независимых потоков, мы сможем измерить выигрыш по быстродействию, достигнутый в результате распределения разных клиентов по разным потокам:

```
  Public Sub WorkingOperation()
    Dim counter As Long
    Dim upperlimit As Long
    Dim temp As Long
    myTimeSpan = TimeSpan.FromTicks(DateTime.Now.Ticks)
    upperlimit = 50000000
    If LongDuration Then upperlimit = 5 * upperlimit
    For counter = 1 To upperlimit
      temp = 5
```

```
    Next
    myTimeSpan = _
      TimeSpan.FromTicks(DateTime.Now.Ticks).Subtract(myTimeSpan)
  End Sub
```

Метод SynchronousRequest отделяет присваивание переменной myTimespan от самой операции. Это необходимо, поскольку мы будем последовательно вызывать метод SynchronousOperation для каждого объекта, чтобы измерить быстродействие одного потока, последовательно обрабатывающего серию клиентских запросов. Нас интересует суммарное время, затраченное с начала первой операции, а не текущие затраты. В остальном метод SynchronousOperation идентичен методу WorkingOperation.

```
Public Sub SynchronousRequest()
  myTimeSpan = TimeSpan.FromTicks(DateTime.Now.Ticks)
End Sub

Public Sub SynchronousOperation()
  Dim counter As Long
  Dim upperlimit As Long
  Dim temp As Long
  upperlimit = 50000000
  If LongDuration Then upperlimit = 5 * upperlimit
  For counter = 1 To upperlimit
    temp = 5
  Next
  myTimeSpan = TimeSpan.FromTicks(DateTime.Now.Ticks).Subtract(myTimeSpan)
End Sub
```

Методы SleepingOperation и SleepingSynchronous похожи на только что приведенные методы, за исключением того, что клиентские запросы не обеспечивают интенсивной загрузки процессора (листинг 7.24). Эти методы имитируют клиентские запросы, сопряженные с файловыми или сетевыми операциями, операциями с базами данных или интенсивным вводом-выводом без загрузки процессора.

Листинг 7.24. Методы SleepingOperation и SleepingSynchronous модуля WorkerThread

```
  Public Sub SleepingOperation()
    Dim sleepspan As Integer
    sleepspan = 1000
    If LongDuration Then sleepspan = sleepspan * 5
    myTimeSpan = TimeSpan.FromTicks(DateTime.Now.Ticks)
    Thread.CurrentThread.Sleep (sleepspan)
    myTimeSpan = _
      TimeSpan.FromTicks(DateTime.Now.Ticks).Subtract(myTimeSpan)
  End Sub

  Public Sub SleepingSynchronous()
    Dim sleepspan As Integer
    sleepspan = 1000
    If LongDuration Then sleepspan = sleepspan * 5
    Thread.CurrentThread.Sleep (sleepspan)
    myTimeSpan = _
      TimeSpan.FromTicks(DateTime.Now.Ticks).Subtract(myTimeSpan)
  End Sub

End Class
```

Программа ThreadingPerformance, оформленная в виде консольного приложения, создает пять объектов `WorkerThread` и пять потоков. Функция `RunTest` создает пять потоков, по одному для каждого объекта `WorkerThread`. Обратите внимание: сначала мы в цикле создаем потоки, а затем в следующем цикле их запускаем. Это сделано для повышения точности измерений, чтобы потоки запускались по возможности одновременно.

Метод `RunTest2` (листинг 7.25) использует метод `SleepingOperation` для тестирования ситуаций без интенсивной загрузки процессора. В остальном этот метод практически идентичен `RunTest`.

Листинг 7.25. Модуль Module1 приложения ThreadingPerformance

```
Module Module1

  Dim WorkerObjects(4) As WorkerThread
  Dim Threads(4) As Threading.Thread

  Sub RunTest()
    Dim x As Integer

    ' Создать потоки
    For x = 0 To 4
      Threads(x) = New Threading.Thread(AddressOf _
        WorkerObjects(x).WorkingOperation)
    Next x

    ' Запустить 5 потоков
    For x = 0 To 4
      Threads(x).Start()
    Next

    ' Ожидать их завершения
    For x = 0 To 4
      Threads(x).Join()
    Next

  End Sub

  Sub RunTest2()
    Dim x As Integer

    ' Создать потоки
    For x = 0 To 4
      Threads(x) = New Threading.Thread(AddressOf _
        WorkerObjects(x).SleepingOperation)
    Next x

    ' Запустить 5 потоков
    For x = 0 To 4
      Threads(x).Start()
    Next

    ' Ожидать их завершения
    For x = 0 To 4
      Threads(x).Join()
    Next

  End Sub
```

Методы RunSynchronous и RunSynchronous2 сначала фиксируют время запуска для каждого объекта вызовом SynchronousRequest, а затем несколько раз вызывают SynchronousOperation или SleepingSynchronous, имитируя последовательную обработку клиентских запросов (листинг 7.26).

Листинг 7.26. Модуль Module1 приложения ThreadingPerformance (продолжение)

```
Public Sub RunSynchronous()
  Dim x As Integer
  For x = 0 To 4
    WorkerObjects(x).SynchronousRequest()
  Next
  For x = 0 To 4
    WorkerObjects(x).SynchronousOperation()
  Next

End Sub

Public Sub RunSynchronous2()
  Dim x As Integer
  For x = 0 To 4
    WorkerObjects(x).SynchronousRequest()
  Next
  For x = 0 To 4
    WorkerObjects(x).SleepingSynchronous()
  Next

End Sub
```

Метод ReportResults (листинг 7.27) выводит суммарные затраты времени по каждому объекту WorkerThread и среднее время по всем объектам. Главная программа проводит тестирование дважды: в первый раз все клиентские запросы имеют равную длину, а во второй раз первый клиентский запрос значительно длиннее остальных (для чего свойству LongDuration объекта WorkerThread задается значение True).

Листинг 7.27. Модуль Module1 приложения ThreadingPerformance (продолжение)

```
  Sub ReportResults()
    Dim x As Integer
    Dim tot As Double
    Dim ms As Double
    For x = 0 To 4
      ms = WorkerObjects(x).ElapsedTimeForCall.TotalMilliseconds
      tot = tot + ms
      Console.Write(Int(ms).ToString + " ,")
    Next
    Console.Write(" Average: " + Int(tot / 5).ToString())
    Console.WriteLine()
  End Sub

  Sub Main()
    Dim x As Integer
    For x = 0 To 4
      WorkerObjects(x) = New WorkerThread()
    Next

    Console.WriteLine ("Running tests...")

    Console.WriteLine ("CPU-Intensive operations")
```

продолжение ⇙

Листинг 7.27 (продолжение)

```
    Console.WriteLine ("Synchronous Equal length operations")
    WorkerObjects(0).LongDuration = False
    RunSynchronous()
    ReportResults()

    Console.WriteLine ("Synchronous one long operation")
    WorkerObjects(0).LongDuration = True
    RunSynchronous()
    ReportResults()

    Console.WriteLine ("Multithreaded Equal length operations")
    WorkerObjects(0).LongDuration = False
    RunTest()
    ReportResults()

    Console.WriteLine ("Multithreaded One long operations")
    WorkerObjects(0).LongDuration = True
    RunTest()
    ReportResults()

    Console.WriteLine ("Non CPU-Intensive operations")

    Console.WriteLine ("Synchronous Equal length operation")
    WorkerObjects(0).LongDuration = False
    RunSynchronous2()
    ReportResults()

    Console.WriteLine ("Synchronous one long operation")
    WorkerObjects(0).LongDuration = True
    RunSynchronous2()
    ReportResults()

    Console.WriteLine ("Multithreaded Equal length operations")
    WorkerObjects(0).LongDuration = False
    RunTest2()
    ReportResults()

    Console.WriteLine ("Multithreaded One long operations")
    WorkerObjects(0).LongDuration = True
    RunTest2()
    ReportResults()

    Console.ReadLine()

  End Sub

End Module
```

Результаты работы программы ThreadPerformance

Начнем с рассмотрения синхронных результатов (конкретные значения зависят от мощности процессора, конфигурации компьютера и версии .NET). В данном случае имитируются клиентские запросы, которые поступают одновременно и последовательно обрабатываются одним потоком.

```
Running tests...
CPU-Intensive operations
Synchronous Equal length operations
2093, 4186, 6238, 8271, 10284, Average: 6214
Synchronous one long operation
10184, 12217, 14250, 16303, 18346, Average: 14260
```

Как видите, продолжительность каждой операции с интенсивной загрузкой процессора составляет около 2 секунд. Таким образом, при последовательном выполнении первая операция занимает 2 секунды, вторая увеличивает суммарные затраты времени до 4 секунд (2 секунды тратится на ожидание завершения первой операции) и т. д.

Среднее время равно 6,2 секунды, что достаточно близко к теоретическому среднему (6 секунд).

Тем не менее, если первая операция занимает много времени (в нашем примере около 10 секунд), это оказывает огромное влияние на общее быстродействие, поскольку всем коротким запросам приходится ждать, пока будет обработан длинный запрос.

Давайте посмотрим, что происходит, когда одновременно полученные запросы с интенсивной загрузкой процессора обрабатываются разными потоками.

```
Multithreaded Equal length operations
8442, 8331, 8221, 8111, 7991, Average: 8219
Multithreaded One long operation
15241, 7921, 8301, 8181, 7921, Average: 9513
```

Когда все запросы имеют равную длину, время обработки каждого запроса заметно увеличивается. Дело в том, что многопоточность не следует воспринимать как магическое повышение мощности процессора — ресурсы процессора распределяются между потоками, что приводит к замедлению каждой операции. Суммарное время чуть превышает 8 секунд, что ниже теоретического значения (10 секунд). Вероятно, это объясняется тем, что алгоритм распределения процессорного времени операционной системой не сводится к простому делению 100 % доступного времени между потоками конкретного приложения. Когда процессорное время запрашивается несколькими потоками, ОС выделяет им больше процессорного времени, чем однопоточному приложению.

При повторении этого теста с длинной первой операцией становится видно, что медленные запросы практически не влияют на обработку других запросов и в меньшей степени сказываются на среднем быстродействии.

Сравнивая результаты, полученные для однопоточного и многопоточного случая, мы приходим к важному выводу.

Если клиентские запросы имеют примерно одинаковую длину и конкурируют за ограниченные системные ресурсы (например, процессорное время), последовательная обработка обеспечивает лучшее быстродействие по сравнению с обработкой запросов в разных потоках!

Преимущества многопоточности проявляются только при обработке запросов разной длины или при отсутствии нехватки ресурсов. Принимая решение о применении многопоточности, необходимо тщательно проанализировать типы запросов, которые должны обрабатываться вашим приложением. Иногда программа может в процессе выполнения оценить предполагаемую длину запроса и организовать последовательную обработку для коротких запросов и многопоточную обработку для длинных запросов.

Это правило несколько изменяется при отсутствии конкуренции за ограниченные ресурсы (в нашем случае — для операций, не обеспечивающих интенсивной загрузки процессора). Если для синхронных операций результат почти точно совпадает с теоретическим значением, многопоточный результат достигает идеа-

ла. Поскольку потоки практически не расходуют процессорного времени, системе не приходится делить процессор между ними, и ограниченность процессорного времени в данном случае значения не имеет.

```
Non CPU-Intensive operations
Synchronous Equal length operation
1001, 2002, 3004, 5007, Average: 3004
Synchronous one long operation
5007, 6008, 7010, 8011, 9012, Average: 7010
Multithreaded Equal length operations
1001, 1001, 1001, 1001, 1001, Average: 1001
Multithreaded One long operation
5007, 1001, 1001, 1001, 1001, Average: 1802
```

При анализе различий между ситуациями с конкуренцией и без нее (в нашем примере — для операций с интенсивной загрузкой процессора и без) не забывайте о том, что речь идет о двух крайностях. Редко когда клиентские запросы требуют полного привлечения какого-либо ресурса. Но даже если клиентский запрос просто ждет завершения файловой/сетевой операции или операции с базой данных, сама операция использует процессор и другие ресурсы.

Итоги

Хотя многие программисты VB6 хорошо знакомы с многопоточностью, все трудности и опасности параллельной обработки в VB6 были надежно замаскированы. В VB .NET они стали играть очень важную роль.

Из этой главы вы узнали, что проблемы многопоточности обусловлены возможностью совместного доступа к данным со стороны нескольких потоков, а возможность прерывания операций на ассемблерном уровне заставляет предполагать, что прерывание может происходить внутри отдельных команд VB. Даже выполнение простейшей команды типа `A=A+1` может завершиться неудачей, если несколько потоков, использующих общую переменную `A`, попытаются выполнить ее одновременно.

Что еще хуже, вероятность ошибки бывает ничтожно малой. В примерах этой главы ошибки встречались раз-другой на миллионы операций, причем их последствия могли быть весьма разнообразными, от взаимной блокировки до неприметных ошибок в вычислениях. Следовательно, диагностика многопоточных проблем не может опираться на одно лишь тестирование. Вы должны тщательно подходить к проектированию многопоточных приложений и жестко управлять доступом к общим объектам и переменным.

В этой главе описаны некоторые приемы синхронизации потоков и защиты программного кода. Вы также узнали, что в CLR существуют и другие средства синхронизации, не упоминавшиеся в тексте.

Наконец, если вас не устрашили трудности, связанные с проектированием многопоточных приложений, при правильном применении многопоточность способна значительно улучшить быстродействие приложения (как объективное, так и субъективное). Многопоточность способствует снижению общей загрузки системы, поскольку она снимает необходимость в постоянном циклическом опросе, на котором строились некоторые операции ожидания в VB6.

Часть 3

Программы

Бог проявляется в мелочах.
Людвиг Мис ван дер Роэ

Типы данных и операторы

Часть 1 этой книги посвящалась стратегическим вопросам, связанным с платформой .NET: что это такое, какое место она займет в мире Microsoft и в вашем собственном мире. В части 2 рассматривались ключевые концепции языкового уровня, которые необходимо твердо усвоить, чтобы старый стиль программирования не приводил к возникновению серьезных недочетов на стадии проектирования.

Эта часть книги посвящена программам, а точнее, изменениям в самом языке. Основное внимание будет уделяться не изменениям синтаксиса, а их последствиям и влиянию на стиль программирования.

Чтобы лучше представить масштаб изменений, с которыми предстоит столкнуться программистам Visual Basic, мы начнем с рассмотрения фундаментальных типов данных языка.

Числовые типы

В табл. 8.1 приведен список числовых типов данных, поддерживаемых в VB6 и VB .NET. Число в скобках указывает размер данных в байтах. В последнем столбце приводится имя типа переменной в CLR — исполнительной среде, используемой VB .NET и другими языками .NET.

Таблица 8.1. Числовые типы данных, поддерживаемые VB6 и VB .NET

Тип VB6	Тип VB .NET	Тип CLR
Byte(1)	Byte(1)	System.Byte
String*1(1)	Char(2)	System.Char
Boolean(2)	Boolean(4)	System.Boolean
Нет	Decimal(12)	System.Decimal
Currency(8)	Нет	Нет
Double(8)	Double(8)	System.Double
Integer(2)	Short(2)	System.Int16
Long(4)	Integer(4)	System.Int32
Нет	Long(8)	System.Int64
Single(4)	Single(4)	System.Single

Присмотритесь повнимательнее. Если у вас и были сомнения относительно того, действительно ли VB .NET является новым языком, при взгляде на эту таблицу они должны были рассеяться. Модификация фундаментальных числовых типов языка — это действительно серьезное изменение. Конечно, программа преобразования обеспечивает автоматическую адаптацию большинства типов данных, но факт остается фактом: программы, написанные на VB .NET, сильно отличаются от программ на VB6.

String*1 и Char

В VB6 не существует типа `Char`. Почему же символьный тип понадобился в VB .NET? Потому что в VB .NET не существует строк фиксированной длины (эта тема рассматривается далее в этой главе). Конечно, можно использовать динамическую строку и работать с первым символом, но это слишком расточительно, поскольку все строки VB .NET создаются в куче. Конечно, выделение памяти из кучи в VB .NET выполняется очень быстро, но по эффективности эта операция все же уступает выделению памяти в стеке для типа `Char`. Тип `Char` является 16-разрядным, чтобы он мог использоваться для представления символов Unicode. По этой причине я и разместил его среди числовых типов данных, хотя в действительности четко классифицировать его довольно трудно.

Логические величины

Логические величины (`Boolean`) теперь занимают 4 байта вместо двух. Впрочем, этот факт не должен влиять на работу большинства программ.

Помню, на первых порах существования Visual Basic шли жаркие споры относительно природы переменной `Boolean`. Дело в том, что переменная `Boolean` поддерживает всего два значения (`True` и `False`, представленные значениями –1 и 0), но Visual Basic 6 выполняет с переменными типа `Integer` поразрядные операции.

Например, в следующем фрагменте:

```
If 4 And 8 Then
  Debug.Print "Should be true"
Else
```

условие считается ложным, и строка не выводится. Операторы VB6 `And` и `Or` работают с числами не как с логическими величинами, а как с совокупностью разрядов. Применительно к числам они работают правильно лишь в том случае, если `True` всегда представляется –1, а `False` всегда представляется 0. Логические операции между числами часто дают логически непоследовательные результаты. В этом случае числа 4 и 8 (отличные от 0) с позиций логики относятся к `True`, поэтому выражение `4 And 8` вроде бы должно быть равно `True`. Тем не менее результат поразрядной операции `4 And 8` равен 0, то есть `False`.

Это может приводить к ошибкам, особенно при использовании функций API, возвращающих 1 в случае удачного завершения. Например, функция API `IsWindow` возвращает 1 для действительного манипулятора окна. Иначе говоря:

- проверка условия `If IsWindow()` работает правильно, поскольку 1 интерпретируется как `True`;

- проверка условия `If Not IsWindow()` работает неправильно, поскольку результат `Not 1` равен –1, что тоже интерпретируется как `True`!

Программистам VB (особенно работающим с функциями API) приходится проявлять дополнительную осторожность, сравнивать результаты только с 0 и никогда не опираться на фактическое значение сравниваемой величины.

В VB .NET используется объект `System.Boolean`, принимающий только два значения: `True` и `False`. В версии бета-1 компания Microsoft героически попыталась внести толику здравого смысла в традиционные правила работы с логическими переменными в Visual Basic и переопределила операторы `And` и `Or` как логические вместо поразрядных. Кроме того, для выполнения поразрядных операций были определены новые операторы `BitAnd` и `BitOr`. При таком разграничении логические операции `Not`, `And` и `Or` всегда работают только с логическими типами (или переменными, преобразованными к логическому типу), что устраняет все возможные неоднозначности. Если вы хотите выполнить поразрядную операцию, используйте `BitNot`, `BitAnd` и `BitOr`.

К сожалению, некоторым недальновидным программистам VB6 удалось убедить Microsoft, что это изменение каким-то образом нарушает те принципы, благодаря которым Visual Basic завоевал такую популярность, и Microsoft по столь же непостижимым причинам уступила.

В результате в VB .NET `True` по-прежнему определяется как –1, а логические и поразрядные операции выполняются одной командой.

В этом нетрудно убедиться при помощи проекта Boolean (листинг 8.1).

Листинг 8.1. Проект Boolean[1]

```
' Демонстрация работы с логическими переменными
' Copyright ©2001 by Desaware Inc.
' All Rights Reserved.
Module Module1

Sub Main()
  Dim A As Boolean
  Dim I As Integer
  A = True
  I = CInt(A)
  Console.WriteLine ("Value of Boolean in Integer is: " + I.ToString())
  Console.WriteLine ("Value of a System Boolean in Integer is: " +
I.ToString())

  I = 5
  A = CBool(I)
  Console.WriteLine ("Value of Boolean assigned from 5 is : " + A.ToString())
  I = CInt(A)
  Console.WriteLine ("And converted back to Integer: " + I.ToString())
  Console.WriteLine("5 And 8: " + (CBool(5) And CBool(8)).ToString)
  Console.WriteLine("5 And 8: " + (5 And 8).ToString())
  Console.ReadLine()
End Sub

End Module
```

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Результат выглядит следующим образом:

```
Value of Boolean in Integer is: 1
Value of Boolean assigned from 5 is : True
And converted back to Boolean: 1
4 And 8: True
4 BitAnd 8: 0
```

Логические переменные могут принимать два значения: `True` и `False`. Пока логические операции выполняются с логическими переменными, все замечательно. Но помните: когда VB .NET встречает числовую переменную вместо логической, он выполняет с ней поразрядную операцию, что может приводить к возникновению непредвиденных ошибок.

Currency и Decimal

На смену `Currency` пришел тип `Decimal`. Он обеспечивает большую точность (12 байтов вместо 8), а позиция десятичной запятой не фиксируется. Вряд ли это изменение окажет сколько-нибудь заметное влияние на ваши программы. Основное преимущество такого изменения заключается в том, что `Decimal` принадлежит к числу «родных» типов .NET и поэтому поддерживается любыми компонентами во всех языках .NET.

Integer, Long и Short

Расширение числовых типов (с 16 до 32 разрядов для типа `Integer`, с 32 до 64 разрядов для типа `Long`) — одно из нововведений VB .NET, вызывающих неоднозначную реакцию. В VB тип `Integer` всегда представлялся 16-разрядной величиной, и это было вполне нормально, пока программа ограничивалась VB и не пользовалась внешним сервисом. Однако в 32-разрядном мире C++ (а следовательно, и во всей документации Microsoft) целочисленный тип представляется 32 битами, поэтому программистам VB постоянно приходилось помнить о том, что «короткий» тип API соответствует типу VB `Integer`, а «целый» тип API соответствует типу VB `Long`.

При проектировании VB .NET Microsoft оказалась перед выбором. Можно было сделать тип `Integer` 32-разрядным и обеспечить единую интерпретацию термина «`Integer`» во всех языках и документации .NET. С другой стороны, можно было ограничить тип VB `Integer` 16 битами и ввести новый тип `Int64`.

Изменение определения `Integer` приводит документацию VB в соответствие с остальной документацией .NET. С другой стороны, это несколько затрудняет работу программистов, привыкших программировать на разных языках, поскольку им приходится постоянно мысленно переключаться из VB6 в VB .NET и обратно.

Лично я одобряю это изменение. Думаю, в конечном счете согласованность типов перевесит все мелкие неудобства. Впрочем, моя точка зрения субъективна: для человека, которому приходится постоянно работать с Win32 API, согласованность типов важнее сохранения старых привычек программирования. Но я готов признать, что это решение было неоднозначным, и против него можно привести обоснованные возражения.

При программировании в VB .NET следует помнить, что в 32-разрядных операционных системах тип `Integer` является самым эффективным числовым типом данных. Старайтесь избегать типа `Long`, если только вам действительно не требуется 64-разрядное представление числа.

Также приготовьтесь к тому, что в какой-нибудь предстоящей 64-разрядной версии CLR тип `Integer` будет переопределен как 64-разрядный. Вероятно, к этому моменту Microsoft определит специальный 32-разрядный тип данных для работы с 32-разрядными переменными.

Беззнаковые типы

Беззнаковые переменные в VB .NET не поддерживаются. Впрочем, не спешите огорчаться. Учтите, что беззнаковые типы, определенные в CLR, не соответствуют спецификации CLS. Другими словами, если вы используете беззнаковую переменную в открытом методе, свойстве или структуре данных сборки, нет гарантий, что другой CLS-совместимый язык сможет воспользоваться этой сборкой.

Следовательно, для большинства программистов беззнаковые переменные (если бы они существовали в VB .NET) могли бы использоваться лишь в границах отдельной сборки.

В VB6 реальная потребность в беззнаковых типах данных возникала только при работе с функциями Win32 API. Как вы вскоре убедитесь, в VB .NET эта потребность практически исчезла, поскольку программы в основном ориентируются на работу с классами .NET Framework, нежели на прямой вызов функций API. При использовании функций API пространство имен `PInvoke` обеспечивает автоматическое преобразование знаковых переменных (см. главу 15).

Итак, некоторым программистам будет не хватать беззнаковых типов данных. Возможно, для кого-то это даже станет аргументом в пользу C# перед VB .NET, но это слишком слабый аргумент.

Типы CLR

В третьем столбце табл. 8.1 приведены имена типов данных, используемые CLR. Я привел их для того, чтобы подчеркнуть одну из важных особенностей .NET, особо выделяемых Microsoft, — совместимость языков программирования.

К сожалению, в изложении Microsoft это выглядит довольно глупо. Особое внимание почему-то уделяется удобству построения приложений с применением разных .NET-языков. Но поскольку три основных языка .NET (C++, C# и VB .NET) обладают примерно одинаковыми возможностями, скорее всего, большинство программистов выберет самый привычный язык и будет пользоваться им.

На совместимость языков можно взглянуть и с другой, более правильной точки зрения. Совместная работа компонентов, написанных на разных языках, играет очень важную роль в .NET и является огромным шагом вперед по сравнению с библиотеками типов COM или ручным преобразованием, выполняемым при использовании команд `Declare` при вызове функций API.

Как видно из табл. 8.1, каждому типу данных VB .NET соответствует определенный тип данных CLR, что позволяет использовать его в других языках .NET, соответствующих спецификации CLS.

Другие типы данных

Изменения, внесенные Microsoft, не ограничиваются числовыми типами данных.

Прощай, Variant (наконец-то!)

Я никогда не любил универсальный тип `Variant`. Не стану углубляться в объяснение причин, так как эта тема рассматривается в одной из предыдущих книг, посвященных разработке компонентов COM/ActiveX. Ограничусь кратким перечнем.

- `Variant` медленно работает.
- Если вы хотите, чтобы ваша программа была устойчивой к вводу разнотипных величин, всегда следует помнить, что применение `Variant` требует проверки типа во время выполнения программы.
- При работе с `Variant` выполняются замаскированные преобразования, которые не всегда работают так, как предполагалось (злостное искажение типов).
- Ошибки, возникающие при использовании `Variant`, с большей вероятностью возникают на стадии выполнения программы (например, «пустые» переменные, ошибки преобразования или переполнение).
- `Variant` медленно работает (на тот случай, если вы не обратили внимания на первый пункт).

В Visual Basic 6 универсальный тип данных реально использовался только там, где это было неизбежно, например при работе с базами данных или объектными моделями, в которых этот тип был задействован, а также при ограниченной перегрузке параметров (что позволяло передавать одной функции параметры нескольких типов[1]).

Универсальный тип и объекты

Некоторые программисты склонны рассматривать тип `Object` как своего рода замену универсального типа, поскольку он позволяет ссылаться на произвольные данные. Даже в документации Microsoft упоминается, что тип `Object` заменяет `Variant`. На самом деле это неточно.

Переменная универсального типа содержит ссылку на блок памяти с произвольными данными. Сам блок памяти содержит поле, определяющее тип хранимых данных. Подсистема OLE поддерживает функции для очистки, присваивания и преобразования значений универсальных типов.

На рис. 8.1 показано, что происходит при объявлении переменной типа `Variant` с последующим присваиванием ей числового и строкового значения.

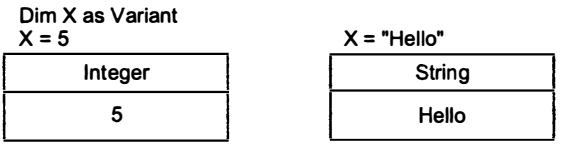


Рис. 8.1. Изменение содержимого переменной типа Variant при присваивании

[1] В языке VB .NET предусмотрена прямая поддержка этой возможности.

Как видите, в результате присваивания содержимое переменной `Variant` изменяется, а внутренний признак типа приводится в соответствие с новым типом данных.

На рис. 8.2 изображен совершенно другой процесс, сопровождающий присваивание объектной переменной в VB .NET.

Рис. 8.2. Присваивание объектной переменной изменяет ссылку

При первоначальном присваивании `x=5` целочисленное значение упаковывается в объект, содержащий величину 5. Переменная `x` содержит указатель (ссылку) на упакованное целое число. При следующем присваивании строковой величины `x` присваивается ссылка на объект `String`, созданный в куче. Что происходит с объектом `Integer`? Ссылки на него утрачены, и этот объект уничтожается при следующей сборке мусора.

Возможность ссылаться на любые типы переменных из типа `Object` объясняется тем фактом, что все типы данных .NET Framework являются производными от типа `Object` и не имеют никакого отношения к универсальному типу `Variant`.

Проверка типа объекта

Хотя тип `Object` и не является универсальным, он позволяет создавать общие переменные, ссылающиеся на объекты разных типов. По аналогии с оператором `VarType`, позволяющим определить тип переменной, можно определить тип объекта, на который ссылается переменная типа `Object` (и вообще любая переменная). При этом используется механизм получения общей информации об объектах, их методах и свойствах, а также других метаданных сборки. Этот механизм называется *рефлексией* (reflection). Рефлексия подробно рассматривается в главе 11.

Получение информации о типе произвольного объекта основано на использовании метода `GetType`, класса `System.Object` (базовый класс любого объекта .NET). Данная возможность продемонстрирована в простой программе DataTypes.

```
Module Module1
  Sub Main()
    Dim o As Object
    o = 5
    console.WriteLine (o)
    console.WriteLine (o.GetType.FullName)
    o = "Hello"
    console.WriteLine (o)
    console.WriteLine (o.GetType.FullName)
    console.ReadLine()
  End Sub
End Module
```

Программа выводит следующий результат:

```
5
System.Int32
Hello
System.String
```

Объект `Type`, возвращаемый методом `GetType`, содержит ряд свойств, предназначенных для получения информации об объекте или классе. При помощи свойства `FullName` можно получить полное имя объекта. Метод `GetTypeCode` возвращает числовой код типа, с которым удобнее и эффективнее работать в программе.

Прочее

Если при умножении двух объектов `Integer` (32-разрядных) происходит переполнение, то результатом будет величина типа `Long` (64-разрядная).

Строки

В области работы со строками в VB .NET наблюдаются два крупных изменения. Первое, которое с большей вероятностью отразится на работе существующих программ, заключается в том, что VB .NET не поддерживает строк фиксированной длины. Строки фиксированной длины VB6 хорошо подходили для работы с записями фиксированного размера, а память для них выделялась из стека, что было дополнительным преимуществом. В VB .NET все строки являются динамическими, а память для них выделяется из кучи.

Второе изменение, весьма тонкое и с меньшей вероятностью влияющее на работу существующих программ, заключается в том, что строки VB .NET являются *неизменными*. Иначе говоря, любые функции и операторы, которые на первый взгляд модифицируют строку, в действительности создают и возвращают новый экземпляр строки. Благодаря этому CLR не приходится беспокоиться о переполнении строковых буферов или перемещении строк в куче в результате модификации. Кроме того, такое решение повышает эффективность работы со строковыми литералами за счет их совместного использования. Но самое важное последствие неизменности строк заключается в том, что это делает работу со строками относительно безопасной в многопоточных условиях. Конечно, вы по-прежнему можете столкнуться с проблемами при многопоточном доступе к общим строковым переменным, но по крайней мере вам не придется беспокоиться об одновременной модификации содержимого строки несколькими потоками.

Но самый важный аргумент в пользу неизменности строк, о котором многие и не задумываются, связан с особенностями передачи объектов в качестве параметров. Мы еще вернемся к этой теме в настоящей главе.

Для удобного и эффективного конструирования строк можно воспользоваться классом `System.Text.StringBuilder`. Тем не менее программисты Visual Basic, вероятно, предпочтут чрезвычайно гибкую команду `Mid`, с которой еще проще работать.

Пример Strings (листинг 8.2) показывает, как организовано совместное использование строковых литералов в VB .NET.

Листинг 8.2. Пример Strings

```
Module Module1

  Sub Main()
    Dim A As String
    Dim B As String
    A = "Hello"
    B = "Hel" + "lo"
    Console.WriteLine (A = B)
    Console.WriteLine (A Is B)

    Mid(A, 1, 1) = "X"
    Mid(B, 1, 1) = "X"
    Console.WriteLine (A = B)
    Console.WriteLine (A Is B)
    Console.WriteLine (A)
    Console.ReadLine()
  End Sub

End Module
```

Результат выглядит так:

```
True
True
True
False
Xello
```

Строка `Hello` присваивается переменным `A` и `B` на стадии компиляции (у компилятора хватает сообразительности понять, что `B` присваивается уже существующий строковый литерал, и использовать ту же ссылку). Переменные `A` и `B` не только имеют одинаковые значения, но и ссылаются на один и тот же объект (как видно из проверки условия `A Is B`). Если бы строки могли модифицироваться, подобное совмещение литералов в VB .NET оказалось бы невозможным, поскольку изменение одной переменной отражалось бы на другой переменной.

После модификации первого символа в обеих строках переменные остаются равными, но уже не ссылаются на один и тот же объект.

В VB .NET была исключена функция `String$`, поскольку объект `String` содержит ряд удобных конструкторов. Например, для создания строки длины N, состоящей из одних нуль-символов, можно воспользоваться следующей командой:

```
s = New String(Chr(0), N)
```

Массивы

В области работы с массивами в VB .NET наблюдаются два принципиальных изменения. Первое чрезвычайно важно, а второе для большинства программистов относительно несущественно.

Индексация массивов теперь начинается с нуля. Это означает, что строка

```
Dim X(10) As Integer
```

всегда создает массив из 11 элементов с индексами от 0 до 10. Она эквивалентна следующему объявлению VB6:

```
Dim X(0 To 10) As Integer
```

Вероятно, это огорчит многих программистов VB, которым назначение начального индекса массива кажется удобным. Программисты C# и C++ к подобной схеме индексации уже успели привыкнуть. С точки зрения программиста это изменение полезно тем, что оно обеспечивает единый стиль работы с массивами. Смена базовых индексов в приложении чревата ошибками, поскольку программисту приходится постоянно помнить о том, какой индекс используется в каждом конкретном случае, и переключаться между ними. Применение постоянного базового индекса упрощает чтение программы и помогает разобраться в чужом коде. Вероятно, многие программисты VB .NET без труда привыкнут к нулевой индексации или будут использовать элементы, начиная с первого, попросту игнорируя нулевой элемент.

Обычно я с энтузиазмом отношусь ко всем аспектам языка, защищающим программиста от ошибок, но, честно говоря, мне кажется, что на этот раз Microsoft просчиталась. Сначала я подумал, что это изменение как-то связано со спецификой .NET Framework, однако после знакомства с классом `Array` выяснилось, что он поддерживает смену базового индекса. В этом нетрудно убедиться при помощи примера, приведенного в листинге 8.3.

Листинг 8.3. Приложение ArrayExample

```
Module Module1

  Sub Main()
    Dim X As Array
    Dim I As Integer
    Dim Lengths(0) As Integer
    Dim LowerBounds(0) As Integer

    Lengths(0) = 10
    LowerBounds(0) = 1

    X = Array.CreateInstance(GetType(System.Int32), _
      Lengths, LowerBounds)

    Try
      X.SetValue(5, 0)
    Catch e As Exception
      Console.WriteLine (e.Message)
    End Try

    X.SetValue(6, 1)
    Console.WriteLine (X.GetValue(1))

    Console.ReadLine()


  End Sub

End Module
```

Все массивы .NET, в том числе и массивы VB, основаны на типе `Array`. Как в этом убедиться? Потому что для массива, объявленного в Visual Basic, можно вызывать все методы объекта `Array`. В листинге 8.3 мы объявляем переменную-массив `X` и создаем экземпляр целочисленного массива длины 10 с базовым индексом,

равным 1. После этого программа пытается присвоить значения двум элементам массива с индексами 0 (этот элемент не существует) и 1. Результат выглядит так:

```
An exception of type System.IndexOutOfRangeException was thrown.
6
```

При обращении к элементу с индексом 0 инициируется исключение `IndexOutOfRangeException`. С индексом 1 все нормально.

Приведенный код выглядит неуклюже, но он убедительно доказывает, что смена базового индекса поддерживается на уровне .NET Framework.

Почему же эта возможность была исключена из VB .NET?

Хотя объект .NET Framework `Array` поддерживает изменение базового индекса, спецификация CLS требует, чтобы индексация массивов всегда начиналась с нуля. В CLS определяется минимальный набор требований, который обеспечивает возможность свободного использования сборки в любом другом CLS-совместимом языке.

Microsoft оказалась перед выбором. С одной стороны, можно было включить изменение базового индекса в CLS и оставить эту возможность в VB .NET; с другой стороны, можно было сохранить CLS в прежнем виде и каким-то образом решить проблему с созданием CLS-несовместимых сборок в VB .NET.

Что ж, решение Microsoft вполне очевидно. Лично я предпочел бы добавить эту возможность в C++ и C#, нежели исключать ее из VB. Конечно, принятое решение, каким бы болезненным оно ни было, прежде всего направлено на обеспечение CLS-совместимости для программистов VB .NET, особенно широко использующих компоненты в процессе программирования.

Все массивы VB .NET могут переобъявляться с изменением размера. Вероятно, для большинства программистов VB это изменение несущественно, однако оно сказывается на работе с функциями API (см. главу 15).

Дата

При работе с датой и временем в VB .NET используется тип объекта `System.DateTime`. Для программистов VB самое существенное последствие заключается в том, что теперь не существует простого преобразования даты в `Double` и обратно. Впрочем, класс `DateTime` содержит методы для преобразования данных в формат OLE Automation. К счастью, объект `DateTime` обладает очень богатыми возможностями, включая средства для сравнения даты/времени, а также прибавления и вычитания временных интервалов.

Перечисляемые типы

Перечисления почти не изменились со времен VB6 — во всяком случае, с точки зрения синтаксиса. Главное изменение заключается в том, что `Enum` теперь интерпретируется как любой другой объявленный тип. Иначе говоря, перед ключевым словом `Enum` теперь могут находиться указатели класса доступа `Public`, `Private`, `Protected` и `Friend`, а также ключевое слово `Shadows`, управляющее видимостью `Enum` в производных классах и других сборках.

В VB .NET базовый класс Enum содержит разнообразные методы для прямых и обратных преобразований между непосредственным значением и его строковым (именованным) представлением.

С точки зрения программирования единственным реальным изменением является поддержка атрибута Flags, использование которого продемонстрировано в программе EnumExample (листинг 8.4).

Листинг 8.4. Приложение EnumExample

```
Module Module1

  Enum E
    A = 5
    B
    C = 6 ' Обе переменные, B и C, равны 6
  End Enum

  <Flags()> Enum B
    A = &H1
    B = &H2
    C = &H4
  End Enum

  Enum C
    A = &H1
    B = &H2
    C = &H4
  End Enum

  Sub F(ByVal X As E)

  End Sub

  Sub FB(ByVal X As B)
  End Sub

  Sub Main()
    Dim I1, I2, I3 As Integer
    I1 = E.A: I2 = E.B: I3 = E.C
      ' Метод ToString нельзя использовать непосредственно
      ' для переменной перечисляемого типа, поскольку
      ' он вернет имя перечисления.
    Console.WriteLine (I1.ToString() + I2.ToString() + I3.ToString())
    F (E.C)
    FB (B.A Or B.C Or B.B)
    Console.WriteLine (E.Format(GetType(E), 5, "G"))
    Console.WriteLine (C.Format(GetType(C), (C.A Or C.C), "G"))
    Console.WriteLine (B.Format(GetType(B), (B.A Or B.C), "G"))
    Console.ReadLine()
  End Sub

End Module
```

Программа доказывает, что перечисляемый тип может содержать несколько элементов с одинаковым значением (в нашем примере E.B и E.C равны 6). Метод Format типа Enum возвращает имя заданной величины в перечислении — эта информация часто используется в процессе отладки или при построении отчетов.

```
566
A
```

Перечисляемые типы обычно используются в двух ситуациях: при логической группировке констант, следующих в некоторой последовательности, и при создании списков флагов — констант с одним установленным битом, объединяемых оператором `Or`. Эти два типичных применения `Enum` продемонстрированы выше в функциях `F` и `FB`.

Атрибут `Flags` типа `Enum` сообщает исполнительной среде о том, что параметры `Enum` объединяются при помощи поразрядных логических операций. Перед тем как возвращать результат[1], метод `Format` читает значение этого атрибута, используя механизм рефлексии. Таким образом, для переменной `C` возвращается значение величины, но для переменной `B` с установленным атрибутом `Flags` будет возвращена строка `A|C` — поразрядная дизъюнкция двух переменных в синтаксисе C++/C#. Следовательно, результат будет выглядеть так:

```
5
A,C[2]
```

Объявления

С объявлениями переменных в VB .NET все просто и мило: язык позволяет объявлять несколько переменных в одной строке (см. приложение Declarations). Впрочем, на самом деле это несущественно.

VB .NET позволяет инициализировать переменные и массивы при объявлении, как показано ниже для переменных `C` и `D`. Все выглядит вполне логично:

```
Module Module1

  Sub Main()
    Dim A, B As Integer
    Dim C As Integer = 6
    Dim D() As String = {"A", "B", "C", "D", "E"}
    Dim E As Integer, F As String
  End Sub

End Module
```

Списки инициализации массивов заключаются в фигурные скобки, которые впервые вошли в синтаксис Visual Basic.

Объявления `DefType` (`DefInt`, `DefLong` и т. д.) теперь не поддерживаются. Впрочем, это небольшая потеря, поскольку они всего лишь упрощали чтение программ и обеспечивали единый стиль выбора имен в приложениях VB.

Преобразования и проверка типа

Из-за обилия рекламной шумихи, сопровождающей VB .NET, можно упустить одно из самых важных и интересных нововведений. То, что оно одновременно вызывает массу хлопот и раздражения, нисколько не... Впрочем, начнем с самого начала.

---

[1] В качестве параметров оператору Format передается тип перечисления, форматируемая величина и код формата (код G означает, что метод должен возвращать имя величины в перечисляемом типе, если это возможно).

[2] Согласно документации должна возвращаться строка «A|C», но версия бета-2 возвращает «A,C». Что будет в окончательной версии? Будущее покажет.

В Visual Basic 6 поддерживается так называемое «злостное искажение типов» (evil type coercion). Ниже приведен классический пример.

```
Sub Evil()
  Debug.Print 15 + "15" + "15"
End Sub

Sub Main()
  Evil
End Sub
```

При выполнении этого фрагмента в окне отладки выводится число 45.

Встретив выражение со смешанными типами, VB6 пытается угадать, какие же преобразования вы подразумевали. В простых и очевидных случаях (например, при преобразовании `Long` в `Integer`) он действует вполне логично.

К сожалению, иногда VB6 угадывает неправильно и не предупреждает вас о том, что преобразование может привести к ошибкам при выполнении программы.

Некоторые считают, что это не такая уж серьезная проблема. Обычно VB6 угадывает правильно, а опытный программист способен заранее предвидеть самые очевидные проблемы. Автоматическое преобразование типов подавляет выдачу предупреждений, большинство из которых все равно игнорируется — ведь программист знает, что он делает.

Недостаток подобной аргументации заключается в том, что она не учитывает реального распределения затрат на протяжении жизненного цикла программы. Как бы нас ни раздражали предупреждения, сопровождающие самые очевидные преобразования, их ликвидация обходится дешево — программист искореняет их при первой компиляции программы. Возможно, на это потребуется несколько лишних минут, но этим все заканчивается.

Если ошибка пережила стадию начальной разработки, затраты на ее исправление заметно возрастают. Они складываются из затрат на отслеживание ошибки, на взаимодействие между разработчиками, пытающимися найти ошибку и поручить ее исправление соответствующему работнику, на проверку всех исправлений и регрессионное тестирование всей программы, защищающее от случайного внесения новых ошибок.

Но если ошибка пережила все предварительные стадии и добралась до конечного продукта, затраты достигают астрономических величин.

В VB .NET появился новый флаг жесткой проверки типов; к сожалению, он не устанавливается по умолчанию в новых проектах VB .NET. Привыкните к тому, чтобы работа над новым проектом всегда начиналась с установки этого флажка[1].

Вы быстро привыкнете пользоваться функциями преобразования там, где это нужно, и убедитесь, что предупреждения компилятора только повышают качество ваших программ. Они привлекают ваше внимание к использованию смешанных типов и напоминают, с каким же объектом или интерфейсом вы работаете. Вы научитесь находить ошибки до того, как их увидит кто-нибудь другой.

Я твердо в этом убежден, поэтому флажок Strict Type Checking устанавливается во всех примерах этой книги. Более того, при обсуждении перехода на VB .NET

[1] В версии бета-1 флажок жесткой проверки типов устанавливался по умолчанию. К сожалению, в версии бета-2 Microsoft решила, что по умолчанию этот флажок следует сбрасывать. В обычной ситуации я бы высказался покрепче, но я так рад, что этот флажок вообще существует, что воздержусь от сильных выражений... Хотя это, конечно, чудовищная ошибка.

я даже не рассматриваю возможность его сброса. Единственный случай, когда об этом стоит подумать хотя бы теоретически — при непосредственной адаптации кода мастером Upgrade Wizard. Во всем новом коде и в большинстве адаптированных программ этот флажок следует устанавливать.

Правила жесткой проверки типов весьма просты. При каждом преобразовании переменной от одного типа к другому компилятор проверяет, может ли это привести к потере данных и возникновению ошибок во время выполнения программы. Для примера рассмотрим листинг 8.5.

Листинг 8.5. Приложение Conversions

```
Module Module1

  Sub Main()
    Dim I As Integer, L As Long
    I = 50
    L = I
    Console.WriteLine (L)
    L = 50
    I = CInt(L) ' Необходимо явное преобразование.
    Console.WriteLine (I)
    L = &H100000000
    Try
      I = CType(L, Integer)
    Catch E As Exception
      Console.WriteLine (E.Message)
    End Try

    Console.WriteLine (I)
    Console.ReadLine()

  End Sub

End Module
```

Тип `Integer` может быть неявно преобразован к типу `Long`, так как последний способен вместить все допустимые значения типа `Integer`. Однако обратное неверно. При выполнении этой программы будет выведен следующий результат:

```
50
50
An exception of Type System.OverflowException was thrown.
50
```

В листинге 8.5 продемонстрированы два способа преобразования `Integer` в `Long`: функции `CInt` и `CType`. Эти функции сообщают компилятору, что вы *действительно* знаете, что делаете, присваивая значение `Long` переменной типа `Integer`. Без них на стадии компиляции возникли бы ошибки. В нашем примере функции `CInt` и `CType` делают одно и то же, просто функция `CType` является более универсальной.

Преобразования и классы

Как было показано в главе 5, функция `CType` преобразует объекты от одного типа к другому. Правила явных и неявных преобразований объектов аналогичны правилам преобразования для переменных структурных типов.

Производный класс может быть неявно преобразован к базовому классу. Это вполне логично, поскольку производный класс является «частным случаем» базового класса и содержит все его свойства и методы.

Также допускается неявное преобразование класса к интерфейсу, реализованному этим классом.

Преобразование от базового класса к производному должно быть явным. Поскольку нет гарантии, что полученный объект действительно относится к производному классу, неудачное преобразование может привести к ошибкам во время выполнения программы.

Преобразование от интерфейса к классу тоже должно быть явным. Успех этого преобразования не гарантирован.

Программа ObjectConversions (листинг 8.6) демонстрирует правила преобразования объектов. В ней определяется интерфейс `I`, класс `A` (реализующий интерфейс `I`) и класс `B`, производный от `A`. В комментариях указаны правила, действующие в каждом конкретном случае.

Листинг 8.6. Приложение ObjectConversions

```
Module Module1
  Interface I
    Sub MyInterfaceFunc()
  End Interface

  Class A
    Implements I
    Protected Overridable Sub MyFunc()

    End Sub
    Public Sub MyPublicFunc()

    End Sub
    Public Sub MyInterfaceFunc() Implements I.MyInterfaceFunc

    End Sub
  End Class

  Class B
    Inherits A

  End Class

  Sub Main()
    Dim myA As New A()
    Dim myB As New B()
    Dim myAReference As A
    Dim myBReference As B
    Dim myIReference As I

    myAReference = myA ' Same type - ok
    myAReference = myB ' Базовый тип - неявное преобразование
    myIReference = myA ' Реализованный интерфейс - неявное

    myA = CType(myIReference, A) ' Ссылка может относиться
             ' к любому объекту - явное преобразование.
             ' На стадии компиляции нельзя определить,
             ' будет ли работать эта команда
```

продолжение ↓

Листинг 8.6 (продолжение)

```
    Try
      myB = CType(myIReference, B) ' Не работает
    Catch e As Exception
      Console.WriteLine (e.Message)
    End Try

    myIReference = myB ' Реализованный интерфейс - неявное
    ' Класс B, производный от A, наследует интерфейс I.

    Console.ReadLine()

  End Sub

End Module
```

Преобразования и структуры

На момент написания книги VB .NET не поддерживал нестандартных преобразований для структур, определяемых в программе. Например, если вы создаете сложный тип данных и хотите, чтобы ему можно было непосредственно присваивать значения типа `Integer` или `Long` (явно или косвенно), сделать это вам не удастся. Вместо этого следует определить `Shared`-функцию с именем `FromXxx` (исходный тип).

Пример:

```
Shared Function FromLong(ByVal l As Long) As YourStructureType
```

Операторы

Многие изменения в операторах не требуют подробных объяснений и достаточно хорошо описываются в разделе «Language Changes» документации Visual Studio .NET.

Операторы AndAlso и OrElse

Как упоминалось выше в этой главе, в VB .NET операторы `And` и `Or` работают так же, как они работали в VB6. В дополнение к ним Microsoft определила два новых оператора, работающих только с логическими величинами: `AndAlso` и `OrElse`. Использование этих операторов продемонстрировано в листинге 8.7.

Листинг 8.7. Операторы AndAlso и OrElse

```
' Комбинированные логические операторы
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Module Module1

  Class A
    Public ReadOnly Property IsTrue() As Boolean
      Get
        Console.WriteLine ("IsTrue was called")
        Return True
      End Get
    End Property
```

```
    Public ReadOnly Property IsFalse() As Boolean
      Get
        Console.WriteLine ("IsFalse was called")
        Return False
      End Get
    End Property

  End Class

  Sub Main()
    Dim testvar As New A()
    Console.WriteLine ("Before IsFalse And IsFalse")
    If testvar.IsFalse And testvar.IsFalse Then
    End If
    Console.WriteLine ("Before IsFalse AndAlso IsFalse")
    If testvar.IsFalse AndAlso testvar.IsFalse Then
    End If

    Console.WriteLine ("Before IsTrue Or IsTrue")
    If testvar.IsTrue Or testvar.IsTrue Then
    End If
    Console.WriteLine ("Before IsTrue OrElse IsTrue")
    If testvar.IsTrue OrElse testvar.IsTrue Then
    End If
    Console.ReadLine()
  End Sub

End Module
```

Программа выводит следующий результат:

```
Before IsFalse And IsFalse
IsFalse was called
IsFalse was called
Before IsFalse AndAlso IsFalse
IsFalse was called
Before IsTrue Or IsTrue
IsTrue was called
IsTrue was called
Before IsTrue OrElse IsTrue
IsTrue was called
```

Как видите, операторы `AndAlso` и `OrElse` оптимизируют проверку логических условий. Если одно условие в связке `And` равно `False`, проверять остальные условия бессмысленно, поскольку результат заведомо равен `False`. Иногда эти операторы заметно улучшают быстродействие программы, особенно если в условиях используются свойства объектов, получение которых занимает относительно много времени.

Строковые операторы

Ранее в этой главе упоминалось о том, что в VB6 иногда выполнял автоматические преобразования типов, приводящие к непредвиденным результатам. Например, выражение `15 + "15"` могло оказаться равным 30. Строгая проверка типов предотвращает автоматические преобразования такого рода, отдельный оператор конкатенации становится ненужным, поэтому программисты могут вернуться

к более наглядному оператору «+». Ниже приведен фрагмент приложения Operators.

```
' Console.WriteLine(15 + "15" + "15") ' Ошибка компиляции
Console.WriteLine("Evil Typing")
Console.WriteLine(15.ToString + "15" + "15")
```

В результате выводится строка 151515.

Конечно, вы можете продолжать пользоваться оператором конкатенации &. При работе со строками следует помнить о некоторых тонкостях, связанных с подсчетом символов (хотя эта проблема не имеет прямого отношения к операторам, она нередко вызывает недоразумения).

Во встроенных командах VB .NET отсчет производится привычным способом, от 1 до длины строки. С другой стороны, класс .NET `String` (на котором основаны переменные `String`) считает символы в строках, начиная с нуля. Так, фрагмент приложения Operators

```
Dim A As String = "ABCD"
Console.WriteLine ("String Characters")
Console.WriteLine (InStr(A, "C"))
Console.WriteLine (A.IndexOf("C"))
```

выводит следующий результат:

```
String Characters
3
2
```

Комбинированные операторы

В VB .NET поддерживаются новые комбинированные операторы, хорошо знакомые программистам C++. Комбинированный оператор выполняет с переменной некоторую операцию и присваивает результат исходной переменной. Например, команда `A = A + B` заменяется командой `A += B`.

В листинге 8.8 приведена функция Concatonators из приложения Operators.

Листинг 8.8. Комбинированные операторы

```
Sub Concatonators()
  Dim S As String
  Dim A As Integer
  S = "Hello"
  S += " Everybody"
  A = 5
  A += 10
  Console.WriteLine ("Concatonators")
  Console.WriteLine (S)
  Console.WriteLine (A)
End Sub
```

Выводится следующий результат:

```
Concatonators
Hello Everybody
15
```

Новые операторы перечислены в табл. 8.2.

Таблица 8.2. Комбинированные операторы в VB .NET

A &= B	A = A & B (конкатенация строк)
A *= B	A = A * B
A += B	A = A + B
A /= B	A = A / B
A -= B	A = A – B
A \= B	A = A \ B
A ^= B	A = A ^ B

Комбинированные операторы делают программу более наглядной и лаконичной, поэтому я рекомендую пользоваться ими.

Eqv и Imp

Оператор VB6 `Eqv` заменен оператором =, который работает точно так же. Оператор VB6 `Imp` заменяется следующей конструкцией:

```
(Not A) Or B ' В логических операциях
```

Итоги

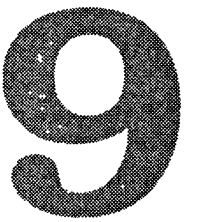
В этой главе было показано, что переход на архитектуру .NET потребовал значительных изменений в фундаментальных типах данных VB .NET и принципах их работы. Чтобы VB .NET был CLS-совместимым языком, его типы данных должны быть основаны на типах данных .NET.

Важнейшим изменением для большинства разработчиков VB станет появление жесткой проверки типов. Поведение VB .NET при отключенной проверке типов в этой главе не рассматривается — тот, кто сознательно отключает ее, слишком глуп для чтения этой книги[1]. Не стоит и говорить о том, что флажок Option Explicit тоже должен быть установлен.

При внесении столь фундаментальных изменений проектировщики Microsoft воспользовались случаем и подправили синтаксис языка. В частности, в новой версии поддерживается инициализация при объявлении — огромное усовершенствование, особенно при инициализации массивов, столь неудобной в VB6. Среди приятных нововведений стоит отметить и комбинированные операторы, хорошо знакомые программистам C++ и C#.

[1] Наверное, я выразился слишком сильно (а может быть даже оскорбительно, чего я стараюсь избегать) — но, черт возьми, жесткая проверка типов *действительно* важна! С позиции общих затрат на разработку программного продукта она легко выходит на первое место среди всех новшеств VB .NET. Итак, если в вашей программе проверка типов отключена, поскорее включите ее и исправьте ошибки (о которых вы наверняка и не подозревали)!

Синтаксис



От изменений в базовых типах данных и операторах Visual Basic мы переходим к изменениям в самом языке. В этой главе рассматриваются изменения, *не связанные* с объектами и объектно-ориентированным программированием (ибо последние заслуживают отдельной главы).

Не стоит рассматривать эту главу как подробный справочник по всем изменениям в языке — эта информация приведена в электронной документации Visual Studio .NET. Прежде чем читать дальше, просмотрите раздел «What's New in Visual Basic». Как всегда, я лишь описываю эти изменения в практическом контексте и рассматриваю их возможное влияние на программный код.

Вызовы функций и параметры

В Visual Basic .NET в области вызова функций произошли определенные изменения. Одни сразу бросаются в глаза, другие менее заметны, но все равно важны.

Рациональный механизм вызова

В VB6 существует четкое разделение между функциями и процедурами (subroutines). Функция всегда возвращает некоторое значение или 0, если возвращаемое значение не указано. Процедуры никогда не возвращают значений. Синтаксис вызова функций и процедур выглядит так:

```
' Функции
x = F(Y)      ' Результат используется
F(Y)          ' Результат не используется
Call F(Y)     ' Результат не используется
F(Y)          ' Передавать Y по значению
' Процедуры
F Y
Call F(Y)
F(Y)          ' Передавать Y по значению
```

Наличие двух вариантов синтаксиса (с круглыми скобками и без) само по себе неприятно, но дело обстоит еще хуже. Допустим, у вас имеется функция `F`. Чем отличаются следующие два вызова:

```
x = F(Y)
F(Y)
```

В одном случае результат используется, а в другом — нет. Но это еще не все! Если параметр `Y` функции `F` был объявлен с ключевым словом `ByRef`, при первом вызове он будет передан по ссылке, а при втором — по значению. Это может привести к неприметным и неожиданным ошибкам.

В VB .NET при вызове функций и процедур используется следующий синтаксис:

```
x = F(Y)      ' Результат используется
Call F(Y)     ' Результат не используется
F Y           ' Результат не используется
F()           ' Вызов функции без параметров
```

После имени функции всегда следуют круглые скобки, даже если функция не имеет параметров. Процедуры и функции вызываются одинаково, если не считать того, что процедура не возвращает результата.

Круглые скобки, как и прежде, могут использоваться для передачи параметров по значению. В этом случае синтаксис вызова выглядит так:

```
F((Y))        ' Принудительная передача по значению
```

В документации Microsoft отмечается, что некоторые изменения VB .NET исправляют недостатки языка. Прекрасным примером такого недостатка, который давно следовало исправить, является синтаксис вызова функций.

Возвращение значений

В VB .NET появился новый способ возвращения значений функциями. Перед нами один из случаев, когда количество вариантов выполнения базовых операций в языке увеличилось.

```
Function F(ByRef As Integer) As Integer
  Return (6)
  Return 6
  F = 6
End Function
```

Команда `Return` позаимствована из языков C++ и C#. Вероятно, это было сделано для удобства программистов, работающих на нескольких языках. Лично мне команда `Return` нравится гораздо больше, чем присваивание значения имени функции. Она упрощает чтение программы и снижает вероятность ошибок при возвращении результата — особенно если ваша функция имеет несколько точек выхода.

К сожалению, компилятор VB .NET, в отличие от компилятора C# не предупреждает о том, что функция не возвращает никакого значения. Надеюсь, Microsoft одумается и предусмотрит хотя бы выдачу предупреждения.

По умолчанию параметры передаются по значению

В VB .NET по умолчанию параметры передаются не по ссылке, а по значению. Честно говоря, я никогда не понимал, почему в Visual Basic по умолчанию использовался механизм передачи по ссылке. Правда, в VB6 передача `ByRef` работает эф-

фективнее, но с точки зрения программиста она увеличивает риск внесения ошибок при непреднамеренной модификации параметров в функции. В VB .NET передача параметров по ссылке уже не обеспечивает сколько-нибудь заметного выигрыша по быстродействию. Причины объясняются в следующем разделе.

Новый механизм передачи по значению

В VB6 смысл ключевых слов `ByRef` и `ByVal` был вполне понятным. При передаче по значению в стеке создается копия переменной, которая и передавалась в качестве параметра функции. Если полученный параметр изменялся внутри функции, то все изменения относились только к копии. При передаче по ссылке функция получает указатель на переменную, поэтому все изменения параметра внутри функции отражались и на исходной переменной.

Единственное исключение из этих правил VB6 составляли строки в командах `Declare` (используемые для передачи функциям API строк, завершенных нуль-символами) и объекты.

Давайте посмотрим, что происходит при передаче объекта в качестве параметра функции. Дальнейшее описание относится как к VB6, так и к VB .NET[1].

На рис. 9.1 показано, как объектные параметры передаются по ссылке. Функция получает указатель на объектную переменную.

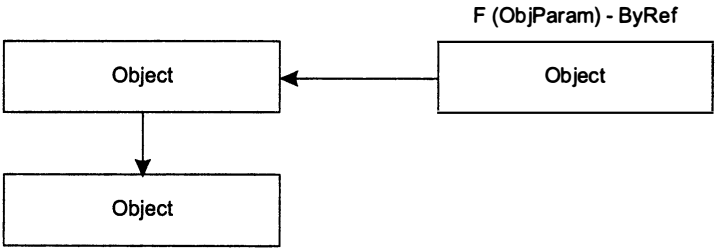


Рис. 9.1. Передача объектного параметра по ссылке

На рис. 9.2 показано, что происходит при передаче параметра по значению. Функция получает объектную переменную, содержащую дополнительную копию адреса объекта.

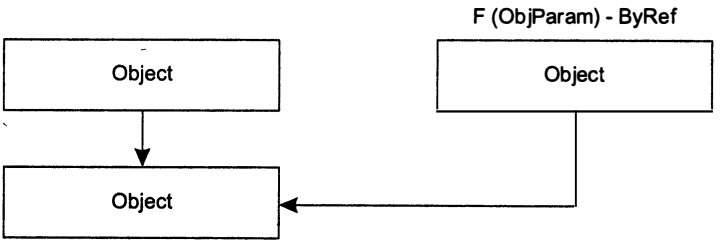


Рис. 9.2. Передача объектного параметра по значению

[1] Конечно, в непосредственной реализации существуют различия. Прежде всего, они связаны с тем, что VB6 средствами COM (точнее, вызовом `QueryInterface`) получает вторую ссылку на объект, тогда как в .NET используется простое присваивание с управлением памятью на базе обычного отслеживания корневых переменных.

Хотя переменная передается по значению, функция получает копию адреса объекта, а не копию самого объекта. Иначе говоря, атрибуты `ByVal` и `ByRef` относятся к переменной, ссылающейся на объект, а не к самому объекту. При передаче по значению объекты не копируются. Передача по значению просто гарантирует, что после вызова исходная переменная будет ссылаться на прежний объект.

Приложение ObjectParams (листинг 9.1) демонстрирует происходящее на примере простого объекта с одним открытым свойством. В приложении объявляются две функции с передачей параметров по ссылке и две функции с передачей по значению. В каждой паре одна функция просто изменяет свойство `X`, а другая присваивает параметр новому экземпляру класса `MyObject`.

Листинг 9.1. Передача объектов[1]

```
Module Module1

  Class MyObject
    Public x As Integer
  End Class

  Public Sub FObjByRef1(ByRef Y As MyObject)
    Y.x = 5
  End Sub

  Public Sub FObjByRef2(ByRef Y As MyObject)
    Y = New MyObject()
    Y.x = 5
  End Sub

  Public Sub FObjByVal1(ByVal Y As MyObject)
    Y.x = 5
  End Sub

  Public Sub FObjByVal2(ByVal Y As MyObject)
    Y = New MyObject()
    Y.x = 5
  End Sub

  Public Sub ObjectTests()
    Dim A As New MyObject()
    Dim B As MyObject = A
    A.x = 1
    Console.WriteLine ("Initial state")
    Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
    Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + _
      B.x.ToString())
    FObjByRef1 (B)
    Console.WriteLine ("After FobjByRef1")
    Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
    Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + _
      B.x.ToString())
    A.x = 1
    B = A
    FObjByRef2 (B)
    Console.WriteLine ("After FobjByRef2")
```

продолжение ⇙

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Листинг 9.1 (продолжение)

```
    Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
    Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + _
      B.x.ToString())
    A.x = 1
    B = A
    FObjByVal1 (B)
    Console.WriteLine ("After FobjByVal1")
    Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
    Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + _
      B.x.ToString())
    A.x = 1
    B = A
    FObjByVal2 (B)
    Console.WriteLine ("After FobjByVal2")
    Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
    Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + _
      B.x.ToString())

  End Sub
```

Результат выглядит следующим образом:

```
Initial state
Are A and B the same? True
A.x: 1 B.x 1
After FobjByRef1
Are A and B the same? True
A.x: 5 B.x 5
After FobjByRef2
Are A and B the same? False
A.x: 1 B.x 5
After FobjByVal1
Are A and B the same? True
A.x: 5 B.x 5
After FobjByVal2
Are A and B the same? True
A.x: 1 B.x 1
```

Сначала свойство `X` инициализируется значением 1, а переменным `B` и `A` присваиваются ссылки на один и тот же объект.

После вызова `FobjByRef1` свойство, как и следовало ожидать, становится равным 5. При этом `A` и `B` продолжают ссылаться на один объект, поскольку параметр `Y` не изменился.

При вызове `FobjByRef2` параметру `Y` присваивается новый объект. Поскольку параметр передавался по ссылке, в переменную `B` заносится ссылка на новый объект. Модификация свойства `X` относится только к новому объекту, поэтому `A.X` сохраняет первоначальное значение 1.

Вызов `FobjByVal1` демонстрирует один важный факт: хотя объект передается по значению, значение свойства объекта изменилось.

С вызовом `FobjByVal2` дело обстоит несколько сложнее. Функция создает новый объект, присваивает ссылку на него параметру `Y` и модифицирует свойство `X`. Тем не менее этот новый объект присваивается временной переменной, а не ссылке на переменную, переданной при вызове. Модификация свойства `X` относится к этому новому объекту. При возвращении из `FobjByVal2` переменная `B` ос-

тается в прежнем состоянии. Она продолжает ссылаться на исходный объект (на который также ссылается переменная A), свойство X которого не изменялось. Что же происходит с объектом, созданным функцией? При возврате из функции FobjByVal2 временная переменная Y выходит из области видимости и перестает ссылаться на созданный объект. Объект, на который не осталось ни одной ссылки, будет уничтожен при следующей сборке мусора.

Все это всегда происходило в Visual Basic 6, но не представляло особого интереса для большинства программистов, которые обычно использовали принятый по умолчанию механизм ByRef и по возможности старались обходиться без присваивания объектным переменным.

Однако в VB .NET механизмы передачи параметров играют значительно более важную роль и в них необходимо хорошо разбираться. Почему? Потому что в VB .NET каждая переменная является объектом.

Не торопитесь погружаться в уныние и позвольте мне подчеркнуть одно обстоятельство.

Объекты структурных типов (числовые переменные, структуры и другие объекты, производные от класса System.ValueType) работают именно так, как следует ожидать. При передаче их с ключевым словом ByVal передается копия объекта структурного типа.

Передача параметров структурного типа

В программе ObjectParams (см. листинг 9.1) продемонстрирована также передача параметров структурного типа. Фрагмент, приведенный в листинге 9.2, практически идентичен фрагменту объектного типа, однако результат оказывается совершенно иным.

Листинг 9.2. Передача структурных параметров

```
Structure MyStruct
  Public x As Integer
End Structure

Public Sub FStructByRef1(ByRef Y As MyStruct)
    Y.x = 5
End Sub

Public Sub FStructByRef2(ByRef Y As MyStruct)
    Y = New MyStruct()
    Y.x = 5
End Sub

Public Sub FStructByVal1(ByVal Y As MyStruct)
    Y.x = 5
End Sub

Public Sub FStructByVal2(ByVal Y As MyStruct)
  Y = New MyStruct()
  Y.x = 5
End Sub

Public Sub StructTests()
  Dim A As MyStruct
  Dim B As MyStruct
```

продолжение

Листинг 9.2 (продолжение)

```
  A.x = 1
  B = A
  Console.WriteLine ("Initial state StructTests")
  Console.WriteLine("Are A and B the same? " + (A.Equals(B)).ToString())
  Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + _
    B.x.ToString())
  FStructByRef1 (B)
  Console.WriteLine ("After FStructByRef1")
  Console.WriteLine("Are A and B the same? " + _
    (A.Equals(B)).ToString())
  Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + _
    B.x.ToString())
  A.x = 1
  B = A
  FStructByRef2 (B)
  Console.WriteLine ("After FStructByRef2")
  Console.WriteLine("Are A and B the same? " + (A.Equals(B)).ToString())
  Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + B.x.ToString())
  A.x = 1
  B = A
  FStructByVal1 (B)
  Console.WriteLine ("After FStructByVal1")
  Console.WriteLine("Are A and B the same? " + (A.Equals(B)).ToString())
  Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + B.x.ToString())
  A.x = 1
  B = A
  FStructByVal2 (B)
  Console.WriteLine ("After FStructByVal2")
  Console.WriteLine("Are A and B the same? " + (A.Equals(B)).ToString())
  Console.WriteLine ("A.x: " + A.x.ToString() + " B.x " + B.x.ToString())

End Sub
```

В программе встречается ряд серьезных изменений, связанных с использованием структурных типов. Во-первых, экземпляры структур создаются без ключевого слова `New`. Переменная структурного типа всегда ссылается на действительный объект — для структурных типов аналога `Nothing` не существует.

При сравнении структур используется метод `Equals`. Это объясняется тем, что VB не разрешает переопределять оператор = для пользовательских структур, но метод `Equals` при желании можно переопределить. В нашем примере используется стандартная реализация метода `Equals`, которая осуществляет *поверхностное* сравнение, то есть просто сравнивает значения соответствующих полей структур.

Для структурных параметров результат выглядит следующим образом:

```
Initial state StructTests
Are A and B the same? True
A.x: 1 B.x 1
After FStructByRef1
Are A and B the same? False
A.x: 1 B.x 5
After FStructByRef2
Are A and B the same? False
A.x: 1 B.x 5
After FStructByVal1
```

```
Are A and B the same? True
A.x: 1 B.x 1
After FStructByVal2
Are A and B the same? True
A.x: 1 B.x 1
```

Как видно из приведенных результатов, изменилось само понятие равенства двух переменных. В нашем примере переменные считаются равными лишь в том случае, если совпадают их свойства X.

В результате вызова `FStructByRef1` свойство `X` объекта `B` становится равным 5, поэтому при выходе из функции `A` и `B` перестают быть равными. Вызов `FStructByRef2` приводит к тому же результату, но по другим причинам. Параметру `Y` присваивается новая структура (что приводит к модификации переменной B), а ее свойство `X` становится равным 5.

В двух последних функциях переменная `B` остается неизменной. Это доказывает, что параметру `Y` действительно была присвоена копия всей структуры.

По этому принципу работают все структурные типы, включая `Boolean`, `Byte`, `Char`, `Decimal`, `Double`, `Enum`, `Single`, `Integer`, `Short` и `Long`. Для них ключевые слова `ByVal` и `ByRef` интерпретируются именно так, как подсказывает здравый смысл.

Однако в приведенном списке отсутствуют два важных случая: строковые переменные и массивы.

Передача строковых параметров

Строковые переменные заслуживают особого внимания. Во-первых, строковый тип очень часто используется, а во-вторых, его поведение трудно понять на интуитивном уровне. В листинге 9.3 приведена программа ObjectParams, видоизмененная для строковых параметров. Внесено лишь одно серьезное изменение — новая функция `FStringByRef3` не изменяет содержимого строки, а лишь заменяет первый символ строки его же текущим значением. Вскоре вы поймете, в чем смысл этой странной операции.

Листинг 9.3. Передача строковых параметров

```
Public Sub FStringByRef1(ByRef Y As String)
  Mid$(Y, 1, 1) = "A"
End Sub

Public Sub FStringByRef2(ByRef Y As String)
  Y = "Hello"
End Sub

Public Sub FStringByRef3(ByRef Y As String)
  Mid$(Y, 1, 1) = Mid$(Y, 1, 1)
End Sub

Public Sub FStringByVal1(ByVal Y As String)
  Mid$(Y, 1, 1) = "A"
End Sub

Public Sub FStringByVal2(ByVal Y As String)
  Y = "Hello"
End Sub
```

продолжение ⇩

Листинг 9.3 (продолжение)

```
Public Sub StringTests()
  Dim A As String = "Hello"
  Dim B As String
  B = A
  Console.WriteLine (Chr(10) + "Initial state StringTests")
  Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
  Console.WriteLine ("A: " + A + " B: " + B)
  FStringByRef1 (B)
  Console.WriteLine ("After FStringByRef1")
  Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
  Console.WriteLine ("A: " + A + " B: " + B)
  A = "Hello"
  B = A
  FStringByRef2 (B)
  Console.WriteLine ("After FStringByRef2")
  Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
  Console.WriteLine ("A: " + A + " B: " + B)
  A = "Hello"
  B = A
  FStringByRef3 (B)
  Console.WriteLine ("After FStringByRef3")
  Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
  Console.WriteLine("Are A and B equal? " + (A = B).ToString())
  Console.WriteLine ("A: " + A + " B: " + B)
  A = "Hello"
  B = A
  FStringByVal1 (B)
  Console.WriteLine ("After FStringByVal1")
  Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
  Console.WriteLine ("A: " + A + " B: " + B)
  A = "Hello"
  B = A
  FStringByVal2 (B)
  Console.WriteLine ("After FStringByVal2")
  Console.WriteLine("Are A and B the same? " + (A Is B).ToString())
  Console.WriteLine ("A: " + A + " B: " + B)
End Sub
```

Не забывая о том, что строки являются объектами ссылочного типа, давайте проанализируем результат строку за строкой:

```
Initial state StringTests
Are A and B the same? True
A: Hello B: Hello
```

В исходном состоянии переменные `A` и `B` ссылаются на один и тот же объект. Иначе говоря, существует всего один объект `String`, содержащий текст «Hello». Переменные `A` и `B` относятся к одному объекту (об этом свидетельствует результат `True` оператора `Is`). Содержимое `A` и `B` тоже считается равным, что вполне очевидно, поскольку переменные ссылаются на один объект:

```
After FStringByRef1
Are A and B the same? False
A: Hello B: Aello
```

Функция `FStringByRef1` заменяет символ строки командой `Mid$`:

```
Mid$(Y,1,1) = "A"
```

Задумайтесь: параметр Y, переданный по ссылке, относится к тому же объекту, что и A. Было бы логично предположить, что модификация символа этой строки приведет к модификации строки A. В приложении ObjectParams изменение свойства X переменной Y приводило к модификации свойства X переменной A, ссылавшейся на тот же объект.

На самом деле функция Mid$ модифицирует не тот объект String, на который ссылается Y. Она создает новую строку с другим первым символом и присваивает ее параметру Y, что приводит к модификации переменной B.

Почему это происходит? Потому что строки VB .NET являются неизменными (см. главу 8). Любая операция, которая на первый взгляд модифицирует строку, в действительности создает новый экземпляр строки. Таким образом, хотя переменная B была изменена и теперь ссылается на новую строку, переменная A продолжает ссылаться на исходную строку «Hello»:

```
After FStringByRef2
Are A and B the same? True
A: Hello B: Hello
```

Функция FStringByRef2 присваивает Y новую строку, совпадающую с исходной строковой величиной:

```
Y = "Hello"
```

Интуиция подсказывает, что B будет ссылаться на новый объект String со строкой «Hello». Но на практике выясняется, что B и A по-прежнему ссылаются на один и тот же объект! Дело в том, что компилятор VB .NET старается разумно подходить к работе со строковыми литералами. Он видит, что Y присваивается строка «Hello», уже присутствующая во внутренней таблице строк, поэтому компилятор просто присваивает Y ссылку на тот же строковый литерал. Эта возможность основана на принципе неизменности строк: CLR знает, что литерал ни при каких условиях не изменится, поэтому там, где это возможно, строковым переменным присваиваются уже существующие литералы.

Пример FStringByRef3 еще нагляднее поясняет сказанное. На первый взгляд эта строка вообще не изменяется:

```
Mid$(Y, 1, 1) = Mid$(Y, 1, 1)
```

Но выходные данные наглядно показывают, что A и B соответствуют разным объектам, хотя они и равны!

```
After FStringByRef3
Are A and B the same? False
Are A and B equal? True
A: Hello B: Hello
```

Другими словами, A и B ссылаются на разные переменные с одинаковым содержимым. Почему? Хотя содержимое строки не изменяется, CLR об этом не знает. В соответствии с принципом неизменности строк переменная Y должна быть связана с новой строкой, даже если ее содержимое будет таким же, как прежде.

Для пары функций с передачей по значению выводятся следующие результаты:

```
After FStringByVal1
Are A and B the same? True
A: Hello B: Hello
```

```
After FStringByVal2
Are A and B the same? True
A: Hello B: Hello
```

Именно такой результат был бы получен, если бы тип `String` был структурным, так как изменения не отражаются на переменной, переданной при вызове (в отличие от примера с параметрами-объектами, где изменение свойства `X` параметра, переданного с ключевым словом `ByVal`, отражалось в свойстве `X` исходного объекта). Такое поведение обусловлено принципом неизменности строк. Изменения в содержимом строки не могут отражаться в исходной переменной, поскольку содержимое строк вообще никогда не изменяется. Вместо этого создается новая строка, присваиваемая параметру `Y`. Так как параметр `Y` передавался по значению, его изменения не распространяются на исходную переменную `B`.

Поскольку передача строки по значению не сопровождается созданием новой копии, передача по ссылке уже не обеспечивает выигрыша по быстродействию! Следовательно, вы можете спокойно передавать строки по значению, если только у вас нет веских доводов против этого.

Передача массивов

Массивы являются объектами. На массивы, в отличие от строк, не распространяется требование неизменности. Это означает, что для массивов должны выводиться те же результаты, которые были получены раньше для параметров-объектов. Изменения элементов массива внутри функции всегда должны распространяться на исходный объект, даже если массив передавался по значению (по аналогии со свойством `X` класса `MyObject`). Присваивание параметру `Y` нового объекта должно распространяться на исходную переменную лишь в том случае, если массив был передан по ссылке.

Я опускаю программный код для экономии места (вы найдете его в приложении ObjectParams). Результат, приведенный в листинге 9.4, выглядит именно так, как мы предполагали.

Листинг 9.4. Передача параметров-массивов

```
Initial state ArrayTests
Are A and B the same? True
Array A:
1, 2, 3, 4, 5
Array B:
1, 2, 3, 4, 5
After FArrayByRef1
Are A and B the same? True
Array A:
5, 2, 3, 4, 5
Array B:
5, 2, 3, 4, 5
After FArrayByRef2
Are A and B the same? False
Array A:
1, 2, 3, 4, 5
Array B:
2, 3, 4, 5, 6
After FArrayByVal1
Are A and B the same? True
```

```
Array A:
5, 2, 3, 4, 5
Array B:
5, 2, 3, 4, 5
After FArrayByVal2
Are A and B the same? True
Array A:
1, 2, 3, 4, 5
Array B:
1, 2, 3, 4, 5
```

Подведем итог всему, о чем говорилось выше.

1. Поведение структурных типов полностью соответствует нашим интуитивным представлениям. При передаче по значению функция получает копию переменной, и любые изменения этой копии не отражаются на исходной переменной. При передаче по ссылке функция получает указатель на переменную, и изменения параметра распространяются на исходную переменную.
2. С объектами дело обстоит сложнее. Ключевые слова `ByVal` и `ByRef` относятся не к самому объекту, а к переменной, ссылающейся на него. Передача по значению не сопровождается копированием объекта. `ByVal` всего лишь гарантирует, что исходная переменная после вызова будет ссылаться на прежний объект. Вызовы методов и обращения к свойствам могут привести к изменению объекта.
3. Массивы являются объектами и подчиняются правилам, представленным в пункте 2.
4. Строки являются объектами, но на их поведение дополнительно влияет принцип неизменности. Поскольку вызовы методов и обращения к свойствам не приводят к модификации строки, может показаться, что передача по значению сопровождается копированием, но это не так. Одно из последствий заключается в том, что передача объекта `String` по значению не уступает по быстродействию его передаче по ссылке (в отличие от VB6, где передача строковых параметров по значению приводила к существенным затратам на копирование строки).

Главное преимущество механизма передачи параметров в .NET заключается в том, что он соответствует вполне определенным правилам, даже при том, что вам придется привыкать к этим правилам.

Необязательные параметры и значения по умолчанию

Visual Basic .NET требует, чтобы для необязательных параметров задавались значения по умолчанию (в VB6 значения по умолчанию могли отсутствовать, в этом случае пропущенным параметрам присваивался 0 или пустая строка в зависимости от типа переменной).

Поскольку в Visual Basic .NET отсутствует универсальный тип `Variant`, естественно, в нем не предусмотрена поддержка необязательных параметров `Variant` и не поддерживается функция `IsMissing` для их проверки.

ParamArray

Ключевое слово `ParamArray` позволяет вызывать функцию с переменным количеством параметров. В VB6 реализация массива параметров выглядела примерно так:

```
Public Function A(ParamArray V() As Variant)
  Dim x As Integer
  If IsMissing(X) Then
    Debug.Print "V is missing"
  Else
    For x = 0 To UBound(V())
      Debug.Print V(x)
    Next x
  End If
End Function
```

Параметры `Variant` (если они присутствуют) передаются по ссылке. Прежде чем перебирать содержимое массива, приходится убеждаться в его наличии при помощи функции `IsMissing`.

В Visual Basic .NET массив параметров может определяться с любым типом (в том числе и с типом `Object`, если ваша функция должна поддерживать разнотипные параметры). Массив передается по значению со всеми последствиями, описанными выше в этой главе.

Если при вызове функции параметры не указаны, массив имеет нулевой размер и может перечисляться напрямую, как показано в следующем фрагменте:

```
Public Sub ParamArrayTest1(ParamArray ByVal A() As Integer)
  Dim x As Integer
  For x = 0 To UBound(A)
    Console.WriteLine(A(x))
  Next
End Function
```

Функция `UBound` возвращает –1 для массива нулевой длины, что упрощает программный код, необходимый для перебора элементов массива.

Правила видимости

Когда программисты говорят об области видимости переменных, они часто имеют в виду сразу два понятия: доступность переменной и ее продолжительность жизни. Доступность определяет те части программы, в которых можно работать с переменной. Продолжительность жизни определяет моменты создания и уничтожения переменной.

Правила видимости для модулей, объектов и сборок рассматриваются в главе 10, поскольку они в большей степени относятся к объектно-ориентированным средствам языка, нежели к его синтаксису. В этой главе мы лишь рассмотрим изменения в области видимости переменных внутри функций.

Рассмотрим следующий пример из приложения ScopingVB6.

```
Sub Main()
  Dim Counter As Integer
  ' X = 3 ' В этой точке переменная не определена
  For Counter = 1 To 3
```

```
    If True Then ' Всегда входить в этот блок
      Dim X As Integer
      Debug.Print X
      X = Counter
    End If
  Next Counter
  Debug.Print "Outside of block: " & X
End Sub
```

Выходные данные в окне отладки выглядят так:

```
0
1
2
Outside of block: 3
```

Программа демонстрирует ряд ключевых моментов, относящихся к видимости переменных в VB6.

- К переменной нельзя обращаться до ее объявления.
- VB6 инициализирует переменные нулями.
- Видимость переменной распространяется за пределы блока, в котором она была объявлена (например, к переменной X можно обращаться вне того блока, в котором она была определена).
- Продолжительность жизни X определяется начальной и конечной передачей управления функции (иначе говоря, вы можете выйти из блока, затем снова войти в него, и значение останется неизменным). Инициализация выполняется при входе в функцию, а не при входе в блок.
- В функции нельзя объявить две локальные переменные с одинаковыми именами.

Рассмотрим эквивалентный код VB .NET из проекта Scoping.

```
Module ScopingMod

  Sub Main()
    Dim Counter As Short
    ' X = 3 ' В этой точке переменная не определена
    For Counter = 1 To 3
      If True Then ' Всегда входить в этот блок
        Dim X As Short
        Console.WriteLine (X)
        X = Counter
      End If
    Next Counter
    'Console.WriteLine("Outside of block: " & X)
    Console.ReadLine()
  End Sub
End Module
```

Результат выглядит так:

```
0
1
2
```

Принципиальное изменение заключается в том, что мы уже не можем обращаться к переменной вне того блока, в котором она была определена. Мастер

Upgrade Wizard при необходимости перемещает объявления локальных переменных за пределы блока, чтобы переменные были доступны при всех обращениях к ним.

Просто для сравнения давайте посмотрим, как подобные ситуации выглядят в новом языке C#. В листинге 9.5 приведен фрагмент примера ScopingCSharp.

Листинг 9.5. Пример ScopingCSharp

```
static void Main(string[] args)
{
  short counter;
  //short x;
  //x = 50; // В этой точке переменная не определена
  for(counter = 1;counter<=3;counter++)
  {
    if (true)
    {
      short x=0; // В C# необходима инициализация
      Console.WriteLine(x);
      x= counter;
    }
  }
  //Console.WriteLine("Outside of block: " + x.ToString());
  Console.ReadLine();
}
```

Результат выглядит так:

```
0
0
0
```

Продолжительность жизни x, как и прежде, определяется продолжительностью работы функции. Тем не менее, язык C# требует, чтобы переменные инициализировались перед использованием, и выполняет инициализацию в точке объявления. Возможно, это чуть снижает быстродействие, но программа становится более предсказуемой (хотя подобные ошибки обычно исправляются на стадии начальной разработки).

В листинге 9.6 приведен интересный фрагмент на языке C++ из приложения ScopingCPP.

Листинг 9.6. Приложение ScopingCPP

```
int main(void)
{
  short counter;
  short x;
  x = 50; // В этой точке переменная не определена
    for(counter = 1;counter<=3;counter++) {
      if (true) {
      short x; // В C++ инициализация не обязательна
      Console::WriteLine(x);
      x= counter;
      }
    }
  Console::WriteLine(x.ToString());
  Console::ReadLine();
    return 0;
}
```

Результат выглядит так:

```
0
1
2
50
```

Справедливости ради замечу, что компилятор C++ предупреждает об использовании неинициализированных переменных. Тем не менее самое интересное в этом фрагменте — объявление во вложенном блоке локальной переменной с существующим именем. Переменная x, объявленная во внутреннем блоке, временно скрывает переменную x, объявленную во внешнем блоке.

Недостаток такого решения состоит в том, что в одной функции может определяться несколько одноименных переменных, что приводит к всевозможным недоразумениям.

Впрочем, есть и преимущество: объявления переменных располагаются поблизости от обращения к ним в конкретных блоках, что позволяет заново использовать имена переменных для других целей.

Откровенно говоря, мне не хватает этой особенности C++, даже несмотря на то, что программы, написанные по правилам C#/VB .NET, легче читаются[1].

Статические переменные

Статическими (static) называются переменные, продолжительность жизни которых совпадает с жизненным циклом всей программы, а видимость ограничивается функцией, в которой эти переменные были объявлены. Статические переменные инициализируются при первом вызове функции и сохраняют свое значение между ее последующими вызовами.

В VB6 вы могли объявить статическую функцию, все переменные которой по умолчанию были статическими. За долгие годы программирования на Visual Basic я ни разу не использовал статические функции в своих программах, поэтому не могу сказать, что меня огорчит их утрата.

Статические переменные существуют на уровне экземпляров класса. Рассмотрим приложение Statics (листинг 9.7).

Листинг 9.7. Приложение Statics

```
Class C
  Public Shared Sub SharedTest()
    Static x As Integer
    x = x + 1
    console.WriteLine (x)
  End Sub

  Public Sub Test()
    Static x As Integer
    x = x + 1
    console.WriteLine (x)
```

продолжение ⇩

[1] Извините, что я потратил столько места на обсуждение второстепенной проблемы. В свое оправдание скажу лишь то, что некоторые аспекты языков программирования интересны сами по себе, независимо от их практической пользы. К тому же я считаю, что любой программист должен хорошо разбираться во всех вопросах видимости переменных, даже в самых изощренных ситуациях.

Листинг 9.7 (продолжение)

```
  End Sub
End Class

Module Module1

  Sub Main()
    Dim c1 As New C()
    Dim c2 As New C()
    c1.Test()
    c1.Test()
    c2.Test()
    c2.Test()
    c.SharedTest()
    c.SharedTest()
    console.ReadLine()


  End Sub

End Module
```

Программа выводит следующий результат:

```
1
2
1
2
1
2
```

Переменные `c1` и `c2` относятся к разным экземплярам класса `C`. Из приведенных результатов видно, что метод `Test` каждого экземпляра, как и метод `SharedTest`[1], обладает собственной копией статических переменных.

Замечание: при знакомстве с платформой .NET неизбежно возникает путаница между статическими (static) и общими (shared) переменными. Причина заключается в том, что ключевое слово `static` в C++ и C# используется для определения переменных, которые являются как общими, так и статическими. В документации и статьях, посвященных .NET, авторы нередко не учитывают двойственной трактовки этих терминов.

Следующие простые правила помогут вам избежать недоразумений.

- Общими переменными в VB .NET называются переменные, принадлежащие всем экземплярам класса.
- Общие методы в VB .NET не связываются с конкретными экземплярами классов (см. главу 10).
- Статические локальные переменные видимы только внутри метода или функции, в которых они были определены, но существуют на протяжении всего жизненного цикла программы
- Методы и переменные, ранее называвшиеся «статическими», в терминологии VB .NET называются «общими».

[1] Общие методы и переменные рассматриваются в главе 10.

Обработка ошибок

Начиная разговор об обработке ошибок в VB .NET, давайте признаем один фундаментальный факт: обработка ошибок в Visual Basic 6 организована на редкость паршиво[1].

Откровенно говоря, в C++ она ничуть не лучше.

Вероятно, сейчас программисты VB6 удивляются, что же я имею против старой доброй команды `On Error Goto`, а программисты C++ решили, что автор совсем выжил из ума, ведь в C++ предусмотрена Структурированная обработка ошибок, о которой я собираюсь поведать.

Не торопитесь негодовать. У меня есть веские причины для подобных заявлений.

История

Прежде чем переходить к обработке ошибок в VB6, стоит заглянуть на более низкий уровень и разобраться в основных источниках ошибок. Большинство ошибок времени выполнения возникает в следующих ситуациях:

- при обращении к методам или свойствам объектов COM;
- при вызове функций API;
- при выполнении операций языка VB6 (например, числовое переполнение).

Давайте рассмотрим все эти причины.

Одно из требований к объектным моделям (в частности, к модели COM) заключается в наличии стандартного и последовательного механизма обработки ошибок. Без такого механизма компонент не сможет оповещать клиентов о том, что в процессе работы возникли ошибки. При отсутствии стандарта становится невозможным взаимодействие компонентов, написанных разными пользователями на разных языках программирования.

В механизме обработки ошибок COM используется 32-разрядный код результата `HRESULT`. Сейчас мы не будем останавливаться на `HRESULT` (подробное описание можно найти в MSDN). Достаточно сказать, что `HRESULT` обеспечивают классификацию ошибок по категориям (элементы ActiveX, Automation, системные ошибки) и числовым кодам. При вызовах методов COM, при обращениям к свойствам и вызове практически всех функций подсистемы OLE возвращаются коды `HRESULT`, которые могут интерпретироваться стандартным образом. Кроме того, объект может дополнительно реализовать интерфейс `ISupportErrorInfo`, при помощи которого компонент передает клиенту дополнительную информацию об ошибке (текстовые сообщения, данные об источнике и даже путь к справочному файлу с описанием ошибки).

Ошибки, происходящие в функциях API, описываются возвращаемым значением функции. Для каждой функции определяются собственные коды ошибок. Одни функции в случае ошибки возвращают –1, другие возвращают 0, третьи — величину, отличную от 0. Обычно в программе можно получить дополнительную

[1] Простите за грубость. Мне трудно выразить свои чувства по этому поводу, не прибегая к не совсем цензурной лексике.

информацию об ошибке при помощи функции API `GetLastError`, которая использует список стандартных ошибок из заголовочного файла winerror.h, входящего в поставку Windows Platform SDK (и Visual C++). Непоследовательность в возвращении кодов ошибок — характерная черта необдуманной, хаотичной эволюции Windows API. Значения, возвращаемые `GetLastError`, были определены в процессе перехода Windows от 16-разрядных версий к 32-разрядным. В частности, одна из категорий кодов `HRESULT` соответствует стандартным кодам ошибок API.

В процессе языковых операций могут возникать ошибки времени выполнения (выход за границы массива, числовое переполнение и т. д.), обрабатываемые в VB6 командой `On Error Goto`. Поскольку VB6 маскирует от программиста вызовы методов COM и обращения к свойствам, ошибки в компонентах COM также могут приводить к возникновению ошибок времени выполнения. В этом случае возвращаемый код `HRESULT` преобразуется в ошибку VB времени выполнения.

Обработка ошибок в VB6

В Visual Basic ошибки перехватываются командой `On Error Goto`. Чем плохо подобное решение?

- В каждой функции или процедуре может действовать лишь один активный обработчик ошибок. Вы можете перехватывать некоторые ошибки и передавать остальные вызывающей функции, однако «вложенные» обработчики ошибок такого рода определяются на уровне функции, а не в блоках внутри нее.
- Нетривиальная обработка ошибок приводит к частой передаче управления, за которой трудно проследить при чтении программы.
- Команда `On Error Resume Next` решает проблему запутанной передачи управления, однако программа при этом загромождается, поскольку встроенную проверку ошибок приходится включать везде, где они могут произойти.
- Непоследовательность получения данных об источнике ошибки (особенно при работе с функциями API) часто порождает общую непоследовательность обработки ошибок в VB. Нередко встречаются программы, в которых одни функции возвращают коды ошибок, а другие инициируют ошибки методом `Raise`.

Обработка ошибок в VC++

Программисты Visual C++ полагают, что обработка ошибок — одна из областей, доказывающих превосходство C++ над Visual Basic. Дело в том, что в Visual C++ поддерживается механизм структурной обработки ошибок, значительно превосходящий синтаксическую конструкцию VB `On Error Goto`. Вскоре мы поговорим о структурной обработке ошибок, а пока я скажу, почему программисты C++ ошибаются.

Да, в VC++ поддерживается структурная обработка ошибок, и она действительно лучше конструкции `On Error Goto`. К сожалению, в VC++ 6.0 этот механизм обработки ошибок очень плохо поддерживается на уровне Windows.

- Функции API, используемые в VC++ значительно чаще, чем в VB, по-прежнему возвращают коды ошибок вместо инициирования ошибок, перехватываемых механизмом структурной обработки.
- Ошибки методов COM и обращений к свойствам, которые VB6 автоматически преобразует в ошибки времени выполнения, для программистов C++ просто возвращаются в виде `HRESULT`. Для этих ошибок механизм структурной обработки тоже не подходит.
- Большинство ошибок исполнительной среды и библиотек C++ не перехватывается механизмом структурной обработки. Обычно эти ошибки вообще не обрабатываются и инициируют исключения защиты памяти. В лучшем случае это приводит к запуску отладчика, в худшем — к аварийному завершению приложения (а в Windows 95/98/ME возможно «зависание» системы).

Другими словами, структурная обработка ошибок, встроенная на уровне языка, приносит мало пользы, если она не поддерживается библиотеками и объектами той среды, для которой вы программируете.

Структурная обработка ошибок

Что такое «структурная обработка ошибок»? В этом термине главным словом является слово «структурная». Visual Basic, C# и C++ принадлежат к классу языков, имеющих блочную структуру. В таких языках программист определяет блоки программного кода, выполняемые как единое целое. Иначе говоря, программист может использовать конструкции вида «если *некоторое условие*, выполнить *блок*; в противном случае выполнить *другой блок*». При этом каждый блок может иметь произвольный размер и содержать другие вложенные блоки.

В C# и C++ границы блоков задаются символами { и }. В Visual Basic блоки ограничиваются синтаксическими элементами самого языка. Мы сейчас говорим не о конкретном синтаксисе, а о возможности группировки команд в блоки, выполняемые на основании единожды принятого решения, без необходимости принимать это решение для каждой строки внутри блока.

Структурная обработка ошибок просто расширяет этот принцип в области обработки ошибок.

Обобщенный синтаксис структурной обработки ошибок выглядит следующим образом:

```
Try
  блок
Catch тип ошибки
  блок, выполняемый при возникновении ошибки
Catch другой тип ошибки
  блок, выполняемый при возникновении ошибки
Finally
  блок, выполняемый перед выходом из блока Try
End Try
```

Внутри каждого блока могут находиться другие блоки, в том числе и вложенные блоки `Try...End Try`!

Различные типы ошибок представляются объектами исключений (exceptions), производными от класса `System.Exception`. Блок `Catch` также может содержать

секцию When с указанием дополнительного условия. Блок Catch выполняется только при возникновении ошибки определенного типа и при истинности условия, заданного в секции When.

Но важнейшие изменения в области обработки ошибок в VB .NET связаны не с синтаксисом языка (при всей их важности) и даже не с тем, что все ошибки времени выполнения языка VB инициируют исключения, перехватываемые механизмом структурной обработки ошибок. Самое главное заключается в том, что *все ошибки, возникающие во всех методах и свойствах объектов .NET Framework, также инициируют исключения, перехватываемые механизмом структурной обработки ошибок*!

Безусловно, синтаксическая конструкция Try...End Try удобна, но по-настоящему фантастически выглядит ее полная интеграция с исполнительной средой, начиная с самого нижнего уровня. Более того, все ошибки, возникающие в используемых объектах COM, средствами .NET тоже отображаются на исключения!

Таким образом, ошибки, несовместимые с механизмом структурной обработки, возникают только при вызове функций API, причем в VB .NET это происходит гораздо реже, чем в VB6 (см. главу 15).

Проект ErrorHandling: общие сведения

Проект ErrorHandling демонстрирует структурную обработку ошибок. Тщательный анализ этого приложения поможет вам лучше понять не только принципы работы механизма структурной обработки, но и разобраться в том, как им следует пользоваться.

Консольное приложение ErrorHandling выполняет несложные файловые операции. Оно читает из файла числа и выводит в консольном окне частное от деления 100 на каждое прочитанное число. Прежде чем углубляться в подробности, приведу краткое описание программы на псевдокоде (листинг 9.8).

Листинг 9.8. Описание проекта ErrorHandling на псевдокоде

```
Sub Main
Try
  Try
    Открыть файл
  Catch-файл не найден
    Создать файл и записать данные
  Catch-остальные ошибки
    Выйти из программы
  End Try

  Try
    Прочитать строку из файла
    Try
      Вывести 100/числовое значение строки
    Catch-строка не преобразуется в число
      Выполнить обработку ошибок
    Catch-деление на ноль
      Выполнить обработку ошибок
    Catch-любые другие ошибки
      Перевести на следующий уровень с дополнительной информацией
    End Try
```

```
  Catch-любая ошибка
    Выполнить обработку ошибок
  Finally
    Закрыть файл и удалить его
End Try
Finally
  Console.ReadLine
End Try
End Sub
```

Функция `Main` начинается с объявления блока `Try`. Возникает мысль, что мы пытаемся перехватывать все ошибки, возникающие в программе. И правда, *это* является веской причиной для заключения всей программы в блок `Try`. У вас появляется возможность самостоятельно обработать ошибки времени выполнения, не полагаясь на стандартные средства .NET Framework. Например, сообщение об ошибке .NET можно сопроводить контактной информацией (адресом электронной почты или телефоном). Возможны и более радикальные варианты, скажем, обработчик строит отчет с полным содержимым стека и отправляет его в вашу службу технической поддержки по электронной почте!

Однако в данном случае вся программа заключена в блок `Try` по совершенно иным соображениям. Возможно, вы заметили, что большинство консольных приложений в этой книге завершается строкой `Console.Readline`. Причина очевидна: при наличии этой команды окно остается открытым до того момента, когда пользователь нажмет клавишу Enter, что позволяет просмотреть результаты ее работы. Однако в проекте ErrorHandling некоторые перехваченные ошибки требуют немедленного завершения программы, для чего проще всего воспользоваться командой `Exit Sub`. Но как вы увидите результат, если выход из функции `Main` приводит к немедленному закрытию окна вывода (поскольку команда `Console.ReadLine` при этом не выполняется)?

Проблема решается заключением всей программы в блок `Try` с размещением итоговой команды `Console.ReadLine` в блоке `Finally`.

Блок `Finally` гарантировано будет выполнен перед выходом из блока `Try`. Он обладает более высоким приоритетом по сравнению с командами `Exit Sub` и `Exit Function`. Другими словами, даже при вызове `Exit Sub` и `Exit Function` в блоке `Try...End Try` все итоговые блоки `Finally` будут выполнены перед выходом из программы.

Перед нами не только принципиальные изменения в смысле команд `Exit Sub` и `Exit Function`, но и принципиальное улучшение, поскольку величайший недостаток команд `Exit Sub` и `Exit Function` заключался именно в том, что они затрудняли деинициализацию и завершающие действия. В VB6 это либо приходилось делать при каждом вызове `Exit Sub`, либо выполнять безусловный `Goto`-переход[1] в общий блок, завершаемый вызовом `Exit Sub` и `Exit Function`.

---

[1] Конечно, команда `Goto` вредна. Тем не менее централизованное выполнение завершающих действий перед выходом из функции — одна из тех областей VB6, где `Goto` оказывается меньшим злом по сравнению с размещением завершающего кода в каждой точке выхода. Другая область VB6, в которой приходится мириться с командой `Goto`, — это, конечно, конструкция `On Error Goto`, поскольку у вас просто нет другого выбора. В VB .NET оба этих случая уже не актуальны, поэтому команда `Goto` еще вреднее, чем в VB6.

Вложенные блоки Try, использованные в программе, демонстрируют одно из основных преимуществ структурной обработки исключений перед командой On Error Goto. Подобная иерархия помогает организовать логическое разделение обработки ошибок даже внутри функций. Аналогичного эффекта можно добиться при помощи нескольких обработчиков и нескольких команд On Error Goto, передающих управление нужному обработчику, но это затруднит чтение программы и отслеживание частой передачи управления.

Вложенные блоки Try также снимают необходимость в команде Resume. В любой точке программы, где потребуется обработать ошибку и продолжить выполнение, достаточно добавить еще один вложенный блок Try!

Проект ErrorHandling: подробный анализ кода

Перейдем к рассмотрению кода приложения ErrorHandling.

В настоящем примере для демонстрации разных режимов оповещения об ошибках используются две логические константы. Если константа ShowErrors равна True, при обнаружении в файле недопустимых чисел на консоль выводится сообщение — либо о том, что строку не удается преобразовать в целое число, либо о том, что прочитанное число (например, ноль) приводит к ошибке деления. Если константа ThrowOnBadFormat равна True, при обнаружении строки, не преобразуемой в целое число, программа инициирует ошибку, которая должна быть перехвачена и обработана блоком следующего уровня (в нашем примере это внешний блок Try, но в компонентах ошибка после некоторой предварительной обработки может передаваться клиенту).

```
' Обработка ошибок
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Imports System.IO
Module Module1

  Const ShowErrors As Boolean = True
  Const ThrowOnBadFormat As Boolean = True
```

Программа начинается с блока Try. В нашем примере этот блок, прежде всего, гарантирует выполнение команды Console.ReadLine из блока Finally:

```
Sub Main()
  Dim FileToRead As String
  Try
    ' Размещение программы в блоке Try позволяет легко
    ' перехватить вызов Exit Sub
    FileToRead = CurDir + "\TestFile.txt"
    Dim TestFile As FileStream
```

Следующий блок Try готовит файл к чтению. Если файл существует, он открывается, а если не существует — программа создает его. При возникновении каких-либо ошибок, препятствующих чтению из файла, программа завершает работу.

В области файлового ввода-вывода в VB .NET также произошли существенные изменения. Эта тема рассматривается в главе 12.

Класс FileStream объявлен производным от Stream — класса, предназначенного для работы с потоками данных. Класс FileStream предназначен для чтения и записи данных в файлах. Если файл существует, объект FileStream успешно

создается программой; в противном случае инициируется исключение FileNotFoundException. А как же другие типы ошибок? В этом блоке имеется вторая секция Catch для перехвата обобщенного объекта Exception[1].

Итак, мы знаем, что тип FileNotFoundException является производным от System.Exception и что System.Exception соответствует любой ошибке, а FileNotFoundException — только ошибке «файл не найден». Возникает вопрос: из чего следует, что ошибка «файл не найден» будет перехвачена именно обработчиком System.Exception, а не более общим обработчиком System.Exception?

Ответ прост: потому что обработчик FileNotFoundException стоит на первом месте.

При возникновении ошибки времени выполнения программа проверяет все блоки Catch и двигается сверху вниз до тех пор, пока не будет найден подходящий блок. Следовательно, обработчики конкретных ошибок должны стоять в начале, а обработчики общих ошибок — в конце списка.

```
Try
  TestFile = New FileStream(FileToRead, FileMode.Open, FileAccess.Read)
Catch E As FileNotFoundException
  ' Если файл не найден, создать новый файл
```

Если файл не найден, мы создаем новый файл и пытаемся записать в него данные примера. Для этого программа сначала создает новый объект FileStream (на этот раз с установленным флагом Create), а затем создает связанный с ним объект StreamWriter. Методы объекта StreamWriter предназначены для выполнения с потоком различных операций чтения и записи.

После записи данных в поток необходимо вызвать метод Flush. Дело в том, что класс FileStream для повышения быстродействия записывает данные в промежуточный буфер. Мы должны очистить буфер и произвести физическую запись в файл, прежде чем переходить в начало файла и читать записанные данные.

Если файл не удается создать или в процессе записи происходит ошибка, инициируется исключение. В нашем примере программа просто перехватывает обобщенный объект System.Exception и завершается, поскольку причины возникновения ошибки несущественны — важно то, что программа не может продолжать работу.

```
Try
  TestFile = New FileStream(FileToRead, FileMode.Create, _
    FileAccess.ReadWrite)
  Dim Writer As New StreamWriter(TestFile)
  Writer.WriteLine ("8")
  Writer.WriteLine ("7")
  Writer.WriteLine ("0")
  Writer.WriteLine ("ABC")
  Writer.WriteLine ("5")
  ' Не забудьте вызвать Flush!!!!
  Writer.Flush()
  TestFile.Seek(0, SeekOrigin.Begin)
Catch CantCreate As Exception
  ' Маловероятно в нашем примере
  Console.WriteLine ("Can't create or write the file")
```

[1] Блок, начинающийся со строки Catch E2 As Exception (см. ниже).

```
  Exit Sub
End Try

Catch E2 As Exception
  Console.WriteLine ("Some other errror occurred")
  ' Внимание: блок Finally выполняется
  ' даже при вызове Exit Sub!
  Exit Sub
End Try
```

В настоящий момент файл открыт и готов к чтению (в противном случае программа бы уже завершилась). Следующим шагом станет чтение из файла. Мы открываем другой блок `Try`, но не для того, чтобы перехватывать ошибки, а в первую очередь для определения блока `Finally`, в котором файл закрывается и удаляется (в нашем примере удаление файла производится в демонстрационных целях). Последовательное чтение записей из файла осуществляется в цикле `Do`.

```
' Если мы находимся в этой точке,
' значит, файл TestFile был благополучно открыт.

Try
  Dim Reader As New StreamReader(TestFile)
  Do
    Dim I As Integer, S As String
    Dim Result As Integer
```

Для каждой строки файла мы входим в новый блок `Try`. Почему? Потому что ошибки, происходящие в этом блоке, обычно не являются фатальными — это означает, что обработка ошибки не помешает нам продолжить чтение остальных записей файла.

```
Try
  S = Reader.ReadLine()
  I = CInt(S)
  Result = 100 \ I
  Console.WriteLine (Result)
```

С исключением `DivideByZeroException` все более или менее понятно. Обратите внимание на использование команды `Exit Try` для выхода из блока `Try`. Команда `Exit Try` передает управление строке, следующей за `End Try`. Тем не менее код в блоке `Finally` выполняется даже в случае вызова `Exit Try`. Вы можете поэкспериментировать с этой командой при помощи константы `ShowErrors`[1].

```
Catch DivByZero As System.DivideByZeroException
  If Not ShowErrors Then Exit Try
  Console.WriteLine ("** Divide by zero **")
```

Исключение `System.InvalidCastException` происходит, когда строку, прочитанную из файла, не удается преобразовать в целое число. Если константа `ThrowOnBadFormat` равна `True`, программа инициирует новое исключение — но какое?

---

[1] Да, я прекрасно знаю: в этом конкретном примере было бы лучше проверять значение `ShowErrors` при вызове `Console.WriteLine` вместо того, чтобы использовать слегка извращенную логику с `Exit Try`. Но тогда я бы не смог продемонстрировать команду `Exit Try`! Откровенно говоря, вам вряд ли придется использовать `Exit Try`, если только код обработки ошибок не отличается особой сложностью.

Вы можете инициировать любое исключение командой Throw. Для этого достаточно создать новый объект Exception и передать его Throw в качестве параметра. При этом можно инициировать исключение стандартного типа или определить свой собственный тип (создайте новый класс, производный от System.Exception или другого класса исключения). В нашей программе заново инициируется исключение InvalidCastException, но с нашим собственным сообщением об ошибке. Более того, вторым параметром передается исходный объект BadConversion. Этот параметр показывает, что инициированное исключение было инициировано в результате другого исключения и содержит дополнительную информацию для внешнего блока.

```
Catch BadConversion As System.InvalidCastException
  If ShowErrors Then Console.WriteLine ("** " + S + " is not a number **")
  If ThrowOnBadFormat Then Throw New _
    System.InvalidCastException(_
    "My own exception happened here", BadConversion)
```

При возникновении других ошибок (маловероятно, но возможно) программа просто инициирует их заново.

```
Catch OtherErrors As Exception
  ' Бессмысленный блок Catch
  Throw OtherErrors
End Try
```

Что произошло бы, если эти «остальные» ошибки не перехватывались бы в нашей программе?

Ошибки, для которых не был найден подходящий блок Catch, автоматически переходят в следующий блок Try! В итоге все выглядит так, словно мы перехватили ошибку и инициировали ее заново. Короче, этот блок Catch бесполезен и его следовало бы убрать из программы.

В контексте текущего блока Try при вызове Throw (в блоке Catch OtherErrors или при возникновении ошибки System.InvalidCastException с ThrowOnBadFormat = True) ошибка выйдет за пределы блока Try, и чтение файла прекратится. Но что, если ошибка преобразования произойдет при ThrowOnBadFormat = False или будет выполнено деление на 0? Команда Throw не вызывается, чтение файла продолжается, и функция подходит к команде Loop:

```
Loop While Reader.Peek <> -1
```

Почему мы используем метод Reader.Peek (возвращающий –1 при достижении конца файла)? Почему бы просто не перехватить исключение, возникающее при достижении конца файла, и не прекратить на этом чтение?

Ответ носит несколько философский характер. Два исключения, перехватываемых нами — деление на 0 и ошибка форматирования, — представляют (в контексте данного примера) настоящие ошибки. Нормально созданный файл с правильными данными не должен содержать нулей или нечислового текста. Другими словами, оба эти состояния представляют аварийные, ненормальные состояния программы.

Однако достижение конца файла в нашем примере ошибкой не является. Алгоритм предусматривает, что в какой-то момент будут исчерпаны все записи, а ожидаемое поведение не является аварийным.

Исключения должны представлять аварийные состояния — ошибки или непредвиденные ситуации. Да, для достижения конца файла можно воспользоваться исключением, но это плохое решение, затрудняющее чтение программного кода и его техническую поддержку.

В каком случае конец файла может представляться исключением?

Предположим, в первой строке файла указывается общее число записей. Программа входит в цикл и читает заданное количество строк. В этом случае достижение конца файла будет представлять непредвиденную ошибку и применение исключения будет вполне оправдано.

Следующий блок `Catch` показывает, как контейнер мог бы обрабатывать инициированные исключения. Он выводит в консольное окно собственное сообщение об ошибке вместе с сообщением, полученным в исключении, и сообщением внутреннего исключения, после чего выводится состояние стека (завидуй, VB6!).

```
Catch E As Exception
  ' Обработка ошибок, возникающих в другой сборке
  Console.WriteLine(ControlChars.CrLf + _
    "An internal error occurred")
  Console.WriteLine("Message: " + E.Message)
  Console.WriteLine("Source Message: " + _
    E.InnerException.Message)
  Dim F As Integer, S As New StackTrace(E)
  Console.WriteLine("StackFrame: ")
  For f = 0 To S.FrameCount - 1
    Console.WriteLine(S.GetFrame(f).ToString())
  Next F
```

Мы подходим к выходу из блоков `Try`. Остается лишь проследить за тем, чтобы файл был должным образом закрыт и удален. В нашем примере файл создавался самим приложением, поэтому какие-либо проблемы с его удалением практически исключены. Впрочем, если вас это все же беспокоит, вы всегда можете включить дополнительный блок `Try` в блок `Finally`. В нашем примере все ошибки, возникающие на этой стадии, будут перехвачены блоком `Catch` внешнего уровня. Приложение завершается итоговым вызовом `Console.ReadLine`:

```
    Finally
      ' Закрытие файла практически всегда
      ' завершается удачно.
      TestFile.Close()
      ' Попытка удаления может оказаться неудачной,
      ' если файл защищен системой безопасности.
      File.Delete (FileToRead)
    End Try

  Catch Unhandled As Exception
    ' На случай, если мы что-то просмотрели.
    ' Вы сами выбираете, какие ошибки передаются наружу!
    Console.WriteLine ("An unhandled exception occurred " + _
      Unhandled.Message)
  Finally
    Console.ReadLine()
  End Try
End Sub

End Module
```

Рекомендую поэкспериментировать с этой программой. Я не буду подробно описывать результаты из-за большого количества вариантов, а лишь приведу некоторые возможные.

- Поэкспериментируйте со значениями констант `ShowErrors` и `ThrowOnBadFormat`.
- Добавьте команды `Throw`, инициирующие разные ошибки в разных местах.
- Включите команды `Exit Sub` в блоки `Catch` и убедитесь в том, что блок `Finalize` выполняется всегда.
- Включите команды `Exit Try` в блоки `Catch` и убедитесь в том, что блок `Finalize` выполняется всегда.
- Включите секции `When` в команды `Catch`, чтобы реализовать проверку дополнительных условий в обработчиках ошибок.

А как же On Error Goto?

Visual Basic .NET продолжает поддерживать старый синтаксис `On Error Goto` и объект `Err`. Возможно, разработчики из Microsoft решили, что изменения в языке и так достаточно масштабны, и им не захотелось брать на себя новые хлопоты. А может, это был акт милосердия по отношению к тем программистам, которым придется адаптировать готовые программы VB6, поскольку код структурной обработки ошибок радикально отличается от кода, использующего `On Error Goto`. А может, это объясняется практическими соображениями: проектирование Upgrade Wizard, правильно преобразующего `On Error Goto` в эквивалентные конструкции `Try...Catch`, было бы чрезвычайно сложной задачей.

Если вам приходится часто адаптировать готовый код и вы знаете, как работает механизм обработки ошибок в вашей программе, возможно, стоит ограничиться применением `On Error Goto` — преимущества структурной обработки исключений лучше всего проявляются на стадии исходного проектирования.

И все же, я рекомендую: стисните зубы и переходите на структурную обработку исключений. Ваш код станет более надежным и устойчивым, что в конечном счете приведет к снижению затрат на сопровождение.

Другие изменения в языке

В этом разделе я кратко обрисую другие серьезные изменения в языке, не касаясь изменений, относящихся к объектно-ориентированному программированию (эта тема рассматривается в следующей главе).

Передача управления

Когда вы в последний раз пользовались конструкциями `Gosub`, `On...Gosub` или `On...Goto`?

Лично я ими вообще никогда не пользовался — это пережитки тех древних времен, когда язык BASIC еще не обладал блочной структурой.

Если вы применяли `Gosub` только потому, что это позволяло заново использовать блок программного кода без хлопот с передачей переменных, подумайте, не

стоит ли определить класс или структуру и передать ссылку функции в качестве параметра. Такое решение заметно упростит чтение и сопровождение вашей программы.

Конструкция `While...Wend` теперь превратилась в блок `While...End While`. К этому второстепенному изменению легко привыкнуть, а язык становится более логичным и последовательным.

Объединение строковых функций

Многие функции VB6 существуют в двух версиях: одна возвращает `Variant`, а другая возвращает строку. Эти функции перечислены в табл. 9.1.

Таблица 9.1. Строковые функции VB6, существующие в двух версиях

Chr	Chr$
CurDir	CurDir$
Dir	Dir$
Format	Format$
Hex	Hex$
LCase	LCase$
Left	Left$
LTrim	LTrim$
Mid	Mid$
Oct	Oct$
Right	Right$
RTrim	RTrim$
Space	Space$
Trim	Trim$
UCase	UCase$

Предполагается, что в VB6 в каждом конкретном случае используется более эффективная форма. Другими словами, если вы работаете с `Variant`, используйте функцию, возвращающую `Variant`, а при работе со строками следует выбрать функцию, возвращающую строку. К сожалению, на практике многие программисты избегают `Variant`, но забывают использовать строковые версии этих функций, что приводит к снижению быстродействия, поскольку VB6 при вызове функции преобразует строковый результат в `Variant`, а затем проводит обратное преобразование при присваивании.

В VB .NET все эти функции существуют в одной форме и всегда возвращают только строки (что вполне логично, поскольку `Variant` больше не существует).

Перед нами еще один пример разумных изменений в языке.

Другие второстепенные изменения

В VB .NET функция `UBound` для массива нулевой длины возвращает –1. Изменения в работе с массивами в VB .NET описаны в главе 8.

Исчезнувшие команды

Первоначальный успех Visual Basic был в значительной мере обусловлен его новаторским подходом к Windows-программированию, а замечательные возможности VB проистекали от инкапсуляции Win32 API на уровне самого языка. Последнее обстоятельство и позволяло легко создавать приложения Windows в Visual Basic. Отчасти это упрощало задачу программиста, так как вместо десятков хитроумных функций API было достаточно изучить несколько команд VB. К счастью, вы также могли прибегать к помощи Win32 API в тех случаях, когда речь шла о нетривиальных возможностях уровня ОС.

Шли годы. Visual Basic развивался, обогащаясь новыми командами и возможностями. Возможно, VB6 до сих пор остается самым удобным средством создания Windows-приложений, но «простым» его уже не назовешь. В наши дни изучение Visual Basic потребует от программиста немалых усилий.

В Visual Basic .NET изменяется сама философия программирования. Да, с точки зрения программиста средства операционной системы по-прежнему инкапсулируются, однако теперь инкапсуляция происходит не на уровне языка, а скорее на уровне .NET Framework. Большинство команд языка и типов данных имеют прямые аналоги среди методов и объектов .NET.

В некоторых случаях Microsoft просто исключила из языка некоторые команды, и соответствующая функциональность теперь обеспечиваются объектами исполнительной среды.

Из всего сказанного следует, что программисты VB .NET располагают большими возможностями, чем когда-либо в прошлом, но за это приходится расплачиваться возрастающей сложностью языка и трудностями при обучении[1].

Графические команды

Следующие методы и свойства Visual Basic 6 *не поддерживаются* в VB .NET: `PSet`, `Scale`, `Circle` и `Line`.

Соответствующие возможности (или их аналоги) находятся в пространствах имен `System.Drawing` и `System.Graphics`. Адаптация кода, использующего эти команды и методы, рассматривается в главе 12.

Не торопитесь скорбеть об утрате. Позвольте напомнить вам синтаксис метода VB6 `Line`:

```
объект.Line [Step] (x1, y1) [Step] - (x2, y2), [цвет], [B][F]
```

Этот синтаксис абсолютно не соответствует общему синтаксису других графических методов. И кому только это в голову пришло?

Решившись на «чистку языка», разработчики Microsoft были вынуждены исключить из него многие знакомые элементы. Конечно, это означает, что нам придется многое учить заново. Однако сам термин «чистка» подразумевает, что изменившиеся аспекты языка были «нечистыми», и во многих случаях это действительно так.

---

[1] Не подумайте, будто я жалуюсь. Я уважаю выбор Microsoft, и мои собственные программы от этого только выиграют. Просто читатель должен четко представлять себе все «плюсы» и «минусы».

Команды, связанные с типом Variant

С исчезновением типа `Variant` пропали многие функции, предназначавшиеся для работы с этим типом.

Функции `IsNull` и `IsEmpty` стали бессмысленными. Чтобы узнать, соответствует ли объект ссылочного типа значению `Nothing`, можно воспользоваться оператором `Is`:

```
If obj Is Nothing Then…
```

Объект структурного типа можно обнулить (как и все его члены), однако сама переменная всегда существует и всегда действительна.

Функция `IsObject` бессмысленна: любая переменная в VB .NET является объектом, поэтому эта функция всегда возвращала бы `True`. Чтобы узнать, соответствует ли переменная ссылочному или структурному типу, можно воспользоваться механизмом рефлексии (см. главу 11).

Как будет показано в главе 11, функции `VarType` и `TypeName` также заменяются средствами рефлексии.

Математические функции

.NET Framework содержит ряд специализированных классов для выполнения математических операций. Из VB .NET были исключены функции `Abs`, `Atn`, `Cos`, `Exp`, `Log`, `Sgn`, `Sin`, `Sqr`, `Tan`, `Rnd` и `Round`; теперь вам придется напрямую работать с математическими библиотеками. Некоторые имена функций `System.Math` не имеют точного соответствия с именами функций VB6 — `Atan` вместо `Atn`, `Sign` вместо `Sgn` и `Sqrt` вместо `Sqr`.

Ниже приведен проект MathVB6, демонстрирующий применение математических функций в VB6.

```
Option Explicit

Sub Main()
  Randomize
  Debug.Print Rnd()
  Debug.Print Sqr(4)
  Debug.Print Round(1.4), Round(1.6)
  Debug.Print Sgn(-1), Sgn(0), Sgn(1)
  Debug.Print Atn(0)
End Sub
```

В результате обработки этого кода мастером Upgrade Wizard (и его преобразования в консольное приложение ради ясности) получается следующий код VB .NET:

```
Option Strict Off
Option Explicit On
Module modMath

  ' ПРЕДУПРЕЖДЕНИЕ: Приложение завершится при выходе
  ' из функции Sub Main().
  ' Дополнительная информация:
  ' ms-help://MS.MSDNVS/vbcon/html/vbup1047.htm
  Public Sub Main()
```

```
    Randomize()
    Console.WriteLine (Rnd())
    Console.WriteLine (System.Math.Sqrt(4))
    Console.WriteLine (VB6.TabLayout(System.Math.Round(1.4), _
      System.Math.Round(1.6)))
    Console.WriteLine (VB6.TabLayout(System.Math.Sign(-1), _
      System.Math.Sign(0), System.Math.Sign(1)))
    Console.WriteLine (System.Math.Atan(0))
  End Sub
End Module
```

Начнем с функций `Sqr`, `Round`, `Sgn` и `Atn`, для которых существуют непосредственные аналоги — `System.Math.Sqrt`, `System.Math.Round`, `System.Math.Sign` и `System.Math.Atan`.

Невольно хочется спросить, зачем было исключать из языка исходные функции, если они так тесно связаны с методами `System.Math`? Зачем заставлять программистов VB .NET пользоваться библиотекой `System.Math`?

Думаю, это было сделано для того, чтобы подтолкнуть программистов VB к использованию богатого набора функций библиотеки `System.Math`. В пространстве имен `System.Math` имеется множество полезных элементов, среди которых функция `Log10` и встроенное значение `Pi` (которые в VB6 приходилось вычислять отдельно, для чего в электронной документации даже приводились специальные формулы). Несомненно, объединение всех математических функций в одном месте выглядит вполне логично, будь то сам язык или отдельное пространство имен. Microsoft выбрала второй вариант, и мне трудно осуждать этот выбор.

С функцией `Rnd` дело обстоит сложнее. Ей соответствует объект `VBMath` из пространства имен `Microsoft.VisualBasic`, однако вы можете воспользоваться классом `System.Random`, обладающим более гибкими возможностями, чем функция `Rnd`. В некоторых случаях Upgrade Wizard выбирает встроенную функцию, тогда как вы, возможно, предпочтете использовать один из общих системных классов. Мы еще вернемся к этой теме.

Другие команды

Функции `LSet` и `RSet` работают только со строками. В .NET присваивание структур выполняется на уровне присваивания полей.

Класс `Debug` заменяется классом `System.Diagnostics.Debug`, а метод `Debug.Print` заменяется методами `System.Diagnostics.Debug.Write/WriteLine`.

Функция `String$` заменена конструктором класса `String` (см. главу 8).

Пространства имен Microsoft.VisualBasic и Compatibility

В маркетинговых материалах Microsoft, посвященных .NET Framework и Visual Basic .NET, часто повторяется утверждение о том, что Visual Basic .NET является «первоклассным» .NET-языком. Важность этого утверждения трудно оценить сразу. Для программистов Visual Basic это в первую очередь означает то, что

VB .NET не является побочной ветвью, «игрушечным» или «неполноценным» языком в среде .NET. Microsoft рассматривает Visual Basic .NET как нормальный .NET-язык, позволяющий в полной мере использовать возможности .NET по созданию CLS-совместимого кода (по мнению Microsoft, это единственный тип кода, который вообще должен создаваться в .NET[1]).

Одно дело — видеть эти утверждения в рекламных брошюрах или на слайдах PowerPoint, и совсем другое — рассматривать их с технологической точки зрения. Ничто не доказывает это лучше, чем пространство имен `Microsoft.VisualBasic`.

Возьмем для примера такой язык, как C++ или C. Синтаксис обоих языков очень прост и содержит очень мало ключевых слов. По историческим причинам большая часть функциональности этих языков обеспечивается громадными исполнительными библиотеками, будь то исполнительная библиотека C, библиотека классов MFC или ATL. В мире .NET синтаксис этих двух языков, как и синтаксис нового языка C#, остается очень простым, а функциональность обеспечивается библиотекой классов .NET.

Просто любопытства ради запустите Object Browser и откройте пространство имен `Microsoft.VisualBasic`. В нем отсутствуют некоторые ключевые слова VB (такие, как `If`, `Then` или `For`), но зато встречаются практически все остальные команды, функции и перечисляемые значения VB.

Другими словами, большинство команд Visual Basic обеспечивается пространством имен .NET Framework, и это относится ко всем остальным .NET-языкам.

Чтобы лучше продемонстрировать сказанное, рассмотрим приложение CSharpOrVB (листинг 9.9). Это приложение написано на C# и содержит ссылку на пространство имен `Microsoft.VisualBasic`[2].

Листинг 9.9. Приложение CSharpOrVB

```
// Приложение CSharpOrVB
// Copyright ©2001 by Desaware Inc.
using Microsoft.VisualBasic;
namespace CSharpOrVB
{
  using System;

  /// <summary>
  ///  Summary description for Class1.
  /// </summary>
  class Class1
  {
    static void Main(string[] args)
    {
      String s = "This is a test";
      Console.WriteLine(Strings.Mid(s, 6, 4));
      Console.ReadLine();

    }
  }
}
```

[1] Я разделяю это мнение.

[2] Не забудьте включить в проект ссылку на исполнительный модуль `Microsoft.VisualBasic` командой Project ▸ Add References.

Все верно. Visual Basic .NET и C# настолько тесно связаны с .NET Framework, что VB .NET не только может использовать те же классы, что и C#, но и C# может работать с классами и командами Visual Basic .NET!

Не знаю, как вы, а я был просто поражен.

Пространство имен `Microsoft.VisualBasic` содержит функции и методы Visual Basic .NET, а также ряд элементов, упрощающих «программирование в стиле VB6» в Visual Basic .NET.

Дело мастера боится

Многие новички в процессе освоения VB .NET пишут программы VB6 и смотрят, какой код будет сгенерирован мастером Upgrade Wizard. К сожалению, этот мастер сбрасывает флажок Option Strict, поэтому такие уроки нередко оказываются вредными.

В качестве примера рассмотрим приложение XOrVB6.

```
Option Explicit

Sub Main()
  Dim Var1 As Integer
  Dim Result As Integer

  Result = Var1 Xor 5

End Sub
```

После обработки этого кода мастером Upgrade Wizard вы получите следующий результат:

```
Option Strict Off
Option Explicit On
Module XOrMod

    ' ПРЕДУПРЕЖДЕНИЕ: Приложение завершится при выходе
    ' из функции Sub Main().
    ' Дополнительная информация:
    ' ms-help://MS.MSDNVS/vbcon/html/vbup1047.htm

    Public Sub Main()
        Dim Var1 As Short
        Dim Result As Short

        Result = Var1 Xor 5

    End Sub
End Module
```

Теперь попытайтесь установить флажок Option Strict. При компиляции программы возникает ошибка, обусловленная неявным преобразованием типа `Integer` в `Short`. Существует много других ситуаций, в которых Upgrade Wizard генерирует код с неявным преобразованием типов или вызовами методов с поздним связыванием, при которых флажок Option Strict должен быть сброшен. Все эти преобразования и вызовы методов сопряжены с потенциальными ошибками времени выполнения — ошибками, которые можно было бы исключить из приложения на стадии проектирования при помощи жесткой проверки типов.

Если вы занимаетесь адаптацией готового кода, вероятно, применение Upgrade Wizard является неизбежным первым шагом. Тем не менее, как я упоминал в части 1 этой книги, адаптация во многих случаях экономически не оправдана. VB .NET лучше подходит для написания новых программ.

Компромиссы совместимости

Каждому программисту VB .NET приходится принимать некоторые принципиальные решения, в том числе выбирать, пользоваться ли пространством имен `Microsoft.VisualBasic` или .NET Framework. Существует немало традиционных функций VB .NET, которые могут быть реализованы средствами .NET.

Если вы ожидаете, что вам когда-нибудь придется переводить свое приложение на другой язык (скажем, на C#), я бы не советовал использовать традиционные функции VB .NET, даже при том, что их можно вызывать из C# (см. выше), и вообще не существует сколько-нибудь разумных причин для перевода программ VB .NET на C#.

В таких случаях я рекомендую придерживаться традиционных функций VB .NET (из пространства имен `Microsoft.VisualBasic`) и избегать лишь функций из пространства `Microsoft.VisualBasic.Compatibility.VB6`.

Почему? Потому что функции пространства имен `Microsoft.VisualBasic` (исключая пространство `Compatibility.VB6`) пережили «чистку» языка, а функции `Compatibility.VB6` являются пережитками, поддерживаемыми ради Migration Wizard.

Тем не менее вы должны непременно изучить .NET-аналоги функций VB. В некоторых областях (в частности, при файловом вводе-выводе) .NET Framework обладает более широкими и гибкими возможностями, чем традиционные команды Visual Basic.

Ниже приведены некоторые примеры областей, в которых встроенные функции перекрываются со средствами .NET Framework, а также даются рекомендации о том, как поступать в таких случаях.

Снова о Rnd

Функция VB .NET `Rnd` получает необязательный параметр, управляющий возвращаемым значением. В типичном случае эта функция вызывается без параметра и возвращает следующее число в случайной последовательности. Тем не менее, если управляющая величина меньше, больше или равна нулю, результат будет равен величине, использованной для раскрутки генератора, следующему числу в последовательности или повторению последнего сгенерированного числа.

Функция `Rnd` возвращает вещественное число в интервале от 0 до 1 (исключая 1).

У функции `Rnd` не существует прямого аналога в .NET Framework. Вместо этого в .NET существует класс `System.Random`. После создания экземпляра этого класса можно использовать его методы для получения случайных чисел.

`Next` — возвращает случайное целое число (32-разрядное).

`NextDouble` — возвращает случайное число от 0 до 1 (исключая 1).

`NextBytes` — заполняет массив `Byte` случайными данными.

Генератор можно раскрутить, указав начальное значение при вызове конструктора `System.Random`; по умолчанию используется начальное значение, основанное на текущей дате и времени (следовательно, отпадает необходимость в команде `Randomize`).

Перед нами отличный пример того, почему Upgrade Wizard выбирает реализацию с использованием исполнительной библиотеки VB .NET. Поскольку объект `System.Random` не имеет метода с аналогичными возможностями, имитация `Rnd` должна состоять из нескольких строк программного кода, поэтому при преобразовании кода из VB6 в VB .NET гораздо проще воспользоваться встроенной функцией.

Тем не менее в новых программах рекомендуется использовать класс `System.Random` и избегать лишних затрат, связанных с применением `Rnd`.

Константы

В .NET Framework, как и в Win32 API и VB6, широко используются константы — математические величины (такие, как `Pi`), цветовые (например, `Red`, `Green`, `Blue`) и флаги операций (скажем, открытия файла).

Различия заключаются в области видимости этих констант.

В .NET константы являются объектами перечисляемых типов или общими свойствами классов. Допустим, вы хотите завершить строку комбинацией символов CR+LF.

Следующие две команды эквивалентны:

```
a = a + ControlChars.CrLf
a = a + Constants.vbCrLf
```

В пространство имен `Microsoft.VisualBasic.Constants` входят многие (но не все) константы с префиксом `vb`, от символов до параметров окон сообщений. Пространство `Microsoft.VisualBasic.ControlChars` содержит только управляющие символы.

Аналогично, вместо использования старых цветовых констант VB `vbRed` и `vbBlue` можно воспользоваться константами из объекта `System.Drawing.Color`.

Конечно, прямые ссылки на константы .NET потребуют определенных усилий с вашей стороны, но если вы к ним привыкнете, это упростит не только чтение, но и сопровождение вашей программы. Почему? Да потому, что перечисления являются типизованными переменными, а функция, получающая параметр перечисляемого типа, сможет получать лишь константы из соответствующего набора. Например, функция `MsgBox` примет только значения из перечисления `MsgBoxStyle`. Более того, значения перечисляемого типа будут выводиться в окне подсказки, что упростит программирование.

В Visual Basic 6 имена констант должны были начинаться с префикса `vb`, поскольку при каждом добавлении в проект ссылки на новую библиотеку ее константы объединялись с глобальным пространством имен приложения. Если одно приложение определяло константу `FAILED`, равную –1, а затем в другом приложении эта константа определялась равной 0, окончательное значение `FAILED` в вашем приложении зависело от того, какая библиотека типов стояла на первом месте в списке ссылок, что открывало массу возможностей для ошибок и путаницы. В Visual Basic (и многих библиотеках COM) эта проблема решалась добавлением префиксов (предположительно уникальных) к именам констант. В .NET иерар-

хическое строение пространств имен снижает вероятность ошибки, поскольку при импортировании пространства имен, использующего перечисляемые типы, имя перечисления включается во все упоминания констант этого типа в программе. Единственным исключением является явное импортирование перечисляемого типа, но оно предполагает, что программист действует сознательно.

Строки и совместимость

Строковые операции традиционно считались одной из сильных сторон языка BASIC (и всех его производных). .NET Framework тоже содержит классы и объекты, предназначенные для выполнения строковых операций.

Я бы рекомендовал вам придерживаться знакомых методов Visual Basic. Они просты и удобны, причем обычно у них имеются прямые аналоги среди методов .NET (а иногда они даже обладают возможностями, не предусмотренными в .NET). Единственным исключением являются области, критические по быстродействию. В таких случаях вам придется сравнить разные варианты и проверить, не обеспечивают ли объекты .NET более высокого быстродействия. Например, при построении строк посредством конкатенации класс `StringBuilder` (разработанный специально для этой цели) работает гораздо быстрее, чем стандартный синтаксис объединения строк.

Обязательно познакомьтесь с классами .NET Framework, перечисленными в табл. 9.2.

Таблица 9.2. Строковые классы в .NET

Пространство имен	Содержимое
System.String	Базовый класс для всех строк VB .NET. Содержит методы для выполнения общих операций со строками
System.Text.Encoding	Класс для преобразования строк между кодировками ASCII, Unicode и т. д.
System.Text.StringBuilder	Класс для построения и модификации строк. В отличие от строк содержимое объектов StringBuilder не является неизменным
System.Text.RegularExpressions	Мощный класс для лексического анализа и обработки строк с применением регулярных выражений[1]

Как упоминалось выше, функция `String$` была заменена конструктором объекта `String`.

Файловый ввод-вывод и совместимость

В пространстве имен `Microsoft.VisualBasic` по-прежнему поддерживаются традиционные файловые операции ввода-вывода с применением команд `Open`, `Close`, `Get`, `Put`, `Input`, `Print` и т. д.

[1] Регулярные выражения выходят за рамки этой книги. Если вы с ними незнакомы, я рекомендую изучить документацию и найти дополнительные источники информации по этой теме. Регулярные выражения обладают воистину фантастическими возможностями.

Пространство .NET Framework `System.IO` объединяет мощные средства потокового ввода-вывода. Подробное описание классов этого пространства приведено в электронной документации .NET. Другое, менее формальное изложение можно найти в документации Visual Studio .NET (раздел «Visual Studio .NET/Visual Basic and Visual C#/Visual Basic Programming/ Processing Drives Folders and Files/File, Drive and Folder Access with Visual Basic .NET»).

Пример FileIO дает представление о том, как в VB .NET организуется чтение содержимого файла в текстовое поле. В листинге 9.10 класс `OpenFileDialog` использован для открытия стандартного диалогового окна. После ввода имени мы средствами класса `OpenFileDialog` получаем объект `Stream` для указанного файла. Для чтения содержимого файла используется отдельный объект `StreamReader`. Запутались? Еще бы, к этому нужно привыкнуть. В главе 12 я попытаюсь хотя бы частично разъяснить, как работают классы файлового вывода в VB .NET.

Листинг 9.10. Чтение содержимого файла из приложения FileIO

```
Private Sub menuItem2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles menuItem2.Click
  openFileDialog1() = New OpenFileDialog()
  openFileDialog1().InitialDirectory = "c:\"
  openFileDialog1().Filter = "txt files (*.txt)|*.txt"
  openFileDialog1().FilterIndex = 1
  openFileDialog1().RestoreDirectory = True
  If openFileDialog1().ShowDialog() = DialogResult().OK Then
    Dim fs As Stream
    fs = openFileDialog1().OpenFile()
    Dim sr As New StreamReader(fs)
    textBox1().Text = sr.ReadToEnd()
    textBox1().SelectionLength = 0
  End If
End Sub
```

Итоги

В этой главе рассматривались различия в синтаксисе VB6 и VB .NET. Вы узнали, что такие серьезные архитектурные решения, как исключение универсального типа `Variant` и отказ от COM, приводят к минимальным изменениям в синтаксисе языка, но обладают далеко идущими последствиями. Надеюсь, углубленное обсуждение нетривиальных изменений в видимости переменных и механизме передачи параметров поможет вам избежать столь же нетривиальных ошибок в ваших программах. Также вы узнали, какие логические обоснования лежат в основе многих изменений в языке и почему они должны сделать ваши программы более устойчивыми и наглядными, а также упростить их сопровождение.

Кроме того, в этой главе описан новый синтаксис обработки ошибок с применением исключений, который по своим возможностям заметно превосходит старую команду VB6 `On Error Goto` — даже дети знают, что команда `Goto` вредна[1].

[1] О вреде Goto известно уже давно. Пришло время закрыть и эту лазейку. Если эта тема вас заинтересовала, обращайтесь к классической статье Эдгара Дейкстры (Edsger W.Dijkstra) «Goto Statement Considered Harmful», в настоящее время доступной на web-сайте Ассоциации вычислительной техники по адресу http://www.acm.org/classics/oct95.

Во многих ситуациях можно использовать как библиотеки языка VB .NET, так и библиотеки классов .NET. Не существует четких правил, по которым можно было бы определить, какое средство следует применять в каждом конкретном случае. Но если в процессе обучения вы при помощи мастера VB .NET Upgrade Wizard обновляете код VB6, просматриваете результат и видите, что в полученном коде используется библиотека `Microsoft.VisualBasic.Compatibility.VB6`, — **откажитесь от него**! Лучше потратьте немного времени и решите эту задачу с использованием традиционных функций VB или классов .NET Framework.

Объекты 10

Темы наследования и объектно-ориентированного программирования в VB .NET уже достаточно подробно рассматривались в главе 5. Впрочем, глава 5 в основном была посвящена концепциям, заложенным в основу объектно-ориентированного программирования и наследования, а также синтаксису, используемому для их выражения. В этой главе рассматриваются некоторые дополнительные вопросы, относящиеся к работе с классами и объектами. Материал главы 5 по очевидным причинам здесь лишь упоминается в общих чертах.

Структура приложения .NET

В главе 9 вы познакомились с видимостью переменной на уровне функции. С видимостью на уровнях класса, модуля, сборки и приложения дело обстоит значительно сложнее. Прежде чем вы начнете понимать принципы видимости переменных в .NET, необходимо разобраться в структуре приложений .NET.

Термин «сборка» (assembly) уже упоминался в предыдущих главах. Вероятно, упоминания «домена приложения» и «сборки» попадались вам и в документации .NET, но, скорее всего, полной ясности у вас все еще нет. На нескольких ближайших страницах я не только постараюсь собрать воедино всю разрозненную информацию, разбросанную по документации, но и изложить этот вопрос нормальным человеческим языком.

Приложение

В мире VB6 (и в программировании до появления .NET вообще) исполняемые файлы делились на две основные категории: EXE-файлы и DLL-файлы. Ниже перечислены их основные характеристики.

EXE-файл:

- работает в отдельном процессе;
- изолируется от других процессов;
- выполняется с учетом атрибутов системы безопасности (только в NT/2000/XP);

- состоит из одного главного потока и может создавать другие потоки;
- может предоставлять объекты, используемые другими приложениями, посредством OLE (ActiveX EXE);
- определяет границы загружаемого блока (EXE-файл не может загружаться в память частично);
- отлаживается независимо от других процессов.

Используя термин «приложение» или «программа», мы в действительности имеем в виду EXE-файлы.

DLL-файл:

- загружается клиентским процессом во время работы программы;
- работает в общем пространстве памяти с процессом и всеми остальными DLL, загруженными процессом;
- загружается с учетом атрибутов системы безопасности (только в NT/2000/XP);
- работает с главным потоком процесса и может создавать новые потоки[1];
- может предоставлять объекты, используемые другими приложениями, посредством OLE (ActiveX DLL);
- определяет границы загружаемого блока (DLL не может загружаться в память частично);
- не может отлаживаться независимо от использующего процесса.

Хотя .NET представляет немало радикальных изменений для разработчиков, эта платформа работает на базе Windows и использует EXE- и DLL-файлы, которые работают так же, как и прежде. Зачем же тогда вносить дополнительную путаницу с «доменом сборки» и «доменом приложения»? Другими словами, если работа EXE- и DLL-файлов совершенно не изменилась, а приложения .NET состоят из EXE- и DLL-файлов, стоит ли усложнять вопрос?

Причины этого решения (как и многих других изменений в .NET) на первый взгляд совершенно не относятся к делу. Более того, они вам уже известны.

Вы знаете, что управление памятью в CLR организовано достаточно жестко. Вы знаете, что в управляемой памяти запрещены указатели. Существование указателей не позволило бы точно определить, продолжает ли использоваться тот или иной объект или переменная (а это необходимо для работы сборщика мусора). Кроме того, при использовании указателей возможно случайное разрушение содержимого памяти приложения или даже исполнительной среды .NET. Концепция управляемой памяти как раз и создавалась в первую очередь для борьбы с утечкой памяти и разрушением ее содержимого.

Интересно заметить, что по этой же причине в 32-разрядных версиях Windows были разделены адресные пространства процессов[2]. Изоляция адресных пространств является самым важным различием между EXE и DLL. Каждый

---

[1] VB6 DLL могли создавать новые потоки лишь при помощи компонентов независимых фирм.

[2] В 16-разрядных версиях Windows память была общей, поэтому ошибки работы с памятью в одном приложении могли привести к сбоям других приложений и даже всей системы.

EXE-файл загружается в отдельном адресном пространстве, тогда как каждая DLL загружается в одном адресном пространстве с загрузившим ее EXE-файлом.

Однако с появлением .NET жесткое управление доступом ко всем объектам в CLR позволяет применить аналогичную изоляцию на уровне отдельных составляющих процесса. Иначе говоря, вы можете создать программный компонент, оформленный в виде DLL, и указать, что объекты и переменные этого компонента полностью изолируются от приложения, использующего компонент, и от всех остальных компонентов процесса. Вне архитектуры .NET это можно сделать лишь одним способом — оформить компонент в виде внешнего COM-сервера, чтобы при каждом создании экземпляра компонента запускался новый процесс. Несомненно, подход .NET гораздо эффективнее — запуск процессов требует лишних затрат. Более того, при решении с ActiveX EXE с каждым компонентом должен быть связан отдельный поток[1]. Поскольку компоненты .NET реализованы в виде DLL, они вполне допускают совместное использование потоков. На рис. 10.1, *а* и 10.1, *б* изображены возможные варианты реализации приложения, использующего три компонента для решения самостоятельных задач, например реализации правил бизнес-логики или запуска web-приложений.

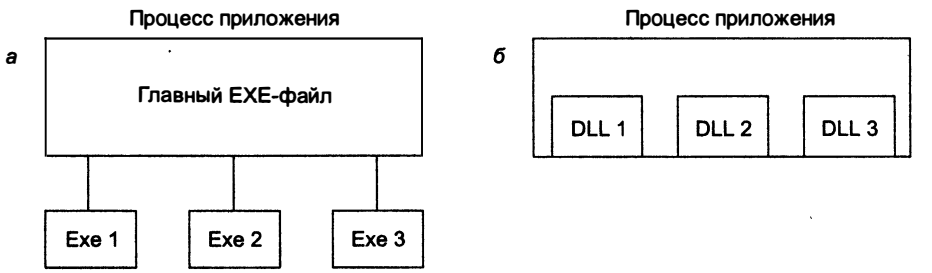


Рис. 10.1. Разделение процессов гарантирует, что компоненты не будут мешать работе других компонентов (*а*). DLL загружаются в пространство основного процесса. С точки зрения операционной системы они используют память совместно, но CLR полностью изолирует их друг от друга (*б*)

В .NET Framework существует термин «домен приложения», который обозначает совокупность объектов, предназначенных для совместной работы, но изолированных друг от друга в управляемой памяти.

Домен приложения может быть реализован в виде EXE-файла с дополнительными DLL-файлами[2]. Возможно и другое решение — оформление в виде одного или нескольких DLL-файлов. При загрузке в память DLL домены приложения изолируются в процессе от других доменов приложения.

Домен приложения:

- состоит из одного или нескольких исполняемых файлов (EXE или DLL), предназначенных для совместной работы;
- изолируется в памяти от других доменов приложения;

[1] Не забывайте: речь идет об «однократных» компонентах ActiveX EXE. Компоненты многократного использования могут совместно использовать потоки, но в этом случае не достигается разделение адресных пространств.

[2] Домен приложения также может включать файлы других типов (например, файлы ресурсов или конфигурационные файлы), но в этой главе рассматриваются лишь программы и исполняемые файлы.

- может выполняться с учетом атрибутов безопасности .NET (см. главу 16);
- может работать в отдельном потоке или использовать другие потоки (в зависимости от управляющего процесса);
- определяет границы выгружаемого блока (домен приложения может загружаться в память по частям, но выгружается как единое целое);
- может отлаживаться независимо от других доменов приложения;
- может предоставлять объекты, используемые другими доменами приложения (или доступные средствами OLE). Доступ к объектам одного домена приложения со стороны другого домена должен осуществляться через промежуточные объекты (proxies), а не через общую память.

Способ оформления домена приложения зависит от типа приложения. В автономных программах домен приложения обычно оформляется в виде EXE-файла. В этом случае термин «домен приложения» достаточно близок по смыслу к традиционным EXE-приложениям.

В web-приложениях домены приложений обычно оформляются в виде DLL. Это обеспечивает безопасную работу web-приложений в процессе web-сервера и одновременно не позволяет им мешать работе друг друга. Отдельные домены приложений даже можно отлаживать из других доменов, работающих под управлением той же программы.

Используя домен приложения, вы используете все его EXE- и DLL-файлы. Впрочем, в .NET существует особый подход к этим файлам и организации программного кода.

Сборки

Вы только что прочитали, что домен приложения состоит из одного или нескольких исполняемых файлов. Означает ли это, что все исполняемые файлы должны загружаться одновременно?

Нет.

По аналогии с тем, как традиционное приложение Windows может загружать отдельные DLL или запускать EXE-серверы ActiveX по мере надобности, так и домен приложения может загружать отдельные сборки при возникновении необходимости в них. Собственно, термин «сборка» (assembly) и определяет минимальную единицу загрузки в домене приложения.

Сборка:

- является минимальной единицей загрузки программного кода (сборки выгружаются при выгрузке домена приложения);
- в VB .NET состоит из одного EXE- или DLL-файла (в других языках сборка может состоять из нескольких файлов);
- является наименьшим компонентом .NET, которому может быть присвоена версия;
- загружается лишь в том случае, если ее выполнение разрешено текущими атрибутами безопасности;
- может проверить наличие и возможность выполнения всех сборок, от которых зависит ее работа.

На рис. 10.2 изображена общая структура приложения .NET. Рисунок относится и к ASP-подобным сценариям, в которых каждый домен приложения представляет отдельное web-приложение или web-службу. Все домены приложения работают в одном процессе и совместно используют потоки по правилам, определяемым процессом. Каждый домен приложения состоит из одной или нескольких сборок, каждая из которых реализуется в виде одного или нескольких исполняемых файлов[1]. Автономное приложение VB .NET представляет собой отдельный процесс с одним доменом приложения, состоящим из одной или нескольких сборок[2].

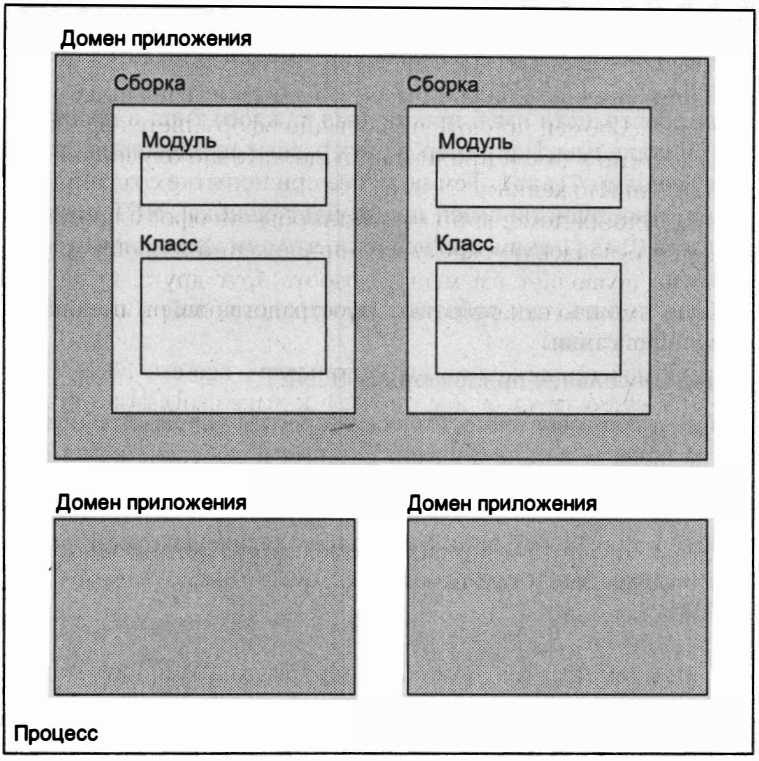


Рис. 10.2. Взаимосвязь между процессами, доменами приложений и сборками

Домены приложений и сборки подробно рассматриваются в главе 16 при обсуждении размещения и контроля версии.

Область видимости в .NET

Как упоминалось выше, понятие «области видимости» в действительности состоит из двух разных аспектов: доступности и продолжительности жизни. Продол-

[1] В VB .NET сборка может состоять только из одного исполняемого файла.

[2] VB .NET позволяет создавать приложения, управляющие несколькими доменами, но это нестандартная ситуация.

жительность жизни определяет сроки существования объекта, а доступность определяет возможность обращения к объекту из программы.

Чтобы понять, как работают области видимости в VB .NET, необходимо сначала разобраться в концепции «пространств имен».

Пространства имен

Домены приложения и сборки определяют структуру программы VB .NET с позиций загрузки, размещения и контроля версии. Пространства имен определяют структуру программы VB .NET с позиций видимости.

Каждый тип объекта в VB .NET обладает именем и входит в некоторое пространство имен.

Концепция пространств имен проста: имя каждого типа в пространстве имен должно быть уникальным. Например, в двух разных приложениях можно определить классы с именами `Class1`. Тем не менее при попытке создания двух классов `Class1` в одном пространстве имен произойдет ошибка. .NET позволит создать два разных класса `Class1` лишь при условии, что они принадлежат к разным пространствам имен.

Чтобы лучше понять, как работают пространства имен, попробуйте выполнить следующие действия.

1. Создайте консольное приложение VB .NET.
2. Откройте диалоговое окно Project Properties (щелкните правой кнопкой мыши на проекте в окне Solution Explorer и выберите команду **Properties**).
3. Очистите текстовое поле **Root Namespace**.
4. Включите в программу блок `Namespace` следующего вида:

```
Namespace MovingToVB.CH10.Class1Example
  Module Module1

  Sub Main()
  End Sub

  End Module
End Namespace
```

В VB .NET используемое по умолчанию пространство имен сборки задается в диалоговом окне свойств проекта. Если вы захотите переопределить пространство имен сборки, поле **Root Namespace** обычно очищается, поскольку все содержимое этого поля становится префиксом имен, определяемых командой `Namespace`. Другими словами, если бы в нашем примере в текстовом поле остался текст «Class1Example», то вместо `MovingToVB.CH10.Class1Example` было бы определено пространство имен `Class1Example.MovingToVB.CH10.Class1Example`.

Попробуйте включить в пространство имен два класса с именем `Class1`:

```
Namespace MovingToVB.CH10.Class1Example
  Class Class1
  End Class

  Class Class1
  End Class
```

Разумеется, попытка завершится неудачей. Компилятор выводит ошибку вида «Class1Example\Module1.vb(2):class 'Class1' and class 'Class1' conflict in namespace Class1Example»[1].

Короче говоря, создать два класса с одинаковыми именами вы не сможете.

Теперь приведите программу к следующему виду:

```
Namespace MovingToVB.CH10.Class1Example
  Class Class1
  End Class

  Module Module1
    Dim c1 As New Class1
    Dim c1b As New Class1ExampleNS2.Class1()
    Sub Main()
    End Sub
  End Module
End Namespace

Namespace MovingToVB.CH10.Class1ExampleNS2
  Class Class1
  End Class
End Namespace
```

Мы определили два класса `Class1` в одной сборке! Конфликт предотвращается тем, что каждое имя определяется в отдельном пространстве имен. Как нетрудно убедиться, программный код одного пространства имен может легко работать с типами из другого пространства — для этого достаточно уточнить тип, указав пространство имен, к которому он относится.

Итак, вы убедились, что сборка может определять несколько пространств имен[2]. Хотя определять разные пространства имен в *одном* файле обычно не рекомендуется, делать это в разных файлах проекта обычно вполне разумно. Более того, само иерархическое строение пространств имен способствует такому подходу. Если взглянуть на пространство `System`, вы увидите, что под уровнем `System` располагаются пространства имен второго уровня, причем большинство из них входит в одну DLL (сборку). Пространства имен часто используются для логической группировки классов.

Впрочем, определение нескольких пространств имен в одной сборке должно объясняться вескими причинами. Однозначное соответствие между сборками и пространствами имен выглядит более логично и уменьшает вероятность ошибки среди пользователей данной сборки.

Размещение приложений

Заглянув в каталог Namespaces главы 10, вы увидите в нем подкаталоги двух проектов ConsoleApplication1 и GroupSupport. Корневое пространство имен в обоих проектах было очищено. В листинге 10.1 приведен модуль `Module1` проекта ConsoleApplication1.

---

[1] При попытке компиляции также будет выдана ошибка «в пространстве имен не определена функция Sub Main». Вернитесь в диалоговое окно **Project Properties** и выберите пространство имен для запуска программы.

[2] В VB .NET проект компилируется в одну сборку.

Листинг 10.1. Файл module1.vb проекта ConsoleApplication1[1]

```
Imports MovingToVB.CH10.Organization.Members
Namespace MovingToVB.CH10.Organization

  Class Organization
  End Class

  Module Module1
    Sub Main()
      Dim org As New Organization()
      ' Следующие две строки идентичны
      ' благодаря команде Imports.
      Dim grp As New CH10.Organization.Members.MemberCollection()
      Dim grp2 As New MemberCollection()

      ' Внимание - это объявление относится к другой сборке,
      ' хотя оно находится в том же пространстве имен!
      Dim sorter As New CH10.Organization.Members.MemberSorter()
      Dim sorter2 As New MemberSorter()
    End Sub
  End Module
End Namespace

Namespace MovingToVB.CH10.Organization.Members
  Class GroupMember
  End Class

  Class MemberCollection
  End Class
End Namespace
```

В листинге 10.2 приведен модуль class1.vb проекта GroupSupport.

Листинг 10.2. Файл class1.vb проекта GroupSupport

```
Namespace MovingToVB.CH10.Organization.Members
  Public Class MemberSorter

  End Class
End Namespace
```

Проект ConsoleApplication1 содержит два пространства имен. В пространстве `MovingToVB.Ch10.Organization` определяется главная программа и классы для управления гипотетической организацией; объекты для управления членами этой организации определяются во втором пространстве имен `MovingToVB.Ch10.Organization.Members`.

Наибольший интерес в проекте ConsoleApplication1 для нас представляет класс `MemberSorter` — он не является частью проекта! Класс `MemberSorter` определен в проекте GroupSupport, на который имеется ссылка в проекте ConsoleApplication1[2]. Впрочем, это не мешает ему входить в пространство имен `MovingToVB.Ch10.Organization.Members`.

---

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

[2] Чтобы включить соответствующую ссылку в проект, щелкните правой кнопкой мыши на кнопке References в окне Solution Explorer и выберите команду Add Reference. Перейдите на вкладку Projects и выберите проект GroupSupport.

Да, все верно: в пространство имен может входить несколько сборок!

Зачем это нужно?

Предположим, `MemberSorter` — довольно большой и сложный класс, используемый относительно редко. Логически он относится к пространству имен `MovingToVB.Ch10.Organization.Member`, поскольку тесно связан с другими классами этого пространства имен. Однако всегда загружать код редко используемого класса было бы глупо. Если выделить класс `MemberSorter` в отдельную сборку, он будет загружаться только в случае надобности.

В более изощренной схеме размещения (например, при установке приложения по Интернету) возможны даже такие ситуации, когда сборка GroupSupport не устанавливается до ее использования!

Другими словами, пространства имен позволяют разработчикам полностью отделить логическую структуру классов от стратегии их использования и системы безопасности. Вы можете организовать классы таким образом, чтобы их было легче понять, сопровождать и документировать, но при этом объединять их в сборки по другим соображениям, например таким, как эффективность загрузки. Архитектура вашего приложения не всегда совпадает со схемой размещения.

Конечно, с точки зрения программиста подобная гибкость означает дополнительную ответственность. Теперь вы должны не только проектировать объекты и связи между ними, но и определить, в каких сборках хранятся те или иные объекты. Полагаю, расширение возможностей оправдывает эти дополнительные усилия.

Команда Imports

Команда `Imports` позволяет ссылаться на элементы пространств имен в укороченной записи. К объектам импортируемого пространства имен можно обращаться напрямую, не указывая полного имени пространства. Пример встречается в процедуре `Sub Main` проекта ConsoleApplication1, в котором ссылки на классы `MemberCollection` и `MemberSorter` задаются как полностью, так и сокращенно.

Что произойдет, если импортируемое пространство имен содержит класс с таким же именем, как и у класса в текущем пространстве имен? В этом случае предпочтение отдается классу из текущего пространства имен. Вы по-прежнему сможете обратиться к классу из другого пространства, но только по полному имени. При попытке импортировать два пространства с одноименными классами (и если класс с таким именем не определен в текущем пространстве имен) произойдет ошибка, поскольку компилятор не сможет определить, какому из двух импортированных пространств следует отдать предпочтение.

А если две сборки пытаются определить одноименные классы в одном пространстве имен? Возникает ошибка: тот факт, что пространство имен не может содержать два класса с одинаковыми именами, распространяется и на случай, когда эти классы находятся в двух разных сборках.

А что произойдет, если вы дополните существующее пространство имен (например, `System`)? Ваша сборка и все остальные сборки, ссылающиеся на вашу, увидят это дополнение. И все будет работать совершенно нормально — до того момента, когда Microsoft включит в пространство имен `System` класс с конфликтующим именем. Мораль: чужие пространства имен лучше не трогать!

Команда `Imports` также применяется для импортирования объектов перечисляемого типа, что позволяет работать со значениями этих типов без уточнения имени перечисления.

Например, для обращения к константе `CrLf` приходится использовать запись вида:

```
Dim S As String = ControlChars.CrLf
```

Но если включить в программу строку:

```
Imports Microsoft.VisualBasic.ControlChars
```

на `CrLf` можно ссылаться иначе:

```
Dim S As String = CrLf
```

Включение ссылки на пространство имен в проект часто путают с импортированием.

Чтобы включить в проект ссылку на пространство имен (а вернее, на реализующую его сборку), следует воспользоваться окном решения Solution или диалоговым окном свойств проекта. После создания ссылки на пространство имен вы можете обращаться ко всем его объектам и методам по полностью уточненным именам.

Команда `Imports` позволяет использовать укороченную запись для пространств имен, на которые существуют ссылки в проекте, чтобы вам не приходилось при каждом использовании объекта из этого пространства вводить его полностью уточненное имя.

Имена и пространства имен

В VB6 и COM Microsoft решила проблему конфликтов имен посредством идентификации объектов по глобально-уникальным идентификаторам (GUID). В .NET пространства имен имеют иерархическую, наглядную структуру, с которой вы уже неоднократно встречались. Корневое пространство имен желательно выбирать так, чтобы оно было заведомо уникальным, например, название вашей организации.

Но что произойдет, если в системе установлены две конфликтующие версии пространства имен?

В большинстве случаев проблем не будет, поскольку, если сборка не была включена в глобальный кэш, ваша программа будет видеть пространства имен лишь тех сборок, на которые в ней присутствуют ссылки.

Но возможна и другая ситуация: сборка, на которую вы ссылаетесь, может изменить свое пространство имен так, что оно по каким-то причинам станет несовместимо с этой сборкой. Что делать?

Ответ — никто не может изменить сборку, на которую вы ссылаетесь, если только вы специально не разрешите такую модификацию или не выполните ее самостоятельно. Сборкам могут присваиваться «сильные» имена, включающие собственно имя сборки, версию, сигнатуру и другую информацию, по которой эту конкретную сборку можно отличить от других версий. Если приложение связывается с конкретной сборкой по «сильному» имени, вам никогда не придется беспокоиться об изменениях, нарушающих совместимость.

Имена сборок и контроль версий более подробно рассматриваются в главе 16.

Область видимости 1: уровень пространства имен

На уровне пространств имен определяются объекты четырех видов: классы, модули, структуры и перечисления.

Всем этим объектам может быть присвоен атрибут области видимости `Friend` или `Public`[1].

Таблица 10.1. Атрибуты видимости

	Сборка/проект	Везде
Public	Видимый	Видимый
Friend	Видимый	Скрытый

Смысл этих атрибутов продемонстрирован в приложении AssemblyScoping[2].

Область видимости объекта на этом уровне определяет максимальную общую область видимости для всех элементов объекта. Например, `Public`-переменная в `Public`-классе видима и доступна для всех, кто имеет доступ к пространству имен.

Классы, структуры и перечисления знакомы вам как по предыдущему опыту программирования на VB6, так и из предшествующих глав книги, однако на модулях (`Module`) стоит задержаться чуть подробнее.

Как вы, возможно, заметили, в VB .NET не существует различий между файлами программного кода: все они имеют расширение `.VB` и одинаковую внутреннюю структуру. Модули VB .NET работают почти так же, как классы, с двумя принципиальными различиями.

1. Невозможно создать экземпляр модуля.
2. Элементы модуля видимы без включения имени модуля во всем коде, где виден сам модуль. Другими словами, открытые функции/переменные модуля можно использовать как глобальные функции/переменные стандартных модулей VB6.

Как ни странно, C# не позволяет программисту создавать модули, но может использовать модули, определяемые в приложениях VB .NET (см. приложение ModuleScoping, входящее в примеры программ, но не рассматриваемое в книге).

Правила наследования

Visual Basic .NET не позволяет расширять область видимости базового класса в производных классах. Иначе говоря, `Friend`-класс может наследовать от `Public`-класса, поскольку видимость всех унаследованных членов будет меньше `Public`, но `Public`-класс *не может* наследовать от `Friend`-класса, поскольку в этом слу-

---

[1] Атрибуты `Private`, `Protected` и `Protected Friend` применяются только к элементам классов и рассматриваются ниже. В версии бета-1 поддерживались закрытые элементы уровня пространства имен, видимые только в текущем файле.

[2] Приложение AsemblyScoping содержит большое количество всевозможных объявлений с разными областями видимости, однако сама программа ничего не делает. Особых объяснений она не требует, поэтому я не стал включать листинг в книгу.

чае открытые члены класса Friend окажутся видимыми за пределами нормальной области видимости Friend.

Разумеется, наследование возможно лишь в том случае, если производный класс имеет доступ к базовому классу.

Контроль наследования

Ключевое слово MustInherit указывает на то, что экземпляры данного класса создаваться не могут. Допускается лишь создание экземпляров производных классов.

Ключевое слово NotInheritable означает, что данный класс вообще не может использоваться в качестве базового при наследовании. В таких классах нельзя определять защищенные члены (да это и бессмысленно, поскольку, как будет показано в следующем разделе, ключевое слово Protected имеет смысл лишь в контексте наследования).

Ключевое слово Shadows при определении вложенного класса указывает на то, что вложенный класс переопределяет соответствующие члены базового класса. Предположим, у вас имеется фрагмент проекта Inheritance1, приведенный в листинге 10.3.

Листинг 10.3. Приложение Inheritance1

```
Class A
  Class B
    Sub Test()
      Console.WriteLine ("A.B.Test")
    End Sub
  End Class
  Class D
    Sub Test()
      Console.WriteLine ("A.D.Test")
    End Sub
  End Class

End Class

Class C
  Inherits A
  Shadows Class B
    Sub Test()
      Console.WriteLine ("C.B.Test")
    End Sub
  End Class

End Class

Module Module1
  Sub Main()
    Dim abref As New A.B()
    Dim cdref As New C.D()
    Dim cbref As New C.B()
    abref.Test()
    cdref.Test()
    cbref.Test()
    Console.ReadLine()
  End Sub

End Module
```

Программа выводит следующий результат:

```
A.B.Test
A.D.Test
C.B.Test
```

Из-за ключевого слова `Shadows` определение класса `B` в классе `C` маскирует определение класса `B` в классе `A`.

В данном случае маскировка используется по умолчанию, поэтому при отсутствии ключевого слова `Shadows` программа будет выдавать тот же результат, но компилятор выдаст предупреждение и напомнит о том, чтобы вы включили ключевое слово `Shadows`.

Область видимости 1: уровень класса

В классах и модулях атрибуты видимости определяют видимость отдельных членов — методов, свойств и переменных класса[1].

Ниже приведена краткая сводка основных атрибутов доступа.

- `Public` — элемент доступен за пределами класса, в котором он определяется.
- `Private` — элемент недоступен за пределами класса, в котором он определяется.
- `Friend` — элемент доступен за пределами класса, но только в пределах сборки, в которой определяется класс.
- `Protected` — за пределами класса элемент доступен только в классах, производных от данного.
- `Protected Friend` — элемент доступен за пределами класса, но только в пределах сборки, в которой определяется класс, и в производных классах.

Эта тема уже рассматривалась в главе 5, поэтому табл. 10.2 и 10.3 следует рассматривать как исчерпывающую сводку видимости членов классов. Для тестирования использовалось приложение ClassScoping (присутствует среди примеров, но не приводится в книге).

Таблица 10.2. Public-классы

	Private	Friend	Protected	Protected Friend	Public
В классе	X	X	X	X	X
Ссылки на базовый класс в производных классах		X	X	X	X
Ссылки на производный класс в производных классах		X		X	X
Ссылки на класс внутри сборки		X		X	X
Ссылки на класс за пределами сборки					X

[1] Я говорю о классах, однако атрибуты видимости сохраняют смысл также в модулях и структурах (не считая аспектов наследования, относящихся только к классам). Помните, что атрибуты видимости также распространяются на классы и структуры, которые являются членами других классов.

Таблица 10.3. Friend-классы

	Private	Friend	Protected	Protected Friend	Public
В классе	X	X	X	X	X
Ссылки на базовый класс в производных классах		X	X	X	X
Ссылки на производный класс в производных классах		X		X	X
Ссылки на класс внутри сборки		X		X	X
Ссылки на класс за пределами сборки					

Унаследованный метод может быть скрыт от производного класса (даже если он объявлен с ключевым словом `Friend` или `Public`) при помощи ключевого слова `Shadows`.

Видимость классов и методов класса всегда должна находиться на минимальном уровне, необходимом для работы приложения. Это упрощает структуру приложения и уменьшает вероятность ошибочного вызова методов в ваших сборках.

Область видимости и многопоточность

Не забывайте о принципиальном различии между глобальными переменными VB6 и VB .NET. Глобальные переменные VB6 существуют на уровне потока. Следовательно, если вы создаете многопоточный EXE-файл или DLL в VB6, каждый поток будет обладать собственной копией глобальных переменных. В VB .NET глобальные переменные совместно используются всеми потоками, если только они не были объявлены локальными по отношению к потокам при помощи атрибута `ThreadStatic` (см. главу 7).

Учтите, что Upgrade Wizard не преобразует глобальные переменные VB6 в переменные VB .NET, локальные по отношению к потокам, и даже не предупреждает об этих различиях.

Дополнительно о классах

Возможно, определение области видимости классов и их членов является важнейшей фазой объектно-ориентированного проектирования, которая не уступает по значимости разработке объектной модели. В этом разделе описаны некоторые особенности классов, в которых вы должны хорошо разбираться.

Общие переменные

Любую переменную класса можно снабдить атрибутом `Shared`. Синтаксис объявления общих переменных:

```
Private Shared SharedVariable As Integer
```

Общие переменные совместно используются всеми экземплярами классов. У них имеется несколько стандартных применений.

- Отслеживание всех экземпляров класса. Например, общая переменная может содержать коллекцию ссылок на все экземпляры класса или счетчик существующих экземпляров.
- Предварительное вычисление данных, используемых всеми экземплярами класса. Например, класс взаимных фондов может содержать общие переменные со средними показателями по разным категориям, чтобы упростить сравнение данных конкретного экземпляра с этими значениями.
- Определение переменных, используемых в работе общих процедур.

Конечно, общие переменные также могут обладать атрибутами `Friend`, `Protected`, `Friend Protected` и `Public`.

Общие процедуры могут вызываться без указания конкретного экземпляра класса. В них не допускается обращение к членам класса без атрибута `Shared`.

Программисты иногда создают классы, состоящие только из общих членов. Это обеспечивает логическую группировку функций общего назначения, для которых не требуется создавать новые объекты. Тем не менее в VB .NET эта задача решается при помощи модулей.

В следующем фрагменте из проекта AssemblyScoping продемонстрировано использование общих переменных и процедур:

```
Public Class SharingDemo
  Public Shared SharedVariable As Integer
  Public NotSharedVariable As Integer
  Shared Sub ShowShared()
    Console.WriteLine ("Access using Shared Procedure: " & _
    SharedVariable.ToString)
  End Sub
End Class
```

Следующий фрагмент показывает, как происходит обращение к общим членам класса:

```
Dim sh As New Scoping.SharingDemo()
Dim sh2 As New Scoping.SharingDemo()
sh.SharedVariable = 5
Console.WriteLine("5 indicates value was shared: " _
  & sh2.SharedVariable.ToString)
Console.WriteLine("Access without instance: " _
  & Scoping.SharingDemo.SharedVariable.ToString)
Scoping.SharingDemo.ShowShared()
```

К общим переменным можно обращаться как через экземпляр, так и напрямую по имени класса.

Говоря об области видимости общих членов класса, мы не ограничиваемся простой доступностью этих членов: речь также идет об области совместного использования. Другими словами, если у вас имеется общая `Public`-переменная в `Public`-классе, значение этой переменной будет сохраняться для всех экземпляров класса во всем домене приложения. Более того, значение сохраняется и во всех производных классах. Это означает, что общие переменные не подходят для хранения данных, смысл которых определяется производными классами.

Класс, в котором определяются общие переменные, должен управлять их интерпретацией и использованием. Общим переменным, как и всем остальным переменным класса, должен назначаться минимальный уровень доступа.

MyBase и MyClass

Два специальных ключевых слова `MyBase` и `MyClass` предназначены для уточнения того, какой метод должен использоваться в ситуации с наследованием.

Ключевое слово `MyBase` используется производными классами при вызове методов базового класса. Такая возможность бывает полезной в том случае, если производный класс переопределяет или маскирует метод базового класса, который требуется вызвать.

Ключевое слово `MyClass` решает противоположную задачу. Если в базовом классе имеется функция, которая должна вызывать функцию базового класса (а не версию производного класса — возможно, переопределенную), ключевое слово `MyClass` гарантирует, что будет вызвана именно версия базового класса.

Приложение MyBaseMyClass (листинг 10.4) демонстрирует сказанное.

Листинг 10.4. Приложение MyBaseMyClass

```
Module Module1

  Class A
    Overridable Sub Test()
      console.WriteLine ("A.Test called")
    End Sub
    Sub ACallsTest()
      console.Write ("A Calls Test directly: ")
      Test()
      console.Write ("A Calls MyClass.Test: ")
      MyClass.Test()
    End Sub

  End Class

  Class B
    Inherits A
    ' Что произойдет, если объявить Test
    ' с ключевым словом Shadows вместо Overrides?
    Overrides Sub Test()
      console.WriteLine ("B.Test called")
    End Sub
    Sub BCallsTest()
      console.Write ("B Calls Test directly: ")
      Test()
      console.Write ("B Calls MyBase.Test: ")
      MyBase.Test()
    End Sub
  End Class

  Sub Main()
    Dim aref As New A()
    Dim bref As New B()
    console.WriteLine ("Object A")
    aref.ACallsTest()
```

```
    console.WriteLine ("Object B")
    bref.ACallsTest()
    console.WriteLine ("Object B")
    bref.BCallsTest()
    console.ReadLine()
  End Sub

End Module
```

Результат выглядит следующим образом:

```
Object A
A Calls Test directly: A.Test called
A Calls MyClass.Test: A.Test called
Object B
A Calls Test directly: B.Test called
A Calls MyClass.Test: A.Test called
Object B
B Calls Test directly: B.Test called
B Calls MyBase.Test: A.Test called
```

Первые три строки очевидны: класс A вызывает свой собственный метод в любом случае, как напрямую, так и через ключевое слово `MyClass`.

С четвертой и пятой строкой дело обстоит несколько сложнее. Объект является экземпляром класса B, однако вызывается метод класса A, унаследованный классом B. При первом вызове `Test` будет вызвана переопределенная версия. Чтобы вызвать исходную версию `Test` класса A, код класса A должен воспользоваться ключевым словом `MyClass`.

С двумя последними строками опять все просто. Когда функция `Test` вызывается из класса B, при отсутствии ключевого слова `MyBase` будет вызван метод класса B.

В качестве эксперимента попробуйте сменить атрибут функции `Test` в классе B с `Overrides` на `Shadows`. Результат выглядит так:

```
Object A
A Calls Test directly: A.Test called
A Calls MyClass.Test: A.Test called
Object B
A Calls Test directly: A.Test called
A Calls MyClass.Test: A.Test called
Object B
B Calls Test directly: B.Test called
B Calls MyBase.Test: A.Test called
```

Различия проявляются при вызове метода `ACallsTest` из объекта производного класса B. В обычных условиях при вызове `Test` из `ACallsTest` вызывается переопределенная функция `Test` производного класса, но при указании ключевого слова `Shadows` вызывается метод базового класса.

Ключевое слово `Overrides` означает, что метод производного класса должен вызываться вместо метода базового класса через ссылки как на базовый, так и производный класс. Ключевое слово `Shadows` означает, что метод производного класса заменяет метод базового класса лишь при вызове через производный класс.

Квалификаторы `MyClass` и `MyBase` не являются ссылками на классы и не могут передаваться в качестве параметров. Они не нарушают правил видимости;

в частности, при помощи MyBase невозможно обратиться к закрытой переменной базового класса. Они могут применяться как к общим членам, так и к членам конкретных экземпляров.

При использовании MyBase член класса не обязан присутствовать непосредственно в базовом классе: VB .NET просматривает цепочку наследования и находит в ней первый метод с заданным именем.

Вложенные классы

К числу приятных новшеств VB .NET относится возможность определения вложенных классов и структур в других классах. Вложенные классы и структуры прекрасно подходят для упорядочения данных в классах (особенно больших и сложных).

Впрочем, многие программисты предпочитают работать с относительно малыми и простыми объектами, поэтому вложенные классы обычно применяются реже, чем можно было бы ожидать.

Область видимости вложенных классов определяется по тем же правилам, как и для других членов. Например, закрытый вложенный класс недоступен за пределами своего класса-контейнера. Visual Basic .NET не позволяет расширять видимость вложенного класса посредством обращения к нему через свойства или возвращаемые значения методов. Другими словами, в классе нельзя объявить Public-метод, возвращающий экземпляр Private- или Friend-класса.

Методы и свойства

Итак, мы рассмотрели область видимости классов и их членов, а также некоторые концепции .NET, необходимые для понимания правил видимости. Следующим шагом станут методы и свойства, при помощи которых объекты решают полезные задачи.

Перегрузка функций

К числу наиболее впечатляющих новых возможностей VB .NET относится поддержка перегружаемых функций. Представьте себе, что вы разместили в классе или модуле VB три объявления функций:

```
Sub Print (ByVal X As Integer)
Sub Print (ByVal X As String)
Sub Print (ByVal X As SomeObject)
```

В VB6 компилятор выдает ошибку, поскольку программа не может содержать функции или процедуры с одинаковыми именами. В VB .NET такие объявления вполне допустимы. Чтобы явно указать на выполняемую перегрузку, можно воспользоваться необязательным ключевым словом Overloads[1]:

```
Overloads Sub Print (ByVal X As Integer)
Overloads Sub Print (ByVal X As String)
Overloads Sub Print (ByVal X As SomeObject)
```

[1] При определении в классе одноименных методов с разными сигнатурами перегрузка происходит по умолчанию и ключевое слово Overloads указывать не нужно.

В сущности, любая функция или процедура может существовать в любом количестве версий. При этом должны выполняться только два обязательных правила: во-первых, все функции должны возвращать результат одного типа, а во-вторых, вызываемая функция должна однозначно определяться на основании переданных параметров. Перегруженные функции не только могут вызываться с параметрами разного типа, но и могут содержать разное количество параметров.

Перегрузка функций часто используется в .NET Framework. Например, функция `Console.WriteLine`, часто встречающаяся в примерах программ, определяется следующим образом:

```
Public Overloads Shared Sub WriteLine()
Public Overloads Shared Sub WriteLine(Boolean)
Public Overloads Shared Sub WriteLine(Char)
Public Overloads Shared Sub WriteLine(Char())
Public Overloads Shared Sub WriteLine(Decimal)
Public Overloads Shared Sub WriteLine(Double)
Public Overloads Shared Sub WriteLine(Integer)
Public Overloads Shared Sub WriteLine(Long)
Public Overloads Shared Sub WriteLine(Object)
Public Overloads Shared Sub WriteLine(Single)
Public Overloads Shared Sub WriteLine(String)
Public Overloads Shared Sub WriteLine(UInt32)
Public Overloads Shared Sub WriteLine(Uint64)
Public Overloads Shared Sub WriteLine(String, Object)
Public Overloads Shared Sub WriteLine(String, Object())
Public Overloads Shared Sub WriteLine(Char(), Integer, Integer)
Public Overloads Shared Sub WriteLine(String, Object, Object)
Public Overloads Shared Sub WriteLine(String, Object, Object, Object)
```

VB .NET просматривает список параметров, переданных при вызове VB .NET, и вызывает наиболее подходящую версию.

Перегрузка функций хороша прежде всего тем, что она упрощает приложение. Что проще запомнить: одну функцию `WriteLine` с разными наборами параметров или пятнадцать разных функций с именами `WriteStringLine`, `WriteIntLine` и т. д.?

Хотя ключевое слово `Overloads` при перегрузке методов класса необязательно, оно играет важную роль при работе с производными классами. При определении в производном классе метода, имя которого совпадает с именем метода базового класса, возможны три ситуации.

1. Методы базового и производного классов обладают одинаковыми сигнатурами (типами параметров и возвращаемого значения).
   - Ключевое слово `Overrides` обеспечивает полиморфный вызов (метод производного класса всегда вызывается даже через ссылку на базовый класс). Если в базовом классе существуют другие методы с тем же именем и другими параметрами, используйте `Overrides` и `Overloads`.
   - Ключевые слова `Overrides` и `NotOverridable` обеспечивают полиморфный вызов с запретом переопределения метода дальнейшими производными классами. Если в базовом классе существуют другие методы с тем же именем и другими параметрами, используйте `NotOverridable`, `Overrides` и `Overloads`.

   - Ключевое слово `Shadows` (по умолчанию) обеспечивает вызов метода в зависимости от типа ссылки. Через ссылки на производный класс всегда вызывается метод производного класса, а все одноименные методы базового класса маскируются. Через ссылки на базовый класс всегда вызываются методы базового класса.

2. Методы базового и производного классов обладают разными сигнатурами (типами параметров и возвращаемого значения).
   - Ключевое слово `Shadows` (по умолчанию) обеспечивает вызов метода в зависимости от типа ссылки. Через ссылки на производный класс всегда вызывается метод производного класса, а все одноименные методы базового класса маскируются. Через ссылки на базовый класс всегда вызываются методы базового класса.
3. Методы базового и производного классов различаются только по типам параметров.
   - Ключевые слова `Oveloads` обеспечивает перегруженный вызов метода производного и базового классов в зависимости от типа используемых параметров.
   - Ключевое слово `Shadows` (по умолчанию) обеспечивает вызов метода на основании типа ссылки. Через ссылки на производный класс всегда вызывается метод производного класса, а все одноименные методы базового класса маскируются. Через ссылки на базовый класс всегда вызываются методы базового класса.

Ниже приведен ряд относительно простых правил, помогающих лучше разобраться в смысле ключевых слов `Overrides`, `Shadows` и `Overloads`.

- Если класс не является производным от другого класса, об этих ключевых словах вообще можно не беспокоиться. Если в классе определяются два и более методов, различающихся только по типу параметров, перегрузка происходит автоматически.
- Используйте ключевое слово `Shadows` везде, где требуется создать производный класс и скрыть методы базового класса. Вызовы через объект базового класса никогда не будут передаваться методам производного класса, а вызовы через объект производного класса никогда не будут передаваться методам базового класса.
- Используйте ключевое слово `Overrides` в случае, когда метод должен вызываться в зависимости от истинного типа объекта (независимо от того, к какому типу относится переменная, используемая для ссылки на объект, — базовому или производному).
- Используйте ключевое слово `Overloads` при добавлении метода, имя которого совпадает с именем метода базового класса, но набор параметров отличается от всех одноименных методов базового класса. При использовании ключевого слова `Overloads` в производном классе вам придется включать его в объявления всех методов с этим именем.
- Используйте ключевое слово `NotOverridable`, чтобы показать, что метод не может переопределяться в производных классах.

Все еще сомневаетесь? Не беда, компилятор VB .NET поможет во время ввода программы. Следуя его указаниям, я неизменно приходил к желаемому результату.

К перегрузке привыкаешь довольно быстро, однако встречаются и нетривиальные ситуации, особенно если учесть, что разные перегруженные методы могут обладать разной областью видимости.

Конструкторы

Конструктором называется функция, вызываемая при создании объекта. В VB .NET конструктору присваивается имя `New` и его общее объявление выглядит так:

```
Public Sub New()
  MyBase.New()
End Sub
```

Конструктор, вызываемый без параметров, называется конструктором по умолчанию. Для структур он определяться не может, поскольку у структур имеется свой конструктор по умолчанию, обнуляющий все поля структуры.

Обычно в конструкторе присутствует строка `MyBase.New()`, обеспечивающая вызов конструктора базового класса. Конструкторы не наследуются производными классами.

Конструкторы, как и другие методы, могут перегружаться. Это позволяет не только определять разные варианты инициализации объекта, но и управлять самим процессом его создания.

Область видимости конструктора

Каталог Overloads содержит подкаталоги двух проектов: библиотеки классов ClassExamples и консольного приложения OverloadsTest. В библиотеке ClassExamples определяются три класса. Первый класс, `InternalClass1`, имеет два конструктора: конструктор по умолчанию объявлен закрытым, а конструктор с параметром объявлен открытым. Поскольку оба конструктора принадлежат одному классу, указывать ключевое слово `Overloads` не нужно. Класс `InternalClass1` приведен в листинге 10.5.

Листинг 10.5. Библиотека ClassExamples

```
Public Class InternalClass1
  Private m_Name As String
  Shared Sub New()
    Console.WriteLine ("Shared constructor called")
  End Sub

  Private Sub New()
    m_Name = "Default"
  End Sub

  Public Sub New(ByVal NewName As String)
    m_Name = NewName
  End Sub

  Public Shared Function GetDefaultObject() As InternalClass1
    Return New InternalClass1()
  End Function
```

продолжение ⇩

Листинг 10.5 (продолжение)

```
  Public Sub Test()
    Console.WriteLine ("Test in InternalClass1, name is: " & m_Name)
  End Sub
End Class
```

Для чего нужен закрытый конструктор, который может вызываться только в пределах класса?

Закрытые конструкторы полезны в тех случаях, когда объект должен создавать дополнительные экземпляры своего класса или если вы хотите, чтобы создание объекта выполнялось `Shared`-методом класса (см. метод `GetDefaultObject` в проекте Overloads).

В чем различие между параметризованным конструктором и созданием структуры `Shared`-методом?

Единственное различие заключается в том, что при использовании параметризованного конструктора происходит фактическое создание объекта. Для сложных объектов производных классов создание (как и деинициализация) может быть связано с немалыми затратами.

Если нет окончательной ясности в том, будет ли создан объект (например, если создание объекта разрешается лишь для некоторых наборов параметров), лучше воспользоваться `Shared`-методом. В случае ошибки этот метод может вернуть `Nothing` или инициировать исключение, обходясь без фактического создания объекта.

Более того, вы можете объявить закрытыми все конструкторы и разрешить создание объекта только `Shared`-методом. Поскольку сам класс объявлен с атрибутом `Public`, проблем с видимостью объекта и его методов за пределами файла и даже сборки не будет. Но при отсутствии открытого конструктора объекты этого типа не могут создаваться оператором `New`! Для создания может использоваться только `Shared`-метод.

Пример InternalClass2 расширяет эту идею и определяет конструктор по умолчанию с атрибутом `Friend`.

```
Public Class InternalClass2
  Friend Sub New()
    MyBase.New()
  End Sub

  Public Overridable Sub Test()
    Console.WriteLine ("InternalClass2 Test called")
  End Sub

  Public Sub Test(ByVal x As Integer)
    Console.WriteLine ("InternalClass2 Test(ByVal x as integer) called")
  End Sub
End Class
```

В классе `Class1` определяется метод для создания и возвращения объектов класса `InternalClass2`:

```
Public Class Class1
  Public Function GetInternalClass2() As InternalClass2
    ' Здесь можно провести дополнительную инициализацию
    ' и даже выполнить проверку, связанную с безопасностью.
    Return New InternalClass2()
  End Function
End Class
```

Класс `InternalClass3` показывает, как использовать ключевое слово `Overloads` для перегрузки методов базового класса.

```
Public Class InternalClass3
  Inherits InternalClass2

  Public Overloads Overrides Sub Test()
    Console.WriteLine ("InternalClass3() Test called")
  End Sub
  Public Overloads Sub Test(ByVal S As String)
    Console.WriteLine ("InternalClass3 Test(ByVal s as String) called")
  End Sub

End Class
```

Программа OverloadsTest находится в отдельной сборке. Это означает, что она не сможет создавать объекты `InternalClass2` напрямую, поскольку не имеет доступа к конструктору, объявленному с ключевым словом `Friend`. Программа OverloadsTest (листинг 10.6) показывает, как контролировать возможность создания объектов при помощи ограничения видимости конструктора.

Листинг 10.6. Программа OverloadsTest

```
Module Module1
  Sub Main()
    ' Создать объект при помощи закрытого
    ' конструктора невозможно.
    'Dim c As New ClassExamples.InternalClass1()

    ' Можно создавать открытым конструктором.
    Dim c As New ClassExamples.InternalClass1("MyTestName")
    c.Test()

    ' К закрытому конструктору можно обратиться
    ' при помощи открытого метода.
    Dim c2 As ClassExamples.InternalClass1 = _
      ClassExamples.InternalClass1.GetDefaultObject

    ' Объекты класса InternalClass2
    ' никогда не создаются напрямую.
    'Dim c2 As New ClassExamples.InternalClass2()

    ' Однако это можно сделать при помощи другого класса
    ' и конструктора Friend.
    Dim cls1 As New ClassExamples.Class1()
    Dim c3 As ClassExamples.InternalClass2 = cls1.GetInternalClass2
    c3.Test()
    Console.WriteLine (ControlChars.CrLf & "InternalClass3")
    Dim cls3 As New ClassExamples.InternalClass3()
    cls3.Test ("hello")
    cls3.Test (5)
    cls3.Test()
    Console.ReadLine()
  End Sub

End Module
```

Как видите, мы можем создавать объекты класса `InternalClass3` даже несмотря на то, что этот класс является производным от `InternalClass2` (объекты

которого создаваться не могут из-за отсутствия открытого конструктора). Это объясняется тем, что конструкторы не наследуются, поэтому ограниченная область видимости базового конструктора не влияет на производный класс.

Общие конструкторы

Помимо конструкторов экземпляров, в классах могут определяться общие (`Shared`) конструкторы. Общий конструктор вызывается при создании первого объекта заданного типа. Определение общих конструкторов в классах продемонстрировано в примере InitAndDestruct из главы 5 и в примере Overloads этой главы.

Снова о создании экземпляров

Классы VB .NET не поддерживают свойства `Instancing`. Впрочем, это не означает, что они не обладают соответствующей функциональностью, просто она реализуется другими средствами.

Область видимости объекта (в пределах сборки или везде) определяется атрибутами класса, то есть его объявлением с ключевым словом `Public` или `Friend`.

Возможность создания экземпляров зависит от области видимости конструкторов объекта. Ограничивая область видимости конструкторов ключевыми словами `Private` или `Friend`, можно запретить создание объектов за пределами сборки.

Глобальная видимость — возможность обращения к объекту без уточнения имени — достигается импортированием пространства имен, в котором определен этот класс, командой `Imports`[1].

Автоматическое создание объекта при обращении не поддерживается. Для создания объекта необходимо использовать команду `New`.

Методы и свойства

Иногда бывает трудно решить, как лучше реализовать тот или иной член класса: в виде свойства или в виде метода? Такие затруднения вполне понятны, ведь параметризованное свойство может сделать все, что делают методы. С позиций функциональности не существует *никаких реальных различий* между обращением к свойству, доступному для чтения, и вызовом метода, если в обоих случаях возвращается нужное значение.

Однако этот факт помогает принять решение в ситуациях, когда значение не возвращается.

- Правило: если вы хотите выполнить операцию, которая **не возвращает** значения, используйте метод.

В сущности, это единственное правило, а все перечисленное ниже — не более чем настоятельные рекомендации.

---

[1] Помните о различиях между импортированием пространства имен и включением ссылки в проект? Включение ссылки дает возможность обращаться к объектам соответствующей сборки. Последующее импортирование пространства имен или любых его объектов позволяет обращаться к ним без указания полностью уточненного имени.

Используйте процедуры свойств

При предоставлении доступа к простому элементу данных (скажем, к целому числу или строке) вместо синтаксических конструкций вида:

```
Public Class SomeClass
  Public A As Integer
End Class
```

следует использовать процедуры свойств:

```
Public Class SomeClass
  Private m_B As Integer
  Public Property B() As Integer
    Get
      Return (m_B)
    End Get
    Set (ByVal Value As Integer)
      m_B = value
    End Set
  End Property
End Class
```

Почему? Потому что такое решение при небольшом объеме дополнительного кода расширяет ваши возможности. Например, позднее вы сможете легко реализовать проверку присваиваемых значений. В VB6 этот выбор особенно важен, поскольку переход от открытых переменных к процедурам свойств сопровождается нарушением совместимости. В VB .NET дело упрощается тем, что связывание осуществляется в процессе работы JIT-компилятора, поэтому переход на процедуры свойств не отразится на работе существующего кода. Тем не менее это полезная привычка.

Впрочем, с полной уверенностью можно сказать лишь одно: во всех ситуациях, когда вам приходится выбирать между открытой переменной и процедурой свойства, предпочтение однозначно следует отдавать синтаксису свойств, а не синтаксису методов.

Не усложняйте свойства

Хотя свойства классов могут быть параметризованными, обычно эта возможность используется лишь при создании индексаторов (indexers). В этом случае свойству передается один целочисленный параметр, по которому из массива или набора данных выбирается нужный элемент. Если в вашем случае одного параметра оказывается недостаточно, вероятно, вместо свойства стоит воспользоваться методом.

Старайтесь избегать свойств, возвращающих массивы, особенно если этот массив фиксирует текущее состояние неких данных и его содержимое может стать недействительным в ходе дальнейшей работы программы.

Избегайте побочных эффектов

Если обращение к члену класса связано с некими побочными эффектами (например, если при этом изменяется сразу несколько переменных класса), вместо свойств рекомендуется использовать методы. Обычно процедуры свойств должны ограничиваться проверкой данных, диагностикой ошибок и оповещением об

изменениях свойств при помощи событий или другого механизма. Единственное общепринятое исключение — ситуация, когда само свойство тем или иным способом изменяет поведение объекта. В этом случае некоторые программисты предпочитают использовать процедуры свойств.

Другие случаи использования методов

Методы часто применяются для преобразования типов данных.

Кроме того, методы используются и в тех случаях, когда важен порядок выполнения. Большинство программистов считает, что чтение/запись свойств может происходить в произвольной последовательности, тогда как методы класса обычно подразумевают более четкий порядок вызова.

Процедуры свойств

В VB .NET процедуры свойств несколько изменились по сравнению с VB6. Главное изменение уже неоднократно встречалось в примерах, приводившихся выше. Речь идет об изменении синтаксиса: процедуры `Get` и `Set` объединяются, а процедура `Let` исчезла вместе с различиями между объектами и переменными (вспомните: в VB .NET любая переменная является объектом). Атрибуты `ReadOnly` и `ReadWrite` указывают на то, является ли свойство доступным только для чтения или записи.

Синтаксис, принятый в VB .NET, отличается большей наглядностью. Процедуры `Get` и `Set` всегда расположены поблизости друг от друга. С другой стороны, я подозреваю, что любой мало-мальски компетентный программист VB6 старается не отдалять процедуры `Property Get` и `Property Set`.

Вероятно, основное недовольство у программистов VB6 вызовет тот факт, что в новом варианте синтаксиса процедуры `Property Set/Let` и `Property Get` должны обладать одинаковой видимостью. В VB6 очень часто встречались ситуации с объявлениями `Public Property Get` и `Private/Friend Property Set`. Значения свойств, созданных таким образом, могли задаваться только внутри компонента или приложения, однако чтение разрешалось и внешним клиентам.

Честно говоря, меня это изменение тоже не радует, но обвинять в этом разработчиков VB .NET было бы несправедливо. Требование одинаковой видимости процедур `Property Get` и `Property Set` входит в спецификацию CLS (Common Language Specification) и действует не только в VB .NET, но и в C# и .NET-версии C++.

Таким образом, если только Microsoft не пойдет на изменение CLS, я рекомендую использовать открытые процедуры свойств с атрибутом `ReadOnly` и отдельные методы с видимостью `Friend` или `Private` для задания внутренних значений свойств.

Свойства по умолчанию

Рассмотрим следующую команду VB6:

```
myVariable = "Hello"
```

Что делает эта команда? Присваивает строку переменной типа `String`?

А если переменная myVariable является объектом? Приведет ли это к ошибке времени выполнения? А если у этого объекта имеется свойство по умолчанию, будет ли строка присвоена ему? И какое именно свойство будет использоваться по умолчанию?

А если переменная myVariable относится к универсальному типу Variant? Что делает эта команда: присваивает переменной строку? А может, в универсальной переменной хранится объект, у которого имеется свойство по умолчанию?

Мы не знаем.

Чтобы ответить на эти вопросы, необходимо знать не только тип переменной myVariable, но и то, является ли эта переменная объектом, какова структура этого объекта и есть ли у него свойства по умолчанию.

А теперь рассмотрим ту же строку в VB .NET.

Переменной myVariable типа String или Object присваивается ссылка на объект String, содержащий значение «Hello».

Все. Никакой неоднозначности, никакой путаницы.

Свойства по умолчанию в VB6 обходятся слишком дорого. Они экономят программистам лишь несколько нажатий клавиш, но за это приходится расплачиваться многими часами отладки, а то и затратами на долгосрочное техническое сопровождение программы.

В своей практике я не припоминаю ни одного случая сознательного использования класса со свойствами по умолчанию. Каждый раз, когда я включал свойство по умолчанию в свои элементы, я об этом жалел. Работая со стандартными объектами (такими, как текстовые поля), я всегда использую полную запись вида txtEditControl.Text = "string" вместо того, чтобы полагаться на свойства по умолчанию.

В VB .NET свойства по умолчанию могут быть только параметризованными (поскольку на параметризованные свойства не распространяется неоднозначность простого присваивания). Чаще всего эта возможность используется при создании индексаторов.

Практическое использование свойств по умолчанию продемонстрировано в листинге 10.7.

Листинг 10.7. Приложение Properties

```
' Использование свойств по умолчанию
' Copyright ©2001 by Desaware Inc.

Module Module1

  Class DefaultTest
    Implements IDisposable

    Private Shared MyClasses As New ArrayList()

    Public Name As String

    Public Sub New(ByVal myname As String)
      MyBase.New()
      MyClasses.Add (Me)
      Name = myname
    End Sub
```

продолжение ⇙

Листинг 10.7 (продолжение)

```
    Default Public ReadOnly Property OtherDefaultTestObjects(ByVal _
        idx As Integer) As DefaultTest
      Get
        Return CType(MyClasses.Item(idx), DefaultTest)
      End Get
    End Property

    Public Shared ReadOnly Property OtherObjectCount() As Integer
      Get
        Return myclasses.Count
      End Get
    End Property

    Public ReadOnly Property MyIndex() As Integer
      Get
        Return myclasses.IndexOf(Me)
      End Get
    End Property

    Public Sub Dispose() Implements IDisposable.Dispose
      MyClasses.Remove (Me)
    End Sub

  End Class

  Sub ShowOtherObjects(ByVal obj As DefaultTest)
    Dim x As Integer
    Console.WriteLine ("This object is: " & obj.Name)
    Console.WriteLine ("Other objects are: ")
    For x = 0 To DefaultTest.OtherObjectCount - 1
      If x <> obj.MyIndex Then
        Console.WriteLine (obj(x).Name)
      End If
    Next
    Console.WriteLine()

  End Sub

  Sub Main()
    Dim obj1 As New DefaultTest("Firstobject")
    Dim obj2 As New DefaultTest("Secondobject")
    Dim obj3 As New DefaultTest("Thirdobject")

    ShowOtherObjects (obj2)
    obj2.Dispose()
    obj2 = Nothing
    ShowOtherObjects (obj3)
    Console.ReadLine()
  End Sub

End Module
```

В приложении Properties продемонстрированы некоторые приемы, упоминавшиеся в главах 4 и 5, а также в этой главе.

Класс DefaultTest показывает, как организовать отслеживание всех созданных экземпляров класса с минимальными затратами со стороны клиентского кода.

Общая переменная myClasses типа ArrayList содержит список созданных объектов. Объекты включаются в список при вызове конструктора.

Свойство по умолчанию OtherDefaultTestObjects представляет собой индексатор, возвращающий ссылки на объекты из внутреннего списка. С его помощью любой объект может получить список всех созданных объектов DefaultTest.

Общее свойство OtherObjectCount, доступное только для чтения, возвращает текущее количество объектов в коллекции myClasses. Выбор между свойством и методом в данном случае оказывается непростым. Поскольку свойство объявлено с атрибутом Shared, к нему можно обращаться как через экземпляр объекта DefaultTest, так и непосредственно в записи вида DefaultTest.OtherObjectCount (как показывает метод ShowOtherObjects). Свойство MyIndex возвращает индекс заданного экземпляра в коллекции myClasses.

Очень важный метод Dispose должен вызываться клиентом при присваивании экземпляру значения Nothing. Почему? Разве VB .NET не обеспечивает автоматического освобождения объектов?

Да, VB .NET автоматически освобождает объекты, на которые не существует ссылок, а точнее, объекты, не присутствующие в цепочках, начинающихся от корневых переменных. Однако коллекция myClasses *является* корневой переменной! Следовательно, если ссылка на объект встречается в массиве, этот объект не освобождается. Метод Dispose удаляет объект из коллекции myClasses, обеспечивая возможность его нормального освобождения.

Другими словами, VB .NET прекрасно справляется с освобождением переменных, на которые в вашем приложении отсутствуют ссылки, но это не помешает вам устроить утечку памяти, если ваше приложение не позаботится о должном освобождении ненужных объектов. Процедура Main показывает, как организовать освобождение компонентов, реализующих интерфейс IDisposable[1].

Метод ShowOtherObjects демонстрирует использование свойства по умолчанию.

Параметризованные свойства

Параметризованным свойствам все параметры передаются по значению. Это сделано для того, чтобы избежать возможных побочных эффектов от модификации параметров в процедуре свойства. Я знаю, что некоторые программисты не одобрили этого изменения, но, на мой взгляд, во всех ситуациях, когда вызывающая функция должна модифицировать параметр, лучше использовать метод вместо свойства.

Передача свойств по ссылке

Оказывается, требование о передаче всех параметров процедур свойств по значению обладает некоторыми дополнительными преимуществами.

---

[1] Возможно, вы обратили внимание на то, что в формах метод Dispose помечается ключевыми словами Overloaded и Overrides. Дело в том, что класс Form является производным от класса Control, который не только реализует метод Dispose (который вы можете переопределить), но реализует и вторую версию Dispose с параметром логического типа — отсюда необходимость в ключевом слове Overloads.

Рассмотрим пример PropByRef, приведенный в листинге 10.8.

Листинг 10.8. Приложение PropByRef

```
' Приложение PropByRef
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Class PropClass
  Private m_Member As String
  ' Ключевое слово Overloads в данном примере необязательно.
  Overloads Property Member() As String
    Get
      Return m_Member
    End Get
    Set(ByVal Value As String)
      m_Member = Value
    End Set
  End Property

  Overloads Property Member(ByVal x As Integer) As String
    Get
      Return (m_Member)
    End Get
    Set(ByVal Value As String)
      m_Member = Value & " called with " & x.ToString()
    End Set
  End Property

End Class

Module Module1

  Sub FunctionSetsString(ByRef s As String)
    s = "Hello"
  End Sub

  Sub Main()
    Dim obj As New PropClass()
    FunctionSetsString (obj.Member)
    Console.WriteLine (obj.Member)
    FunctionSetsString (obj.Member(5))
    Console.WriteLine (obj.Member)
    Console.ReadLine()
  End Sub

End Module
```

Класс `PropClass` содержит закрытую строковую переменную, для обращения к которой используются две перегруженные функции с именами `Member`. Обе версии `Member` возвращают содержимое закрытой строковой переменной, но присваивают ей разные значения. Параметризованный метод `Member` просто присваивает строку; второй метод `Member` вместе со строкой сохраняет значение целочисленного параметра. Это позволяет определить, какой из перегруженных методов задал значение свойства.

Метод `FunctionSetsString` получает по ссылке одну строковую переменную и присваивает ей значение «Hello».

Процедура `Main` вызывает функцию `FunctionSetsString` и передает ей по ссылке свойство `Member` класса `PropClass`.

В VB6 при этом создается временная переменная, которой задается значение свойства `Member`. Именно эта временная переменная передается в качестве параметра функции `FunctionSetsString`. В результате, хотя параметр передавался по ссылке, свойство `Member` класса `PropClass` так и не изменяется функцией `FunctionSetsString`, модифицируется только временная переменная.

Однако в VB .NET выводится следующий результат:

```
Hello
Hello called with 5
```

Мы видим, что свойство не только правильно передается по ссылке, но и передается его нужная версия со всеми параметрами, используемыми при обращении к свойству.

Другими словами, происходит следующее.

- VB .NET читает `Member` и присваивает значение временной переменной.
- VB .NET вызывает `FunctionSetsString`.
- Полученное значение задается свойству той же версией `Member` и с теми же параметрами, которые использовались при чтении.

Конечно, для сколько-нибудь надежного выполнения этой операции должны выполняться два условия.

1. Параметры процедуры свойства должны оставаться неизменными. Модификация параметров между моментами чтения и записи может стать причиной разнообразных ошибок, от простых недоразумений до серьезной порчи данных. Например, если параметр используется при индексации внутреннего массива, при выходе из `FunctionSetsString` значение может быть присвоено другому элементу.
2. Область видимости процедур `Get` и `Set` должна быть идентичной. В противном случае возможны ошибки времени выполнения, зависящие от того, откуда исходит вызов и какие методы находятся в области видимости.

Не берусь утверждать, что требование о передаче всех параметров процедур свойств по значению предназначалось именно для того, чтобы свойства можно было передавать по ссылке. Также не могу сказать, в какой степени это повлияло на требование об одинаковой области видимости процедур свойств `Get` и `Set`. Если дело обстояло действительно так, я бы сказал, что за эту возможность пришлось заплатить довольно дорого. С другой стороны, вполне вероятно, что эти два требования сначала были установлены по совершенно иным соображениям, а потом выяснилось, что в качестве побочного эффекта это позволяет передавать свойства по ссылке.

Из того, что мне известно о Microsoft, могу уверенно сказать лишь одно: эти решения появились в результате долгих и упорных дебатов между разработчиками.

События и делегаты

Вероятно, некоторые читатели уже интересуются, когда я доберусь до событий. В других книгах о VB .NET речь о них зашла бы гораздо раньше, но у меня были свои причины, чтобы не торопиться с этой темой.

Во-первых, события VB .NET в простейшем случае работают практически так же, как и в VB6. Следующий фрагмент демонстрирует обработку событий мыши в VB .NET.

```
Private WithEvents button1 As System.Windows.Forms.Button

Me.button1 = New System.Windows.Forms.Button()

Private Sub button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
  MsgBox("Simple Event Clicked", MsgBoxStyle.Information, _
  "Event arrived")
End Sub
```

В VB .NET формы, кнопки и другие элементы всего лишь представляются разными типами классов. Как и любые другие классы, они могут инициировать события. Таким образом, переменная, представляющая кнопку, объявляется с ключевым словом `WithEvents`, а затем соответствующий объект создается командой `New`.

Синтаксис обработчиков событий тоже несколько изменился. Вместо невесть откуда взявшегося обработчика с именем `button1_Click` (как это было в VB6) вы создаете процедуру с произвольным именем и при помощи ключевого слова `Handles` указываете, какое событие она обрабатывает. Другими словами, приведенную выше процедуру `button1_Click` можно определить в следующем виде:

```
Private Sub OhNoTheButtonWasClicked(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
  MsgBox("Simple Event Clicked", MsgBoxStyle.Information, _
  "Event arrived")
End Sub
```

Имя процедуры значения не имеет. Мастер Form Designer Wizard автоматически создает процедуру с именем *элемент_событие*, но имя может быть любым — важно лишь имя события, указанное после ключевого слова `Handles`.

Следовательно, в VB .NET используются те же базовые принципы обработки событий, как и в VB6, поэтому не стоило особенно торопиться с ее описанием в предыдущих главах.

Однако в действительности механизм обработки событий в .NET кардинально отличается не только от VB6, но и от обработки событий в COM вообще. Чтобы понять, как работают события, необходимо хорошо разбираться в том, как организовано взаимодействие объектов в .NET — и это другая причина, по которой я рассматриваю события только в этой главе.

К сожалению, документация о работе событий (по крайней мере, с моей точки зрения) написана совершенно невразумительно. Искренне надеюсь, что я справлюсь с этой задачей получше.

Чтобы представить события VB .NET в практическом контексте, мы сначала кратко рассмотрим основные принципы обработки событий в VB6 и COM. При этом я не стремлюсь дать общее представление о том, как работают события за пределами .NET, а просто хочу свести воедино все, что вы уже знаете, чтобы вам было проще это «все» забыть. Только тогда вы поймете, как же организована обработка событий в .NET.

События, функции обратного вызова и COM

Что такое «событие»?

Трудно сказать. Зато мы твердо знаем, что при инициировании события вызывается соответствующий метод вашего объекта.

Чем непосредственный вызов метода отличается от вызова по событию?

С сугубо концептуальной точки зрения — это всего лишь вопрос восприятия.

Рассмотрим два объекта. Клиентский объект является частью вашей программы и создает серверный объект для выполнения некоторой задачи (рис. 10.3). Серверный объект при необходимости может передать клиенту некоторую информацию. Концептуальное различие между вызовом метода и событием связано с направлением вызова. Вызывая метод из своего приложения, вы сами определяете, когда это происходит, то есть вызов входит в нормальную последовательность выполнения программы. События можно рассматривать как вызовы методов, не входящие в нормальную последовательность выполнения, на которые ваше приложение должно определенным образом отреагировать.

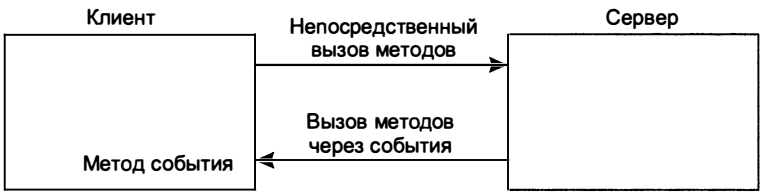


Рис. 10.3. Простейшая схема обработки событий

В VB6 поддерживаются два механизма обработки событий. Первый, стандартный механизм, используется при определении событий в синтаксисе `Event` и ссылки на серверный объект с ключевым словом `WithEvents`. Этот механизм рассматривается ниже. Второй механизм называется механизмом обратного вызова OLE (OLE callback). При использовании этого механизма клиентский объект передает серверу ссылку на самого себя, через которую серверный объект может напрямую вызывать методы клиентского объекта.

Мы не будем подробно обсуждать преимущества и недостатки этих двух механизмов, а также тонкости их работы. Главное, что необходимо понять, — что при использовании обратных вызовов OLE сервер получает ссылку на клиентский объект. Поскольку клиент тоже содержит ссылку на сервер, возникает классическая ситуация с циклическими ссылками. Это означает, что до освобождения одной из ссылок ни один из этих объектов не может быть уничтожен.

Хотя в некоторых ситуациях подобный механизм работает достаточно хорошо, не стоит и говорить, что для работы COM необходим механизм событий, не подверженный проблеме циклических ссылок. Этот механизм основан на так называемых *точках подключения* (connection points), при помощи которых объект сообщает информацию о событиях, которые он может инициировать. Эта информация состоит из имени события и списка параметров. Объект ожидает, что клиент (приемник события) предоставит методы с соответствующими параметрами. Точки подключения реализуются при помощи стандартных интерфейсов COM, которые не только предоставляют информацию о событиях, но и определяют

способ ведения сервером списка всех подключенных клиентов, чтобы одно событие могло инициироваться сразу в нескольких клиентах. Но самое важное заключается в том, что эти интерфейсы требуют, чтобы серверные объекты хранили «слабую» ссылку на клиентское приложение. «Слабой» называется ссылка, которая позволяет серверу вызывать методы клиентского объекта, но не считается ссылкой на клиента с точки зрения механизма подсчета ссылок COM. Это позволяет нормально уничтожить клиентский объект даже в том случае, если он является приемником событий.

Механизм событий, основанный на точках подключения, весьма сложен.

К счастью, приложения .NET не страдают от проблемы циклических ссылок, что избавляет программиста от многих сложностей, связанных с обработкой событий в COM.

Конечно, в .NET можно воспользоваться механизмом обратного вызова и передавать серверу ссылки на клиентские объекты, однако в .NET определен более гибкий механизм, работающий на уровне отдельных методов.

Делегаты

Одной из основных составляющих любого механизма событий (COM, обратных вызовов OLE или .NET) является согласование между клиентом и сервером типов параметров и значений, возвращаемых методами событий. Другая составляющая — механизм для хранения ссылок на объект и вызова методов. Обе задачи решаются при помощи специального объекта, называемого делегатом (рис. 10.4).

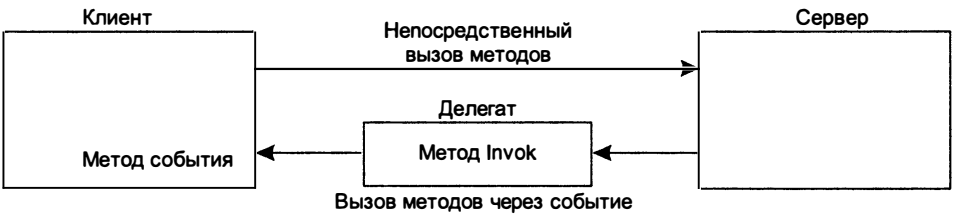


Рис. 10.4. Вызов методов при помощи делегата

Делегат определяет прототип или сигнатуру вызова метода, то есть точное описание типов параметров и возвращаемого значения[1]. При определении делегата с ключевым словом `Delegate` в действительности создается новый класс, производный от базового класса `Delegate`. Этот новый класс определяет типы параметров и возвращаемого значения для данного делегата.

Например, команда

```
Delegate Sub DelegateWithStringSignature(ByVal S As String)
```

из проекта Delegates определяет новый класс с именем `DelegateWithStringSignature`. Этот класс подходит только для методов, которые получают один строковый параметр и не имеют возвращаемого значения.

Оператор `AddressOf` в VB .NET возвращает делегата для заданного метода.

[1] Выражаясь точнее, сигнатуру определяют объекты, производные от базового класса Delegate.

Допустим, у нас имеется класс `CalledClass`, который определяется следующим образом:

```
Class CalledClass
  Public Sub WriteMessage(ByVal s As String)
    Console.WriteLine("Called Class Write Message of " & s)
  End Sub
End Class
```

Затем мы создаем делегата типа `DelegateWithStringSignature`:

```
Dim c As New CalledClass()
Dim d1 As DelegateWithStringSignature
d1 = AddressOf c.WriteMessage
```

Делегат, созданный этим фрагментом, не только имеет правильную сигнатуру (один строковый параметр), но и ссылается на метод `WriteMessage` конкретного объекта `CalledClass`.

Вызов метода `WriteMessage` этого объекта осуществляется любой из двух команд, приведенных ниже:

```
d1.Invoke("Test")
c.WriteMessage("Test")
```

Делегаты без объектов

Но при чем здесь события? Ведь класс `CalledClass` из приведенного выше примера не имеет к событиям никакого отношения?

Верно, однако механизм событий VB .NET основан именно на использовании делегатов. Давайте не будем торопиться и познакомимся с делегатами поближе, прежде чем разбираться, как они применяются по отношению к событиям.

Проект Delegates (листинг 10.9) показывает, что делегаты не обязательно связывать с конкретными объектами. Делегаты могут использоваться для вызова общих методов и даже функций/процедур модулей.

Листинг 10.9. Приложение Delegates

```
Class CalledClass
  Shared Sub SharedMessage(ByVal s As String)
    Console.WriteLine (_
      "Called CalledClass.SharedMessage with parameter: " & s)
  End Sub

  Public Sub WriteMessage(ByVal s As String)
    Console.WriteLine (_
      "Called CalledClass WriteMessage method with parameter: " & s)
  End Sub
End Class

Class OtherCalledClass
  Sub WriteMessage(ByVal s As String)
    Console.WriteLine ("Called OtherCalledClass " _
      & "WriteMessage method with parameter: " & s)
  End Sub
End Class

Module StdModule
  Sub ModuleWriteMessage(ByVal s As String)
```

продолжение ↛

Листинг 10.9 (продолжение)

```
    Console.WriteLine ("AddresssOf works in standard module too")
  End Sub
End Module
```

В процедуру `Sub Main` проекта Delegates входит следующий фрагмент:

```
Dim c As New CalledClass()
Dim o As New OtherCalledClass()
Dim BadObject As New ObjectWithNoWriteMessage()
Dim d1 As DelegateWithStringSignature
Dim obj As Object
Dim Params() As Object = {"DynamicParam"}

' Следующие две строки эквивалентны.
d1 = New DelegateWithStringSignature(AddressOf c.WriteMessage)
'd1 = AddressOf c.WriteMessage

d1.Invoke ("Test")
d1.DynamicInvoke (Params)
Dim d2 As DelegateWithStringSignature = AddressOf CalledClass.SharedMessage
Dim d3 As DelegateWithStringSignature = AddressOf o.WriteMessage
Dim d4 As DelegateWithStringSignature = AddressOf
StdModule.ModuleWriteMessage

d2.Invoke ("Test2")
d3.Invoke ("Test3")
d4.Invoke ("Test4")
```

Три делегата `d1`, `d2` и `d3` демонстрируют вызов метода конкретного объекта, общего метода класса и функции стандартного модуля. Метод `DynamicInvoke` позволяет динамически задавать параметры во время работы программы. Вскоре вы увидите, зачем это может понадобиться. Результат выглядит следующим образом:

```
Called CalledClass WriteMessage method with parameter: Test
Called CalledClass WriteMessage method with parameter: DynamicParam
Called CalledClass.SharedMessage with parameter: Test2
Called OtherCalledClass WriteMessage method with parameter: Test3
AddressOf works in standard module too
```

Позднее связывание

Для успешного вызова `Invoke` компилятору необходима дополнительная информация. Во-первых, он должен знать количество и типы параметров, используемых при вызове `Invoke`, а во-вторых — что объект *действительно* предоставляет метод, который может быть вызван делегатом.

В .NET Framework также поддерживается механизм позднего связывания, при котором объект и имя метода остаются неизвестными до времени выполнения программы. Делегаты работают даже при позднем связывании. Позднее связывание рассматривается в главе 11, а пока давайте посмотрим, как делегаты применяются в этом случае.

Ниже приведено определение нового класса, не поддерживающего метод `WriteMessage`:

```
Class ObjectWithNoWriteMessage
  Sub BadWriteMessage()
```

```
  End Sub
End Class
```

Также определяется тип делегата без параметров:

```
Delegate Sub DelegateWithNoParam()
```

В настоящем примере делегат создается методом `CreateDelegate`, который позволяет задать тип делегата, объект и имя метода следующим образом:

```
Console.WriteLine (ControlChars.CrLf & "Late binding example 1")

Dim d5 As DelegateWithStringSignature
Dim c As New CalledClass()
d5 = CType(System.Delegate.CreateDelegate(GetType(_
  DelegateWithStringSignature), c, "WriteMessage"),
  DelegateWithStringSignature)
d5.Invoke ("Test")
Try
  d5 = CType(System.Delegate.CreateDelegate(GetType(_
    DelegateWithStringSignature), BadObject, _
     "WriteMessage"), DelegateWithStringSignature)
  d5.Invoke ("Test")
Catch e As Exception
  Console.WriteLine (e.Message)
End Try
```

Имя метода может быть представлено в виде переменной. В этом случае связывание объекта и метода с делегатом откладывается до момента выполнения программы. Попытка связать делегата с объектом `BadObject` завершается неудачей, поскольку у этого объекта отсутствует метод `WriteMessage`. То же самое происходит и при несовпадении сигнатуры метода `WriteMessage` объекта с сигнатурой типа `DelegateWithStringSignature` (например, если метод не вызывается с одним строковым параметром). При выполнении этого фрагмента будет выведен следующий результат:

```
Late binding example 1
Called CalledClass WriteMessage method with parameter: Test
Error binding to target method
```

В программе можно определить универсальный метод, позволяющий вызвать любого переданного делегата независимо от типа параметров. Эта возможность продемонстрирована в функции `LateBoundCaller`:

```
Public Sub LateBoundCaller(ByVal d As Delegate)
  Dim params() As Object = {"Test"}
  Try
    d.DynamicInvoke (Params)
  Catch e As Exception
    Console.WriteLine (e.Message)
  End Try
End Sub
```

Функции `LateBoundCaller` могут передаваться делегаты произвольного типа, поскольку все типы делегатов являются производными от базового класса `Delegate` (а на объект производного типа всегда можно ссылаться через переменную базового типа). В следующем примере функции `LateBoundCaller` передаются два совершенно разных делегата:

```
Console.WriteLine (ControlChars.CrLf & "Late binding example 2")

Dim d6 As DelegateWithNoParam = AddressOf BadObject.BadWriteMessage
LateBoundCaller (CType(d1, System.Delegate))
LateBoundCaller (CType(d6, System.Delegate))
```

Первый вызов завершается успешно, поскольку делегат `d1` относится к типу `DelegateWithStringSignature`. Вторая попытка завершается неудачей, поскольку делегат `DelegateWithNoParam` вызывается без параметров. Результат:

```
Late binding example 2
Called CalledClass WriteMessage method with parameter: Test
Parameter count mismatch
```

Другие применения делегатов

Как видите, делегаты являются мощным средством для вызова функций. Имя метода и его параметры могут задаваться динамически во время работы программы (в VB6 подобная задача решалась с большим трудом[1]). Делегаты могут передаваться в параметрах, а это позволяет создавать функции для выполнения определенных операций с делегатами, предоставленными вызывающей стороной. Классическим примером является алгоритм сортировки, получающий метод сравнения двух элементов в виде делегатного параметра.

В качестве примера мы рассмотрим реализацию в VB .NET классической задачи — перечисления всех окон верхнего уровня в системе. Наше решение основано на передаче делегатных параметров и на применении делегатов для реализации обратного вызова при работе с функциями API[2].

При вызове функции API `EnumWindows` передается функция обратного вызова с двумя параметрами: манипулятором (handle) окна и значением, смысл которого определяется пользователем. Определение делегата выглядит так:

```
Delegate Function EnumWindowsCallback(ByVal hWnd As Integer, _
ByVal lParam As Integer) As Integer
```

Обратите внимание: параметры и возвращаемое значение определяются с типом `Integer` (32-разрядным!), а не с типом `Long`[3].

В проект входит модуль с объявлениями функции `EnumWindows` и функции `GetWindowText`, используемой для получения текста заголовков окон (там, где они присутствуют). Кроме того, определяется функция обратного вызова `ShowWindowNamesCallback`, совпадающая по сигнатуре с делегатом `EnumWindowsCallback`. Метод `ShowWindowsNames` при помощи оператора `AddressOf` получает делегата для ссылки на функцию `ShowWindowNamesCallback` и передает его функции `EnumWindows`. Определения этих функций выглядят следующим образом:

---

[1] Функция VB6 `CallIndirect` позволяет указать имя и параметры метода во время работы программы, но количество и типы параметров фиксируются. В статье «Implementing Indirect Calls on ActiveX/COM Objects» по адресу http://www.desaware.com показано, как организовать обобщенные косвенные вызовы с использованием пакета SpyWorks, но и это решение значительно уступает VB .NET по простоте.

[2] Вызов функций API подробно рассматривается в главе 15 (там же объясняется, почему в VB .NET такая необходимость возникает гораздо реже).

[3] Вероятно, это будет самая распространенная ошибка программирования, связанная с использованием функций API в VB .NET.

```
Module StdModule
  Public Declare Ansi Function EnumWindows Lib "User32" _
  (ByVal proc As EnumWindowsCallback, ByVal pval As Integer) As Integer
  Public Declare Ansi Function GetWindowText Lib "User32" Alias _
  "GetWindowTextA" (ByVal hWnd As Integer, ByVal WindowName As _
  String, ByVal BufferLength As Integer) As Integer

  Public Function ShowWindowNamesCallback(ByVal hwnd As Integer, _
  ByVal lParam As Integer) As Integer
    Dim windowname As New String(Chr(32), 255)
    Dim TrimmedName As String
    GetWindowText(hWnd, windowname, 254)
    TrimmedName = Left$(windowname, InStr(windowname, Chr(0)) - 1)
    If TrimmedName <> "" Then Console.WriteLine (TrimmedName)
    Return (1)
  End Function

  Public Sub ShowWindowNames()
    EnumWindows(AddressOf ShowWindowNamesCallback, 0)
  End Sub

End Module
```

Пока все выглядело примерно так же, как в VB6, но дальше начинается самое интересное. Далее следует определение класса `WindowHandles`:

```
Class WindowHandles
  Private col As New Collection()
  Public Sub New()
    MyBase.New()
    EnumWindows(AddressOf GetWindowHandlesCallback, 0)
  End Sub

  Private Function GetWindowHandlesCallback(ByVal hwnd As Integer, _
  ByVal lParam As Integer) As Integer
    col.Add (hwnd)
    Return (1)
  End Function

  Public Sub ShowAllWindows()
    Dim hwnd As Integer
    For Each hwnd In col
      Console.Write (Hex$(hwnd))
      Console.Write (", ")
    Next
    Console.WriteLine()
  End Sub

End Class
```

При создании экземпляра этого класса вызывается функция `EnumWindows`, которой передается делегат для метода `GetWindowHandlesCallback`. При каждом вызове `GetWindowHandlesCallback` полученный манипулятор окна сохраняется в коллекции, содержимое которой выводится на консоль методом `ShowAllWindows`.

Как видите, функция `EnumWindows`, которой в качестве параметра передается делегат, может выполнять различные операции в зависимости от полученного делегата. Но еще интереснее то, что `EnumWindows` может переадресовать обратный вызов методу, связанному с конкретным объектом. В этом VB .NET значительно

превосходит VB6, где оператор `AddressOf` возвращал лишь указатели на функции в стандартных модулях[1].

События

Наконец-то пришло время разобраться с тем, как события работают в VB .NET. Прежде всего следует отметить, что делегаты обладают еще одной особенностью, играющей важную роль при работе с событиями. При определении делегатной переменной в VB .NET вы в действительности определяете новый делегатный тип, производный от класса `System.MulticastDelegate` (который, в свою очередь, является производным от типа `System.Delegate`). Групповые (multicast) делегаты позволяют хранить ссылки на несколько методов, причем при вызове метода `Invoke` вызываются все эти методы (рис. 10.5).

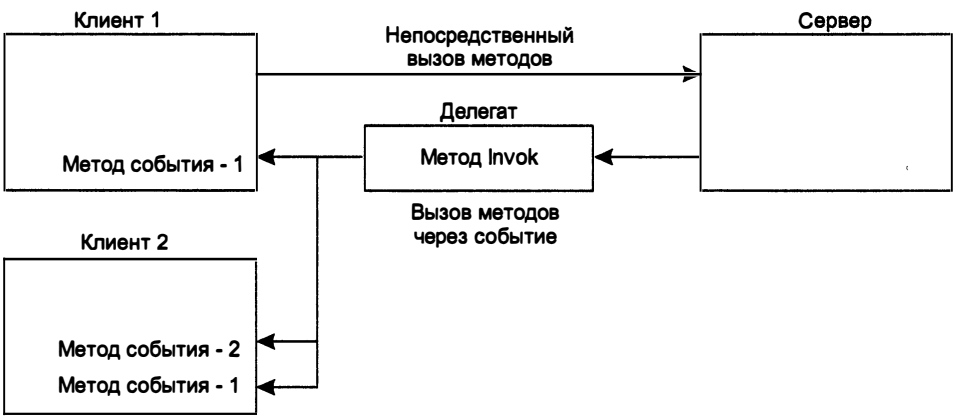


Рис. 10.5. Групповые делегаты

Варианты обработки событий

Приложение EventExample содержит одну форму с четырьмя кнопками и тремя переключателями. Каждый элемент демонстрирует некоторый аспект обработки событий в VB .NET.

Кнопка SimpleEvent работает так, как описано в разделе «События и делегаты» в последнем разделе этой главы. Дизайнер форм создает объявление кнопки с ключевым словом `WithEvents`, указывающим на то, что VB .NET должен автоматически организовать обработку событий. Для кнопки генерируется метод `button1_Click`, а ключевое слово `Handles` указывает на то, что язык должен автоматически создать делегата для метода `button1_Click` и связать его с событием `button1.Click`.

```
Private WithEvents button1 As System.Windows.Forms.Button
Private Sub button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
```

[1] Интересно, а как .NET это делает, если функция EnumWindows работает только с указателями на функции и понятия не имеет об объектах и экземплярах? Что ж, в VB6 за кулисами часто творились всякие чудеса — вот и считайте это новой разновидностью чуда для VB .NET.

```
  MsgBox("Simple Event Clicked", MsgBoxStyle.Information, _
  "Event arrived")
End Sub
```

Кнопка button2 с надписью «Other Approaches» показывает, что существует несколько возможных подходов к обработке событий.

В классе EventClass определяются два события, FirstEvent и SecondEvent. Событие SecondEvent соответствует общепринятой схеме, согласно которой в первом параметре передается ссылка на объект, инициировавший событие. Событие FirstEvent эту схему игнорирует.

Для обоих событий определяются делегаты, хотя в нашем примере это неочевидно.

- При вызове функции RaiseEvent в действительности вызывается метод Invoke делегата.

Впрочем, все это происходит за кулисами VB .NET[1].

```
Public Class EventClass
  Public Event FirstEvent(ByVal myData As Integer)

  Public Event SecondEvent(ByVal Sender As Object, ByVal myData As Integer)

  Public Sub Test1()
    RaiseEvent FirstEvent(5)
  End Sub

  Public Sub Test2()
    RaiseEvent SecondEvent(Me, 5)
  End Sub

End Class
```

На примере этого класса мы рассмотрим два варианта обработки событий.

Процедура события button2_Click просто вызывает методы Test1 и Test2 для двух разных экземпляров класса EventClass:

```
Private Sub button2_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button2.Click
  ec.Test1()
  ec2.Test2()
End Sub
```

Объекты ec и ec2 определяются по-разному:

```
Private ec As New EventClass()
Private WithEvents ec2 As EventClass
```

Как видите, событие ec2 связывается с функцией EventClass_SecondEvent при помощи ключевого слова Handles — точно так же, как событие button1.Click связывалось с методом button1_Click.

```
Private Sub EventClass_SecondEvent(ByVal obj As Object, _
ByVal i As Integer) Handles ec2.SecondEvent
  MsgBox("EventClass_SecondEvent called", _
  MsgBoxStyle.Information, "Event arrived")
End Sub
```

[1] В C# обработка событий организована посложнее, но сейчас это неважно.

Объект ec не определяется с ключевым словом WithEvents. В VB6 это вызвало бы большие трудности, но в VB .NET поддерживается динамическое подключение событий командой AddHandler:

```
AddHandler ec.FirstEvent, AddressOf EventClass_FirstEvent
Private Sub EventClass_FirstEvent(ByVal i As Integer)
  MsgBox("EventClass_FirstEvent called", _
  MsgBoxStyle.Information, "Event arrived")
End Sub
```

Упрощенное объяснение — функция AddHandler связывает метод EventClass_FirstEvent с событием ec.FirstEvent.

На самом деле происходит следующее.

Объект ec определяет событие с именем FirstEvent. Впрочем, мы уже знаем, что FirstEvent в действительности является делегатом, используемым объектом ec для инициирования событий.

Мы также знаем, что оператор AddressOf возвращает делегата.

Следовательно, команда AddHandler на самом деле берет делегата для метода EventClass_FirstEvent и добавляет его к делегату ec.FirstEvent. Я знаю, что само выражение «добавить одного делегата к другому» выглядит довольно странно. Попробуйте взглянуть на происходящее так:

- Делегат представляет собой объект, позволяющий вызвать заданный метод.
- Групповой делегат представляет собой объект, позволяющий вызвать несколько методов.
- При добавлении делегата к групповому делегату вы просто добавляете очередной метод в список методов, вызываемых групповым делегатом.

При добавлении делегата к событию (реализованному групповым делегатом) вы просто включаете очередной метод в список методов, вызываемых при инициировании события. А что произойдет, если воспользоваться командой AddHandler для добавления делегата к уже обрабатываемому событию? Мы знаем, что объект ec2 связан с методом EventClass_SecondEvent, а теперь мы связываем его с дополнительным методом.

```
AddHandler ec2.SecondEvent, AddressOf AnotherSecondEventHandler
Private Sub AnotherSecondEventHandler(ByVal obj As Object, _
  ByVal i As Integer)
  MsgBox("AnotherSecondEventHandler called", _
  MsgBoxStyle.Information, "Event arrived")
End Sub
```

Если щелкнуть на кнопке button2, на экране появятся три сообщения о вызове методов EventClass_FirstEvent, EventClass_SecondEvent и AnotherSecondEventHandler.

Метод AnotherSecondEventHandler также демонстрирует один важный факт: имя метода абсолютно несущественно. Синтаксис «класс_событие» используется лишь для наглядности.

Наследование событий

События, как и методы со свойствами, наследуются производными классами.

Приведенный ниже класс DerivedEventClass демонстрирует некоторые дополнительные особенности обработки событий. Одни из них связаны с наследо-

ванием, другие применимы к событиям в целом. Прежде всего следует отметить, что события могут определяться производными от типа `Delegate` (что неудивительно, если вспомнить, что событие *является* делегатом).

События, как и делегаты, могут объявляться общими. Общее событие инициируется без указания конкретного экземпляра объекта.

```
Public Delegate Sub EventTemplate(ByVal Obj As Object, ByVal i As Integer)

Public Class DerivedEventClass
  Inherits EventClass
  Shared Event ASharedEvent()

  ' Обратите внимание на другой вариант -
  ' эти объявления эквивалентны.
  'Event DerivedEvent(ByVal obj As Object, ByVal i As Integer)
  Event DerivedEvent As EventTemplate
```

Метод `InternalHandler` обрабатывает событие `SecondEvent` своего базового класса. Производные классы вообще обладают способностью принимать события базовых классов. Клиент, использующий этот класс, получает как события, унаследованные от базового класса, так и события, инициированные производным классом. Чтобы пользователь производного класса не мог получать события базового класса, воспользуйтесь ключевым словом `Shadows` и замаскируйте событие базового класса, определив закрытое событие с тем же именем. Закрытое событие недоступно за пределами класса.

Метод `Test2` класса `DerivedEventClass` маскирует одноименный метод базового класса. Он вызывает метод `Test2` базового класса и инициирует общее событие.

```
  Public Sub InternalHandler(ByVal obj As Object, _
  ByVal i As Integer) Handles MyBase.SecondEvent
    RaiseEvent DerivedEvent(Me, i * 2)
  End Sub
  Private Shadows Event FirstEvent(ByVal MyData As Integer)

  Public Shadows Sub Test2()
    MyBase.Test2()
    RaiseEvent ASharedEvent()
  End Sub
End Class
```

Объект `DerivedEventClass` определяется в форме с ключевым словом `WithEvents`. Он связывается с методом `DerivedEventHandler` при помощи ключевого слова `Handles`. Команда `AddHandler` связывает его с методом `AnotherSecondEventHandler`. Да, все верно — метод `AnotherSecondEventHandler` обрабатывает сразу два события от двух разных объектов. Вообще говоря, любой метод может принимать любое число событий от любого числа объектов — при условии, что сигнатура метода соответствует сигнатуре делегата события. Вскоре мы рассмотрим еще один пример подобного рода.

Для наглядности ссылка на общее событие осуществляется по имени класса, а не по имени объекта. Я мог бы сослаться на него и через экземпляр объекта (`dec.ASharedEvent`), однако имя класса подчеркивает тот факт, что речь идет об общем событии. Общее событие связывается с методом точно так же, как и событие объекта. Программный код, демонстрирующий использование `DerivedEventClass`, вызывается событием `button3.Click`.

```
Private WithEvents dec As DerivedEventClass
  AddHandler dec.SecondEvent, AddressOf AnotherSecondEventHandler
  AddHandler DerivedEventClass.ASharedEvent, AddressOf SharedEventHandler

  Private Sub DerivedEventHandler(ByVal obj As Object, _
  ByVal i As Integer) Handles dec.DerivedEvent
    MsgBox("DerivedEventHandler called", MsgBoxStyle.Information, _
    "Event arrived")
  End Sub

  Private Sub SharedEventHandler()
    MsgBox("Shared event handler", MsgBoxStyle.Information, "Event arrived")
  End Sub

  Private Sub button3_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button3.Click
    dec.Test2()
  End Sub
```

Снова о делегатах

В класс `DerivedEventClass` входит открытый делегат `SpecialEventHandler` и метод `TestSpecialEvents` для его вызова. Поскольку этот делегат не определяется с ключевым словом `Event`, он не отображается как событие в окне подсказки и информации типа. Тем не менее это не означает, что его нельзя использовать как обычное событие.

```
Public SpecialEventHandler As EventTemplate
Public Sub TestSpecialEvents()
  SpecialEventHandler.Invoke(Me, 6)
End Sub
```

Сначала делегату `SpecialEventHandler` присваивается делегат, указывающий на метод `FirstSpecialEventHandler` формы. Затем создается другой делегат, ссылающийся на метод `SecondSpecialEventHandler` формы. Наконец, делегату `SpecialEventHandler` присваивается комбинация двух предыдущих делегатов, объединенных методом `Combine` класса `Delegate`.

В результате методы `FirstSpecialEventHandler` и `SecondSpecialEventHandler` связываются с делегатом `dec.SpecialEventHandler`. При вызове объектом `dec` метода `SpecialEventHandler.Invoke` будут вызваны оба метода формы.

Описанный механизм работает практически так же, как механизм событий, если не считать того, что при использовании событий VB .NET берет на себя часть работы, упрощая задачу программиста. Следующий фрагмент демонстрирует этот «не-событийный» подход.

```
dec.SpecialEventHandler = AddressOf FirstSpecialEventHandler
  Dim EventToCombine As EventTemplate = AddressOf _
  SecondSpecialEventHandler
  dec.SpecialEventHandler = CType(dec.SpecialEventHandler.Combine(_
  dec.SpecialEventHandler, EventToCombine), EventTemplate)

  Private Sub FirstSpecialEventHandler(ByVal obj As Object, _
  ByVal i As Integer)
    MsgBox("FirstSpecial event handler", MsgBoxStyle.Information, _
    "Event arrived")
  End Sub
```

```
Private Sub SecondSpecialEventHandler(ByVal obj As Object, _
ByVal i As Integer)
  MsgBox("SecondSpecial event handler", MsgBoxStyle.Information, _
  "Event arrived")
End Sub

Private Sub button4_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles button4.Click
  dec.TestSpecialEvents()
End Sub
```

Обработка нескольких событий

Некоторые программисты VB6 жаловались на то, что в VB .NET не поддерживаются массивы управляющих элементов. Хотя в VB .NET массивы элементов действительно не поддерживаются, это не означает, что желаемого эффекта нельзя добиться другим способом. В приведенном ниже фрагменте три разных переключателя ассоциируются с одним событием.

```
Private Sub Options_CheckedChanged(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles radioButton3.CheckedChanged, _
  radioButton2.CheckedChanged, radioButton1.CheckedChanged
    Dim params() As Object
    Dim NameValue As Object
    NameValue = sender.GetType.InvokeMember("Text", _
      Reflection.BindingFlags.Public Or _
      Reflection.BindingFlags.Instance Or _
      Reflection.BindingFlags.GetProperty, Nothing, sender, params)
    Debug.WriteLine(CStr(NameValue))

    Dim rb As RadioButton
    rb = CType(sender, RadioButton)
    Debug.WriteLine(rb.Name & " " & rb.Text)

End Sub
```

Метод `Options_CheckedChange` связывается с событием `CheckedChanged` трех разных переключателей, для чего все события перечисляются после ключевого слова `Handles`. Вероятно, вы уже поняли, что аналогичного эффекта можно было добиться и командой `AddHandler`, но этот вариант вам придется кодировать самостоятельно, так как в Visual Studio отсутствует механизм, который бы позволял сделать это автоматически.

Параметр `sender` указывает, каким переключателем было инициировано событие. Поскольку массивы элементов в VB .NET не поддерживаются, вам не удастся идентифицировать источник события по значению свойства `Index`. В зависимости от типа приложения для идентификации источника можно воспользоваться любым другим подходящим свойством, например свойствами `Name`, `Text` или позицией элемента. Для получения свойств от `sender` используется механизм рефлексии, описанный в следующей главе. С загадочным методом `InvokeMember` (см. выше) мы разберемся чуть позже, а пока достаточно сказать, что он возвращает заданное свойство элемента `sender`.

Если вы твердо уверены, что все элементы относятся к одному типу, параметр `sender` можно объявить с типом соответствующего элемента и обращаться к свойству напрямую, как показано во втором варианте с переменной `rb`, объявленной с типом `RadioButton` (кнопка-переключатель).

Отключение обработчиков

Метод `Dispose` формы содержит следующие команды:

```
RemoveHandler ec.FirstEvent, AddressOf EventClass_FirstEvent
RemoveHandler ec2.SecondEvent, AddressOf AnotherSecondEventHandler
RemoveHandler DerivedEventClass.ASharedEvent, AddressOf SharedEventHandler
RemoveHandler dec.SecondEvent, AddressOf AnotherSecondEventHandler
```

Команда `RemoveHandler` разрывает связь метода с событием. В нашем примере это не оправдано, поскольку при закрытии формы приложение автоматически завершается, но в других ситуациях это может быть существенно.

Допустим, вы создали десять клиентских объектов и связали их с событием серверного объекта командой `AddHandler`. Но вот надобность в клиентских объектах отпала, и вы хотите освободить ссылки на них в своей программе.

Поскольку проблема циклических ссылок в VB .NET решена, клиентские объекты будут успешно освобождены, верно?

Неверно.

Серверный объект продолжает хранить ссылки на эти объекты с своем делегате события. Пока в вашем приложении сохраняется ссылка на серверный объект, VB .NET будет считать делегатов ссылками на клиентские объекты.

Следовательно, вызов `RemoveHandler` играет очень важную роль: разрыв установленных связей с событиями обеспечивает нормальное освобождение ваших объектов.

А как же события, объявленные с ключевым словом `WithEvents`? С ними проблем быть не должно, поскольку ссылка на источник события автоматически освобождается при освобождении модуля, принимающего события. Например, если при помощи ключевого слова `WithEvents` связать событие с элементом формы, то при удалении ссылки на форму будет освобождена сама форма и все ее объекты, поскольку ни одна из этих ссылок не ведет к переменным корневого уровня вашего приложения.

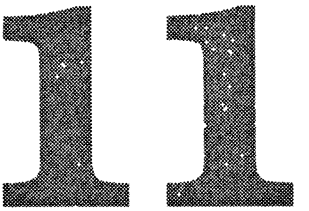
Итоги

Эта глава посвящена использованию объектов в VB .NET. Поскольку книга написана не для новичков, а для опытных программистов, вместо базовых концепций (для чего нужны классы, как работают свойства и методы и т. д.) мы разбирали более сложные вопросы. В частности, была рассмотрена структура приложений .NET и использование пространств имен для логической группировки классов, не зависящей от их физической группировки в сборках.

Далее были изложены практически все аспекты применения классов в VB .NET. Особое внимание уделялось изменениям в трактовке области видимости, обусловленным как специфической архитектурой приложений .NET, так и введением наследования в язык. Также были описаны некоторые изменения в методах и свойствах и приведены логические обоснования.

Глава завершается описанием событий и делегатов — новой концепции, на которой построен механизм событий .NET.

Рефлексия и атрибуты



В Visual Basic 6 часто встречаются конструкции следующего вида:

```
Public Class TestClass
  Public X As Integer
  Public Y As Integer
End Class
```

Классы и их отдельные члены обладают характеристиками `Public`, `Private` и т. д., которые можно рассматривать как атрибуты класса или члена. Рассмотрим следующий код VB .NET:

```
<Modified("Dan","1/10/2001")> Public Class TestClass
  <Modified("Dan","1/13/2001")> Public X As Integer
  Private Y As Integer
End Class
```

То, что заключено в угловые скобки (<>) — это тоже атрибуты[1]. Перед нами одно из принципиальных усовершенствований языка Visual Basic, но для чего оно нужно?

Чтобы понять всю важность атрибутов, необходимо сделать небольшой шаг назад.

Компиляторы и интерпретаторы

Для начала рассмотрим фундаментальные различия между компиляторами и интерпретаторами[2].

Компиляторы и интерпретаторы представляют два принципиально разных способа обработки программного кода. Интерпретатором называется программа, которая читает другую программу и выполняет указанные в ней действия (рис. 11.1).

---

[1] Не пытайтесь использовать атрибут Modified в своих программах, пока не прочтете эту главу и не поймете, откуда он взялся.

[2] Дальнейшее описание относится к «эталонным» интерпретаторам и компиляторам. На практике часто встречаются гибридные решения, в которых интерпретатор наделяется некоторыми возможностями компилятора, или наоборот.

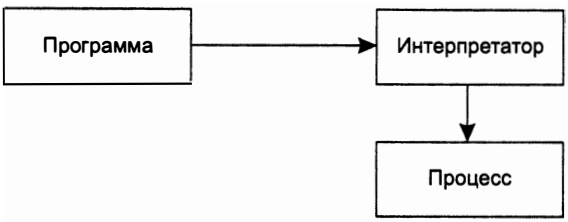


Рис. 11.1. Принцип действия интерпретатора

Например, при получении команды

```
A = 5
```

интерпретатор BASIC сохраняет число 5 в некоторой ячейке памяти, с которой связано имя «A». Поскольку интерпретатор фактически читает программный код и выполняет предписанные действия, он в любой момент располагает полной информацией обо всей программе. В частности, это позволяет легко прервать выполнение программы, изменить ее и продолжить работу, просматривать переменные и изменять их текущие значения и даже вводить команды в окне отладки и обрабатывать их вне обычной последовательности выполнения.

Интерпретаторы не всегда работают непосредственно с исходным текстом. Довольно часто они осуществляют предварительное преобразование программы в промежуточный код (иногда называемый «P-кодом»). Тем не менее интерпретатор всегда может восстановить исходную программу по предварительно преобразованному коду, позволяя программисту работать на уровне исходного текста.

Главным недостатком интерпретатора является низкая скорость работы. Интерпретатор постоянно обрабатывает исходный текст или P-код и выполняет полученные инструкции.

Компилятор преобразует исходный текст программы в стандартный машинный код (рис. 11.2).

Рис. 11.2. Принцип действия компилятора

Допустим, компилятор встретил следующую команду:

```
A = 5
```

Он генерирует машинный код для выполнения этой операции и записывает его в итоговый исполняемый файл. Завершив свою работу, компилятор уже не участвует в выполнении программы. Поскольку откомпилированная программа состоит из машинного кода, она отличается превосходным быстродействием (в зависимости от качества компилятора).

Для работы откомпилированной программы уже не нужен исходный текст: компилятор не включает его в файл, благодаря чему программы получаются более компактными и эффективными. К сожалению, отсутствие исходного текста затрудняет отладку программы, поскольку исходный текст используется для идентификации переменных и управления выполнением программы во время отладки. По этой причине компиляторы позволяют создавать отладочные версии файлов, содержащие исходный текст программы вместе с информацией о том, какому фрагменту исходной программы соответствует тот или иной блок исполняемого кода. Впрочем, даже при наличии отладочной информации отладчику будет очень трудно продолжить выполнение после модификации прерванной программы, поскольку в результате модификации код обычно перемещается в памяти, а указатели и содержимое памяти становятся недействительными. Многие современные среды программирования обеспечивают ограниченную поддержку «редактирования с продолжением»[1], но интерпретаторы позволяют делать это значительно проще и быстрее.

В Visual Basic 6 были объединены лучшие стороны обоих подходов. В среде программирования VB6 поддерживается как интерпретатор со всеми присущими удобствами, так и полноценный компилятор. Одна из самых замечательных особенностей VB6 заключается в том, что любая программа всегда одинаково работает как в интерпретируемом, так и в откомпилированном варианте.

Visual Basic .NET является «чистым» компилятором. Для многих программистов VB6 уже только к этому изменению придется долго привыкать. Я довольно долго работал на Visual C++, поэтому в среде Visual Studio .NET я чувствую себя вполне комфортно, но должен признаться, что я скучаю по интерпретируемой среде VB6. Я прекрасно понимаю, почему Microsoft решила объединить среды программирования, и новая среда *действительно* обладает массой классных возможностей... и все же мне немного не хватает интерпретатора VB6.

Раз компилятор, два компилятор...

Как упоминалось в главе 4, Visual Basic .NET и C# компилируют программы не в машинный код. Исходные тексты преобразуются в сборки, содержащие код на промежуточном языке (IL) и манифест с информацией обо всех объектах программы (рис. 11.3).

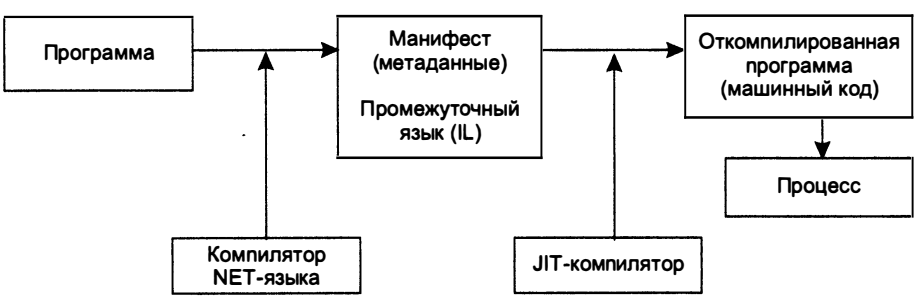


Рис. 11.3. Принцип действия языков .NET

[1] По текущей бета-версии трудно сказать, как эта возможность будет реализована в окончательной версии VB .NET.

Программа преобразуется в машинный код лишь в фазе JIT-компиляции, происходящей либо во время загрузки (когда сборка загружается в первый раз), либо во время установки. На основании информации об объектах, хранящейся в манифесте, устанавливаются связи с объектами ссылки, с их методами и свойствами. Кроме того, это позволяет сборке устанавливать связи с другими сборками. Данные манифеста и промежуточный код при этом остаются в программе, что позволяет JIT-компилятору восстановить сборку в случае необходимости, например при установке новой зависимой сборки. Возможность связывания методов и свойств с обновленными сборками помогает предотвратить «кошмар DLL», так часто возникающий при работе с компонентами COM. Вскоре мы вернемся к этой теме.

Стадия компиляции и стадия выполнения

Понятия «стадия компиляции» и «стадия выполнения» очень хорошо знакомы программистам, работающим на C++ и на ассемблере. Рассмотрим следующий фрагмент, написанный на псевдокоде:

```
If ABooleanConstantOrVariable Then
  Блок 1
End If
#If ABooleanConstant Then
  Блок 2
#End If
```

Блок 1 выполняется или не выполняется в зависимости от значения `ABooleanConstantOrVariable`. Даже если этот блок не выполняется (вследствие проверки условия), его код все равно присутствует в программе. Достигнув команды `If`, программа проверяет значение `ABooleanConstantOrVariable` и решает, следует ли выполнять следующий блок. Говорят, что условие проверяется *на стадии выполнения*, потому что это происходит во время работы откомпилированной программы.

Директивы `#If` и `#EndIf` обрабатываются в процессе компиляции программы. Константа `ABooleanConstant` определяется в настройках проекта или в параметрах командной строки компилятора. Если значение `ABooleanConstant` равно `True`, блок 2 включается в программу, а если оно равно `False`, блок 2 полностью игнорируется, словно обычный комментарий. В любом случае проверка условия в программу не входит. Происходит так называемая «условная компиляция»: решение о включении блока в программу принимается на основании условия, проверяемого на стадии компиляции (не на стадии выполнения!). Условная компиляция поддерживается и в VB .NET.

Условная компиляция является мощным средством, позволяющим определять разные конфигурации программы в одном исходном файле. С каждой конфигурацией связывается отдельный набор констант компиляции, управляющих включением/исключением кода.

Я часто использую условную компиляцию для включения отладочного кода в отладочные версии программ, для создания специальных конфигураций с трассировочным кодом и для изменений кода при локализации.

Атрибуты

Средства условной компиляции VB .NET имеют много общего с аналогичными возможностями VB6, однако в VB .NET также поддерживаются атрибуты, играющие важную роль как на стадии компиляции, так и на стадии выполнения. Прежде всего следует запомнить, что атрибуты доступны для компилятора. Это означает, что компилятор может проанализировать атрибуты класса, метода или свойства и на основании их значений внести изменения в сгенерированный код. Смысл языковых атрибутов (таких, как атрибуты области видимости `Public` или `Private`) очевиден, однако сказанное относится и к атрибутам, являющимся частью .NET Framework. Например, если пометить класс атрибутом синхронизации (см. главу 7), компилятор сгенерирует IL-код для поддержки синхронизации класса (далее этот код будет преобразован JIT-компилятором в машинный код).

Однако настоящие возможности атрибутов связаны с тем, что .NET хранит сведения об атрибутах объектов в манифесте. Как было сказано выше, это означает, что данная информация не теряется при компиляции.

А значит, ее можно получить во время выполнения программы.

Процесс получения атрибутов программного кода и других данных, хранящихся в манифесте сборки, называется *рефлексией* (reflection). Атрибуты будут подробно описаны ниже, а пока давайте разберемся с рефлексией.

Рефлексия

Следующий класс и перечисляемый тип взяты из приложения Reflection:

```
Public Class Class1

  Public Class TestClass
    Public X As Integer
    Private Y As Integer
  End Class

  Public Enum TestEnum
    FirstMember = 1
    SecondMember = 2
  End Enum
End Class
```

Наша цель — открыть сборку и не только обнаружить в ней этот класс и перечисление, но и определить имена и типы всех членов класса, а также получить список значений перечисляемого типа.

Исследуем манифест

Прежде всего мы импортируем пространство имен `System.Reflection`. Это позволит использовать сокращенную запись для всех объектов этого пространства (ссылка на пространство имен `System.Reflection` автоматически включается во все приложения .NET). Пространство `System.Reflection` содержит объекты, необходимые для чтения манифеста.

```
' Рефлексия и атрибуты
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Imports System.Reflection
Module Module1

  Sub Main()
    AssemblyTypes()
    Console.ReadLine()

  End Sub
```

Функция AssemblyTypes получает список всех открытых типов сборки, выполняемой в настоящий момент.

```
Sub AssemblyTypes()
  Dim TypeIndex As Integer
  Dim A As System.Reflection.Assembly
  Dim ATypes() As Type
  ' Исследовать текущую сборку
  A = A.GetExecutingAssembly()
  ' Получить все типы, предоставляемые данной сборкой
  ATypes = A.GetTypes()
```

Метод GetExecutingAssembly является статическим. С таким же успехом его можно вызвать без указания конкретного объекта, конструкцией вида

```
A = System.Reflection.Assembly.GetExecutingAssembly()
```

а также Reflection.Assembly.GetExecutingAssembly или [Assembly].GetExecutingAssembly. Существуют и другие статические методы, позволяющие открыть сборку по имени сборки, DLL или пространства имен. Метод GetTypes() возвращает массив всех типов, определенных в сборке. Продолжение метода AssemblyTypes выглядит так:

```
For TypeIndex = 0 To UBound(ATypes)
  ' Обратите внимание на полные имена типов.
  Console.WriteLine ("Type: " + ATypes(TypeIndex).FullName)
  ' Если это перечисление, вывести список значений
  If ATypes(TypeIndex).IsEnum Then
    Dim EnumStrings() As String
    ' Получить имена
    EnumStrings = System.Enum.GetNames(ATypes(TypeIndex))
    Console.WriteLine ("   Enumeration names are: ")
    Dim estemp As String
    For Each estemp In EnumStrings
      ' Вывести имена со значениями
      Console.WriteLine ("   " + estemp + " = " + _
      System.Enum.Format(ATypes(TypeIndex), _
      System.Enum.Parse(ATypes(TypeIndex), estemp), "D"))
    Next
  End If
```

Функция в цикле перебирает элементы массива. Свойство IsEnum переменной Type равно True, если тип является перечисляемым. В этом случае вызывается статический метод GetNames типа System.Enum (базового типа всех перечислений), возвращающий строковый массив с именами всех значений перечисляемого типа. Статический метод Parse типа System.Enum возвращает значение, соот-

ветствующее имени величины перечисляемого типа. Статический метод `Format` преобразует значение перечисляемого типа в строку для вывода.

Если текущий тип является вложенным (то есть определяется внутри сборки и не принадлежит к числу типов, реализующих саму сборку), вызывается функция `ShowMembers`, которая выводит список членов заданного типа. И класс `TestClass`, и перечисление `TestEnum` будут отнесены к вложенным типам.

```
    ' Для вложенных типов (определяемых программистом
    ' в сборке) вывести список всех членов.
    If ATypes(TypeIndex).MemberType = _
    MemberTypes.NestedType Then
      ' Вывести пользовательские атрибуты, которые будут
      ' определены в следующем примере.
      ' ShowCustomAttributes(_
        ATypes(TypeIndex).GetCustomAttributes())
      ShowMembers (ATypes(TypeIndex))
    End If
    Console.WriteLine()
  Next

End Sub
```

Функция `ShowMembers` получает список членов заданного типа (в нашем примере возвращается информация только о полях данных). Вы также можете получить список свойств, методов, интерфейсов и т. д.

```
Sub ShowMembers(ByVal ThisType As Type)
  Dim Index As Integer
  Dim idx2 As Integer
  Dim members() As MemberInfo
  ' Получить все поля данных для типа ThisType.
  ' Также можно получить информацию о методах,
  ' свойствах, интерфейсах и т.д.
  ' В список включаются закрытые и открытые
  ' (но не статические!) члены.
  members = ThisType.FindMembers(MemberTypes.Field, _
  BindingFlags.Public Or BindingFlags.Instance _
  Or BindingFlags.NonPublic, Type.FilterName, "*")
```

Метод `FindMembers` типа данных `Type` заполняет массив объектами `MemberInfo`, содержащими информацию об искомых членах классов. В нашем примере используется информация только о полях. Параметр `BindingFlags` определяет искомую категорию членов класса. В нашем примере возвращается информация об открытых и закрытых членах, а также о членах, ассоциированных с экземплярами объектов (в частности, информация о статических членах не включается). Мы используем стандартный фильтр класса `Type` и принимаем любые имена полей.

Следующий фрагмент в цикле перебирает содержимое полученного массива и выводит информацию о членах:

```
For Index = 0 To UBound(members)
  Dim fi As FieldInfo
  ' Поскольку мы знаем, что это поле
  ' данных, возможно безопасное преобразование
  ' к типу FieldInfo.
  fi = CType(members(Index), FieldInfo)
```

Тип FieldInfo, производный от MemberInfo, позволяет получить подробную информацию о полях данных. Оператор CType преобразует объект MemberInfo к типу FieldInfo. Такое преобразование заведомо возможно, поскольку каждый найденный объект MemberInfo фактически является объектом FieldInfo (при вызове FindMembers мы запрашивали информацию только о полях данных).

Тип поля определяется при помощи свойства FieldType объекта FieldInfo. При помощи свойства Attributes можно узнать, является ли поле закрытым или открытым.

```
    ' Получить имя и тип поля
    Console.Write ("  Member: " + members(Index).Name + _
      " Type:" + fi.FieldType.ToString())
    ' Прочитать атрибуты поля и проверить,
    ' является ли оно закрытым или открытым.
    If (fi.Attributes And FieldAttributes.Public) <> 0 Then
      Console.WriteLine (" - is Public")
    End If
    If (fi.Attributes And FieldAttributes.Private) <> 0 Then
      Console.WriteLine (" - is Private")
    End If
    ' См. следующий пример.
    ' ShowCustomAttributes(fi.GetCustomAttributes())
  Next Index
End Sub
```

Результат выглядит следующим образом:

```
Type: Reflection.Module1
Type: Reflection.Class1
Type: Reflection.Class1+TestClass
  Member: X Type:System.Int32 - is Public
  Member: Y Type:System.Int32 - is Private

Type: Reflection.Class1+TestEnum
  Enumeration names are:
    FirstMember = 1
    SecondMember = 2
  Member: value__ Type:System.Int32 - is Public
```

Как видите, пространство имен System.Reflection позволяет легко получить информацию о сборке во время выполнения программы. В оставшейся части этой главы основное внимание уделяется практическому применению этой возможности.

Пользовательские атрибуты

Откуда берутся атрибуты?

Одни встраиваются в язык, другие определяются в .NET Frameworks. Впрочем, у вас также есть возможность определять собственные атрибуты.

Проект Attributes показывает, как определить атрибуты для хранения в манифесте служебной информации, например истории вносимых изменений.

Атрибут представляется классом, производным от класса System.Attributes. Имя класса задается в форме имяAttribute. Так, в нашем примере для создания атрибута Modified определяется класс ModifiedAttribute. При ссылках на атрибут указывается его имя с суффиксом Attribute или без него. Например, на атри-

бут AttributeUsage можно ссылаться как в виде AttributeUsage, так и в виде AttributeUsageAttribute.

Поведение атрибута в новом классе также определяется специальными атрибутами. Атрибут AttributeUsage имеет конструктор, которому в качестве параметра передается комбинация флагов перечисляемого типа AttributeTargets. Флаги AttributeTargets определяют типы объектов, к которым может применяться данный атрибут. Как показывает следующий фрагмент, атрибут Modified из нашего примера может устанавливаться для методов, свойств, классов и полей данных.

```
' Рефлексия и атрибуты, пример II
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Public Class Class1
  <AttributeUsage(AttributeTargets.Method Or _
  AttributeTargets.Property Or _
  AttributeTargets.Class Or AttributeTargets.Field, _
  AllowMultiple:=True)> Public Class ModifiedAttribute
  Inherits System.Attribute
```

Для атрибута AttrubuteUsage можно установить два дополнительных свойства. Свойство Inherited указывает, должен ли новый атрибут наследоваться производными классами, а свойство AllowMultiple определяет возможность многократного применения атрибута к объекту.

Возникает интересный вопрос: как задать значения свойств в конструкторе?

Конструктор атрибута

Рассмотрим простой класс:

```
Class MyClass
  Public Sub New()
  End Sub
  Public X As Integer
End Class
```

Экземпляр этого класса можно создать командой вида:

```
Dim C As New MyClass()
```

Однако для того, чтобы значение поля X задавалось в конструкторе, необходимо определить конструктор с целочисленным параметром:

```
Public Sub New(ByVal NewX As Integer)
  X = NewX
End Sub
```

Конечно, при создании класса можно определить несколько конструкторов, однако для атрибутов предусмотрена и другая возможность. Ниже приведен простой атрибут MyAttribute, который может устанавливаться только для классов.

```
<AttributeUsage(AttributeTargets.Class)>Class MyAttributeAttribute
  Inherits System.Attribute
  Public Sub New()
  End Sub
  Public X As Integer
End Class
```

Естественно, для применения этого атрибута можно воспользоваться конструктором по умолчанию:

```
<MyAttribute()> Public Class B
End Class
```

Как видите, один из недостатков атрибутов заключается в том, что в момент установки атрибута нельзя вызывать методы соответствующего класса. Да, методы атрибутов можно вызывать при обращении к ним посредством рефлексии, но в момент непосредственного применения атрибута это не помогает. Тем не менее вы можете задавать значения свойств и полей данных, используя знакомый синтаксис именованных свойств:

```
<MyAttribute(X:=5)> Public Class B

End Class
```

Таким образом, на практике бывает удобно определять атрибуты с различными свойствами и полями, значения которых задаются в момент установки атрибута без применения длинных и сложных конструкторов.

Оператор присваивания :=, используемый при передаче именованных параметров функциям, знаком многим программистам VB. Приведенный выше синтаксис, при котором он задает значения свойств, может использоваться только для атрибутов.

Снова об атрибуте Modified

Давайте вернемся к атрибуту `Modified`. В соответствии со значениями `AttributeUsage` этот атрибут может устанавливаться для методов, свойств, классов и полей данных. Также допускается многократное применение этого атрибута. Атрибут `Modified` предназначен для пометки изменений программного кода, поэтому многократное применение выглядит вполне логично: элемент программы может изменяться несколько раз.

Помимо кода, приведенного выше, для этого атрибута определяются три открытых поля: `Author`, `ModDate` и `SomeIntValue`. В листинге 11.1 приведен конструктор, при вызове которого передается автор и дата изменения.

Листинг 11.1. Определение и использование пользовательских атрибутов[1]

```
Public Class Class1
  <AttributeUsage(AttributeTargets.Method Or _
  AttributeTargets.Property Or _
  AttributeTargets.Class Or AttributeTargets.Field, _
  AllowMultiple:=True)> Public Class ModifiedAttribute
    Inherits System.Attribute
    Public Author As String
    Public ModDate As String
    Public Sub New(ByVal SetAuthor As String, _
    ByVal SetModDate As String)
      MyBase.New()
      Author = SetAuthor
      ModDate = SetModDate
    End Sub
```

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

```
    Public Overrides Function ToString() As String
      Return ("Modified by " + Author + " on " + ModDate)
    End Function
    Public SomeIntValue As Integer
  End Class
```

Класс `ModifiedAttribute` также переопределяет метод `ToString` для вывода сообщений об изменениях. В следующем фрагменте определяется класс с именем `TestClass`, измененный автором Dan 10 октября 2001 года. Класс содержит поле X, которое изменялось дважды (авторы изменений — Dan и Joe). Атрибут также присваивает полю `SomeIntValue` значение 5 (просто для того, чтобы вы лучше поняли, как это делается).

```
  <Modified("Dan", "1/10/2001")> Public Class TestClass
    <Modified("Dan", "1/13/2001"), Modified("Joe", "1/25/2001", _
      SomeIntValue:=5)> Public X As Integer
    Private Y As Integer
  End Class

  Public Enum TestEnum
    FirstMember = 1
    SecondMember = 2
  End Enum
End Class
```

Чтение пользовательских атрибутов

Программа, демонстрирующая операции с пользовательскими атрибутами, построена на базе примера Reflection (см. раздел «Рефлексия» этой главы). В функцию `ShowMembers` добавляется команда

```
ShowCustomAttributes(fi.GetCustomAttributes(True))
```

Метод `GetCustomAttributes` класса `FieldInfo` возвращает массив пользовательских атрибутов для заданного поля.

В метод `AssemblyTypes` добавляется команда

```
ShowCustomAttributes(ATypes(TypeIndex).GetCustomAttributes(True))
```

При помощи метода `GetGustomAttributes` мы получаем массив пользовательских атрибутов для классов, найденных методом `AssemblyTypes`.

Метод `ShowCustomAttributes` определяется следующим образом:

```
Private Sub ShowCustomAttributes(ByVal TheAttributes() As Object)
  Dim ca As System.Attribute
  Dim idx As Integer
  For idx = 0 To UBound(TheAttributes)
    ca = CType(TheAttributes(idx), System.Attribute)
    Console.WriteLine ("    Attribute: " + ca.ToString())
  Next
End Sub
```

Методу `ShowCustomAttributes` в качестве параметра передается массив объектов, возвращаемый методом `GetCustomAttributes`. Объекты массива соответствуют пользовательским атрибутам, которые, как и все пользовательские атрибуты, являются производным от класса `System.Attributes`. Функция перебирает все элементы массива, обращается к каждому объекту через переменную

типа `System.Attributes` и затем выводит пользовательский атрибут методом `ToString`.

Хотя механизм вызова методов при установке атрибутов в Visual Basic .NET не предусмотрен, при работе с объектами атрибутов можно легко организовать вызов методов посредством рефлексии.

При выполнении программы будет получен следующий результат:

```
Type: Attributes.Class1

Type: Attributes.Class1+ModifiedAttribute
  Attribute: System.AttributeUsageAttribute
Member: Author Type:System.String - is Public
Member: ModDate Type:System.String - is Public
Member: SomeIntValue Type:System.Int32 - is Public

Type: Attributes.Class1+TestClass
  Attribute: Modified by Dan on 1/10/2001
Member: X Type:System.Int32 - is Public
  Attribute: Modified by Joe on 1/25/2001
  Attribute: Modified by Dan on 1/13/2001
Member: Y Type:System.Int32 - is Private

Type: Attributes.Class1+TestEnum
  Enumeration names are:
    FirstMember = 1
    SecondMember = 2
  Member: value__ Type:System.Int32 - is Public

Type: Attributes.Module1
```

В данном примере мы воспользовались рефлексией для чтения атрибута `Modified`. Представьте себе утилиту, которая использует рефлексию для документирования сборки и получает не только имена и свойства объектов, но и информацию об авторах и датах модификаций, а также любые другие сведения, для которых вы определите атрибуты. По аналогии с тем, как исполнительная среда использует атрибуты .NET Framework для управления поведением объектов и приложений, вы тоже можете читать атрибуты и управлять поведением своих программ.

При программировании компонентов и управляющих элементов в VB .NET вам часто придется использовать такие атрибуты, как `Browsable`, `Category`, `Description` и `Bindable`, определенные в классе `System.ComponentModel`. От этих атрибутов зависит интерпретация свойств средой разработки .NET. Например, если вы при помощи атрибута `Category` укажете, что свойство принадлежит к категории `Appearance`, то при работе с компонентом на стадии конструирования это свойство появится в группе `Appearance` окна свойств. IDE использует рефлексию для получения данных о принадлежности свойств к категориям[1].

Приложение DumpLib дополняет приложения Reflection и Attributes и показывает, как получить информацию о методах, свойствах и параметрах для каждо-

[1] Да, помимо стадий компиляции и выполнения в VB .NET также приходится учитывать поведение компонентов на стадии конструирования. Все программисты VB, которым приходилось программировать элементы ActiveX, хорошо знакомы с этой концепцией.

го объекта в сборке. Оно строит простую базу данных с информацией о членах классов и выводит ее содержимое в стандартном формате CSV (разделение данных запятыми)[1].

Связывание

Во время первой загрузки сборки JIT-компилятор на основании информации манифеста компилирует IL-код в машинный код. Обращения к членам классов в полученном коде реализуются очень эффективно: компилятор может установить местонахождение каждого члена и сгенерировать прямое обращение по соответствующим адресам[2]. Такой механизм называется ранним связыванием (early binding).

Раннее связывание используется в тех случаях, когда компилятору известны типы объекта и его члена.

Раннее связывание и «кошмар DLL»

COM тоже использует раннее связывание в тех случаях, когда компилятору известны типы объекта[3] и члена. Допустим, у вас имеется COM-программа, использующая объект из COM DLL. Приложение просматривает библиотеку типов DLL и осуществляет связывание членов на основании прочитанных данных. Таким образом, если приложение располагает указателем на объект, находящийся в DLL, оно будет обращаться к его свойствам и методам через этот указатель.

Но что произойдет, если изменить исходный текст DLL с изменением порядка членов в объекте и построить DLL заново?

Приложение COM о перемещении ничего не знает, поэтому при попытке вызвать метод с фиксированным смещением из предыдущей версии в итоге может быть вызван другой метод новой DLL или попытка вызова несуществующего метода завершится сбоем. Поэтому в COM очень важно, чтобы новые DLL были полностью совместимы со своими предыдущими версиями. Ошибки совместимости часто приводят к ошибкам защиты памяти. Даже простая перестановка членов в объекте может стать причиной несовместимости. Переименование членов и параметров приводит к еще худшим бедам.

Приложения .NET тоже используют раннее связывание, хотя машинный код генерируется на основании манифеста вызываемой .NET DLL, а не библиотеки типов. Однако при создании сборки приложения компилятор .NET сохраняет манифест используемой .NET DLL в виде сигнатуры[4]. При каждой загрузке прило-

---

[1] Я создал программу DumpLib для того, чтобы мне было проще сравнивать методы и константы в VB6 и VB .NET во время работы над книгой. Эта программа приводится как дополнительный пример использования рефлексии без каких-либо пояснений.

[2] Подробности низкоуровневой реализации раннего связывания совершенно несущественны. Неважно, вызывается ли функция напрямую, через v-таблицу (таблицу виртуальных функций), через смещение указателя и т. д. Достаточно знать, что раннее связывание работает быстро, а указатели и смещения при нем жестко фиксируются.

[3] В терминологии COM правильнее было бы сказать «тип интерфейса».

[4] Несколько упрощенное представление. На самом деле сохраняемая информация зависит от конфигурации, использованной при построении. Дополнительная информация приведена в главе 16.

жение проверяет, соответствует ли сигнатура DLL той, что была сохранена при создании приложения.

Допустим, автор .NET DLL создает новую версию DLL, изменяет порядок следования методов и добавляет несколько новых членов.

Приложение, использующее эту DLL, в процессе загрузки обнаруживает, что сигнатура DLL изменилась. Если конфигурация приложения допускает использование обновленных компонентов[1], CLR понимает, что все вызовы с ранним связыванием могут завершиться неудачей. Кэшированная версия приложения на машинном коде уничтожается, и приложение обрабатывается заново JIT-компилятором с использованием данных манифеста из новой DLL.

Что если создатель обновленной DLL допустил ошибку и изменил тип параметра при вызове метода?

В этом случае JIT-компилятор обнаружит ошибку на стадии компиляции и приложение не загрузится.

Теперь вы понимаете, почему в сборках .NET данные манифеста хранятся вместе с IL-кодом даже после их компиляции в машинный код?

Безопасное применении раннего связывания в приложениях .NET позволяет решить многие проблемы, из-за которых в приложениях COM приходилось прибегать к позднему связыванию.

Позднее связывание

Термин «позднее связывание» (late binding) означает, что при вызове метода приложение вычисляет его местонахождение во время выполнения программы. Иначе говоря, программа знает только имя метода и определяет его местонахождение по информации типа того объекта, к которому оно обращается. При позднем связывании может оказаться, что вызываемый метод не существует, поэтому в программе необходимо организовать перехват и обработку ошибок.

Позднее связывание в COM реализуется при помощи интерфейса `IDispatch`, реализованного всеми классами VB6. Этот интерфейс содержит метод `Invoke`, который получает имя метода/свойства и вызывает его с заданными параметрами[2]. В Visual Basic 6 позднее связывание используется при вызове методов обобщенного типа `Object`.

Visual Basic .NET поддерживает позднее связывание, хотя при этом он не использует ни средства COM, ни интерфейс `IDispatch`. Чуть позже в этой главе вы узнаете, как позднее связывание реализовано в .NET, а пока давайте рассмотрим один из способов (неправильный!) применения позднего связывания в .NET (листинг 11.2).

Листинг 11.2. Неверный подход к позднему связыванию в VB .NET

```
' Позднее связывание пример #1
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Option Strict Off
```

[1] Сборку можно настроить так, чтобы она работала только с конкретной версией DLL.

[2] Для экспертов в области COM такое объяснение выглядит несколько упрощенно, но мы не будем вдаваться в тонкости.

```
Interface ITestInterface1
  Sub Test()

End Interface

Interface ITestInterface2
  Sub Test()

End Interface

Class A
  Implements ITestInterface1
  Implements ITestInterface2
  Sub Test1() Implements ITestInterface1.Test
    Console.WriteLine ("Test1 called")
  End Sub
  Sub Test2() Implements ITestInterface2.Test
    Console.WriteLine ("Test2 called")
  End Sub
End Class

Module Module1

  Sub Main()
    Dim obj As Object
    Dim Aclass As New A()
    Dim it1 As ITestInterface1
    Dim it2 As ITestInterface2

    Aclass.Test1()
    Aclass.Test2()
    obj = Aclass
    obj.Test1()
    obj.Test2()
    Try
      obj.Test3()
    Catch e As Exception
      Console.WriteLine ("Late binding error: " & e.Message)
    End Try
    it1 = Aclass
    it1.Test()
    obj = it1
    Try
      obj.Test()
    Catch e As Exception
      Console.WriteLine ("Can't late bind to implemented interface")
    End Try

    obj.Test1()
    Console.ReadLine()
  End Sub

End Module
```

Результат:

```
Test1 called
Test2 called
Test1 called
```

```
Test2 called
Late binding error: Method "LateBinding.A.Test3" not found
Test1 called
Can't late bind to implemented interface
Test1 called
```

Вызовы `Aclass.Test1` и `Aclass.Test2` проходят раннее связывание с непосредственным вызовом методов `Test1` и `Test2`, реализованных классом.

Методы `obj.Test1` и `obj.Test2` проходят позднее связывание. Поскольку эти методы вызываются для типа `Object` (который сам по себе не содержит методов `Test` и `Test2`), CLR приходится искать этот метод во время выполнения программы и вызывать его. Как видите, попытка вызова `Test3` завершается неудачей с исключением «Method not found» («метод не найден»).

Как и в VB6, на объекты можно ссылаться по реализованным ими интерфейсам, однако для реализованного интерфейса нельзя использовать позднее связывание. В VB6 при присваивании интерфейса переменной типа `Object` вы получаете доступ к интерфейсу и можете вызывать его методы через объектную переменную. В VB .NET этот вариант не работает.

Почему такой подход к позднему связыванию завершается неудачей?

Во всем виновата первая строка программы:

```
Option Strict Off
```

Как было сказано выше, в приложениях VB .NET всегда следует включать жесткую проверку типов, и этот пример лишь подтверждает правило: недопустимый вызов `obj.Test3()` обнаруживается лишь во время выполнения. Подобные проблемы всегда должны обнаруживаться на стадии компиляции, поэтому Visual Basic .NET не разрешает позднее связывание такого рода при установленном флажке `Option Strict`.

Позднее связывание: правильный подход

Итак, флажок Option Strict установлен. Теперь я покажу, как правильно организовать позднее связывание в VB .NET.

Как упоминалось выше, в COM позднее связывание основано на интерфейсе `IDispatch`, поэтому объекты с поддержкой позднего связывания должны реализовать `IDispatch`. Реализация должна знать все члены, к которым объект желает предоставить доступ посредством позднего связывания, и обеспечить вызовы этих методов и свойств при вызове метода `IDispatch.Invoke`. Эта задача бывает довольно сложной, хотя при создании компонентов COM в ATL и MFC построение необходимого кода автоматизировано. Конечно, в VB6 все делается автоматически.

В приложениях .NET вся информация, необходимая для вызова методов объектов, находится в манифесте, поэтому вполне логично предположить, что в VB .NET позднее связывание основано на применении рефлексии[1].

[1] В главе 10 было показано, что позднее связывание может быть реализовано с применением делегатов. Делегаты обладают всей информацией, необходимой для позднего связывания, поскольку они ассоциируются с конкретным объектом и/или конкретным методом и сигнатурой.

На форме приложения IndirectCalls (рис. 11.4) пользователь вводит имя и значение параметра вызываемой функции.

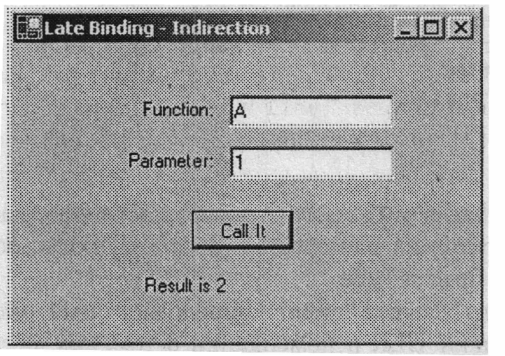


Рис. 11.4. Форма приложения IndirectCalls

Текстовым полям присвоены имена `txtFunction` и `txtParameter`. Кнопка называется `cmdCallIt`, а статическая надпись под кнопкой, предназначенная для вывода результата — `lblResult`. Приведенный ниже класс `myTestClass` демонстрирует позднее связывание (или косвенный вызов функции — в данном контексте эти термины имеют одинаковый смысл).

```
Public Class myTestClass
  Public Function A(ByVal InputValue As Integer) As Integer
    Return InputValue * 2
  End Function
  Public Function B(ByVal InputValue As Integer) As Integer
    Return InputValue * 3
  End Function
  Public Function C(ByVal InputValue As Integer) As Integer
    Return InputValue * 4
  End Function

End Class
```

Процедура события `cmdCallIt_Click` (листинг 11.3) показывает, как использовать рефлексию для косвенного вызова функции.

Листинг 11.3. Позднее связывание на базе метода InvokeMethod

```
Private Sub cmdCallIt_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdCallIt.Click
  Dim obj As New myTestClass()
  Dim T As Type
  Dim Params(0) As Object
  Dim result As Integer
  ' Целое число упаковывается в объект.
  Try
    Params(0) = CInt(txtParameter().Text)
  Catch ex As Exception
    MsgBox ("Must enter a number")
    Exit Sub
  End Try
  T = obj.GetType()
  Try
```

продолжение

Листинг 11.3 (продолжение)

```
    result = CInt(T.InvokeMember(txtFunction().Text, _
      Reflection.BindingFlags.Default Or _
      Reflection.BindingFlags.InvokeMethod, Nothing, obj, Params))
    lblResult().Text = "Result is " + result.ToString()
  Catch ex As Exception
    MsgBox (ex.ToString())
  End Try

End Sub
```

Метод `GetType` класса `myTestClass` возвращает информацию типа для объекта (информация берется из манифеста). Вызов функции осуществляется методом `InvokeMember` объекта `Type`.

Первый параметр `InvokeMember` определяет имя члена класса. Флаги `BindingFlags` передают CLR рекомендации о том, как должно осуществляться связывание. В нашем примере выбирается стандартный поиск с последующим вызовом метода. Следующий параметр определяет объект, выполняющий связывание. Мы передаем `Nothing`, чтобы использовать стандартное связывание (нестандартные объекты связывания создаются в тех случаях, когда вам по какой-либо причине требуется дополнительно управлять выбором члена при вызове `InvokeMember`). Следующий параметр определяет объект, метод которого вы хотите вызвать. Помните, что метод `InvokeMethod` вызывается не для объекта класса `myTestClass`, а для объекта `Type`, содержащего информацию типа для объекта `myTestClass`. Следовательно, вы должны предоставить ссылку на объект для вызова метода. Наконец, в последнем параметре передается массив объектов. У `InvokeMethod` существуют дополнительные перегруженные версии, обеспечивающие поддержку именованных параметров и данных локального контекста[1]. Они используются в ситуациях, когда метод должен вызываться в другом локальном контексте.

Динамическая загрузка

В последнем примере этой главы концепция позднего связывания доведена до логического завершения. Приложение DynamicLoading показывает, что динамически выбирать можно не только метод, но и объекты со сборками.

Каталог DynamicLoading содержит подкаталоги двух вспомогательных проектов. В первом каталоге находится сборка LaterBinding со следующим классом:

```
' Демонстрация "очень позднего" связывания
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Public Class LoadItDynamically
  Public Sub Test()
    MsgBox("The LoadItDyamically Test method was invoked", _
    MsgBoxStyle.Information, "Important message")
  End Sub

End Class
```

[1] В .NET вместо термина «локальный контекст» (locale) используется термин «культура» (culture).

В сборке LaterBinding выбрано корневое пространство имен `MovingToVB.LaterBinding`. Сборка компилируется в отдельную DLL. Мы хотим загрузить DLL во время выполнения программы, создать экземпляр класса `LoadItDynamically` и вызвать метод `Test`. Задача решается во втором проекте LaterBindingCaller при помощи кода, приведенного в листинге 11.4.

Листинг 11.4. Проект LaterBindingCaller

```
' Пример позднего связывания
' Copyright ©2001 by Desaware Inc. All Rights Reserved.
Imports System.Reflection
Module Module1

  Sub Main()
    Dim A As Reflection.Assembly
    Dim LaterBindingDLL As String
    Dim obj As Object
    Dim Params() As Object

    ' Navigate to the other DLL
    LaterBindingDLL = CurDir() & _
    "\..\..\laterbinding\bin\laterbinding.dll"
    A = A.LoadFrom(LaterBindingDLL)

    obj = A.CreateInstance("MovingToVB.LaterBinding.LoadItDynamically")
    obj.GetType().InvokeMember("Test", BindingFlags.Default Or _
    BindingFlags.InvokeMethod, Nothing, obj, Params)
  End Sub

End Module
```

Как видите, в программе нет ничего сложного. Метод `Assembly.LoadFrom` получает имя DLL в виде параметра и загружает сборку. Метод `CreateInstance` создает объект по полностью уточненному имени.

После того как объект будет создан, мы вызываем метод `InvokeMember`, как это было сделано в проекте IndirectCall.

Итоги

В начале этой главы мы рассмотрели основные различия между компиляторами и интерпретаторами, играющие важную роль для понимания различий между стадией компиляции и стадией выполнения. Вы узнали, что атрибуты определяются на стадии компиляции и обладают значительно большими возможностями, чем традиционный механизм условной компиляции. Также было показано, что хотя атрибуты в первую очередь предназначены для управления компиляцией, они также могут использоваться для управления поведением программ на стадии выполнения.

Далее было показано, как читать из манифеста сборки различную информацию, включая имена типов и методов, а также значения атрибутов. Мы рассмотрели процесс определения пользовательских атрибутов, их применение для документирования сборок и чтение на стадии выполнения.

Завершающая часть главы посвящена связыванию. В архитектурах COM и .NET раннее связывание обеспечивает более высокую эффективность, однако в COM оно часто порождает проблемы совместимости. В .NET эти проблемы решаются хранением манифеста и IL-кода, что позволяет перекомпилировать сборки по мере необходимости.

В конце главы представлены два подхода к позднему связыванию в .NET: правильный и неправильный. Развивая концепцию позднего связывания, мы рассмотрели возможность динамической загрузки сборки и объекта на стадии выполнения.

Лирическое отступление

Прежде чем продолжать, я хочу поговорить с вами по душам.

Я хочу, чтобы вы знали, чего ожидать от оставшихся глав, а чего ожидать не стоит.

Видите ли, до сих пор моя задача была относительно простой.

В части 1 были рассмотрены основные стратегические факторы, которые следует учитывать при переходе на VB .NET.

В части 2 излагались ключевые концепции, необходимые для проектирования приложений VB .NET.

В предыдущих четырех главах было приведено подробное, добросовестное описание изменений в синтаксисе языка VB .NET.

Но с остальными главами дело обстоит иначе. Они посвящены изменениям языка, связанным с классами .NET Framework и с исполнительной средой. Иногда речь идет о замене некоторых средств VB6 целыми пространствами имен .NET, но многие изменения обусловлены изменениями базовой архитектуры (например, новый механизм форм, новые web-технологии и т. д.) или появлением новых концепций .NET Framework, совершенно незнакомых программистам VB6. Во всех перечисленных случаях соответствующая функциональность представлена десятками объектов, каждый из которых обладает многочисленными свойствами и методами, а иногда она складывается из нескольких пространств имен с десятками объектов.

Если вы рассчитываете получить подробное описание по всем подобным темам, вы будете разочарованы. Более того, по многим из них вполне можно написать отдельную книгу![1]

Что мне делать?

Единственное, что я могу, — придерживаться того курса, который я выбрал с первых страниц. Как было сказано выше, я не собираюсь пересказывать документацию Microsoft. В своей книге я хочу познакомить читателя с VB .NET, описать некоторые базовые концепции и поделиться своим мнением.

---

[1] Это замечание будет часто встречаться в следующих главах. Кстати, если вам вдруг захочется написать такую книгу, пожалуйста, дайте мне знать.

А теперь запомните самое главное правило для всех следующих глав.

- **Обязательно** прочитайте электронную документацию. Вы должны хорошо ориентироваться в пространствах имен, в объектах и их методах.

Многие программисты VB6 не привыкли читать документацию. У них есть для этого веские основания: документация Microsoft для программистов Visual Basic написана очень неудобно. Объявления C++ во многих случаях так плохо переносились в VB6, что мне пришлось написать отдельную книгу лишь с одной целью: научить программистов VB читать MSDN и создавать объявления функций для программ VB6. Многие функции, определенные в MSDN, несовместимы с Visual Basic, причем иногда было трудно отличить совместимые функции от несовместимых.

Теперь ситуация кардинально изменилась.

Язык VB .NET обеспечивает полную CLS-совместимость. Классы .NET Framework полностью совместимы с VB .NET. В документацию Microsoft включены синтаксические конструкции VB .NET для вызовов методов и обращений к свойствам. Многие примеры написаны на VB .NET, а там, где код VB .NET отсутствует, примеры C# практически построчно переводятся на VB .NET — код вызова методов/свойств в C# и VB .NET выглядит практически одинаково.

Что касается меня, то я постараюсь сразу выделить основные концепции и ключевые классы, с которыми вам предстоит работать. Но я при всем желании не смогу привести подробные описания всех технических мелочей, которые приходится учитывать при переходе с VB6 на VB .NET. У каждого элемента, у каждого объекта имеются свои особенности. Вы должны быть готовы к самостоятельному чтению документации MSDN. Со временем на рынке непременно появятся книги, которые помогут вам разобраться в отдельных темах.

А пока можете рассматривать следующие главы как введение в .NET-составляющую VB .NET (тогда как до настоящего момента речь шла об основной функциональности языка). В этой части книги я постараюсь дать представление о доступных возможностях и заложить основу для дальнейшего изучения VB .NET. Мы рассмотрим стратегические и концептуальные аспекты разных компонентов .NET и даже разберем несколько примеров программ. Главное — помнить о том, что очень многое осталось «за кадром», и быть готовым к самостоятельным исследованиям.

Часть 4

Удивительный мир .NET

Путешествие длиной в тысячу миль начинается с одного шага.

Лао Цзы

Пространства имен .NET: общие сведения

12

Дамы и господа, мы начинаем большую экскурсию по пространствам имен Microsoft .NET. Вас сопровождает Дэниел Эпплман, опытный экскурсовод, который не только укажет на основные достопримечательности, но и познакомит с историей предмета начиная с времен Visual Basic 1. Итак, садитесь поудобнее, пристегните ремни и смотрите внимательнее.

Главное, что нужно знать о пространствах имен .NET

Всю суть этой главы можно сформулировать в двух предложениях. Их необходимо хорошенько понять и запомнить, даже если вы пропустите все остальное.

.NET Framework

Многое из того, что происходило в VB6, напоминало чудо. В начале работы над проектом программист выбирал тип создаваемого приложения: ActiveX EXE, ActiveX DLL, стандартный EXE-файл, приложение IIS, элемент управления и т. д. Различия между этими классами приложений в большой степени определялись внутренними механизмами самого VB.

В Visual Basic .NET компилятор определяет лишь синтаксис языка, на котором вы работаете, — *и ничего больше.*

Позвольте мне еще раз подчеркнуть это обстоятельство.

В Visual Basic .NET никаких чудес *нет.* Компилятор решает только одну задачу: он обрабатывает группу текстовых файлов с ключевыми словами VB .NET по правилам, определяемым синтаксисом языка.

В новом языке Microsoft C# чудес тоже нет. Собственно, он отличается от VB .NET лишь тем, что обрабатываемые текстовые файлы написаны по правилам синтаксиса C#.

Все чудеса, все эффектные возможности (формы, web-приложения и web-службы, управляющие элементы .NET и компоненты) существуют благодаря

библиотеке классов .NET и Common Language Runtime. Даже поведение компонентов, элементов и окон на стадии конструирования определяется спецификой классов, созданных вами, — классов, производных от классов .NET Framework. Атрибуты, управляющие поведением классов, также создаются на базе классов .NET Framework.

Следует особо подчеркнуть, что выбор языка особой роли не играет. Формы VB .NET и C# содержат более или менее одинаковый код. Все различия сводятся к синтаксису языка, который был использован для определения классов, производных от классов .NET Framework, и вызова их методов.

Я уверен, что большинство книг, посвященных VB .NET, будет начинаться с демонстрации различных вариантов приложений — на базе форм Windows, web-форм, web-служб и т. д. Возможно, такой подход действительно поможет быстрее взяться за практическое программирование, но при этом читатель рискует упустить очень важное обстоятельство. В .NET формы не являются отправной точкой для программирования. Форма — всего лишь результат определения класса, производного от базового класса .NET. Логичнее было бы начать с наследования и объектно-ориентированного программирования, а затем перейти к .NET Framework. После этого изучение форм, web-приложений и различных типов приложений, включая web-приложения, сводится к простому изучению методов базового класса, на основе которого создан ваш производный класс, а также правил расширения этих классов посредством наследования. Именно по этой причине многие приложения, приведенные выше, были консольными. Консольные приложения позволяют сосредоточить все внимание на программном коде, не отвлекаясь на более сложные пространства имен .NET.

Из предыдущих глав вы узнали, чем VB .NET отличается от VB6 на уровне синтаксиса. Практически все, что вам осталось узнать, так или иначе связано с классами .NET Framework и средой Common Language Runtime.

Мы подходим ко второму факту, который необходимо хорошо усвоить...

.NET учитывает интересы программистов VB .NET

Давайте поговорим о Windows-программирования вообще, без технологий Microsoft .NET.

Как программисты C++ учатся программировать для Windows?

Большинство начинает с элементарных книг, в которых базовые принципы изложены лучше, чем в документации Microsoft (совершенно не предназначенной для начинающих: MSDN представляет собой технический справочник, а не учебник).

Программисты среднего и высокого уровня, а также работающие с новыми технологиями, еще не описанными в книгах, черпают информацию из двух источников:

1. Документация Microsoft.
2. Специализированная литература по конкретным темам.

Они могут читать документацию Microsoft, поскольку она предназначена для программистов C++ — например, в объявлениях функций API приводятся прото-

типы C++. Работа с многими параметрами функций API в VB6 сильно затруднена. Во многих интерфейсах COM используются типы данных, не поддерживаемые в VB6[1]. Любые архитектурные допущения основываются на том, что программист работает на C++, причем результат часто оказывается абсолютно несовместим с VB6.

В результате у большинства программистов Visual Basic нет привычки искать нужную информацию в документации Microsoft. Они предпочитают книги, в которых документация Microsoft «переводится» на язык VB.

Откровенно говоря, я от этого даже выиграл: моя книга «Visual Basic Programmer's Guide to the Win32 API»[2] стала популярной именно потому, что она отвечала реальным потребностям программистов Win32 API, ведь чтобы эффективно работать с Win32 API, необходимо было много знать.

Но представьте себе, что при разработке Win32 API были бы учтены интересы программистов VB, каждая функция сопровождалась объявлением VB, а среди примеров наряду с программами C++ были бы представлены и версии, написанные на VB6.

Что бы стало с моей книгой?

Во всяком случае, такой успех ей бы и не снился.

Конечно, тираж бы постепенно разошелся — всегда найдутся люди, которые попросту не любят документацию Microsoft или предпочитают работать с печатными документами (но не хотят печатать свою собственную копию). К тому же я надеюсь, что в книге хватало примеров и полезной информации, выходящей за рамки документации.

Но я могу уверенно заявить: если бы Win32 API был спроектирован для программистов VB, абсолютное большинство программистов работало бы, используя документацию Microsoft, и не стало бы покупать мою книгу[3].

Мы подходим к самому главному.

Документация .NET Framework написана для программистов Visual Basic .NET. Все классы и все методы могут использоваться из VB .NET[4]. Синтаксис вызовов VB .NET включен в нее наряду с синтаксисом C# и C++.

Пожалуйста, заведите новую привычку: *сначала прочитайте документацию .NET Framework!*

Если раньше, столкнувшись с проблемой, вы начинали рыться в книгах или технических журналах, если боялись взяться за документацию Microsoft — настало время повзрослеть. Документация Microsoft должна стать вашим *первым* источником информации, а все поиски должны начинаться с приведенных в ней примеров.

Не поймите меня превратно. Скоро рынок будет завален книгами по .NET Framework. Среди этих книг будет много полезных, которые описывают нетривиальные приемы, содержат более доступные объяснения, материал, ориентирован-

---

[1] Впрочем, продукты независимых фирм, в частности пакет Desaware SpyWorks, позволяют программистам VB6 нормально использовать и реализовывать эти несовместимые интерфейсы.

[2] Русский перевод: Win32 API и Visual Basic. СПб.: Питер, 2000. — *Примеч. ред.*

[3] ...А моя карьера пошла бы в совершенно другом, неизвестном направлении...

[4] Существуют отдельные исключения, несовместимые со спецификацией CLS, — впрочем, пользоваться ими все равно не рекомендуется.

ный на начинающих или опытных программистов, или же углубленно рассматривают отдельные типы приложений. Другие книги будут написаны со специфическими целями, например чтобы помочь программистам VB6 перейти на новую технологию[1].

Но вам уже не понадобятся книги и статьи с описанием стандартных операций и простых задач. Прослойка между VB и миром «серьезного» Windows-программирования исчезнет.

Начинаем экскурсию

Попадая в новый город, мы стремимся познакомиться с ним поближе. Существуют два стандартных способа: купить путеводитель или отправиться на экскурсию, в ходе которой вам расскажут об основных достопримечательностях и истории.

В этой главе я постараюсь объединить оба подхода.

Ни из путеводителя, ни от экскурсовода вы никогда не получите полной информации обо всем, что вы видите на своем пути. Не ждите этого и от меня.

Я несколько раз начинал писать эту главу. В одной из неудачных попыток я решил привести список всех объектов .NET Framework, привести для каждого объекта комментарий из одной-двух строк и перечислить его методы. На всякий случай я решил оценить объем своей работы. Оказалось, что глобальный кэш сборок содержит свыше 6700 объектов (классов и перечислений)[2]. Даже если бы каждый объект был представлен всего одной строкой, список занял бы свыше 100 страниц.

Проку от такого списка было бы немного, все свелось бы к обычному пересказу документации (чего я терпеть не могу).

Поэтому данную главу не следует рассматривать как «справочник» по пространствам имен .NET. Я постарался привести общий обзор пространств имен, указать на их взаимосвязи и отдельно подчеркнуть некоторые особенно эффектные возможности. Надеюсь, к моменту чтения этой главы вы уже будете достаточно хорошо представлять себе структуру связей между различными компонентами .NET Framework.

Я также попытаюсь описать ключевые концепции, заложенные в основу многих пространств имен: это заложит прочную основу для самостоятельных исследований.

Как известно, любая задача считается на 90 % решенной, если вы достоверно знаете, что решение существует. Я постараюсь показать, какими возможностями вы располагаете, чтобы вы знали, в какой области следует искать решение.

Карта

В любом путеводителе центральное место занимает карта. Наверное, сейчас мне бы следовало нарисовать красивую блок-схему с основными компонентами .NET

---

[1] Одну из таких книг вы как раз держите в руках.

[2] При создании нового Windows-приложения по умолчанию загружается около 2700 объектов. Приведенные цифры были получены при помощи приложения Derivations2, представленного ниже в этой главе.

Framework, однако мои исследования показали, что здесь требуется нечто помощнее. Итак, познакомьтесь с тремя важнейшими инструментами, которые помогут вам ориентироваться в пространствах имен.

Справочная документация MSDN

Документация .NET Framework SDK станет вашим главным источником информации, но основным справочником станет раздел «.NET Framework Class Library». Содержимое этого раздела упорядочено по пространствам имен, а внутри каждого пространства объекты отсортированы в алфавитном порядке.

Справочник лучше всего подходит для поиска подробных описаний конкретных объектов и методов. Вы также можете просмотреть его, чтобы составить общее представление о библиотеке классов, хотя просмотр такого справочника — задача нелегкая и непростая.

Раздел «Programming with the .NET Framework» содержит полезную информацию о конкретных областях .NET. Не ограничивайтесь одной документацией VB .NET; помните, что общая документация .NET Framework тоже была рассчитана на программистов VB .NET.

Наконец, не пренебрегайте примерами. Большинство примеров входит в документацию VB .NET, но вы без особого труда разберетесь и в других примерах, поскольку операции с классами .NET Framework выполняются во всех языках практически одинаково.

Утилита WinCV

Утилита WinCV, входящая в .NET Framework SDK, предназначена для просмотра классов. Информация о членах заданного класса извлекается посредством рефлексии.

Используйте утилиту WinCV, если вы обнаружите какие-либо расхождения между документацией и реальной функцией. WinCV поможет вам узнать, из каких членов состоит объект на самом деле[1].

Проект Derivation2

Разбираясь в пространствах имен, я понял, что главным недостатком существующей документации является ее логическая организация. Группировка методов по пространствам имен хороша для высокоуровневых классов, но неудобна на более низком уровне. Например, как получить список всех возможных исключений, которые могут инициироваться исполнительной средой, или всех атрибутов, используемых при разработке компонентов?

К счастью, сама архитектура .NET Framework поддерживает механизм, позволяющий легко получить ответы на подобные вопросы. Поскольку в основе .NET лежит механизм наследования, объекты естественным образом группируются в соответствии с базовыми классами.

---

[1] Документация Microsoft чаще страдает от недостаточной глубины и четкости, нежели от фактических ошибок, но и это не исключено, особенно при выходе новых продуктов или сильно обновленных версий.

Например, чтобы ответить на вопрос: «Какие исключения могут инициироваться системой?», достаточно запросить список всех объектов, производных от `System.Exceptions.SystemException`. Результат приведен в табл. 12.1.

Таблица 12.1. Системные исключения (неполный список — только для базовых DLL)

System.AppDomainUnloadedException
System.ApplicationException
System.ArgumentException
System.ArgumentNullException
System.ArgumentOutOfRangeException
System.ArithmeticException
System.ArrayTypeMismatchException
System.BadImageFormatException
System.CannotUnloadAppDomainException
System.Configuration.ConfigurationException
System.Configuration.Install.InstallException.
System.ContextMarshalException
System.Data.ConstraintException
System.Data.DBConcurrencyException
System.Data.DeletedRowInaccessibleException
System.Data.DuplicateNameException
System.Data.EvaluateException
System.Data.InRowChangingEventException
System.Data.InvalidConstraintException
System.Data.InvalidExpressionException
System.Data.MissingPrimaryKeyException
System.Data.NuNullAllowedException
System.Data.OleDb.OleDbException
System.Data.ReadOnlyException
System.Data.RowNotInTableException
System.Data.SqlClient._ValueException
System.Data.SqlTypes.SqlException
System.Data.SqlTypes.SqlNullValueException
System.Data.SqlTypes.SqlTruncateException
System.Data.StrongTypingException
System.Data.SyntaxErrorException
System.Data.TypedDataSetGeneratorException
System.Data.VersionNotFoundException
System.DivideByZeroException
System.DllNotFoundException
System.Drawing.Printing.InvalidPrinterException

продолжение

Таблица 12.1 (продолжение)

System.DuplicateWaitObjectException
System.EntryPointNotFoundException
System.Exception
System.ExecutionEngineException
System.FieldAccessException
System.FormatException
System.IndexOutOfRangeException
System.InvalidCastException
System.InvalidOperationException
System.InvalidProgramException
System.IO.DirectoryNotFoundException
System.IO.EndOfStreamException
System.IO.FileLoadException
System.IO.FileNotFoundException
System.IO.InternalBufferOverflowException
System.IO.IOException
System.IO.IsolatedStorageException
System.IO.PathTooLongException
System.MemberAccess.Exception
System.MethodAccessException
System.MissingFieldException
System.MissingMemberException
System.MissingMethodException
System.MulticastNotSupportedException
System.Net.CookieException
System.Net.ProtocolViolationException
System.Net.Sockets.SocketException
System.Net.WebException
System.NotFiniteNumberException
System.NotImplementedException
System.NotSupportedException
System.NullReferenceException
System.ObjectDisposedException
System.OutOfMemoryException
System.OverflowException
System.PlatformNotSupportedException
System.RankException
System.Reflection.AmbiguousMatchException
System.Reflection.CustomAttributeFormatException

System.Reflection.InvalidFilterCriteriaException
System.Reflection.ReflectionTypeLoadException
System.Reflection.TargetException
System.Reflection.TargetInvocationException
System.Reflection.TargetParameterCountException
System.Resources.MissingManifestResourceException
System.Runtime.InteropServices.COMException
System.Runtime.InteropServices.ExternalException
System.Runtime.InteropServices.InvalidComObjectException
System.Runtime.InteropServices.InvalidOleVariantTypeException
System.Runtime.InteropServices.MarshalDirectiveException
System.Runtime.InteropServices.SafeArryRankMismatchException
System.Runtime.InteropServices.SafeArrayTypeMismatchException
System.Runtime.InteropServices.SEHException
System.Runtime.Remoting.MetadataServices.SUDSGeneratorException
System.Runtime.Remoting.MetadataServices.SUDSParserException
System.Runtime.Remoting.RemotingException
System.Runtime.Remoting.RemotingTimeoutException
System.Runtime.Remoting.ServerException
System.Runtime.Serialization.SerializationException
System.Security.Policy.PolicyException
System.Security.SecurityException
lparSystem.Security.VerificationException
System.Security.XmlSyntaxException
System.ServiceProcess.TimeoutException
System.StackOverflowException
System.SystemException
System.Threading.SynchronizationLockException
System.Threading.ThreadAbortException
System.Threading.ThreadInterruptedException
System.Threading.ThreadStateException
System.ThreadStopException
System.TypeInitializationException
System.TypeLoadException
System.TypeUnloadedException
System.UnauthorizedAccessException
System.UriFormatException

Таблица 12.1 демонстрирует еще один важный аспект логической организации классов в пространствах имен. Вы можете легко выбрать все исключения, относящиеся к многопоточности: они группируются ближе к концу списка с префиксом `System.Threading`.

Помимо поиска классов, производных от заданного, проект Derivation2 выводит иерархию наследования для любого объекта, а также может использоваться для поиска всех объектов, реализующих заданный интерфейс.

Проект Derivation2 действует методом «грубой силы»: при помощи рефлексии он загружает информацию типа для всех объектов, на которые имеются ссылки в домене приложения (проект содержит ссылки на большинство пространств имен .NET). Затем методом линейного поиска во всех иерархиях наследования и списках интерфейсов строится список всех заданных объектов (реализующих интерфейс или наследующих от класса)[1].

Как обычно, полный проект находится среди примеров книги. Листинги не приводятся, поскольку этот проект просто демонстрирует некоторые приемы, описанные в главе 11.

В этой главе я буду указывать те места, в которых я бы рекомендовал воспользоваться утилитой Derivation2, чтобы получить карту объектов для дальнейших самостоятельных исследований.

Системные классы

Пространство имен `System` содержит важнейшие классы, используемые практически в любой программе.

Базовые классы

Все классы пространства имен `System` строятся на основе трех основных классов.

System.Object

Базовый тип в иерархии наследования любого объекта (как ссылочного, так и структурного типа). Центральное место в классе занимает метод `ToString`. Даже если этот метод не используется в вашей программе, его рекомендуется переопределять, так как вывод строкового представления класса часто помогает в процессе отладки.

System.Type

Класс представляет тип объекта и широко используется при рефлексии.

- Метод `GetType` этого и других классов позволяет получить информацию о типе объекта или получить ее из манифеста сборки.
- Многочисленные свойства с префиксом `Is` позволяют определить, относится ли объект к заданному типу (например, классу или перечисляемому типу) или обладает определенным атрибутом (скажем, `Private` или `Public`).
- При помощи свойства `Assembly` можно определить сборку, к которой относится объект.

[1] Решение не слишком эффективное, но зато легко программируемое. Поначалу меня сильно беспокоила проблема быстродействия, и я был убежден, что мне придется переделывать приложение. Я был приятно удивлен тем, что при большом количестве объектов (по моим оценкам, около 25 000) быстродействие оказалось вполне приемлемым. Очевидно, сказываются усилия Microsoft по оптимизации создания и выполнения операций с объектами.

- Свойство `BaseType` возвращает информацию о базовом типе класса (см. проект Derivation2 этой главы).
- Поддерживаются разнообразные методы для получения информации о членах объекта. Например, можно получить список методов, свойств, полей и т. д.
- Метод `InvokeMember` предназначен для вызова методов объекта посредством позднего связывания.

System.ValueType

Базовый тип для всех объектов структурного типа. Переопределяет метод `Equals` для сравнения двух структурных объектов на уровне полей. Такое переопределение вполне логично, поскольку переменные объектов ссылочного типа считаются равными, когда они ссылаются на один объект, а переменные структурного типа — когда они содержат одинаковые данные.

Вспомогательные языковые классы

К этой категории относятся классы, которые редко напрямую используются в программах, однако они закладывают основу для работы языков .NET. Например, класс `System.Int32` представляет целочисленный примитив в языках VB .NET и C#. С этими классами следует познакомиться по двум причинам: во-первых, если вы знаете, что типы данных языка являются классами .NET Framework, то будете помнить, что для них можно вызывать методы объектов `System.Object` и `System.ValueType`; во-вторых, некоторые классы содержат дополнительные методы, которые не имеют встроенных аналогов в языке. Эти методы тоже могут использоваться в программах.

System.Array

Примеры непосредственного использования класса встречаются редко, но вы должны помнить, что массивы VB .NET являются производными от этого базового класса. Обратите внимание на метод `Sort`, предназначенный для сортировки массивов. Ниже приведен фрагмент из проекта Misc1:

```
Sub ArrayDemo()
  Dim A() As Integer = {5, 4, 10, 2, 1}
  Array.Sort(A)
  Dim i As Integer
  Console.WriteLine ("Array Tests")
  For Each i In A
    Console.WriteLine (i)
  Next
  Console.WriteLine()
End Sub
```

System.Random

По сравнению с функцией `Rnd` класс обладает большими возможностями построения случайных чисел. В частности, предусмотрена возможность заполнения буфера случайными данными. За дополнительной информацией об этом классе обращайтесь к главе 9.

System.String

В Visual Basic .NET сохранились строковые функции, традиционно считавшиеся одной из сильных сторон VB. Как правило, при работе со строками используются встроенные функции языка, однако класс `String` (заложенный в основу строк VB) содержит некоторые интересные методы, о которых следует знать. Эти методы позволяют дополнять строки заданными символами, удалять из них символы, легко преобразовывать строки в символы и символьные массивы, а также производить лексический анализ строк по заданным символам-разделителям. Рассмотрим пример из приложения Misc1:

```
Public Sub StringSprintDemo()
  Dim S As String = "a,b,c,d,e,f"
  Dim Separators() As Char = {",".Chars(0)}
  Dim SArray() As String
  SArray = S.Split(Separators)
  For Each S In SArray
    Console.WriteLine (S)
  Next
End Sub
```

Результат:

```
a
b
c
d
e
f
```

Не забывайте, что объекты `String` являются неизменными.

System.Text (пространство имен)

Пространство имен `System.Text` содержит ряд дополнительных классов, работающих в тесном взаимодействии с классом `String` для расширения возможностей обработки текстов.

Самым полезным из этих классов, вероятно, является класс `StringBuilder`. По многим возможностям он эквивалентен классу `String`, но не является неизменным. Другими словами, изменения, вносимые в объекты `StringBuilder` (добавление или удаление символов или любые другие модификации текста), приводят к модификации содержимого объекта `StringBuilder`. Объекты `StringBuilder` повышают эффективность построения строковых объектов из нескольких строк меньшего размера (поскольку при этом не приходится создавать несколько временных объектов).

В пространство имен `System.Text` также входят различные объекты для кодирования и декодирования строк в байтовые массивы. В частности, предусмотрены объекты для преобразования строк из ANSI в Unicode и обратно.

Дата и время

В следующую категорию входят объекты, предназначенные для работы с датой, временем и временными интервалами.

System.DateTime

Объект предназначен для хранения даты/времени и заменяет тип VB6 `Date`. Содержит большое количество методов, позволяющих:

- инициализировать объект текущими значениями даты и времени;
- сравнивать два объекта `DateTime`;
- выводить дату и/или время в разных форматах;
- преобразовывать дату в формат OLE (вещественное значение двойной точности) и обратно;
- инициализировать объект `DateTime` по строковому представлению даты и/или времени;
- выполнять операции сложения и вычитания интервалов с объектом DateTime;
- извлекать из объекта нужную информацию (год, месяц, день и т. д.);
- учитывать високосные годы и преобразования из UTC в локальный часовой пояс.

System.TimeSpan

Если объект `DateTime` содержит конкретное значение даты или времени, то объект `TimeSpan` содержит временной интервал. Методы этого объекта позволяют инициализировать его на основании различных единиц измерения (часов, минут, секунд, тактов и т. д.). С объектами `TimeSpan` выполняются операции вычитания, сложения и сравнения.

System.TimeZone

Объект предназначен для работы с информацией о часовых поясах. Позволяет определить локальный часовой пояс, смещение UTC, а также проверить действие режима летнего времени.

Системные классы общего назначения

System.AppDomain

Класс предназначен для работы с доменами приложений.

- Статический метод `GetCurrentThreadId` возвращает идентификатор текущего программного потока.
- Методы `BaseDirectory` и `RelativeSearchPath` позволяют определить каталог и путь к домену приложения.
- Класс содержит методы для загрузки, выгрузки и перечисления сборок. Также предусмотрены средства для загрузки объектов из сборок.

System.Console

Класс упрощает создание консольных приложений. Кстати, он часто использовался в примерах этой книги. Консольные приложения в VB .NET создаются проще оконных приложений и отличаются большей эффективностью.

Консольные приложения используют три потока данных. Они читают данные из входного потока, записывают в выходной поток и выводят сообщения об ошибках в поток ошибок. По умолчанию входной поток ассоциируется с клавиатурой, выходной поток ассоциируется с консольным окном режима командной строки, ошибки тоже направляются в консольное окно. Для работы с этими потоками используются методы `Read`, `ReadLine`, `Write` и `WriteLine`.

Класс `Console` позволяет перенаправить любые из этих потоков в объекты классов `TextReader` или `TextWriter` (см. ниже).

System.Environment

Класс предназначен для получения информации о процессе, работающем в системе. Он заменяет некоторые функции API, а также и методы объекта `App` в VB6. В частности, при помощи класса `System.Environment` вы можете узнать:

- содержимое командной строки;
- текущий каталог;
- имя компьютера;
- версию операционной системы;
- объем используемой памяти;
- версию сборки;
- логические диски, доступные в системе;
- текущее содержимое стека;
- текущее время с момента запуска системы (в тактах);
- текущие значения переменных среды.

Ниже приведен простой пример из проекта Misc1:

```
Sub EnvironmentDemo()
  Console.WriteLine (Environment.CurrentDirectory)
  Console.WriteLine (Environment.OSVersion.ToString)
  Console.WriteLine (Environment.SystemDirectory)
  Console.WriteLine()
End Sub
```

Результат выглядит примерно так:

```
D:\CPBknet\Src1\CH12\Misc1\bin
Microsoft Windows NT 5.0.2195.0
J:\WINNT\System32
```

System.GC

Класс управляет работой сборщика мусора CLR. Примеры использования класса `GC` приведены в главе 5.

System.MarshalByRefObject

Базовый класс для всех объектов, которые должны передаваться по ссылке за пределы домена приложения. Как вы узнали из главы 10, домен приложения определяет границы разделения памяти в .NET (тогда как процесс определяет гра-

ницы разделения памяти в других приложениях Windows). Это означает, что за пределами домена приложения указатели становятся недействительными.

Чтобы вы могли сослаться на объект, находящийся вне домена приложения, создается специальный объект-посредник. Обращения к методам и свойствам этого объекта преобразуются в обращения к методам и свойствам реального объекта — этот процесс называется маршалингом (marshaling). Обычно CLR обеспечивает маршалинг автоматически, не требуя никаких дополнительных усилий с вашей стороны.

Объявление объекта производным от `System.MarshalByRefObject` сообщает CLR о необходимости создания объекта-посредника; обращения к методам и свойствам этого объекта передаются реальному объекту посредством маршалинга. В другом, реже используемом классе `System.ComponentModel.MarshalByValue` маршалинг выполняется по-другому — данные объекта сохраняются посредством сериализации и передаются за пределы домена приложения, где по ним создается копия компонента. Механизм сериализации рассматривается ниже в этой главе.

Исключения

В документации для каждого метода указано, какие исключения он может инициировать. Исключения делятся на категории, которым соответствуют классы, производные от корневого класса `SystemException`. Пользовательские типы исключения следует объявлять производными от `System.ApplicationException`.

Если компоненты .NET используются в COM, то классы исключений преобразуются в разные коды `HRESULT`.

Исключения делятся на две категории: пользовательские и системные. Таким образом, при инициировании исключений программист часто оказывается перед философским выбором: определить собственный тип исключения или же остановиться на системном исключении, если оно подходит для данного случая?

Я полагаю, что если системное исключение достаточно адекватно описывает состояние ошибки, вы вполне можете воспользоваться им. Кроме того, я предпочитаю пользоваться системными исключениями еще и потому, что они хорошо знакомы пользователям моих компонентов. Тем не менее, если системное исключение не отвечает вашим потребностям, я рекомендую определять новые исключения производными от `System.ApplicationException`, а не от классов иерархии `System.SystemException`. Чтобы получить список всех системных исключений, воспользуйтесь описанным выше приложением Derivation2.

System.Exception

Базовый класс для всех исключений. Методы класса позволяют определить источник исключения, получить сообщение с описанием исключения или ссылку на справочный файл с дополнительной информацией об исключении и т. д. Кроме того, при обработке исключения можно произвести трассировку стека.

System.ApplicationException

Класс, производный от `Exception` и обладающий теми же возможностями. Используется в качестве базового класса для всех исключений, инициируемых при-

ложением (в отличие от исключений, инициируемых CLR). Все пользовательские типы исключений следует объявлять производными от этого класса, а не от `System.Exception`.

System.SystemException

Класс, производный от `Exception` и обладающий теми же возможностями. Используется в качестве базового класса для всех исключений, инициируемых системой, и большинства исключений, инициируемых CLR.

Атрибуты

Атрибуты рассматривались в главе 11. Их значения можно получить во время выполнения программы посредством рефлексии; кроме того, атрибуты влияют на работу компилятора и компонентов среды программирования. Чтобы получить список всех системных атрибутов, воспользуйтесь утилитой Derivation2 для поиска всех классов, производных от `System.Attribute`. Особого внимания заслуживают следующие категории.

- Атрибуты, управляющие работой приложения на стадии выполнения (например, `System.ThreadStaticAttribute`).
- Атрибуты сборок. В основном сосредоточены в пространстве имен `System.Reflection` и задаются в файл Assembly.vb. Вероятно, самым важным в этой категории является атрибут `Version`.
- Атрибуты разработки компонентов. В основном находятся в пространстве имен `System.ComponentModel` и позволяют управлять работой компонентов в режиме конструирования, особенно при манипулировании свойствами. В частности, предусмотрены различные возможности управления окном свойств вплоть до создания специализированных редакторов свойств, используемых в режиме конструирования.
- Атрибуты пространства имен `System.Diagnostics` управляют поведением отладчика.

Некоторые важные классы атрибутов перечислены ниже.

System.Attribute

Базовый класс для всех атрибутов.

Общий метод `GetCustomAttributes` позволяет получить пользовательские атрибуты, связанные с типом.

System.AttributeUsageAttribute

Атрибут используется при определении новых атрибутов, производных от класса `System.Attribute`. С его помощью можно указать, должен ли новый атрибут наследоваться производными классами и может ли он устанавливаться многократно.

System.ThreadStaticAttribute

Устанавливается для общих переменных классов или глобальных переменных и указывает на то, что для каждого потока должен создаваться отдельный экземпляр переменной. Такие переменные хранятся в локальной памяти потока.

System.Reflection.AssemblyVersionAttribute

Задает версию сборки. Контроль версии рассматривается в главе 16.

Интерфейсы

В пространстве имен `System` также определяются некоторые часто используемые интерфейсы.

IAsyncResult

Интерфейс реализуется классами, выполняющими асинхронные операции. При помощи этого интерфейса программа может получить информацию о состоянии асинхронной операции или манипулятор, используемый для ожидания завершения асинхронной операции. Обычно интерфейс `IAsyncResult` присутствует в классах файлового ввода-вывода или сетевых сокетов (чтобы получить список классов, реализующих этот интерфейс, воспользуйтесь утилитой Derivation2).

ICloneable

Мы уже рассматривали различия между присваиванием объектов ссылочного и структурного типов. По умолчанию присваивание объекту структурного типа осуществляется поверхностным (shallow) копированием (то есть попарным присваиванием переменных). Следовательно, если структура содержит внутренний объект и вы присваиваете ее другой структуре, обе структуры будут ссылаться на одни и те же внутренние объекты.

Для объектов ссылочных типов новой переменной просто присваивается ссылка на исходный объект.

Интерфейс `ICloneable` обеспечивает стандартный механизм копирования как ссылочных, так и структурных объектов. При реализации этого интерфейса программист определяет метод `Clone`, который должен создавать полностью независимую копию исходного объекта. Фрагмент приложения Misc1, приведенный в листинге 12.1, показывает, как это делается.

Листинг 12.1. Реализация интерфейса ICloneable[1]

```
Class WillClone
  Implements ICloneable
  Public X As Integer
  Public Y As String
  Public Function Clone() As Object Implements ICloneable.Clone
    Dim n As New WillClone()
    n.X = X
```

продолжение ⇙

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Листинг 12.1 (продолжение)

```
    ' Наряду с копированием основного объекта
    ' обычно создаются копии внутренних объектов.
    n.Y = String.Copy(Y)
    Return n
  End Function
End Class

Public Sub WillCloneDemo()
  Dim obj1 As New WillClone()
  obj1.Y = "Test"
  Dim obj2 As WillClone = obj1
  If obj1 Is obj2 And obj1.Y Is obj2.Y Then
    Console.WriteLine ("Objects are the same")
  End If
  obj2 = CType(obj1.Clone(), WillClone)
  If (Not obj1 Is obj2) And (Not obj1.Y Is obj2.Y) And _
  (obj1.Y = obj2.Y) Then
    Console.WriteLine ("Objects are not the same but strings are equal")
  End If
  Console.WriteLine()
End Sub
```

При вызове метода `WillCloneDemo` будут выведены оба сообщения.

IComparable

Интерфейс `IComparable` при сравнении объектов играет примерно такую же роль, как и интерфейс `ICloneable` при присваивании. Он позволяет определить критерии сравнения двух объектов для сортировки и идентификации объектов на основании их содержимого. Интерфейс `IComparable` реализуется более чем 400 объектами .NET Framework.

Используйте этот интерфейс в тех случаях, когда объекты должны сравниваться на основании их содержимого.

IDisposable

В главе 6 было показано, что CLR не обеспечивает детерминированного завершения объектов с использованием секций `Finalize`. С другой стороны, в .NET Framework предусмотрен логически последовательный вариант детерминированного завершения, основанный на взаимодействии контейнеров. Реализуя интерфейс `IDisposable`, объект фактически сообщает пользователям компонента о том, что перед освобождением компонента они должны вызвать метод `Dispose` этого интерфейса. Если пользователь этого не сделает, это может помешать нормальной деинициализации компонента.

Интерфейс `IDisposable` реализуется для тех компонентов, которые должны участвовать в этой схеме. Также следует предусмотреть «страховочную» деинициализацию в обработчике `Finalize` на случай, если пользователь компонента забудет вызвать метод `Dispose`.

Visual Studio автоматически генерирует вызов метода `Dispose` для компонентов, размещаемых в режиме конструирования.

Другие интересные системные классы

Некоторые классы из пространства имен `System` заслуживают внимания хотя бы своими неординарными возможностями.

System.BitConverter

Вас когда-нибудь интересовало, как различные типы данных представляются на физическом уровне? Класс `BitConverter` позволяет преобразовать любой примитивный тип данных в массив байтов, и наоборот. Следующий пример взят из приложения Misc1:

```
Sub BitConverterDemo()
  Dim d As Double = 1.5E+64
  Dim myBitArray() As Byte = BitConverter.GetBytes(d)
  Console.WriteLine (BitConverter.ToString(BitArray))
  Console.WriteLine (BitConverter.ToDouble(BitArray, 0))
  Console.WriteLine()
End Sub
```

Результат выглядит так:

```
78-FB-EF-EE-42-3B-42-4D
1.5E64
```

System.Uri, System.UriBuilder

Два класса содержат разнообразные методы для операций с URL: построения, преобразования относительных URI в абсолютные, разбиения на компоненты и т. д. Следующий пример из приложения Misc1 демонстрирует некоторые возможности класса `System.Uri`:

```
Public Sub URIDemo()
  Dim u As New Uri("http://www.desaware.com")
  Console.WriteLine (u.Host)
  Console.WriteLine (u.Port)
  Console.WriteLine (u.Scheme)
End Sub
```

Приведенный фрагмент выводит следующий результат:

```
www.desaware.com
80
http
```

Коллекции

В Visual Basic .NET существует тип `Collection`, более или менее похожий на объект `Collection`, к которому вы привыкли (самое принципиальное различие заключается в том, что несуществующий тип `Variant` в ссылках заменен типом `Object`).

Сколь бы приятным и знакомым ни выглядел класс `Collection`, он далеко не исчерпывает всех возможностей программиста. Даже те, кто раньше использовал класс VB `Dictionary`, найдут много нового в области работы с коллекциями.

Прежде чем рассматривать новые типы коллекций, стоит присмотреться повнимательнее к принципам их работы.

Класс `Microsoft.VisualBasic.Collection`, как и большинство классов коллекций, реализует следующие три интерфейса.

- `System.Collection.ICollection`. Интерфейс содержит метод `Count` для получения количества объектов в коллекции и некоторые дополнительные методы (проверка безопасности по отношению к потокам и возможности синхронизации доступа к коллекции).
- `System.Collection.IEnumerable`. Интерфейс представляет объект, реализующий интерфейс `IEnumerator`, обеспечивающий последовательный перебор содержимого коллекции. Команда `For...Each` может использоваться для любых объектов, реализующих интерфейс `IEnumerable`.
- `System.Collections.IList`. Интерфейс предоставляет списковый доступ к коллекции и содержит методы для добавления, удаления и поиска объектов. Свойство `Item` этого интерфейса является свойством по умолчанию объекта коллекции, что обеспечивает возможность прямого индексированного доступа к элементам коллекции.

Ассоциативные коллекции реализуют интерфейс `System.Collections.IDictionary`. К этой категории относятся коллекции, в которых хранятся пары «ключ/значение» с возможностью быстрого поиска по ключу и обеспечением уникальности ключей.

System.CollectionBase и пользовательские коллекции

При программировании открытых коллекций я всегда рекомендую использовать сильную типизацию элементов. Это избавит вас от необходимости включать в программу проверку ошибок на случай включения в коллекцию посторонних объектов, с которыми программа не умеет работать.

В Visual Basic .NET создание типизованных коллекций заметно упрощается. Все, что от вас потребуется, — это объявить класс коллекции производным от класса `CollectionBase`, спроектированного специально для этой цели. Проект CollectionDemo (листинг 12.2) показывает, как это делается.

Листинг 12.2. Файл Module1.vb проекта CollectionDemo

```
' Использование For...Each при работе с коллекциями
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Module Module1
  Enum MyFish
    OneFish
    TwoFish
    RedFish
    BlueFish
  End Enum


  Public Class FishCollection
    Inherits CollectionBase
    Public Function Add(ByVal value As MyFish) As Integer
```

```
    MyBase.List.Add (value)
  End Function
  Public Sub Insert(ByVal index As Integer, ByVal value As MyFish)
    List.Insert(index, value)
  End Sub

  Public Function IndexOf(ByVal value As MyFish) As Integer
    Return List.IndexOf(value)
  End Function

  Public Function Contains(ByVal value As MyFish) As Boolean
    Return List.Contains(value)
  End Function

  Public Sub Remove(ByVal value As MyFish)
    List.Remove (value)
  End Sub

  Public Sub CopyTo(ByVal array() As MyFish, ByVal index As Integer)
    List.CopyTo(array, index)
  End Sub

  Default Property Item(ByVal index As Integer) As MyFish
    Get
      Return CType(MyBase.List.Item(index), MyFish)
    End Get
    Set(ByVal Value As MyFish)
      MyBase.List.Item(index) = value
    End Set
  End Property

  Protected Overrides Sub OnInsert(ByVal index As Integer, _
  ByVal value As Object)
    If Not TypeOf (value) Is MyFish Then
      Throw New ArgumentException("Invalid type")
    End If
  End Sub

  Protected Overrides Sub OnSet(ByVal index As Integer, _
    ByVal oldValue As Object, ByVal newValue As Object)
    If Not TypeOf (newValue) Is MyFish Then
      Throw New ArgumentException("Invalid type")
    End If
  End Sub

  Protected Overrides Sub OnValidate(ByVal value As Object)
    If Not TypeOf (value) Is MyFish Then
      Throw New ArgumentException("Invalid type")
    End If
  End Sub
End Class


Sub Main()
  Dim col As New Collection()
  Console.WriteLine ("VB Collection Type is: " & _
    col.GetType.FullName)

  Dim F As New FishCollection()
```

продолжение ⇩

Листинг 12.2 (продолжение)

```
    Dim il As IList
    F.Add (MyFish.OneFish)
    F.Add (MyFish.TwoFish)
    F.Add (MyFish.RedFish)
    F.Add (MyFish.BlueFish)
    il = F
    Try
      il.Add ("Something not a fish")
    Catch e As Exception
      Console.WriteLine (e.Message)
    End Try

    Dim afish As MyFish
    For Each afish In F
      Console.WriteLine (afish.ToString)
    Next

    afish = F(1)

    Console.WriteLine (afish.ToString)

    Console.ReadLine()

  End Sub

End Module
```

В программе определяется перечисляемый тип с именем `MyFish`. Коллекция `FishCollection` предназначена только для работы с объектами этого типа. Методы `Add`, `Insert`, `IndexOf`, `Contains`, `Removes` и `CopyTo` принимают только объекты типа `MyFish` и ограничиваются простым вызовом соответствующего метода базового класса, принимающего любые объекты. Свойство `Item` является свойством по умолчанию (допустимо для параметризованных свойств) и также передает обращение свойству `Item` базового класса.

Помимо реализации методов коллекции, мы также переопределяем методы `OnInsert`, `OnSet` и `OnValidate`. Эти методы предоставляются базовым классом, чтобы производный класс мог выполнить дополнительную проверку перед выполнением операции. В приведенном примере мы убеждаемся в том, что объект действительно относится к правильному типу.

Зачем это нужно?

А что произойдет, если для ссылки на класс будет использована переменная типа `IList`? Ссылка будет относиться к реализации интерфейса, принадлежащей базовому классу, что позволит сохранять в коллекции объекты других типов. Наш класс спроектирован таким образом, чтобы производный класс выполнял дополнительную проверку типа, тем самым закрывая эту «лазейку».

Подробнее о коллекциях

Любой объект можно наделить возможностью перебора содержимого при помощи синтаксической конструкции `For...Each`. Для этого класс реализует интерфейс `IEnumerable`. В листинге 12.3 продемонстрировано использование `IEnumerable` на примере проекта ForEach.

Листинг 12.3. Реализация интерфейса IEnumerable

```
Public Class PseudoCollection
  Implements IEnumerable
  Private DummyData() As String = {"One Fish", "Two Fish", _
    "Red Fish", "Blue Fish"}

  Function GetEnumerator() As IEnumerator Implements _
  IEnumerable.GetEnumerator
    Dim myEnumerator As New EnumeratorClass()
    ReDim myEnumerator.Snapshot(UBound(DummyData))
    DummyData.CopyTo(myEnumerator.Snapshot, 0)
    Return myEnumerator
  End Function

  Public Class EnumeratorClass
    Implements IEnumerator

    Dim CurrentIndex As Integer = -1
    Public Snapshot() As String

    ' Перейти в начало (перед первым элементом)
    Sub Reset() Implements IEnumerator.Reset
      CurrentIndex = -1
    End Sub

    ReadOnly Property Current() As Object Implements IEnumerator.Current
      Get
        Return (Snapshot(CurrentIndex))
      End Get
    End Property

    Function MoveNext() As Boolean Implements IEnumerator.MoveNext
      CurrentIndex += 1
      Return (CurrentIndex <= UBound(Snapshot))
    End Function

  End Class
```

Класс PseudoCollection содержит массив строк, содержимое которого должно перебираться в цикле `For...Each`. Прежде всего необходимо реализовать интерфейс `IEnumerable`. Этот интерфейс предоставляет единственный метод `GetEnumerator`, который возвращает ссылку на отдельный объект `Enumerator`, реализующий интерфейс `IEnumerator`.

При вызове `GetEnumerator` текущее состояние данных сохраняется в массиве `SnapShot`, содержащем копию данных на момент вызова функции `GetEnumerator`. Перебор данных, хранящихся в массиве, осуществляется методами интерфейса `IEnumerator`: `Reset`, `Current` и `MoveNext`.

Конечно, при желании можно создать пользовательскую коллекцию с самостоятельной реализацией `IList`, `IComparable` и `ICollection`, но в большинстве случаев двух продемонстрированных решений оказывается вполне достаточно.

Другие коллекции

В начале этого раздела я упомянул о том, что помимо класса коллекций Visual Basic (`Microsoft.VisualBasic.Collection`) существуют и другие варианты.

В пространствах имен System.Collections и System.Collections.Specialized определено несколько типов коллекций. В листинге 12.4 продемонстрирован пример использования класса SortedList из приложения ForEach.

Листинг 12.4. Использование класса SortedList

```
Dim I() As Integer = {3, 8, 4, 6, 10, 7, 9}
Dim C As New Collections.SortedList()
Dim P As New PseudoCollection()

Dim x As Integer
For x = 0 To UBound(I)
  C.Add(I(x), I(x))
Next

Dim IntegerIterator As Object

Console.WriteLine ("Array iteration")
For Each IntegerIterator In I
  Console.Write (IntegerIterator.ToString & ", ")
Next
Console.WriteLine (ControlChars.CrLf & "Collection iteration")

Dim DictIterator As DictionaryEntry
For Each DictIterator In C
  Console.Write (DictIterator.Value.ToString & ", ")
Next

Dim StringIterator As String
Console.WriteLine (ControlChars.CrLf & "Custom enumerable object")
For Each StringIterator In P
  Console.Write (StringIterator.ToString & ", ")
Next

Console.ReadLine()
```

Несомненно, коллекция с автоматической сортировкой содержимого заслуживает внимания. В оставшейся части этого раздела будут представлены другие примеры коллекций.

Приведенный ниже список не полон. Я рекомендую самостоятельно исследовать пространства имен System.Collections и System.Collections.Specialized. Помните, что большинство классов коллекций небезопасно по отношению к потокам.

System.Collection.ArrayList

Удобная разновидность коллекций общего назначения.

System.Collections.BitArray

Коллекция содержит массив битов, принимающих значения True и False, и обеспечивает самый компактный способ хранения большого количества флаговых битов.

Но самая интересная особенность этого объекта заключается в том, что он может инициализироваться байтовым массивом. Это позволяет взять произвольный тип данных (вероятно, преобразованный классом System.BitConverter) и проверить и/или установить в нем отдельные биты.

System.Collections.DictionaryBase

Используется для создания ассоциативных массивов с сильной типизацией.

System.Collections.HashTable

Класс реализует хэш-таблицу — эффективную разновидность ассоциативных коллекций, обеспечивающую быстрый доступ к любому объекту.

System.Collections.Queue

Реализует очередь (принцип FIFO — «первым пришел, первым вышел»).

System.Collections.SortedList

Ассоциативная коллекция, в которой элементы сортируются по значению ключа.

System.Collections.Stack

Стек (принцип LIFO — «последним пришел, первым вышел»).

System.Collections.Specialized.BitVector

Объект позволяет разделить 32-разрядное число на несколько полей, к каждому из которых можно обращаться напрямую. Читателям, знакомым с C++, он напомнит битовые поля в C++.

System.Collections.Specialized.String

Коллекция для работы со строками, безопасная по отношению к типам.

Вывод

Вскоре появятся целые книги, посвященные выводу графики и текста в VB .NET. Впрочем, для описания основных концепций графического вывода много места не требуется, однако восприятие этого материала во многом зависит от того, как вы предпочитаете работать с графическими операциями в приложениях VB6.

Если вы привыкли использовать графические методы VB6...

Забудьте все, что вы знали о графическом выводе, в том числе и о работе с растровыми изображениями. Графические примитивы Visual Basic (такие, как `Line`, `Circle` и `PSET`) ушли навсегда и больше не вернутся. Пропало даже простейшее присваивание при копировании изображений[1]:

```
Picture1.Picture = Picture2.Image
```

[1] Свойство Image графического поля (picture box) аналогично свойству Picture VB6, однако оно возвращает ссылку на ранее присвоенное изображение вместо текущего содержимого окна.

Забудьте о дурацкой необходимости измерять расстояние в твипах, что приводило к ошибкам округления при позиционировании элементов и выборе размера шрифта. Отныне практически во всех операциях расстояния задаются в пикселах. Исчезли такие свойства, как `ScaleMode`, `ScaleWidth` и `ScaleHeight`. Конечно, нетривиальные режимы масштабирования по-прежнему доступны, но для работы с ними используются средства .NET Framework.

У нового подхода есть как положительные, так и отрицательные стороны. В вашем распоряжении оказываются гораздо более мощные графические возможности, но чтобы научиться ими пользоваться, придется приложить немало усилий.

В нескольких ближайших разделах я помогу вам правильно взяться за дело.

Если вы привыкли выполнять графические операции средствами Win32 API...

Забудьте все, что вы знали о графическом выводе.

Шутка.

На самом деле вам повезло, и вы очень быстро освоите новые графические возможности. А если вы к тому же прочитали описание графических функций в моей книге «Visual Basic Programmer's Guide to the Win32 API»[1], то сможете быстро просмотреть несколько ближайших разделов, обращая внимание лишь на изменения относительно Win32 API.

GDI умер, да здравствует GDI+

Сокращение GDI означает «интерфейс графического устройства» (Graphics Device Interface). В доисторические времена[2] программа, пожелавшая вывести что-либо на экран или принтер, должна была располагать полной информацией об устройстве вывода, в том числе знать размеры поверхности вывода и набор управляющих команд. Таким образом, в каждую программу приходилось включать самостоятельную реализацию графического вывода для разных устройств при разных вариантах разрешения.

Библиотека GDI обеспечивает аппаратно-независимый подход к графическому выводу. Приложения работают с универсальными графическими методами, а GDI в сочетании с драйверами устройств Windows выводят изображение на устройство, будь то принтер, окно или метафайл (записанная последовательность графических команд, которая может воспроизводиться приложением). Приложение может ограничиться конкретными значениями разрешения и размера, а может определить логическое окно со стандартной или пользовательской системой координат и поручить GDI его отображение в конкретную область устройства.

Термин «контекст устройства» (DC, device context) относится к числу важнейших понятий GDI. Контекст устройства представляет собой объект, ассоциирующий графические команды с устройством вывода. В Windows его можно по-

[1] См. сноску на с. 296.

[2] Эпоха DOS, период с конца 70-х годов до начала 90-х.

лучить для любого устройства вывода. Все графические команды GDI работают с контекстами устройств (рис. 12.1).

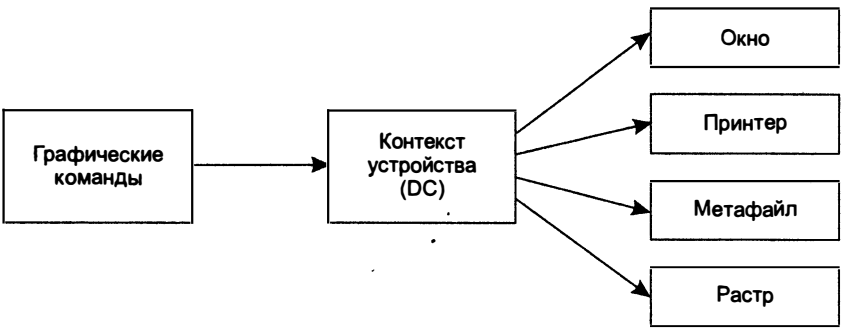


Рис. 12.1. Графические операции в GDI

Результаты выполнения традиционных графических функций Win32 API зависят от объектов GDI (перьев, кистей, шрифтов и растров), «выбранных» в контексте устройства. Таким образом, в Win32 контексты устройств обладают определенным состоянием, зависящим от текущего набора выбранных объектов. Программа должна перевести контекст устройства в нужное состояние перед выводом и, что еще важнее, — восстановить прежний контекст после завершения вывода, чтобы не мешать работе других частей программы. Это особенно важно при вызове функций GDI в Visual Basic 6. Если при вызове функций API программа не восстановит прежнее состояние контекста, это может привести к нарушению работы обычных графических команд VB6.

В .NET Framework на смену GDI приходит GDI+.

Главное различие между GDI и GDI+ заключается в том, что графический вывод теперь не зависит от текущего состояния. Аналогом контекста устройства в GDI+ является объект `System.Drawing.Graphics` (рис. 12.2). Используемые перья, кисти и шрифты задаются непосредственно при вызове графических команд. Вам уже не придется беспокоиться о том, что вызов графических функций может привести к побочным эффектам.

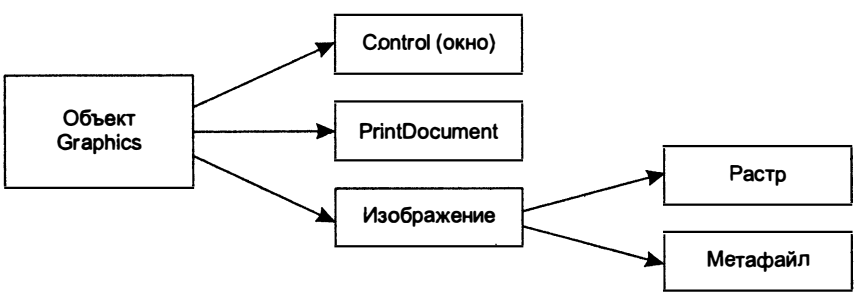


Рис. 12.2. Графические операции в GDI+

Следующий фрагмент приложения GraphicsDemo демонстрирует использование объектов `Graphics`, `Brush`, `Pen` и `Font`.

```
Private Sub cmdDraw_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdDraw.Click
  Dim g As Graphics
  ' Также можно использовать стандартные кисти.
  Dim b As SolidBrush = New SolidBrush(color.Beige)
  g = pictureBox1().CreateGraphics() ' Получить объект Graphics
  g.FillRectangle(b, pictureBox1().ClientRectangle)
  ' Конструктор для быстрой модификации существующих шрифтов
  Dim f As Font = New Font(Me.Font, FontStyle.Bold)
  g.DrawString("Some Text", f, brushes.Black, 10, 10)
  g.DrawRectangle(pens.Blue, 50, 50, 50, 50)
  g.DrawLine(pens.Red, 50, 50, 150, 150)
  g.Dispose() ' Не забывайте о вызове Dispose!
  b.Dispose()
  f.Dispose()
End Sub
```

Метод `CreateGraphics` класса `System.Windows.Forms.Control` возвращает объект `Graphics`, предназначенный для вывода в элементе управления. Все элементы пользовательского интерфейса Windows (в том числе и формы) определяются производными от `System.Windows.Forms.Control`, поэтому они всегда поддерживают этот метод.

На основе базового класса `System.Drawing.Brush` определяются кисти разных типов, от текстурных до однородных. Мы используем однородную кисть, созданную на базе одного из стандартных цветов, определенных в классе `System.Drawing.Colors`. Конструктор шрифта, использованный в нашем примере, берет за основу уже существующий шрифт и изменяет атрибуты его начертания. Существует много конструкторов для создания шрифтов.

Класс `System.Drawing.Graphics` содержит большое количество графических методов. Обратите внимание: при вызове каждого метода указываются перья, кисти и шрифты, необходимые для выполнения графической операции. Объект `Graphics` не поддерживает информации о состоянии.

Объекты `Brush`, `Font`, `Pen`, `Graphics` и другие объекты GDI+ представляют собой «обертки» для базовых объектов GDI, поэтому разработчики должны были предусмотреть средства для освобождения последних. Все перечисленные объекты реализуют интерфейс `IDisposable`, поэтому после завершения графического вывода следует вызвать метод `Dispose`. Впрочем, если вы забудете это сделать, ничего особо страшного не произойдет. Метод `Finalize` этих объектов автоматически освободит базовые объекты GDI, однако вызов `Dispose` работает эффективнее.

Преимущества GDI+ перед GDI не исчерпываются отсутствием зависимости от состояния. В GDI+ поддерживаются такие графические операции, как наложение (blending) и сглаживание (anti-aliasing), а также изощренные алгоритмы масштабирования растровых изображений. Большинство новых возможностей сосредоточено в пространстве имен `System.Drawing.Drawing2D`. Ниже приведен простой пример: заполнение формы градиентной заливкой.

```
Private Sub cmdGradient_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles cmdGradient.Click
  Dim lb As New Drawing2D.LinearGradientBrush(Me.DisplayRectangle, _
  color.Blue, color.Red, Drawing2D.LinearGradientMode.Horizontal)
  Dim g As Graphics = Me.CreateGraphics()
```

```
  g.FillRectangle(lb, Me.DisplayRectangle)
  lb.Dispose()
  g.Dispose()
End Sub
```

Растровые изображения

Класс `System.Drawing.Bitmap` предназначен для загрузки растровых изображений в разных форматах и из разных источников. Для полученного растра можно создать объект `Graphics`, упрощающий вывод в этом растре. Объект `Bitmap` также позволяет задавать значения отдельных пикселов (замена команды `PSet` VB6). Данная возможность продемонстрирована во фрагменте приложения GraphicsDemo, приведенном в листинге 12.5.

Листинг 12.5. Операции с растровыми изображениями

```
Private Sub cmdBitmap_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles cmdBitmap.Click
  Dim g, g2 As Graphics
  g = pictureBox1().CreateGraphics()
  Dim bm As New Bitmap(pictureBox1().ClientRectangle.Width, _
  pictureBox1().ClientRectangle.Height, g)

  g2 = Graphics.FromImage(bm)
  dwgraphics.CopyImage(g, g2, pictureBox1().ClientRectangle.Width, _
  pictureBox1().ClientRectangle.Height)

  g2.DrawLine(pens.Green, 60, 50, 160, 150)
  bm.SetPixel(1, 1, color.Red)
  bm.SetPixel(1, 2, color.Blue)
  bm.SetPixel(1, 3, color.Green)

  ' Другой способ получения графических объектов
  g = Graphics.FromHwnd(pictureBox2().Handle)
  Dim g3 As Graphics = graphics.FromHwnd(pictureBox2().Handle)
  g3.DrawImage(bm, 0, 0, pictureBox2().DisplayRectangle.Width, _
  pictureBox2().DisplayRectangle.Height)
  g.Dispose()
  g2.Dispose()
  g3.Dispose()
  bm.Dispose()

End Sub
```

В GDI+ не предусмотрено простой возможности копирования растровых изображений из существующих окон. Никаких объяснений, кроме невероятного просчета разработчиков, предложить не могу. В листинге 12.6 приведен простой класс, копирующий текущее содержимое окна в объект `Graphics`, представляющий растровое изображение или другое окно. Копирование осуществляется традиционной функцией API `BitBlt`.

Листинг 12.6. Копирование изображения в окне функцией BitBlt

```
Private Module APIDeclarations
  Friend Const SRCCOPY As Integer = &HCC0020&
  Friend Declare Ansi Function BitBlt Lib "gdi32" (ByVal hDestDC _
    As IntPtr, ByVal x As Integer, ByVal y As Integer, _
```

продолжение ⇩

Листинг 12.6 (продолжение)

```
    ByVal nWidth As Integer, ByVal nHeight As Integer, _
    ByVal hSrcDC As IntPtr, ByVal xSrc As Integer, _
    ByVal ySrc As Integer, ByVal dwRop As Integer) As Integer
End Module

Public Class dwGraphics
  Shared Sub CopyImage(ByVal Source As System.Drawing.Graphics, _
  ByVal Dest As System.Drawing.Graphics, ByVal Width As Integer, _
  ByVal Height As Integer)
    Dim dhdc, shdc As IntPtr
    dhdc = Dest.GetHdc
    shdc = Source.GetHdc
    BitBlt(dhdc, 0, 0, Width, Height, shdc, 0, 0, SRCCOPY)
    Dest.ReleaseHdc (dhdc)
    Source.ReleaseHdc (shdc)
  End Sub
End Class
```

Могу лишь предположить, что в окончательной версии этот недостаток будет устранен, поскольку графическими функциями API пользоваться не рекомендуется (причины описаны в главе 15).

Стратегии изучения GDI+

Как было сказано выше, по GDI+ вполне можно написать отдельную книгу. Тем не менее основные правила графического вывода просты.

- Получите объект `System.Drawing.Graphics` для той поверхности, на которой вы хотите выводить.
- Создайте перья, кисти, шрифты и другие объекты, используемые командами вывода.
- Воспользуйтесь графическими методами объекта `Graphics` для вывода на поверхности.
- После завершения вывода освободите все объекты методом `Dispose`.

Мой совет: обратитесь к документации и внимательно прочитайте описания классов следующих пространств имен.

- `System.Drawing` — все базовые классы GDI+.
- `System.Drawing.Design` — классы диалоговых окон для загрузки растровых изображений, выбора шрифтов и других типовых операций пользовательского интерфейса, используемых при графическом выводе.
- `System.Drawing.Drawing2D` — классы для выполнения более сложных графических операций (в частности, наложения и градиентных заливок).
- `System.Drawing.Imaging` — классы поддержки метафайлов и нетривиальных графических операций (например, преобразований цветовых пространств).
- `System.Drawing.Printing` — классы поддержки печати (этой теме посвящен следующий раздел).

Утилита Derivation2 поможет вам лучше разобраться в иерархии объектов в этих пространствах. Например, проверка класса `System.Drawing.Brush` показывает, что этот класс является базовым для следующих классов:

```
System.Drawing.Drawing2D.HatchBrush
System.Drawing.Drawing2D.LinearGradientBrush
System.Drawing.Drawing2D.PathGradientBrush
System.Drawing.SolidBrush
System.Drawing.TextureBrush
```

Печать

Само название пространства имен печати, `System.Drawing.Printing`, подсказывает принцип его работы. Как только что было сказано, GDI+ заставляет программистов VB6 перейти от старомодных графических команд VB на новый стиль графического программирования, знакомый программистам Win32 API. Вероятно, вас не удивит, что аналогичные изменения произошли и в области печати.

Суть этих изменений можно выразить всего четырьмя словами: *объекта* `Printer` *больше нет.*

Нет, он не был переименован. Исчезла сама концепция объекта печати.

Не паникуйте. Стоит усвоить несколько базовых принципов, и вы поймете, что подход .NET к печати невероятно прост и удобен — и при этом обладает огромными возможностями.

В этом разделе я постараюсь представить печать в .NET в нужном ракурсе, а остальное вы найдете в документации.

На рис. 12.3 изображена простейшая модель печати, использовавшаяся в VB6.

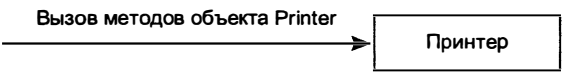


Рис. 12.3. Печать в VB6

Вы передаете на принтер команды создания документа, вывода графики и текста, формирования новой страницы и фактического начала печати.

В .NET используется более сложная модель, показанная на рис. 12.4.

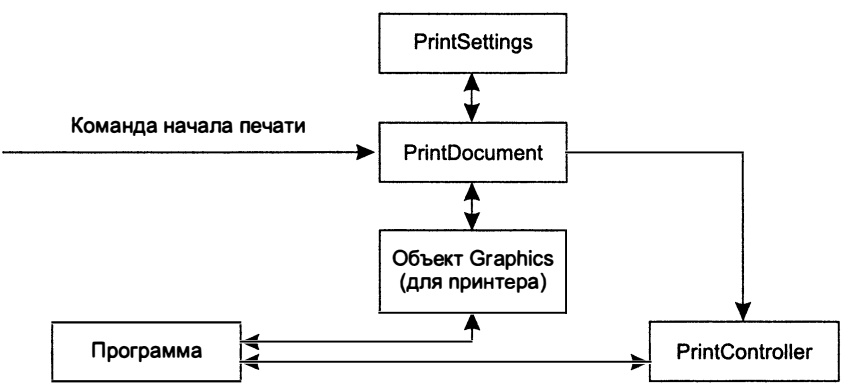


Рис. 12.4. Печать в .NET

И я еще говорю об упрощении модели печати?!

Да, в большинстве случаев. Присмотритесь повнимательнее.

Суть происходящего заключается в следующем. Вместо того чтобы печатать каждую страницу отдельной функцией, мы вызовом одной функции сообщаем .NET о своем намерении приступить к печати. Далее .NET Framework инициирует событие, сообщая о готовности к выводу новой страницы. В процессе обработки этого события ваша программа формирует печатную страницу.

На происходящее можно взглянуть иначе. Печать трех страниц в VB6 описывается следующим фрагментом псевдокода:

```
Sub Print
  Напечатать страницу 1
  Напечатать страницу 2
  Напечатать страницу 3
End Sub
```

В VB .NET аналогичный фрагмент выглядит иначе:

```
Sub Print
  Печать документа
End Sub

Sub PrintDocument_PrintPage Handles событие PrintPage документа
  Напечатать страницу
  Вернуть признак продолжения печати
End Sub
```

В проекте PrintingDemo эта абстрактная схема преобразуется в реальный код.

На форме находится текстовое поле для ввода печатаемой строки. Мы несколько упрощаем свою задачу и используем для печати шрифт, ассоциированный с текстовым полем. Конечно, вместо него можно воспользоваться любым объектом `Font` (в том числе и выбранным в стандартном диалоговом окне `System.Windows.Form.FontDialog`). На форме также присутствует графическое поле с растровым изображением.

Печать начинается с создания объекта `System.Drawing.Printing.PrintDocument`. С этим объектом связаны два других объекта, `PrinterSettings` и `PrintController`. Объект `PrinterSettings` определяет принтер и параметры печати. Объект `PrintController` непосредственно управляет печатью. Обычно при печати используется стандартный объект `PrintController`, но вы можете определить производный класс для нестандартной печати, например для отображения статусной информации в процессе печати.

В нашем примере для настройки параметров печати (объект `PrintSettings`) используется стандартное диалоговое окно, знакомое каждому пользователю Windows. Для работы с этим окном используются объекты `PrintDialog`. При выходе из вызова `prDialog.ShowDialog` объект `PrinterSettings` соответствует принтеру, выбранному в окне.

Таким образом, в VB .NET печать на любом доступном принтере производится элементарно — вам не придется беспокоиться о том, какой принтер выбран по умолчанию. Естественно, выбрать принтер и изменить настройки конкретного задания печати можно и без диалогового окна, простыми операциями с соответствующими объектами из пространства имен `System.Drawing.Printing`.

Задание печати создается вызовом prDoc.Print. Но перед вызовом этого метода необходимо указать событие, которое должно инициироваться для каждой страницы. Как было показано в главе 10, событие в действительности является делегатом и при помощи команды AddHandler элементарно связывается с любым методом — в нашем примере это метод PagePrintFunction.

```
Private Sub cmdPrint_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdPrint.Click
  Dim prDialog As New PrintDialog()
  Dim prDoc As New Drawing.Printing.PrintDocument()
  prDoc.DocumentName = "My new printed document"
  prDialog.Document = prDoc
  prDialog.ShowDialog()

  ' Подключить событие, вызываемое для каждой страницы
  AddHandler prDoc.PrintPage, AddressOf Me.PagePrintFunction
  prDoc.Print()
  prDoc.Dispose()
  prDialog.Dispose()
End Sub
```

Метод PagePrintFunction (обработчик события PrintDocument.PrintPage) получает два параметра: ссылку на объект PrintDocument (отправитель) и ссылку на объект PrintPageEventArgs, содержащий большое количество свойств для получения сведений о печатаемой странице.

Важнейшим из этих свойств является объект Graphics. Да, все верно — это тот самый объект Graphics, упоминавшийся в предыдущем разделе. В нашем примере выводятся две строки текста: данные ограничивающего прямоугольника (для наглядного представления координат, используемых страницей) и содержимое текстового поля. Обратите внимание на вызов метода GetHeight объекта Font, при помощи которого мы получаем высоту строки текста на текущей поверхности устройства. Метод DrawImage масштабирует растровое изображение по свободной части страницы (при запуске этой программы приготовьтесь к тому, что печать займет много времени, особенно на принтерах PostScript). Задавая значение свойства PrintPageEventArgs.HasMorePages, можно сообщить объекту PrintController, будут ли печататься дополнительные страницы[1].

```
Public Sub PagePrintFunction(ByVal sender As Object, _
  ByVal e As Printing.PrintPageEventArgs)
  Dim LineHeight, LineNumber As Single
  LineHeight = txtText().Font.GetHeight(e.Graphics)

  Dim TextRect As New RectangleF(0, LineHeight * LineNumber, _
  e.PageBounds.Width, e.PageBounds.Height - LineHeight * LineNumber)
  Dim SF As New StringFormat(StringFormatFlags.LineLimit)

  e.Graphics.DrawString(e.PageSettings.Margins.ToString, _
  txtText().Font, brushes.Black, TextRect)
  LineNumber += 2
  e.Graphics.DrawString(txtText().Text, txtText().Font, _
  Brushes.Black, 0, LineHeight * LineNumber)
  LineNumber += 1
```

[1] В данном примере эта возможность не используется.

```
  e.Graphics.DrawImage(pictureBox1().Image, 0, _
  LineNumber * LineHeight, e.PageBounds.Width, _
  e.PageBounds.Height - LineNumber * LineHeight)
End Sub
```

Прежде чем двигаться дальше, следует учесть два обстоятельства.

- Делегаты определяют `PagePrintFunction` и другие обработчики событий объектов `PrintDocument` и `PrintController`. Как говорилось выше, делегат может связываться с конкретным экземпляром класса. Таким образом, в приложениях, работающих с несколькими документами, делегата можно связать с конкретным экземпляром класса документа и использовать переменные этого класса для отслеживания служебной информации (например, количества страниц).
- Поскольку во всех операциях вывода вместо конкретного графического устройства используется объект `Graphics`, вы можете написать обобщенные графические функции, передавать им объект `Graphics` в качестве параметра и использовать их как для печати, так и для вывода на экране.

Последняя возможность продемонстрирована в обработчике `cmdPreview_Click` приложения PrintingDemo.

```
Private Sub cmdPreview_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdPreview.Click
  Dim previewDialog As New PrintPreviewDialog()
  Dim prDoc As New Drawing.Printing.PrintDocument()
  prDoc.DocumentName = "My new printed document"
  ' Подключить событие, вызываемое для каждой страницы
  AddHandler prDoc.PrintPage, AddressOf Me.PagePrintFunction
  previewDialog.Document = prDoc
  previewDialog.ShowDialog()
  prDoc.Dispose()
  previewDialog.Dispose()
End Sub
```

Объект `PrintDocument` создается точно так же, как было показано выше. Вызов диалогового окна `PrintDialog` пропускается (хотя с его помощью можно направить вывод на другой принтер для предварительного просмотра). Далее мы создаем объект `PrintPreviewDialog`, обеспечивающий стандартное окно предварительного просмотра, и указываем, что для вывода должен использоваться объект `PrintDocument`. Объект `PrintDocument` связывается с методом `PagePrintFunction`, описанным выше.

При вызове метода `Show` для объекта `PrintPreviewDialog` на экране появляется предварительное изображение вашего документа, причем для этого используется тот же программный код, что и при непосредственном выводе на печать!

Завидуй, VB6!

Итак, к новой архитектуре печати действительно нужно привыкнуть, но стоит вам понять базовые принципы — и проблем с печатью не будет. Я уже не говорю о том, что вы сможете пользоваться новыми возможностями .NET по настройке принтера и выбору конфигурации печати и новыми средствами графического вывода.

За дополнительной информацией о печати в .NET обращайтесь к описанию объектов пространства имен `System.Drawing.Printing`.

Ввод-вывод

В VB .NET сохранена поддержка традиционных файловых команд BASIC (`Get`, `Put`, `Print#` и т. д.) и традиционных операций файловой системы (`CurDir`, `ChDir`, `Kill` и т. д.). Все эти команды входят в пространство имен `Microsoft.VisualBasic`.

Конечно, никто не запрещает вам пользоваться этими командами, если вы к ним привыкли, но все же я советую познакомиться с пространством имен `System.IO`. При помощи объектов этого пространства можно выполнять все традиционные файловые операции VB, но, что еще важнее, — в нем предусмотрены объекты для других типов операций ввода-вывода. Объекты `System.IO` применяются в широком спектре приложений, от сериализации до криптографии и сетевой пересылки данных.

Впрочем, использование средств `System.IO` связано с определенными затруднениями. Дело в том, что в .NET Framework принят новый подход к вводу-выводу, незнакомый многим программистам VB6 (и даже программистам C++). На нескольких ближайших страницах я постараюсь представить основные принципы ввода-вывода в .NET, чтобы вы поняли логику взаимодействия этих объектов и научились успешно пользоваться ими на практике.

Главный принцип ввода-вывода в .NET понять несложно: по аналогии с тем, как GDI реализует абстрактный уровень вывода на разных графических поверхностях, классы `System.IO` реализуют абстрактный уровень чтения и записи для разных устройств ввода-вывода.

Большинство классов, представляющих устройства ввода-вывода, являются производными от класса `System.IO.Stream`. Этот класс интерпретирует устройство как поток байтов (доступный для чтения или записи) и позволяет выполнять следующие операции:

- чтение одного или нескольких байтов данных;
- запись одного или нескольких байтов данных;
- асинхронное чтение или запись (с дополнительной возможностью оповещения о завершении операции);
- физическая запись данных из промежуточного буфера на устройство;
- переход к заданной позиции в потоке данных;
- закрытие потока (устройства) после завершения всех операций.

Конечно, для некоторых устройств поддерживается лишь часть операций, но вы можете проверить факт поддержки той или иной операции для каждого конкретного устройства.

В любом классе устройства, производном от `Stream`, могут определяться дополнительные методы. Например, вы можете полностью или частично заблокировать объект `FileStream`, чтобы предотвратить одновременный доступ к нему со стороны нескольких процессов.

Некоторые классы потоковых устройств изображены на рис. 12.5.

Классы, показанные на рисунке, представляют файлы, блоки памяти, зашифрованные данные, сокеты, сетевые данные HTTP, данные XML и т. д. Полный

список классов, производных от `System.IO.Stream`, можно получить при помощи утилиты Derivation2.

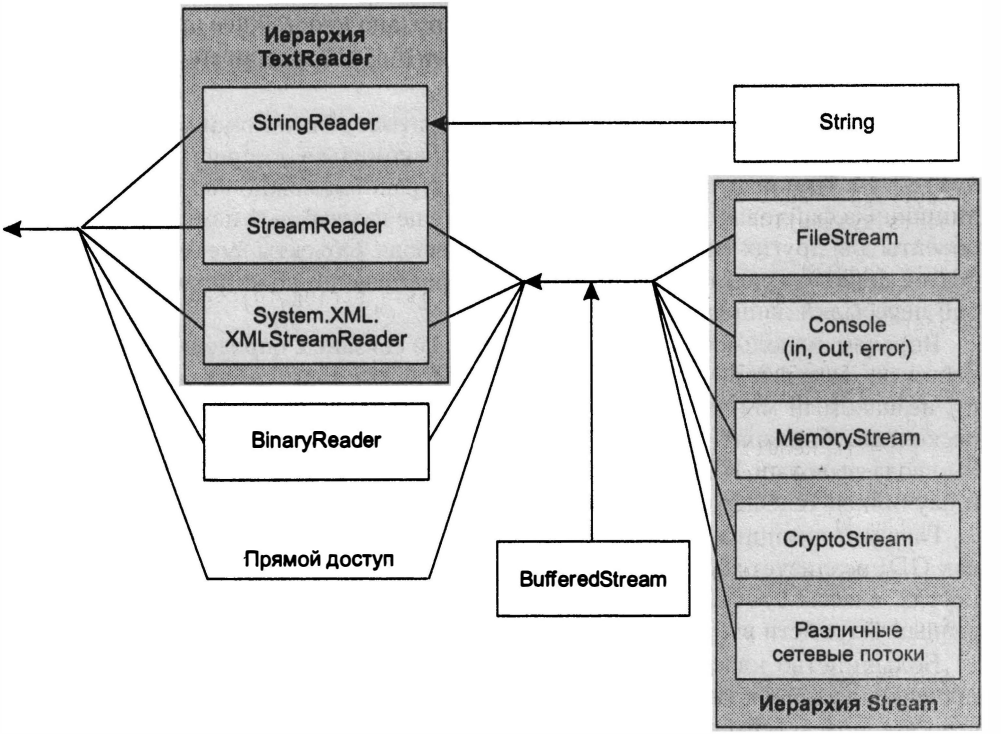


Рис. 12.5. Ввод-вывод в .NET

Каждый поток может быть связан с объектом `BufferedStream`. Буферизованный вывод повышает быстродействие при выполнении большого числа коротких операций ввода-вывода, например при посимвольном чтении данных из устройства. Объект `BufferedStream` обеспечивает промежуточный буфер заданного размера для уменьшения количества обращений к физическому устройству.

Хотя операции чтения и записи могут выполняться непосредственно с потоками, в вашем распоряжении также находятся специализированные классы, производные от класса `TextReader`. Этот класс оптимизирован для обработки текста. Например, объект `StreamReader` поддерживает чтение текста по строкам (до следующего признака конца строки), а класс `XMLTextReader` позволяет интерпретировать поток как данные в формате XML.

Объявляя классы производными от `TextReader`, можно определять специализированные объекты для работы с потоковыми данными.

Класс `System.IO.Stream` позволяет загрузить один или несколько байт в байтовые переменные или массивы. Объект `BinaryReader` специализируется на чтении произвольных данных; он позволяет интерпретировать поток данных как источник любых примитивных типов данных. Таким образом, вы можете читать данные непосредственно в переменные `Boolean`, `Integer`, `Long`, `String` и других примитивных типов.

Не стоит и говорить, что все сказанное выше о чтении данных в равной степени относится и к записи. Для этого используются объекты `TexWriter`, `StringWriter`, `BinaryWriter`, `StreamWriter` и т. д.

В приложении IODemo (листинг 12.7) показано простейшее применение этих базовых принципов на примере функции `ShowTheStream`, успешно обрабатывающей данные двух разнотипных потоков.

Листинг 12.7. Приложение IODemo

```
' Ввод-вывод
' Copyright ©2001 by Desaware Inc.
Imports System.IO

Module Module1
  Const StringToReadFrom As String = "This is a string to read with a string
reader"

  Private Sub ShowTheStream(ByVal tr As TextReader)
    Dim i As Integer
    Do
      i = tr.Read()
      If i >= 0 Then
        Console.Write (Chr(i))
      End If

    Loop While i >= 0

  End Sub

  Sub Main()
    Dim sr As New StringReader(StringToReadFrom)
    Dim i As Integer
    ShowTheStream (sr)
    Console.WriteLine()

    Dim fs As New FileStream("..\demo.txt", FileMode.Open)
    Dim strread As New StreamReader(fs)
    ShowTheStream (strread)
    fs.Close()
    strread.Close()

    Console.ReadLine()
  End Sub

End Module
```

Другие классы System.IO

В пространстве имен `System.IO` присутствуют и другие классы, заслуживающие внимания.

System.IO.Directory и System.IO.DirectoryInfo

Классы предназначены для выполнения различных операций с каталогами, в том числе создания, удаления и перемещения каталогов. Кроме того, они позволяют получить или задать время создания и последней модификации каталога.

System.IO.File и System.IO.FileInfo

Классы предназначены для выполнения различных операций с файлами, в том числе создания, удаления, копирования, перемещения и проверки существования файлов. Также с их помощью можно открывать файлы (функции открытия файлов возвращают объекты `System.IO.FileStream`, используемые при последующих операциях чтения и записи).

System.IO.FileSystemInfo

Базовый класс для классов `System.IO.DirectoryInfo` и `System.IO.FileInfo`. Используется при перемещении в иерархии каталогов для получения информации о каталогах и файлах.

System.IO.FileSystemWatcher

Класс предназначен для отслеживания событий файловой системы (создания, удаления и модификации файлов).

System.IO.IsolatedStorage (пространство имен)

Классы этого пространства имен позволяют связать уникальный каталог со сборкой или доменом приложения. Принцип аналогичен хранению данных в системном реестре с ключом, уникальным для заданного приложения, но в данном случае вместо фрагментов данных сохраняются целые файлы.

Сериализация и управление данными

Типичные задачи, решаемые на Visual Basic, так или иначе связаны с перемещением и хранением данных. Учитывая это обстоятельство, логично предположить, что в .NET Framework включена основательная поддержка хранения, управления и преобразования данных. В пространстве имен `System.Data` реализована модель ADO .NET — новейшая версия модели ADO, хорошо знакомой программистам VB. Пространства имен иерархии `System.XML` предназначены для работы с данными в формате XML — нового стандарта разметки данных. Наконец, пространства иерархии `System.Runtime.Serialization` позволяют сохранять объекты в потоках данных с последующим восстановлением.

Любые попытки описать эту тему на нескольких страницах обречены на неудачу. Сначала я хотел полностью проигнорировать проблемы сериализации, но потом выяснилось, что ей все же придется посвятить хотя бы несколько слов. Представьте, что наш экскурсионный автобус едет по длинной, скучной дороге к очередной достопримечательности, а экскурсовод тем временем рассказывает о других замечательных городах, расположенных неподалеку. Огни этих городов видны на горизонте, но их посещение откладывается на будущее.

Сериализация

Итак, мы хотя бы в общих чертах познакомились с потоками данных. Устрашающий термин «сериализация» в действительно имеет вполне понятный смысл:

речь идет о сохранении состояния объекта в потоке данных. В будущем сериализованный объект восстанавливается по данным, прочитанным из потока. Объект с поддержкой сериализации должен знать, как сохранять и восстанавливать свое состояние.

Разработчики компонентов VB6 знакомы с концепцией сериализации по наборам свойств (property bags). Некоторые даже знают, что на уровне внутренних механизмов COM для выполнения сериализации применяются интерфейсы `IPersistStream` и `IPersistStorage`. С точки зрения программиста в VB .NET происходит нечто очень похожее, хотя на самом деле все работает совершенно иначе.

Сериализация объектов используется очень часто — например, для сохранения и восстановления объектов в web-приложениях и службах, не обладающих собственной информацией состояния, или для сохранения состояния объектов перед их временной выгрузкой. Кроме того, сериализация позволяет преобразовать объект в данные, легко передаваемые по сети, поскольку сериализованные данные не содержат ссылок.

В приложении Serialization (листинг 12.8) определяется простой класс SerializationTest, который умеет сохранять свое состояние. Класс содержит две открытые переменные `m_MyString` и `m_MyInteger`. Чтобы наделить его возможностью сериализации, мы устанавливаем атрибут `Serializable`. Этот атрибут приказывает .NET организовать автоматическое сохранение всех переменных и свойств класса, не помеченных атрибутом `<NonSerialized>`.

Класс также содержит два метода. Первый метод сохраняет объект в потоке SOAP, а второй создает новый объект `SerializationTest` по данным, прочитанным из потока SOAP. Учтите, что эти методы не обязаны входить в класс — в нашем примере это сделано лишь для удобства.

Листинг 12.8. Пример сериализации

```
' Простой пример сериализации
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Imports System.Runtime.Serialization

<Serializable()> Public Class SerializationTest

  Public m_MyString As String
  Public m_MyInteger As Integer

  Public Function DumpToSoap(ByVal sc As serializationtest) As String
    Dim ms As New System.IO.MemoryStream()
    Dim sf As New Formatters.Soap.SoapFormatter()
    ' Сохранить класс в потоке данных
    sf.Serialize(ms, sc)
    ms.Flush()
    ' Прочитать данные из потока в строковую переменную
    ms.Seek(0, IO.SeekOrigin.Begin)
    Dim tr As New System.IO.StreamReader(ms)
    Dim res As String
    res = tr.ReadToEnd()
    ms.Close()
    Return res
  End Function
```

продолжение ⇙

Листинг 12.8 (продолжение)

```
  Public Shared Function GetFromSoap(ByVal soapstring As String) As
SerializationTest
    Dim ms As New System.IO.MemoryStream()
    Dim sw As New System.IO.StreamWriter(ms)
    ' Записать строку в поток данных
    sw.Write (soapstring)
    sw.Flush()
    ms.Flush()
    ms.Seek(0, IO.SeekOrigin.Begin)
    Dim sc As serializationtest
    ' Загрузить объект по описанию Soap
    Dim sf As New Formatters.Soap.SoapFormatter()
    sc = CType(sf.Deserialize(ms), serializationtest)
    ms.Close()
    Return (sc)
  End Function

End Class

Module Module1

  Sub Main()
    Dim st As New SerializationTest()
    st.m_MyInteger = 5
    st.m_MyString = "A test string"

    Dim soapstring As String
    soapstring = st.DumpToSoap(st)
    console.WriteLine (soapstring)
    st = serializationtest.GetFromSoap(soapstring)
    console.WriteLine ("Results after object is loaded")
    console.WriteLine (st.m_MyString & " " & CStr(st.m_MyInteger))
    console.ReadLine()

  End Sub

End Module
```

В этом фрагменте сериализация выполняется в двух местах. Объект класса `System.Runtime.Serialization.Formatters.Soap.SoapFormatter` сохраняет объект в потоке данных произвольного типа (в нашем примере — `MemoryStream`), после чего содержимое потока немедленно читается в строковую переменную при помощи объекта `StreamReader`. В .NET Framework также имеется объект `BinaryFormatter`, сохраняющий объект в виде компактного двоичного потока. Создание объекта выполняется в противоположном направлении: строка записывается в поток данных, а затем восстанавливается при помощи объекта `SoapFormatter`. Конечно, вы можете определить собственные объекты для выполнения сериализации или реализовать в сохраняемом объекте интерфейс `ISerializable`.

Если новомодные концепции «объектов в Интернете» обошли вас стороной, на всякий случай поясню: сокращение SOAP означает «простой протокол для работы с объектами» (Simple Object Access Protocol) — стандартный способ описания объектов, их содержимого и методов на языке XML (eXtensible Markup Language). Сериализованная версия объекта в формате SOAP выглядит так:

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3,org/2000/10/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
xmlns:a1="http://schemas.microsoft.com/urt/NSAssem/Serialization/Serializati
on">
<SOAP-ENV:Body>
<a1:SerializationTest id="ref-1">
<m_MyString id="ref-3">A test string</m_MyString>
<m_MyInteger>S</m_MyInteger>
</a1:SerializationTest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Протокол SOAP является текстовым и может транспортироваться при помощи протокола HTTP, что позволяет организовать передачу SOAP-данных через брандмауэры (firewalls) (при условии, что настройка брандмауэра разрешает прохождение HTTP, главного протокола World Wide Web), а благодаря наличию стандарта объекты в формате SOAP могут использоваться в любых операционных системах и средах.

Впрочем, это всего лишь теоретические рассуждения. На практике стандарт SOAP еще не утвердился и пока идут горячие споры относительно того, чьи расширения станут или не станут частью этого стандарта. Другими словами, сейчас слишком рано говорить о том, станет ли SOAP полноценным стандартом или же выродится в семейство плохо совместимых протоколов (как HTML и DHTML).

Будем надеяться на лучшее[1].

ADO .NET и XML

Если вы, как и большинство программистов, будете работать с базами данных в VB .NET, я рекомендую прочитать документацию Microsoft, а затем найти хорошую книгу по ADO .NET.

А пока я лишь приведу несколько фактов, которые необходимо знать об ADO .NET.

- Вы не обязаны переходить на ADO .NET. При желании вы можете продолжать использовать COM ADO.
- В ADO .NET не предусмотрено сохранение текущего состояния. Для работы с данными используется объект `DataSet`, содержащий копию рабочих данных. После внесения необходимых изменений эти данные записываются обратно в источник данных.
- ADO .NET позволяет работать с разными источниками данных, но самой интересной возможностью мне кажется работа с данными XML. Более того, ADO .NET использует XML для взаимодействия с источником данных. Это позволяет избежать проблем с безопасностью и настройкой брандмауэров, из-за которых удаленное подключение к источникам данных в COM считалось делом весьма творческим.

[1] По мнению технического редактора, из этой фразы непонятно, на какой же именно исход надеется автор. Конечно, я бы предпочел, чтобы SOAP стал общепринятым межплатформенным стандартом со стопроцентной совместимостью.

Простое применение ADO .NET продемонстрировано в приложении TVListing. Модуль TVListingDB.vb приведен в листинге 12.9. Класс TVListingDB содержит объект таблицы данных ADO .NET. В процессе динамического построения в таблицу включаются два поля: название телесериала и время показа (для простоты представленное в строковом формате). Свойства объектов `DataColumn` определяют поведение соответствующих полей базы данных.

Метод `LoadInitialData` заполняет таблицу исходными данными. Вызовы методов `Debug` позволяют проследить за динамикой изменения записей. При первоначальном создании объекта `DataRow` и заполнении его данными эта информация еще не находится в базе. После вызова метода `DataRow.Add` объект `DataRow` считается «новым» («New»), но еще не принятым в базу. После вызова `DataTable.AcceptChanges` объект переходит в «неизмененное» состояние («Unchanged»). В этом состоянии он пребывает до ближайшей модификации, после чего переходит в измененное («Modified») состояние и остается в нем до подтверждения изменений. Метод `GetDataSet` является одним из способов получения объекта `DataSet` для всей таблицы. На практике объект `DataSet` обычно загружается в результате выполнения запроса SQL или установки фильтра для таблицы.

Листинг 12.9. Модуль TVListingDB.vb приложения TVListing

```
Imports System.Data
Public Class TVListingDB
  Public TVTable As New DataTable("TVListing")

  Public Sub New()
    Dim showcol As New DataColumn("Name", GetType(String))
    Dim showtime As New DataColumn("Time", GetType(String))
    ' Неопределенные значения недопустимы
    showcol.AllowDBNull = False
    showtime.AllowDBNull = False

    TVTable.Columns.Add (showcol)
    TVTable.Columns.Add (showtime)
  End Sub

  Public Sub LoadInitialData()
    Dim newRow As DataRow

    newRow = TVTable.newRow()
    newRow.Item(0) = "Star Trek"
    newRow.Item(1) = "13:00"
    TVTable.Rows.Add (newRow)
    newRow = TVTable.newRow()
    newRow.Item("Name") = "Babylon 5"
    newRow.Item("Time") = "14:00"
    TVTable.Rows.Add (newRow)
    newRow = TVTable.newRow()
    newRow("Name") = "BattleShip Galactica"
    newRow("Time") = "15:00"
    Debug.WriteLine("Before adding, Row is: " & _
      newRow.RowState.ToString)
    TVTable.Rows.Add (newRow)
    Debug.WriteLine("Before accepting to table, Row is: " & _
      newRow.RowState.ToString)
    TVTable.AcceptChanges()
```

```
    Debug.WriteLine("Before modification, Row is: " & _
      newRow.RowState.ToString)
    newRow("Time") = "15:30"
    Debug.WriteLine("Before accepting changes, Row is: " & _
      newRow.RowState.ToString)
    newRow.AcceptChanges()
    debug.WriteLine(TVTable.Rows.Count)
  End Sub


  Public Function GetDataSet() As DataSet
    Dim ds As New DataSet()
    ds.Tables.Add (TVTable)
    Return ds
  End Function

End Class
```

Объект `DataSet` может содержать данные из нескольких таблиц вместе с объектами, определяющими связи между ними.

Приведенный ниже метод `cmdCreateDB_Click` показывает, как создать объект `TVListingDB` и назначить таблицу источником данных для элемента `DataGrid`.

```
Private Sub cmdCreateDB_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdCreateDB.Click
  Dim tv As New TVListingDB()

  tv.LoadInitialData()
  dataGrid1().DataSource = tv.TVTable
  dataGrid1().PreferredColumnWidth = -1
End Sub
```

Методы `cmdXML_Click` и `cmdXML2_Click` (листинг 12.10) демонстрируют применение методов `DataSet.WriteXmlSchema` и `DataSet.WriteXml` для вывода данных в формате XML.

Листинг 12.10. Вывод XML в проекте TVListing

```
Private Sub cmdXML_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdXML.Click
  Dim tv As New TVListingDB()
  Dim sr As New StringWriter()
  tv.LoadInitialData()
  txtResult().Text = ""
  tv.GetDataSet.WriteXmlSchema (sr)
  txtResult().Text = sr.ToString

End Sub

Private Sub cmdXML2_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdXML2.Click
  Dim tv As New TVListingDB()
  Dim sr As New StringWriter()
  tv.LoadInitialData()
  txtResult().Text = ""
  tv.GetDataSet.WriteXml (sr)
  txtResult().Text = sr.ToString

End Sub
```

Данные XML приведены в листинге 12.11. Одной из приятных особенностей ADO .NET является то, что при актуализации данных в источник передаются не все данные, а только изменившиеся, что способствует снижению сетевого трафика.

Листинг 12.11. Данные XML из проекта TVListing

```
<NewDataSet>
  <xsd:schema id="NewDataSet" targetNamespace="" xmlns=""
xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xsd:element name="TVListing">
    <xsd:complexType>
      <xsd:all>
        <xcs:element name="Name" type="xsd:string"/>
        <xcs:element name="Time" minOccurs="0" type="xsd:string"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="NewDataSet" msdata:IsDataSet="true">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element ref="TVListing"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <xsd:schema>
  <TVListing>
    <Name>Star Trek</Name>
    >Time>13:00</Time>
  </TVListing>
  <TVListing>
    <Name>Babylon 5</Name>
    >Time>14:00</Time>
  </TVListing>
  <TVListing>
    <Name>BattleStar Galactica</Name>
    >Time>15:30</Time>
  </TVListing>
</NewDataSet>
```

В .NET Framework также входит пространство имен `System.XML`. В него входят средства лексического анализа и управления потоками данных XML.

Итоги

На этом завершается наше краткое знакомство с важнейшими пространствами имен. Возможно, вас интересует, о чем я не упомянул? Уверяю вас, «за кадром» осталось гораздо больше классов, чем упоминается в тексте. В частности, совершенно не рассматривалась область разработки компонентов. В .NET Framework можно найти классы для чего угодно, от управления ресурсами до лексического анализа файлов в формате HTML и XML.

Впрочем, вы получили представление о важнейших пространствах имен, играющих ключевую роль в решении многих задач программирования — и, в первую очередь, связанных с возможностями VB6, не поддерживаемых напрямую в

Visual Basic .NET. Мы рассмотрели основные системные объекты и принципы их взаимодействия. Вы познакомились с коллекциями, вводом-выводом, графическим выводом и печатью, основными принципами использования этих пространств имен и отличиями от решений, использованных в Visual Basic 6. Глава завершается кратким введением в сериализацию и ADO .NET (каждая из этих тем заслуживает отдельной книги).

Напоминаю, что многие классы пространства имен `System` небезопасны по отношению к потокам. При использовании этих классов в многопоточных приложениях необходимо действовать очень осторожно (то есть не предоставлять общий доступ к объектам .NET Framework со стороны нескольких программных потоков, если в документации не сказано, что эти классы обладают потоковой безопасностью).

Наша короткая экскурсия подошла к концу, но знакомство с миром VB .NET еще не закончено. В следующих трех главах мы рассмотрим пространство имен `System.Windows.Forms`, используемое для создания приложений Windows, и ряд пространств имен из области сетевых операций и средств взаимодействия COM. Уровень изложения не претендует на сколько-нибудь достаточную глубину, но я надеюсь, что приведенный обзор поможет вам правильно взяться за самостоятельное изучение.

Приложения Windows

13

Первая неудачная попытка

«...Visual Basic .NET предоставляет в распоряжение программиста визуальные средства ускоренной разработки приложений, которые стали одной из главных причин успеха Visual Basic. Создание пользовательского интерфейса сводится к простому выбору нужных элементов из палитры и размещению их на форме. Значения свойств элементов либо вводятся в специальном окне, либо задаются на программном уровне, после чего программист пишет код для обработки различных событий.

Далее следует простое описание приложения „Hello World“ в Visual Basic .NET...»

Многие авторы именно так знакомят читателя с приложениями Windows. У подобного подхода есть два недостатка. Во-первых, он сводится к пересказу документации, поэтому такие книги скучно читать (и писать, кстати, тоже). Во-вторых, если читатель еще не понимает концепций, представленных в таких «введениях», ему вообще слишком рано читать подобные книги.

Большинство программистов VB .NET освоится в Visual Studio IDE за полчаса, и при этом им не понадобится ни моя помощь, ни чтение документации.

Вторая неудачная попытка

Осваивая программирование приложений Windows в VB .NET, программист VB6 среднего или высокого уровня уже знает, что именно его интересует. Остается лишь выбрать из огромного набора свойств, методов и событий именно то, что необходимо для решения конкретной задачи. Формы и элементы VB .NET сохранили практически все возможности VB6 (а также приобрели множество новых), но не всегда понятно, как они соответствуют друг другу. Даже после чтения раздела «What's New in VB .NET» («Что нового в VB .NET») документации Microsoft вам придется в течение некоторого времени поэкспериментировать с новым пакетом форм, чтобы добиться нужного результата.

Итак, большая часть вашего времени будет потрачена на изучение новых свойств и событий объектов, но я даже не буду пытаться описывать их. Простран-

ство имен `System.Windows.Forms` содержит столько классов, методов и событий, что ими можно заполнить отдельную книгу, не говоря уже об одной главе.

Возникает резонный вопрос: о чем же тогда написана эта глава?

Новый пакет форм

В этой главе я попытался представить изменения, произошедшие с пакетом форм, в практическом контексте. Надеюсь, что мои наблюдения помогут вам быстрее освоиться в новой обстановке.

Прежде всего следует запомнить, что пакет форм .NET не имеет ничего общего с тем пакетом, который использовался в VB6. Это совершенно другой пакет, написанный заново[1].

На практике это в первую очередь означает невозможность каких-либо предположений, выходящих за рамки документации, книг по VB .NET или того, что вы узнаете опытным путем. Впрочем, я говорю не о принципиальных особенностях, ведь свойства и события работают примерно так же, как прежде. Скорее, речь идет о практических мелочах вроде правильных способов проверки полей, порядка инициирования событий или принципов связывания элементов с данными. Постепенно вы получите всю нужную информацию и необходимые практические навыки, но это потребует определенного времени, и эта глава лишь укажет нужное направление.

Итак, вам предстоит столкнуться с бесчисленными мелкими изменениями и их практическими последствиями. Тем не менее в области форм VB .NET существует ряд серьезных, принципиальных изменений, о которых следует помнить с самого начала.

Повторное создание окон

Многие свойства элементов VB6 задаются только на стадии конструирования. Например, вам не удастся перевести список с одиночным выделением в режим множественного выделения во время работы программы. На первый взгляд это кажется досадным упущением, но для программиста, знакомого с механикой работы Windows, причина ясна: эти свойства соответствуют стилям Windows, которые не могут изменяться после создания окна.

В VB6 при создании формы или размещении на ней элемента для этой формы/элемента создается окно. Изменить стиль окна можно только одним способом: уничтожить его и создать заново, что приведет к потере текущих данных окна.

В VB .NET значения этих свойств могут изменяться на стадии выполнения, как показывает следующий фрагмент приложения Recreate.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
  Dim x As Integer
  For x = 1 To 20
```

[1] Хотя такие классы, как Form, UserControl и т. д., действительно написаны заново, многие стандартные элементы (например, RichTextBox) представляют собой простые «обертки» для стандартных элементов Windows или их COM-аналогов. Это означает, что они подвержены проблемам обновления (пресловутый «кошмар DLL») в той же степени, как и все остальные компоненты, использующие DLL стандартных компонентов.

```
    listBox1().Items.Add ("Entry # " & CStr(x))
  Next
  lblWindow().Text = "hWnd = " & listBox1().Handle.ToString
End Sub

Private Sub chkMulti_CheckedChanged(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles chkMulti.CheckedChanged
  If chkMulti().CheckState = CheckState.Checked Then
    listBox1().SelectionMode = SelectionMode.MultiExtended
  Else
    listBox1().SelectionMode = SelectionMode.One
  End If
  lblWindow().Text = "hWnd = " & listBox1().Handle.ToString

End Sub
```

Находящийся на форме флажок позволяет выбрать режим выделения для элемента `ListBox` (одиночное или множественное). Поэкспериментируйте с этим приложением и убедитесь в том, что поведение списка соответствует значению свойства `SelectionMode`.

А теперь проследите за изменением надписи при переключении режима. Вы увидите, что манипулятор окна изменяется.

При смене режима все данные списка сохраняются во временном буфере, окно списка создается с новым стилем, после чего данные восстанавливаются.

Первый урок особенно важен для тех, кто использует функции Win32 API: манипуляторы окон в .NET могут изменяться.

Графические элементы

В Visual Basic 4 появилась новая разновидность элементов управления — «упрощенные» или «графические» элементы, не имеющие собственных окон. Их можно рассматривать как инструкции контейнеру о выполнении некоторых графических операций. Графические элементы были включены в OLE, поскольку они расходовали меньше ресурсов и обеспечивали лучшее быстродействие.

В .NET все элементы ассоциируются с окнами. Преимущество нового подхода заключается в том, что модель программирования становится более последовательной, поскольку вам не придется разбираться с ошибками, связанными с различиями в поведении оконных и графических элементов. Впрочем, есть и недостаток: теперь каждый элемент требует дополнительных затрат, связанных с ассоциированным окном.

Если вы привыкли использовать элементы `Line` и `Shape` (самые распространенные графические элементы), учтите, что в VB .NET они не поддерживаются. Вам придется воспользоваться другим, более рациональным решением — прорисовкой всех дополнительных изображений в обработчике события `Paint` формы.

Согласованное поведение контейнеров

Как опытный разработчик элементов ActiveX заявляю, что самый большой недостаток этой технологии — непредсказуемый характер работы элементов ActiveX в каждом конкретном контейнере. Мелкие различия между управляющими приложениями приводят к мелким (а то и крупным) различиям в поведе-

нии элементов ActiveX. Успешное тестирование элемента в VB6 не гарантирует, что он будет нормально работать в Visual C++, Microsoft Word или Excel. Подобная несовместимость объясняется тем, что каждое управляющее приложение содержит собственную реализацию контейнера. Дело в том, что каждый разработчик по-своему толкует спецификации COM (надо признать, местами весьма невразумительные), не говоря уже об ошибках, неизбежно присутствующих в любой сложной программе.

На практике это означает, что разработчики элементов создают и тестируют элементы в VB6 (или еще для одного-двух управляющих приложений), где существует определенный рынок, и фактически игнорируют другие платформы, поскольку они не оправдывают расходов на тестирование и сопровождение элементов. Отсутствие элементов для этих платформ отрицательно сказывается на развитии спроса и подавляет инициативу разработчиков. Возникает порочный круг. Аналогичные проблемы возникают и при создании специализированных элементов для внутреннего использования, хотя и в меньшей степени.

Архитектурные шаблоны и System.Windows.Forms

Вокруг часто слышатся разговоры о применении архитектурных шаблонов при программировании. Проще говоря, вы должны знать общепринятые решения, используемые в стандартных ситуациях, и уметь применять их в конкретных задачах.

Быстрота изучения пространства имен `Windows.Forms` зависит от вашего умения обнаруживать закономерности в методах и свойствах многочисленных элементов пространства имен. Конечно, можно попытаться изучать все элементы независимо друг от друга или же начать с методов и свойств, общих для нескольких элементов, а затем сосредоточиться на относительно немногочисленных методах и свойствах, уникальных для каждого элемента.

Я выбрал второй путь.

Прежде всего необходимо понять, что пакет форм строится на основе наследования. Каждый раз, когда вы создаете форму, вы в действительности определяете класс, производный от класса `System.Windows.Forms.Form`.

На рис. 13.1 показана часть иерархии классов форм .NET. На рисунке изображены не все классы, но и этого вполне достаточно, чтобы проиллюстрировать иерархический характер связей между классами.

Рассмотрим ключевые классы и пространства имен этой иерархии.

System.ComponentModel.Component

Пространство имен `System.ComponentModel` содержит объекты, управляющие работой компонентов и их взаимосвязями.

- Компонент может находиться под управлением контейнера, выполняя при этом функции контейнера по отношению к другим компонентам.
- Компонент следит за своими внутренними компонентами и при вызове метода `Dispose` вызывает метод `Dispose` для каждого из внутренних компонентов.

- Компоненты могут передаваться между доменами приложений посредством маршалинга.
- С компонентами могут связываться средства режима конструирования. В этом случае компонент присутствует на панели инструментов Visual Studio, а его свойства отображаются в окне свойств Visual Studio.
- Компоненты и их свойства могут сериализоваться. Значения их свойств могут преобразовываться в текстовый формат для вывода в окне свойств Visual Studio. Кроме того, вы можете создавать собственные мини-редакторы, работающие в окне свойств.

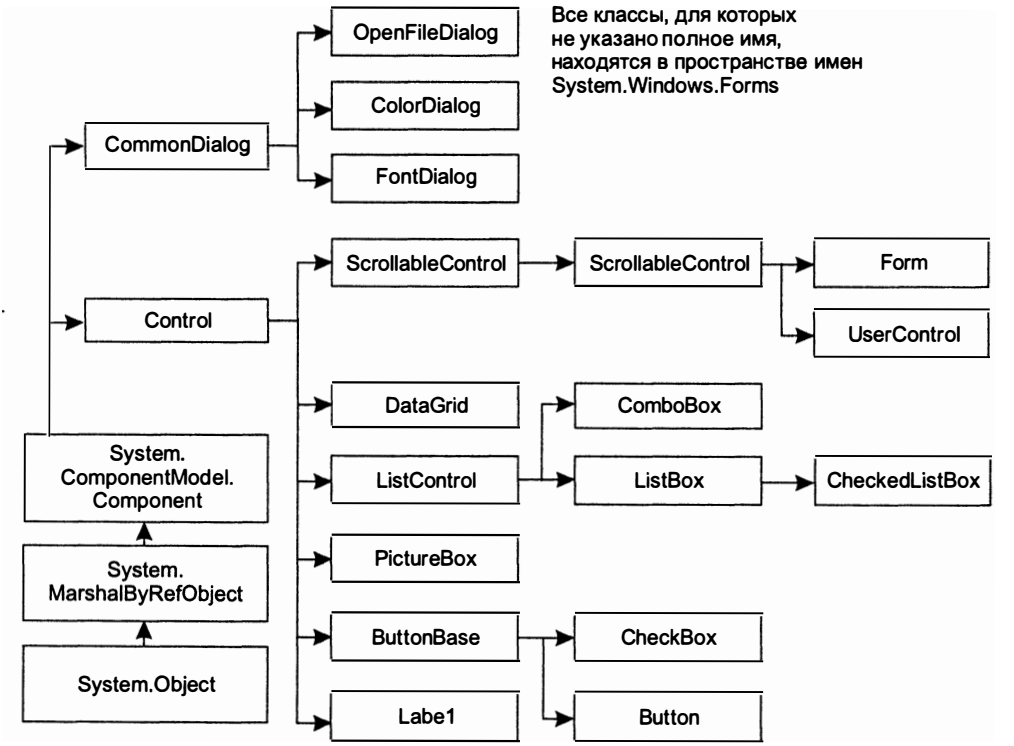


Рис. 13.1. Часть пакета Windows.Forms

System.Windows.Forms.Control

Чтобы лучше разобраться в иерархии форм, мы рассмотрим серию примеров, демонстрирующих функциональные возможности каждого класса в цепочке наследования. Начнем с проекта ControlOnly. Мы рассмотрим его особенно подробно, чтобы вы поняли, как работают формы.

Прежде всего обратите внимание на то, что класс формы объявляется производным от `System.Windows.Forms.Form`:

```
Public Class Form1
  Inherits System.Windows.Forms.Form
```

При создании нового приложения в VB6 имя «Form1» имеет двойной смысл. Оно представляет тип формы и одновременно глобальное имя первой формы этого типа. Например, следующий фрагмент создает две формы типа `Form1` с идентификаторами `Form1` и `f2`:

```
Dim f2 As New Form1
  Form1.Caption = "First form 1"
  f2.Show
  f2.Caption = "Second form 1"
```

В VB .NET эта концепция не поддерживается. Здесь имя `Form1` относится к новому классу объектов, производному от типа `Form`. На этот объект можно ссылаться с ключевым словом `Me`.

Для форм VB .NET определен конструктор по умолчанию, который вызывает конструктор базового класса (как обычно), а затем вызывает `InitializeComponent`:

```
#Region " Windows Form Designer generated code "
  Public Sub New()
    MyBase.New()

    ' Необходимо для дизайнера форм Windows.
    InitializeComponent()

    ' Дальнейшая инициализация выполняется
    ' после вызова InitializeComponent().
  End Sub
```

Формы также реализуют интерфейс `IDisposable`. Форма, как и другие контейнеры, хранит список внутренних компонентов в объекте `System.ComponentModel.Container`, обладающем средствами для освобождения этих внутренних объектов:

```
' Необходимо для дизайнера форм Windows.
Private components As System.ComponentModel.Container

' Форма переопределяет Dispose для очистки списка компонентов.
Public Overloads Overrides Sub Dispose()
    MyBase.Dispose()
    If Not (components Is Nothing) Then
      components.Dispose()
    End If
End Sub
```

На форме расположены три элемента. В данном примере наибольший интерес представляет первый элемент, относящийся к обобщенному типу `Control`. Вряд ли вам когда-нибудь придется создавать элементы типа `Control` на практике, но в данном примере это поможет разобраться в функциональности разных объектов, использованных при построении форм.

```
Private WithEvents ctl1 As System.Windows.Forms.Control
Private WithEvents textBox1 As System.Windows.Forms.TextBox
Private WithEvents textBox2 As System.Windows.Forms.TextBox
```

На примере функции `InitializeComponent` наглядно проявляются принципиальные различия между VB6 и VB .NET. В VB6 чтение и запись свойств происходит «по волшебству», незаметно для программиста. В VB .NET значения свойств задаются программой на стадии выполнения, а окно свойств среды Visual Studio — всего лишь особая разновидность редактора программного кода.

Элемент `ctl1` (тип `Control`) включается в коллекцию `Control` формы. Контейнером текстового поля `textBox1` в действительности является `ctl1`, а не форма. Из чего это следует? Из того, что этот элемент включается в коллекцию `ctl1.Control` (коллекцию элементов элемента `ctl1`) вместо коллекции элементов формы. Процесс инициализации формы показан в листинге 13.1.

Листинг 13.1. Инициализация формы в проекте ControlOnly[1]

```
' ВНИМАНИЕ: следующий фрагмент необходим
' для дизайнера форм Windows.
' Для его модификации следует использовать дизайнер форм.
' Не изменяйте его в редакторе!
<System.Diagnostics.DebuggerStepThrough()> Private _
Sub InitializeComponent()
  Me.ctl1 = New System.Windows.Forms.Control()
  Me.textBox1 = New System.Windows.Forms.TextBox()
  Me.textBox2 = New System.Windows.Forms.TextBox()
  Me.ctl1.SuspendLayout()
  Me.SuspendLayout()
  '
  'ctl1
  '
  Me.ctl1.BackColor = System.Drawing.Color.Bisque
  Me.ctl1.Controls.AddRange(New System.Windows.Forms.Control() _
  {Me.textBox1})
  Me.ctl1.Name = "ctl1"
  Me.ctl1.Size = New System.Drawing.Size(128, 112)
  Me.ctl1.TabIndex = 0
  '
  'textBox1
  '
  Me.textBox1.Location = New System.Drawing.Point(32, 48)
  Me.textBox1.Name = "textBox1"
  Me.textBox1.Size = New System.Drawing.Size(72, 20)
  Me.textBox1.TabIndex = 1
  Me.textBox1.Text = "textBox1"
  '
  'textBox2
  '
  Me.textBox2.Location = New System.Drawing.Point(24, 128)
  Me.textBox2.Name = "textBox2"
  Me.textBox2.Size = New System.Drawing.Size(96, 20)
  Me.textBox2.TabIndex = 1
  Me.textBox2.Text = "textBox2"
  '
  'Form1
  '
  Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
  Me.ClientSize = New System.Drawing.Size(292, 273)
  Me.Controls.AddRange(New System.Windows.Forms.Control() _
  {Me.textBox2, Me.ctl1})
  Me.Name = "Form1"
  Me.Text = "Form1"
  Me.ctl1.ResumeLayout (False)
  Me.ResumeLayout (False)

End Sub
```

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Тот факт, что элемент textBox1 принадлежит ctl1, приводит к весьма странным последствиям. Среда Visual Studio не позволяет непосредственно выбрать текстовое поле на форме. При перемещении между двумя текстовыми полями элемент ctl1 тоже получает фокус. Все эти проблемы возникают из-за того, что объект Control не рассчитан на непосредственное включение внутренних элементов — по крайней мере, в среде Visual Studio.

Но несмотря на это, объект ctl1 типа Control содержит многие свойства, которые мы ожидаем увидеть в элементах управления: BackColor, ForeColor, BackgroundImage, ClientRectangle, ContextMenu, Cursor, Font, Top, Left, Width, Height, Parent, Visible и т. д. Объект класса, производного от Control, автоматически наследует все эти свойства. Стандартные элементы (такие, как TextBox, Label и DataGrid) являются непосредственно производными от Control.

Естественно, одним из первых шагов при самостоятельном знакомстве с пространством имен Windows.Forms должно стать изучение свойств и методов класса Control.

Продолжим описание объектов, используемых при построении форм.

System.Windows.Forms.ScrollableControl

Проект ScrollableOnly аналогичен ControlOnly, однако объект ctl1 в нем объявляется производным от класса ScrollableControl (производного от Control). В следующем фрагменте несколько текстовых полей и надпись:

```
Private WithEvents ctl1 As System.Windows.Forms.ScrollableControl
Private WithEvents label1 As System.Windows.Forms.Label
Private WithEvents textBox2 As System.Windows.Forms.TextBox
Private WithEvents textBox1 As System.Windows.Forms.TextBox
Private WithEvents textBox3 As System.Windows.Forms.TextBox
```

Функция InitializeComponent (листинг 13.2) задает значения свойств этих элементов (для экономии места процедура приводится с сокращениями).

Листинг 13.2. Инициализация формы в проекте ScrollableOnly

```
<System.Diagnostics.DebuggerStepThroughAttribute()> Private _
Sub InitializeComponent()
  Me.textBox2 = New System.Windows.Forms.TextBox()
  Me.textBox1 = New System.Windows.Forms.TextBox()
  Me.ctl1 = New System.Windows.Forms.ScrollableControl()
  Me.label1 = New System.Windows.Forms.Label()
  Me.textBox3 = New System.Windows.Forms.TextBox()
  '
  'ctl1
  '
  Me.ctl1.AutoScroll = True
  Me.ctl1.BackColor = System.Drawing.Color.Crimson
  Me.ctl1.Controls.AddRange(New System.Windows.Forms.Control() _
  {Me.textBox2, Me.textBox1, Me.label1})
  Me.ctl1.Location = New System.Drawing.Point(40, 16)
  Me.ctl1.Name = "ctl1"
  Me.ctl1.Size = New System.Drawing.Size(200, 136)
  Me.ctl1.TabIndex = 0
  '
```

продолжение

Листинг 13.2 (продолжение)

```
  'label1
  '
  Me.label1.BackColor = System.Drawing.Color.FromArgb(0, 192, 0)
  Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
  Me.ClientSize = New System.Drawing.Size(292, 273)
  Me.Controls.AddRange(New System.Windows.Forms.Control() _
  {Me.textBox3, Me.ctl1})
  Me.Name = "Form1"
  Me.Text = "Form1"
End Sub
```

Элементы `textBox2`, `textBox1` и `label1` содержатся в элементе `ctl1`. Элемент `textBox3` содержится в самой форме. Свойству `AutoScroll` элемента `ctl1` задается значение `True`. Это свойство, добавленное классом `ScrollableControl`, позволяет элементу включать другие элементы не только на логическом, но и на визуальном уровне. Когда внутренний элемент выходит за пределы видимых границ объекта `ScrollableControl`, появляются полосы прокрутки, при помощи которых его можно увидеть.

Поэкспериментируйте с этим приложением и понаблюдайте за тем, как действуют полосы прокрутки. Вы увидите, что в режиме конструирования можно выбрать внутренние элементы и разместить новые элементы на `ctl1`.

Впрочем, при перемещении между элементами происходит нечто странное: элемент `clt1` по-прежнему получает фокус.

Контейнеры, формы и класс UserControl

Проект ContainerControl практически идентичен ScrollableOnly. Единственное изменение заключается в том, что объект `ctl1` теперь принадлежит к классу `ContainerControl`. Класс `ContainerControl`, в свою очередь, является производным от `ScrollableControl`, но включает дополнительные средства управления передачей фокуса, необходимые для правильной работы внутренних элементов. При попытке передачи фокуса контейнеру он автоматически передает фокус первому внутреннему элементу. Запустите программу и поэкспериментируйте с передачей фокуса клавишей Tab.

Классы `Form` и `UserControl` являются производными от класса `ContainerControl`. В формах добавлены свойства и методы для работы с рамками и заголовками, поддержки MDI и других возможностей, присущих окнам верхнего уровня. Количество новых членов класса `UserControl` не велико, зато в этом классе имеются атрибуты, указывающие Visual Studio на необходимость использования особых средств конструирования `UserControl`.

Ориентация в пространстве имен System.Windows.Forms

Просматривая электронную документацию по классам пространства имен `Forms`, вы увидите, что для каждого класса в документации приведен полный список всех членов. В документации также сказано, какие члены реализуются (или пере-

гружаются) классом, а какие наследуются от базовых классов. Хотя вы сможете легко получить список членов класса, с первого взгляда трудно понять, какие члены непосредственно реализуются или перегружаются классом — для этого вам придется просмотреть весь список.

Приложение DirectMembers представляет собой утилиту для ускоренного поиска членов, реализованных непосредственно в классе. Кроме того, программа демонстрирует некоторые возможности, общие для многих элементов.

На форме находится иерархический список (класс `TreeView`), свойство `Sorted` которого равно `True`. Во время инициализации формы вызывается функция `LoadTreeview`, подробно описанная ниже.

```
Private Sub LoadTreeview()
  Dim asm As Assembly
  Dim asmtypes() As Type
  Dim ThisType As Type
  asm = Reflection.Assembly.GetAssembly(_
    GetType(System.Windows.Forms.Form))
```

Получение данных сборки посредством рефлексии вам уже хорошо знакомо. В нашем примере запрашивается информация о сборке, содержащей пакет форм.

```
asmtypes = asm.GetTypes()
For Each ThisType In asmtypes
  If ThisType.IsClass And ThisType.IsPublic Then
```

Переменная `asmtypes` заполняется массивом с описанием всех типов сборки. Нас интересуют только классы с атрибутом `Public`.

```
Dim tn As New TreeNode(ThisType.Name)
```

Имена классов отображаются в корневых узлах иерархического дерева `TreeView`. Элемент `TreeView` содержит коллекцию `TreeNodeCollection`, состоящую из объектов `TreeNode`. Каждый объект `TreeNode`, в свою очередь, содержит собственную коллекцию `TreeNodeCollection` — так образуется иерархическая структура.

Подобное решение используется практически всюду, где элемент управления содержит список внутренних объектов. Например, элемент `ListBox` уже не поддерживает методов для непосредственного добавления и удаления элементов. Вместо этого необходимые операции выполняются со свойством `Items` объекта `ListBox`. Хотя по сравнению с VB6 ситуация заметно изменилась, особых проблем быть не должно, поскольку вы уже умеете работать с коллекциями. Но еще интереснее выглядит тот факт, что в список можно включать любые объекты! Элемент `ListBox` просто вызывает метод `ToString` и выводит строковое представление объекта, что снимает необходимость в управлении дополнительной информацией при помощи свойства `ItemData`. Таким образом, хранение сложных данных в списке организуется просто: вы определяете класс, загружаете в него все необходимые данные и переопределяете метод `ToString`, чтобы в списке выводилась нужная информация.

Программисты VB6, жалующиеся на несовместимость списков VB .NET и VB6, просто не разобрались в ситуации. Подход .NET гораздо более интуитивен, а поскольку он применяется всюду, где элементы управляют списками, программирование становится более последовательным. Это ускоряет освоение новых элементов, упрощает чтение и сопровождение программ.

Но я отклонился от темы[1]. Вернемся к проекту DirectMembers. Следующий фрагмент выполняется для каждого класса в пространстве имен `System.Windows.Forms`.

```
Dim members(), mi As MemberInfo
treeView1().Nodes.Add (tn)
members = ThisType.GetMembers(BindingFlags.DeclaredOnly Or _
  BindingFlags.Public Or BindingFlags.Instance Or _
  BindingFlags.Static)
```

Массив `Members` заполняется данными обо всех открытых и общих членах, объявленных на этом уровне (к их числу относятся перегруженные реализации, но не те, которые были в неизменном виде унаследованы от базовых классов). Затем программа в цикле перебирает все элементы массива.

```
      For Each mi In members
        Dim methinfo As MethodInfo
        Select Case mi.MemberType
          Case MemberTypes.Method
            methinfo = CType(mi, MethodInfo)
            If Not methinfo.IsSpecialName Then
              tn.Nodes.Add (StripType(mi.ToString))
            End If
          Case MemberTypes.Event
            tn.Nodes.Add (StripType(mi.ToString) & _
              " event")
          Case Else
            tn.Nodes.Add (StripType(mi.ToString))
        End Select
      Next
    End If
  Next
End Sub
```

Метод `ToString` класса `MethodInfo` создает полное строковое представление метода, включая специальные методы доступа (вида `get_xxx` для свойства `xxx`). В строку включается информация о возвращаемом значении (задается в отдельном пространстве имен). Если функция вызывается для метода, мы проверяем результат и заносим его в список лишь в том случае, если метод не является специальным. Если метод на самом деле оказывается событием, к строке для наглядности добавляется слово «event». Свойства добавляются без дополнительной обработки. Метод `StripType` удаляет информацию о возвращаемом значении, чтобы результат легче читался. Тип возвращаемого значения всегда можно узнать из документации.

```
Private Function StripType(ByVal s As String) As String
  Dim spacepos As Integer
  spacepos = InStr(s, " ")
  If spacepos > 0 Then Return Mid$(s, spacepos + 1)
  Return (s)
End Function
```

[1] Впрочем, это отступление вызвано отнюдь не моей излишней словоохотливостью. Я хотел подчеркнуть весьма важное обстоятельство и одну из главных причин, по которой мы так подробно разбираем этот пример.

При помощи этой утилиты вы сможете легко определить, какие члены были добавлены в класс. Рисунок 13.2 ясно показывает, что количество новых членов класса `UserControl` по отношению к базовому классу `ContainerControl` относительно невелико.

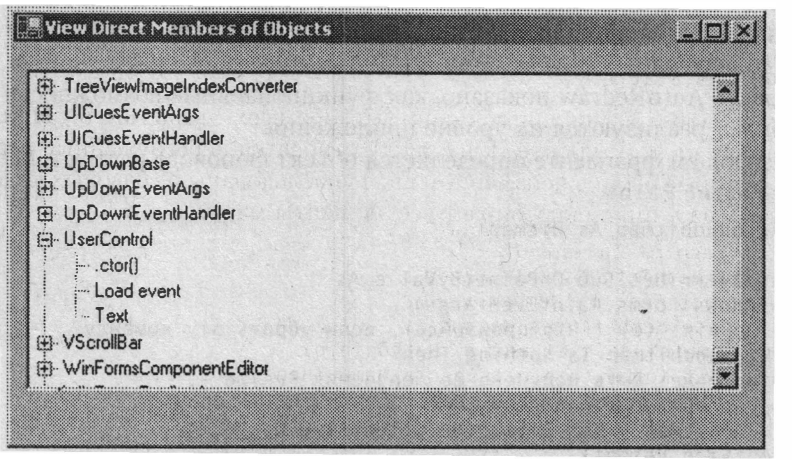


Рис. 13.2. Приложение DirectMember

Дальнейшие исследования

Как я и обещал, мы не будем рассматривать все изменения в пакете форм, однако некоторые классы и изменения заслуживают особого внимания.

AutoRedraw, события и переопределения

Я не люблю свойство `AutoRedraw` VB6. В эпоху VB1 разработчики Visual Basic решили, что программирование, управляемое событиями, в области графического вывода порождает слишком много проблем. Программистам VB и так приходилось адаптировать свои DOS-приложения под обработку событий, и заставлять их обрабатывать событие `Paint` и перерисовывать графику было бы слишком жестоко.

Возможно, они были правы.

Только программисты с большим стажем помнят, сколько трудностей порождало программирование, управляемое событиями, у разработчиков из мира DOS.

Руководствуясь этими соображениями, разработчики Microsoft создали фоновые растровые изображения для всех элементов `Form` и `Picture`. Все графические операции с этими объектами в действительности выполнялись с фоновыми растрами, содержимое которых затем автоматически копировалось в форму или элемент. Использование этого растра зависело от значения свойства `AutoRedraw`, по умолчанию равного `True` (использовать фоновый растр). Хотя такой подход упрощал задачу бывших DOS-программистов, он был связан с большими затратами ресурсов, особенно в 16-разрядных операционных системах того времени.

В последующих версиях Visual Basic это свойство по умолчанию было равно `False`, и перерисовка была основана на использовании события `Paint`.

Вероятно, в наши дни большинство программистов VB не использует свойство `AutoRedraw = True`, однако бывают и исключения, например, если элемент содержит сложное графическое изображение и экономия времени на его построение оправдывает затраты на хранение фонового растра (что в наши дни обходится гораздо дешевле).

В проекте AutoRedraw показано, как функциональные возможности «в стиле `AutoRedraw`» реализуются на уровне приложения.

В следующем фрагменте определяется объект фонового растра и переопределяется событие `Paint`.

```
Dim backgroundbitmap As Bitmap

Protected Overrides Sub OnPaint(ByVal e As
System.Windows.Forms.PaintEventArgs)
  MyBase.OnPaint (e) ' Что произойдет, если убрать эту команду?
  If backgroundbitmap Is Nothing Then
  ' Событие может быть получено до получения Resize -
  ' особенно при первой перерисовке.
  backgroundbitmap = New Bitmap(Me.ClientSize.Width, _
  Me.ClientSize.Height)
  End If
  e.Graphics.DrawImage(backgroundbitmap, 0, 0)
End Sub
```

Если фоновый растр не существует, он создается программой, а если существует — его содержимое просто копируется на форму.

У приведенного фрагмента есть одна довольно странная особенность. В отличие от VB6, он не является обработчиком события с формальной точки зрения!

Подумайте — что такое событие? Механизм, при помощи которого объект сообщает своим клиентам о выполнении некоторого условия. Например, форма инициирует событие, чтобы передать нужную информацию программному коду, находящемуся за пределами формы.

Еще раз: при помощи событий объект передает информацию **внешнему** коду.

Однако в приведенном примере мы не *используем* форму, а реализуем ее. В VB6 вы создавали клиентский код, использующий стандартный объект `Form`. В VB .NET вы создаете собственный уникальный класс, производный от стандартного класса формы. Вместо того чтобы ограничиваться обработкой событий в базовом классе, вы обычно переопределяете реализации базового класса и наделяете свой класс уникальными возможностями.

Но каким образом ваш класс инициирует события `Paint`?

Приложение AutoRedraw содержит метод `cmdOtherForm_Click`, создающий новый экземпляр класса `Form1`. Другими словами, первая форма действительно становится клиентом по отношению к другому экземпляру класса. В качестве клиента она может обнаруживать события этого объекта, как показывает следующий фрагмент:

```
Private Sub OtherFormPaint(ByVal sender As Object, _
  ByVal args As PaintEventArgs)
  Debug.WriteLine("Other paint arrived")
End Sub
```

```
Private Sub cmdOtherForm_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdOtherForm.Click
  Dim f As New Form1()
  f.Visible = True
  AddHandler f.Paint, AddressOf Me.OtherFormPaint
End Sub
```

Попытайтесь запустить приложение и щелкнуть на кнопке Other Form два раза: с командой `MyBase.OnPaint` (приведена выше) и без нее. Вы увидите, что событие `Paint` инициируется только при вызове метода `MyBase.OnPaint`. Из этого можно сделать вывод, что событие инициируется базовым классом (производным от класса `Control`).

При подобном переопределении методов почти всегда следует вызывать реализацию базового класса, или вы рискуете столкнуться с неприятными побочными эффектами.

Предположим, вы хотите, чтобы ваша форма обладала собственным событием `Paint` — возможно, с уникальной сигнатурой. Нет проблем. Просто определите собственное событие, маскирующее событие базового класса:

```
Shadows Event Paint(ByVal sender As Object, ByVal args AS PaintEventArgs)
```

При желании вы можете инициировать новое событие командой `RaiseEvent`.

Различия между реализацией и использованием формы/элемента — вопрос весьма тонкий, и большинству программистов VB6 придется к ним привыкать. К счастью, это один из тех случаев, когда задача решается несколькими способами, и ни одному из них нельзя отдать однозначного предпочтения.

Вернемся к проекту. На форме находятся еще две кнопки: первая кнопка объявляет текущее содержимое формы недействительным (что приводит к форсированной перерисовке и инициированию события `Paint`), а вторая рисует на форме косой крест и объявляет форму недействительной.

```
Private Sub cmdInvalidate_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdInvalidate.Click
  Me.Invalidate()
End Sub

Private Sub cmdDrawStuff_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdDrawStuff.Click
  Dim g As Graphics = graphics.FromImage(backgroundbitmap)
  g.DrawLine(pens.Black, 0, 0, 200, 200)
  g.DrawLine(pens.Red, 0, 200, 200, 0)
  cmdInvalidate_Click(Me, Nothing)
End Sub
```

В реализации фонового растра особый интерес представляет обработка события `Resize`:

```
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
  If Not Me.ClientRectangle.IsEmpty Then
    If Not Me.ClientSize.Equals(New Size(backgroundbitmap.Width, _
    backgroundbitmap.Height)) Then
      Dim newbitmap As Bitmap
      newbitmap = New Bitmap(Me.ClientSize.Width, _
      Me.ClientSize.Height)
      If Not backgroundbitmap Is Nothing Then
        Dim g As Graphics = graphics.FromImage(newbitmap)
        g.DrawImage(backgroundbitmap, 0, 0)
```

```
      End If
      backgroundbitmap = newbitmap
    End If
  End If
End Sub
```

При изменении размеров окна функция сначала убеждается в том, что форма не была свернута (в этом случае структура `ClientRectangle` описывает пустой прямоугольник). Если новые размеры формы соответствуют текущим размерам растра (что происходит при восстановлении свернутой формы), делать ничего не нужно. В противном случае создается новый растр нужного размера и в него копируется текущее содержимое растра.

Поэкспериментируйте с изменением размеров формы. Уменьшите форму так, чтобы крест на ней не помещался, и затем снова увеличьте ее.

Также обратите внимание на то, что во всех вычислениях координаты задаются в пикселах. Пакет графического вывода позволяет работать в других единицах, однако во всех базовых операциях управления окнами используются пикселы. Честно говоря, я никогда не понимал, почему в Visual Basic 6 и более ранних версиях использовались твипы. Этими единицами почти никто не пользуется, к тому же они приводят к ошибкам округления при выводе и вычислении прямоугольников.

Главный аргумент в пользу пикселов — их эффективность. Пикселы и целые числа используются в работе внутренних механизмов Windows. Твипы и вещественные вычисления лишь замедляют работу Visual Basic, не предоставляя программисту никаких новых возможностей.

Формы MDI и принадлежность окон

При попытках организовать связи между формами программисты VB6 постоянно сталкиваются с трудностями. Для передачи элементов между формами или другими контейнерами они используют функцию API `SetParent`. Проблемы возникают и с отношениями принадлежности, поскольку все дочерние окна сворачиваются вместе с родительским окном. Это особенно важно в приложениях, создающих несколько форм верхнего уровня одновременно, — обнаруживать свертывание одной формы, чтобы вручную сворачивать остальные формы, по меньшей мере неудобно.

Наконец, программисты VB6 испытывают трудности с созданием надежно работающих иерархий MDI, в которых дочерние формы могут реализовываться в виде отдельных библиотек DLL. В одной из моих самых популярных статей описывалась динамическая иерархия MDI и было показано, как решить эту проблему при помощи динамически создаваемых пользовательских элементов VB6, однако попытки использования в этой схеме коммерческих элементов сопровождались появлением нетривиальных ошибок и лицензионных проблем.

А теперь посмотрите, как это делается в VB .NET.

Чтобы установить иерархическую связь между окнами, создайте приложение с двумя формами и объявите в первой форме экземпляр второй формы:

```
Dim f2 As New Form2()
```

В дальнейшем связь между формами создается простым включением второй формы в коллекцию OwnedForm первой формы:

```
Me.AddOwnedForm(f2)
```

Ну как, впечатляет? Еще бы! Ведь изменить принадлежность формы после ее создания невозможно не только в VB6, но и в Windows! Принадлежность формы должна определяться при создании окна. Как же .NET Framework добивается такого эффекта? Точно так же, как при изменении стилей окон: существующее окно уничтожается и воссоздается заново с новым владельцем.

Но это еще не все!

Чтобы объявить главную форму приложения формой MDI, достаточно задать значение свойства IsMdiContainer:

```
Me.IsMdiContainer = True
```

Отдельный класс MDIForm не нужен. Допустим, у вас имеется другая форма (возможно, даже реализованная в отдельной сборке). Стоит задать ее свойству MdiParent нужное значение, и она мгновенно превращается в дочернее окно MDI:

```
Me.MdiParent = parent
```

Проще не бывает. Примеры практического использования этих приемов показаны в приложении FormInForm.

А заодно рассмотрите приложение Parenting и посмотрите, как кнопка перемещается с одной формы на другую всего одной строкой программного кода.

В VB6 элементы обычно были статическими и создавались на стадии конструирования. Создавать элементы динамически можно было только с применением массивов элементов.

Итак, в VB6 существовала возможность динамического создания элементов, но пользоваться ей было неудобно, особенно в области обработки событий.

В VB .NET все элементы и формы создаются динамически, а статических элементов «стадии конструирования» не существует в принципе. При размещении элемента на форме на стадии конструирования мастер Design Wizard генерирует код динамического создания этого элемента.

А если вам все же не хватает массивов элементов, обратитесь к проекту EventExample главы 10 — в нем показано, как аналогичные возможности реализуются в VB .NET.

Субклассирование и объект Application

Термин «субклассирование» (subclassing) означает перехват сообщений Windows, получаемых формой или элементом (как правило, для реализации возможностей, не поддерживаемых стандартными событиями Visual Basic).

Моя компания Desaware завоевала свою репутацию за создание лучшего компонента для субклассирования до появления Visual Basic 5, когда субклассирование в VB вообще не поддерживалось[1]. Когда в VB5 появились средства субклас-

[1] Это был наш первый пакет из семейства SpyWorks. В последующих версиях возможности SpyWorks уже не ограничивались субклассированием.

сирования, мы создали компонент VB, который демонстрировал их правильное применение, и включили его исходный текст в поставку SpyWorks.

Приложение Messages показывает, как субклассирование выполняется в .NET.

```
Private Sub button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
  listBox1().Items.Clear()
End Sub

Protected Overrides Sub wndproc(ByRef m As Message)
  Select Case m.Msg

    Case &H20, &H84
      ' Игнорировать многочисленные
      ' сообщения WM_SETCURSOR, WM_NCHITTEST
    Case Else
      listBox1().Items.Add (m.ToString)

  End Select
  MyBase.WndProc (m)
End Sub
```

На форме находится список всех сообщений, полученных формой, кроме `WM_SETCURSOR` и `WM_NCHITTEST` (их слишком много). Все, что от нас требуется, — переопределить функцию `wndproc` базового класса `Control` и обработать интересующие сообщения. Не забудьте вызвать `MyBase.WndProc`, иначе вы рискуете серьезно нарушить работу формы. Метод `MyBase.WndProc` выполняет стандартную обработку сообщений, благодаря чему все окна сохраняют свое характерное поведение.

Правда, этот способ не может применяться к внутренним элементам, а только к тем классам, которые вы создаете самостоятельно. Однако многие методики субклассирования VB6 основаны на субклассировании вашей собственной формы для перехвата системных сообщений, отправляемых окнам верхнего уровня. Такое решение прекрасно подойдет для данного случая.

Переопределяя функцию `PreProcessMessage`, можно установить фильтр предварительной обработки сообщений. В частности, это позволяет организовать специальную обработку некоторых клавиш (например, Tab и клавиш управления курсором).

Кроме того, объект `System.Windows.Forms.Application` позволяет установить фильтр сообщений для программного потока методом `Application.AddMessageFilter`. При этом устанавливается перехватчик уровня потока, который обнаруживает сообщения, выбранные из очереди для конкретного потока. Действуйте осмотрительно, поскольку установка фильтра может отрицательно повлиять на быстродействие программы.

Я рекомендую продолжить самостоятельные исследования класса `Application`. Этот класс содержит ряд полезных свойств и методов, относящихся к работе приложений, включая пресловутый метод `DoEvents` (которым пользоваться не рекомендуется, особенно учитывая поддержку многопоточности в VB .NET).

Если вы хотите выполнять межпроцессное субклассирование, перехватывать сообщения других процессов или всей системы или обнаруживать нажатия клавиш на системном уровне, VB .NET не предложит вам ничего нового по сравне-

нию с VB6. Впрочем, пакет SpyWorks 6.5 (вероятно, он уже будет продаваться к моменту выхода книги) после доработки позволяет решать все эти задачи в приложениях Visual Basic .NET. За подробностями обращайтесь по адресу http://www.desaware.com[1].

Формы и потоки

Надеюсь, вы уже пришли в себя после страха, которого я нагнал на вас в главе 7 по поводу многопоточности, и по крайней мере обдумываете возможности ее практического применения. Что ж, сейчас начну пугать снова.

Шутка.

Но вы должны очень хорошо понять, как работать с формами и элементами в многопоточном приложении.

Проект Threading показывает, как создать форму в отдельном потоке. Как обычно, большая часть автоматически сгенерированного кода пропускается. В процессе обработки события `Form1_Load` в элементе `Label` выводится идентификатор текущего потока, которому принадлежит форма.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
  label1.Text = "Thread ID: " & CStr(AppDomain.GetCurrentThreadId)
End Sub
```

Переменная `OtherThread` определяет объект класса `Thread`, связанный с процедурой `OtherThreadEntryPoint`. При запуске этого потока (нажатием кнопки `button1`) создается новый экземпляр формы[2]. Процедура вызывает метод `Application.Run` и передает ему форму в качестве параметра. При вызове `Application.Run` поток входит в цикл обработки сообщений. Заданная форма отображается на экране, а сообщения обрабатываются вплоть до ее закрытия (CLR автоматически выполняет указанные действия для стартовой формы приложения). Ссылка на новую форму хранится в переменной `OtherForm1`.

```
Dim OtherThread As New Thread(AddressOf OtherThreadEntryPoint)

Module UsedByThread
  Public OtherForm1 As Form1
  Public Sub OtherThreadEntryPoint()
    OtherForm1 = New Form1()
    OtherForm1.DisableButtons()
    Application.Run (OtherForm1)
  End Sub
End Module

Private Sub button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
  OtherThread.Start()
End Sub
```

[1] Периодические упоминания моей компании и продуктов раздражают некоторых читателей. Впрочем, большинство все же понимает, что без этого у меня просто не будет возможности писать книги.

[2] Не нажимайте кнопку во второй раз! Это простой демонстрационный пример и проверка ошибок в нем не предусмотрена.

Метод ThreadIdInLabel2 формы выводит идентификатор текущего потока в элементе-надписи label2.

```
Public Sub ThreadIdInLabel2()
  label2.Text = "Current Thread " & CStr(AppDomain.GetCurrentThreadId)
End Sub
```

При нажатии кнопки button2 вызывается метод ThreadIdInLabel2 формы OtherForm1:

```
Private Sub button2_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button2.Click
  If Not OtherForm1 Is Nothing Then
    OtherForm1.ThreadIdInLabel2()
  End If
End Sub
```

Как показывает эксперимент, на новой форме выводится идентификатор потока исходной, а не новой формы.

Конечно, это плохо.

Дело в том, что в документации ничего не сказано о потоковой безопасности пакета форм. Более того, Microsoft специально оговаривает, что все методы формы или элемента должны вызываться в том потоке, где они создавались.

К счастью, у этой задачи имеется простое решение. Метод Invoke класса Control (который, как было сказано выше, является базовым классом всех форм и элементов) правильно осуществляет маршалинг вызовов методов и процедур в поток элемента. Приведенная ниже процедура button3_Click показывает, как следует вызывать метод Invoke. Если воспользоваться этим приемом, на новой форме будет выводиться правильный идентификатор программного потока.

```
Delegate Sub NoParams()

Private Sub button3_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button3.Click
  If Not OtherForm1 Is Nothing Then
    Dim usedel As NoParams = AddressOf OtherForm1.ThreadIdInLabel2
    OtherForm1.Invoke (usedel)
  End If
End Sub
```

Инициирование событий объектов в других потоках безопасно. При этом используется механизм вызова с применением делегатов, а вызовы методов передаются нужному потоку.

Избегайте вызовов SyncLock и других функций ожидания для потоков, работающих с элементами или другими составляющими пользовательского интерфейса. Мало того, что это может нарушить нормальную работу пользовательского интерфейса — возникает риск взаимной блокировки, поскольку эти объекты часто реентерабельны (например, вызов методов часто приводит к вызовам других методов или инициированию событий).

Но, прежде всего, старайтесь избегать создания форм в разных потоках. Приложения, выигрывающие от многопоточного пользовательского интерфейса, встречаются очень редко — только в ситуациях, когда каждая форма ведет себя как отдельное приложение (например, окна web-браузеров обычно принадлежат к разным потокам, а окна документов — нет). Тем не менее представленные прин-

ципы применимы и к более общему случаю — созданию в фоновом потоке компонента или класса, взаимодействующего с пользовательским интерфейсом (как правило, посредством событий).

Итоги

Эта глава не сводится к простому сопоставлению возможностей VB6 и VB .NET — в ней рассматривается архитектура, и я даже рискну сказать — философия нового пакета форм .NET Framework. Мы познакомились с иерархией основных объектов пространства имен `System.Windows.Forms` и выяснили, что все элементы и формы .NET основаны на базовом классе `Control`. Простое приложение DirectMembers поможет вам быстро определить, какие члены определяются непосредственно самими классами пространства имен.

Далее мы рассмотрели некоторые модификации, обусловленные не столько расширенной функциональностью, сколько изменениями в архитектуре. Вы узнали, как имитировать свойство `AutoRedraw` VB6, причем этот пример был использован для объяснения различий между использованием формы (VB6) и реализацией нового типа `Form` (VB .NET). Были приведены примеры связей между формами и элементами, которые являются естественным следствием того, что все элементы в .NET создаются динамически. Также вы узнали, как в VB .NET организовать обработку сообщений и как обеспечить потоковую безопасность при работе с формами.

Интернет-приложения и службы

14

Я ненадолго прерву нормальное повествование и обращусь к конкретной группе читателей, к тем, кто не читал предыдущих глав, а сразу открыл книгу на этой странице, желая узнать, что в этой книге говорится о web-приложениях и службах и почему эта тема рассматривается только в конце.

Да, я обращаюсь к тем, кто просматривает книгу в магазине и пытается понять, что он держит в руках — серьезную профессиональную книгу о VB .NET или халтурную поделку, призванную заработать на модной теме?

Чтобы ответить на ваш вопрос, позвольте мне задать свой.

Что такое Microsoft .NET?

Ответ зависит от того, кому адресован этот вопрос.

Попробуйте спросить, что такое ActiveX, COM — а еще лучше Windows DNA? Вы услышите множество определений.

.NET захлестнул тот же поток маркетинговых статей, недостоверных и обрывочных сведений, вольных интерпретаций и неразберихи, от которого пострадали многие инициативы Microsoft. Не думаю, что вы получите сколько-нибудь вразумительный ответ от сотрудников Microsoft — их ответы слишком сильно зависят от должности и от того, какую презентацию PowerPoint им показали последней.

Например, недавно я говорил с одним специалистом по маркетингу, который искренне считал, что Biztalk 2000 является существенной, неотъемлемой частью Microsoft .NET.

Конечно, это такая же важная составляющая общей стратегии Microsoft .NET, как .NET Enterprise Services (отчасти это Windows DNA под новым названием) и таинственный проект Hailstorm. Однако это вовсе не означает, что все эти составляющие абсолютно необходимы для работы с .NET — более того, это попросту неверно.

Насколько я понимаю, основной круг читателей этой книги складывается из программистов и менеджеров, организующих их работу. Определение .NET с

точки зрения программиста было приведено в главе 4 — новая виртуальная машина, для которой мы программируем[1].

Но какое место в этом определении занимает Интернет?

Со стратегических и маркетинговых позиций .NET выражает взгляд Microsoft на будущее Интернета — похоже, ярлык «.NET» будут лепить ко всему, что имеет сколько-нибудь отдаленное отношение к Интернету. С точки зрения программиста, Microsoft .NET представляет собой виртуальную машину, благодаря которой Интернет-программирование должно стать таким же простым, как традиционное программирование для Windows.

Я не буду тратить время на дальнейшие поиски определения Microsoft .NET — все равно определение (или по крайней мере стратегические приоритеты) завтра изменится. Для нас .NET — это прежде всего инструментарий программирования и архитектура, и интернет-программирование занимает в этой картине далеко не последнее место.

Причина, по которой я не упоминал об этом до настоящего момента, проста: практически все, о чем говорилось в книге (за исключением главы 13), в равной степени относится и к Интернет-программированию в VB .NET.

Итак, вы просто листаете книгу, интересуясь, почему автор так долго откладывал тему Интернет-программирования? Надеюсь, вы поняли, что вам абсолютно необходимо вернуться к началу и прочитать все предыдущие главы — только так можно понять концепции, необходимые для программистов VB .NET в любой области — как в Интернет-программировании, так и в традиционном программировании для Windows.

Программирование для Интернета

Чтобы понять сущность интернет-программирования, необходимо капитально разобраться в концепции Интернета с точки зрения Microsoft. Но прежде чем объяснять, в чем же эта концепция состоит, я хочу особо подчеркнуть одно важное обстоятельство: она не является неотъемлемой частью .NET. Microsoft .NET — инструмент, разработанный для реализации этой концепции, однако он с таким же успехом может применяться для реализации других идей (в том числе и ваших собственных).

Подход Microsoft к Интернету на самом деле прост и основан на одной принципиальной идее (которая, впрочем, никогда не высказывается вслух): Web в своем текущем состоянии никуда не годится.

Да, я знаю, что это звучит кощунственно, но выслушайте мои аргументы.

Во времена создания Web существовало несколько файловых форматов, которые прекрасно справлялись с воспроизведением текста и графики. Например, формат RTF (Rich Text Format) поддерживался большинством текстовых редакторов, а формат Adobe PDF позволял воспроизводить сложные документы в одинаковом виде практически на любом компьютере. Что же мы имеем сегодня?

[1] Имеется в виду виртуальная машина в классическом понимании, то есть совокупность требований и условий разработки программного обеспечения. Существует и другая трактовка виртуальной машины как специализированного эмулятора (пример — виртуальная машина Java).

HTML — стандарт, который не позволяет одинаково воспроизвести страницу в разных версиях одного и того же браузера, не говоря уже о разных[1].

Что еще хуже, информация web-страниц перемежается с командами форматирования. Программе становится очень трудно извлечь нужную информацию из нагромождения форматных тегов. А поскольку web-сайт практически в любой момент может измениться, практически невозможно написать программу, позволяющую сколько-нибудь надежно читать данные с web-страниц.

Посмотрим, что происходит в области программирования. Сценарии ADP (Active Server Pages) содержат самую невероятную мешанину, которую только можно себе представить. Здесь и HTML-код, готовый для передачи клиенту, клиентские и серверные сценарные команды, следующие в непредсказуемом порядке. Многие из виденных мной сценариев ASP отличались крайне запутанной логикой, при виде которой вспоминалось «спагетти-программирование» и команда `Goto`. Где правила объектно-ориентированного программирования? Где принцип отделения пользовательского интерфейса от программного кода, позволяющий дизайнеру изменить страницу без помощи программиста, и наоборот?

Новые тенденции в web-программировании (в том числе и Microsoft .NET) наконец-то станут шагом в нужном направлении.

XML

XML[2]. Название выглядит устрашающе. Похоже, нужно скорее бежать в магазин за новой толстой книгой. А может, у вас уже есть такая книга?

Возможно, для описания всех тонкостей XML действительно нужна большая книга (хотя я в этом сомневаюсь), но для его практического использования ничего такого не потребуется. XML — всего лишь файловый формат для хранения данных. Используемые в нем теги внешне напоминают теги HTML, однако теги XML определяют произвольные имена полей данных, а между начальным и конечным тегом находится значение указанного поля.

Ниже приведен простой пример: база данных в формате XML из проекта SimpleXML.

```
<?xml version="1.0" encoding="utf-8" ?> <?xml-stylesheet type="text/xsl"
href="BillingExample.xslt"?>
<BillingTable>
  <Heading>Billing records</Heading>
  <data>
    <record>
      <name>Dan</name>
      <hours>5</hours>
      <description>First job</description>
    </record>
    <record>
      <name>Jill</name>
      <hours>3.2</hours>
```

[1] Возможно, я преувеличиваю — но совсем немного.

[2] EXtensible Markup Language.

```
    <description>Second job</description>
   </record>
  </data>
</BillingTable>
```

Нужно ли еще что-нибудь объяснять? Средства XML позволяют создать иерархическую структуру данных при помощи тегов, идентифицирующих фрагменты данных в файле.

Итак, XML описывает данные. Особая разновидность файлов XML — так называемые файлы XSL[1] — описывает способ просмотра этих данных.

Файл XSL предназначен для преобразования данных XML в другую форму, в данном случае — в формат HTML[2]. Файл BillingExample.xslt указан в файле BillingExample.XML в качестве списка стилей (style sheet) — файла, описывающего параметры отображения файла XML. Файл BillingExample.xslt приведен в листинге 14.1.

Листинг 14.1. Файл BillingExample.xslt[3]

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" />
  <xsl:template match="/">
    <HTML>
      <Head>
        <Body>
          <h1>
            <xsl:value-of select="//BillingTable/Heading" />
          </h1>
          <table border="1" width="100%">
            <xsl:for-each select="//data/record">
              <tr>
                <td>
                  <xsl:value-of select="name" />
                </td>
                <td>
                  <xsl:value-of select="hours" />
                </td>
                <td>
                  <xsl:value-of select="description" />
                </td>
              </tr>
            </xsl:for-each>
          </table>
        </Body>
      </Head>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

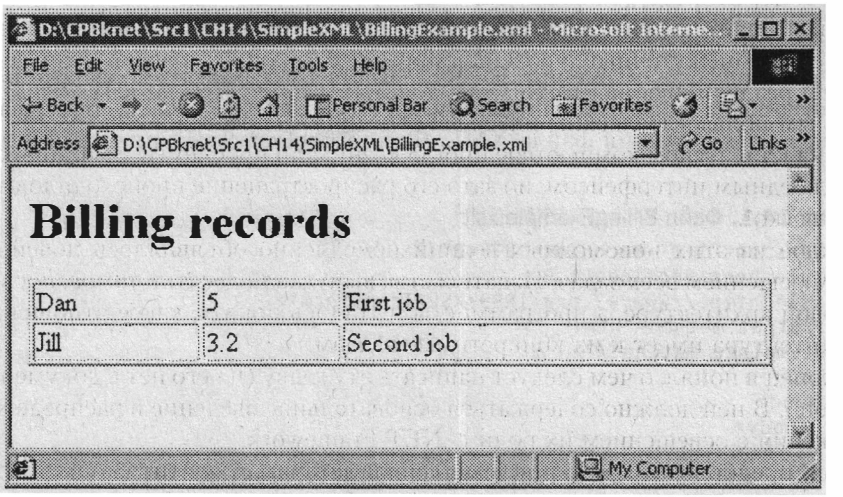
[1] EXtensible Stylesheet Language (или XSLT от слов «XSL Transform»).

[2] XSL также может использоваться для преобразования XML в другие форматы. В сущности, работа Biztalk основана на применении XML в качестве промежуточного формата для преобразования данных из практически любого исходного формата в практический любой итоговый формат.

[3] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Теги, начинающиеся с префикса `xsl:`, представляют команды XSL, интерпретируемые в процессе преобразования в соответствии со спецификацией XSL. В остальных тегах находится непосредственно выводимый код HTML. Тег «`xsl:value-of`» производит замену: данные, указанные в теге, читаются из файла XML и записываются в выходные данные HTML.

На следующем рисунке показано, как файл XML выглядит в виде web-страницы в Internet Explorer.



Файл XSL содержит все теги, необходимые для отображения файла XML в нужном формате.

XML является промышленным стандартом. Это означает, что Microsoft может влиять на развитие XML, но не обладает правами на него. Решение Microsoft взять на вооружение и интегрировать XML выглядит весьма впечатляюще, особенно учитывая традиционную тягу Microsoft к установлению собственных стандартов.

XML помогает отделить данные в Интернете от их визуального представления. Позднее в этой главе будет показано, как проблема разделения функциональности web-сайта и его внешнего вида решается в ASP .NET.

Распределенные приложения

В этой книге я постарался уделять особое внимание тем принципам, на которых основана работа программного кода. Если вы понимаете базовые принципы тех языков и программных инструментов, с которыми вы работаете, найти подробности реализации в документации Microsoft относительно просто. Размышляя над тем, о чем следует написать в этой главе, я был несколько растерян. Эта тема тоже заслуживает отдельной книги (уверен, что даже сейчас к печати готовится немало книг, посвященных ASP .NET и интернет-программированию в VB .NET). К тому же я не хотел пересказывать материал, который можно (и нужно) найти в документации Microsoft.

Что делать?

Ответ нашелся в тот момент, когда я начал писать примеры программ. Подлинная суть Интернет-программирования в .NET заключается не в web-службах, ASP .NET, XML или другой конкретной технологии, а в подходе к написанию приложения.

Когда-то в прошлом типичная программа представляла собой отдельный исполняемый файл (иногда с библиотеками или оверлейными модулями). Позднее вошло в моду программирование «клиент-сервер» — отделение пользовательского интерфейса от уровня операций с базами данных. Затем появилась «трехуровневая архитектура», в которой выделялись уровни пользовательского интерфейса, промежуточный уровень с бизнес-логикой и уровень базы данных. Клиенты разделились на «тонких» и «толстых»: «толстый» клиент обладал расширенным пользовательским интерфейсом, при распространении и запуске которого на клиентском компьютере иногда возникали проблемы; «тонкий» клиент обладал бедным интерфейсом, но зато его распространение происходило относительно легко[1].

Каждое из этих новомодных веяний немедленно объявлялось новейшим и лучшим решением всех задач. Издательства выпускали десятки томов, посвященных новой архитектуре, а программисты пытались понять, какое отношение новая архитектура имеет к их конкретным задачам.

Наконец я понял, о чем следует написать эту главу (и чего нет в документации Microsoft). В ней должно содержаться основательное введение в распределенные приложения с освещением их роли в .NET Framework.

Итак, предположим, некая вымышленная компания платит своим сотрудникам по повременному тарифу, для чего ей требуется точно и быстро получать информацию о затратах рабочего времени.

В следующем разделе предложено одно из возможных решений для программистов .NET.

Подход к проектированию приложений в .NET

Поскольку наше внимание в этой главе сосредоточено на концептуальных и архитектурных аспектах программирования, я постараюсь сделать примеры как можно проще, чтобы нагляднее подчеркнуть концептуальную основу каждой технологии. Начнем с нижнего уровня — операций с базами данных.

Уровень базы данных

Допустим, у организации имеется база данных с информацией о рабочем времени сотрудников. Реализация базы данных не рассматривается даже в общих чертах — эта тема выходит далеко за рамки книги. В нашем примере база данных моделируется простым набором данных в файле XML.

---

[1] По иронии судьбы так называемые «тонкие» клиенты, работающие в браузере, обычно требуют больше системных ресурсов и порождают больше проблем с установкой, чем традиционные «толстые» клиенты.

Все обращения к базе данных осуществляются через компонент, находящийся в каталоге BillingComponent. Этот компонент сохраняет данные в файле XML в соответствии со схемой, определяемой в файле BillingSet.xsd (листинг 14.2).

Листинг 14.2. Файл BillingSet.xsd

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema id="BillingSet"
targetNamespace="http://tempuri.org/BillingSet.xsd"
elementFormDefault="qualified" xmlns="http://tempuri.org/BillingSet.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="BillingTable">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Hours" type="xsd:integer" />
        <xsd:element name="Description" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Файл схемы (schema file) XML описывает структуру данных, но не содержит самих данных. Все элементы данных обладают сильной типизацией, что упрощает их сохранение и загрузку из файлов XML в соответствии со схемой. В данном примере схема состоит из трех полей: имя сотрудника, количество рабочих часов и описание проекта.

Класс BillingComponent (листинг 14.3) используется при всех обращениях к базе данных. В реальных приложениях для разных типов операций с базой данных обычно используются разные объекты. Например, один объект специализируется на записи информации в базу и включается в приложения и компоненты, используемые сотрудниками. Другой объект специализируется на чтении информации из базы данных — вероятно, он будет использоваться в бухгалтерии.

ВНИМАНИЕ

Измените константу BillingXMLLocation в соответствии с местонахождением файла XML. Прежде чем запускать какие-либо примеры, не забудьте скопировать в указанный каталог файл BillingSet.xsd.

Листинг 14.3. Файл BillingComponent.vb

```
' Billing Component
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Public Class BillingComponent

  Inherits System.ComponentModel.Component

  ' Приведите в соответствие с каталогом,
  ' используемым в вашей системе.
  Private Const BillingXMLLocation As String = _
  "d:\MovingToVBNet\Source\ch14\"

#Region " Component Designer generated code "

  Public Sub New(ByVal Container As System.ComponentModel.IContainer)
    MyClass.New()
```

```
    Container.Add(Me)
  End Sub

  Public Sub New()
    MyBase.New()

    ' Необходимо для работы дизайнера компонентов.
    InitializeComponent()

    ' Дальнейшая инициализация выполняется
    ' после вызова InitializeComponent().
    OpenXML()

  End Sub

    ' Необходимо для работы дизайнера компонентов.
  Private components As System.ComponentModel.Container

  ' ВНИМАНИЕ: следующий фрагмент необходим
  ' для работы дизайнера компонентов.
  ' Для его модификации следует использовать дизайнер компонентов.
  ' Не изменяйте его в редакторе!
  <System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
    '
    'BillingComponent
    '
  End Sub

#End Region

  Private myds As New DataSet()
  Private xmllocation As String

  Public Sub OpenXML()
    Dim loc As String
    xmllocation = BillingXMLLocation
    myds.ReadXmlSchema(xmllocation & "billingset.xsd")
    Try
      myds.ReadXml(xmllocation & "BillingData.xml")
    Catch e As Exception
    End Try
    Dim t As DataTable
  End Sub

  Public Sub AddRecord(ByVal Name As String, ByVal Hours As Double, _
  ByVal Description As String)
    Dim dr As DataRow
    dr = myds.Tables(0).NewRow
    dr.Item(0) = Name
    dr.Item(1) = Hours
    dr.Item(2) = Description
    myds.Tables(0).Rows.Add(dr)
    ' Обновить набор данных
    myds.AcceptChanges()
  End Sub

  Public Overloads Overrides Sub Dispose()
    MyBase.Dispose()
```

продолжение

Листинг 14.3 (продолжение)

```
    myds.AcceptChanges()
    myds.WriteXml(xmllocation & "BillingData.xml", _
    XmlWriteMode.IgnoreSchema)
    myds.Dispose()
  End Sub

  Public ReadOnly Property Info() As DataSet
    Get
      Return myds
    End Get
  End Property

End Class
```

Рассмотрим важнейшие методы этого класса.

Сразу же после создания компонент читает файл схемы XML и загружает текущий файл базы данных (если он существует). Данные загружаются в объект `Dataset` ADO .NET. Для обращения к объекту используется свойство `Info`. Объект `Dataset` в ADO .NET содержит «снимок» данных на некоторый момент времени, причем в нашем примере это особенно хорошо видно. База данных XML обновляется лишь в момент фактической записи при вызове `Dispose`.

Метод `AddRecord` включает новую запись в базу данных.

В этом компоненте следует обратить внимание на два обстоятельства. Во-первых, этот простой пример всего лишь демонстрирует целый класс решений по низкоуровневым операциям с базами данных. Конечно, одновременное обращение нескольких пользователей к данным, хранящимся в формате XML, вызовет серьезные проблемы синхронизации. Впрочем, и такой подход может использоваться в ситуации, когда каждый пользователь работает с отдельной копией данных (маловероятно, чтобы пользователь работал с одним экземпляром данных сразу из нескольких приложений).

Во-вторых, помните, что все операции с данными осуществляются через этот компонент. Это означает, что пользователь компонента уже не обязан знать, как организовано хранение данных и каким способом компонент подключается к источнику данных. Клиент лишь должен знать, как обратиться к компоненту в соответствии с его текущим местонахождением.

Вероятно, в .NET компонент BillingComponent будет находиться либо в одном каталоге с базой данных, либо на сервере, связанном с базой данных по локальной сети[1]. В обоих вариантах для работы с клиентом может использоваться локально установленное приложение, играющее роль традиционного «толстого клиента». Этот сценарий рассматривается в следующем разделе.

Традиционное клиентское приложение Windows

Приложение BillingWinApp демонстрирует стандартную ситуацию, при которой среди пользователей локальной сети распространяется специальное приложение

[1] Компонент BillingComponent можно настроить и для удаленного подключения к базе данных, но это потребует подробного описания средств удаленного доступа .NET. Кроме того, как будет показано ниже, у этой задачи есть и другие решения.

для ввода информации в базу данных. Приложение BillingWinApp содержит ссылку на компонент BillingComponent (откройте проект BillingWinApp, включающий как компонент, так и приложение Windows). В данном примере они находятся в одной системе. Отредактируйте ссылку на базу данных в BillingComponent в соответствии с ее текущим местонахождением (или спроектируйте компонент BillingComponent с поддержкой удаленного доступа).

Код приложения очень прост. Объект `Dataset` с загруженными данными назначается в качестве источника данных элемента `DataGrid`:

```
Private BillingInfo As New MovingToVBNet.Billing.BillingComponent()

Private Sub Form1_Load(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
  If components Is Nothing Then components = New _
    System.ComponentModel.Container()
  components.Add (BillingInfo)
  DataGrid1.DataSource = BillingInfo.Info
  DataGrid1.NavigateTo(0, "BillingTable")
  DataGrid1.CurrentRowIndex = 0
End Sub
```

Особого упоминания заслуживает лишь включение компонента BillingInfo в коллекцию компонентов формы. Поскольку с компонентом не ассоциируется собственный дизайнер, он не находится под управлением дизайнера Visual Studio и поэтому не включается автоматически в список компонентов. Включение компонента в коллекцию компонентов обеспечивает вызов его метода `Dispose` при вызове `Dispose` для формы. Вызов `Dispose` очень важен, поскольку в этом методе компонент BillingInfo записывает обновленные данные XML.

Для просмотра текущего содержимого базы данных и добавления новых записей используется элемент `DataGrid` (рис. 14.1).

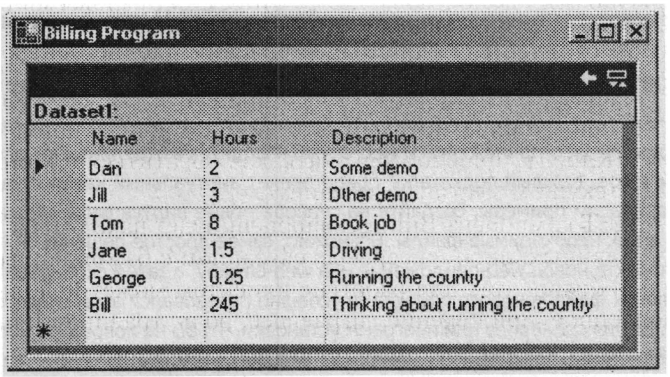


Рис. 14.1. Работа с базой данных в традиционном приложении Windows

То, что вы видели до настоящего момента, было прямым аналогом традиционной трехуровневой архитектуры с «толстым» клиентом.

Но это только начало.

Web-приложение

Web-приложения принадлежат к числу основных новшеств Microsoft .NET и основываются на новой версии ASP (Active Server Pages), называемой ASP .NET. Между ASP и ASP .NET существуют многочисленные различия. Вероятно, самое важное из них заключается в том, что для разработки приложений ASP .NET может использоваться любой язык .NET. Иначе говоря, для написания сценариев ASP .NET вам уже не придется ограничиваться VBScript и JavaScript. Более того, сами термины «сценарии» и «сценарные языки» стали неуместными, поскольку в ASP .NET нет сценариев как таковых: выполняется полноценный код, откомпилированный для .NET. В ASP .NET поддерживаются VB .NET, C# и все остальные языки .NET.

И опытные разработчики ASP, и новички должны запомнить главный принцип Интернет-программирования в .NET: *разработчики Visual Studio .NET стремились к тому, чтобы разработка web-приложений по сложности не превосходила программирование традиционных приложений Windows.*

Другими словами, предполагалось, что вы сможете воспользоваться всем опытом программирования традиционных приложений Windows и применить его непосредственно к интернет-программированию.

Следует заметить, что в целом эта задача была успешно выполнена.

При создании web-приложения вы работаете с дизайнером, очень похожим на форму Windows. Вы размещаете элементы на форме и задаете их свойства. Если дважды щелкнуть на элементе, вам будет предложено ввести код обработки событий данного элемента. Такие приложения работают практически так же, как и приложения VB, если не считать того, что пользовательский интерфейс реализуется передачей HTML-кода браузеру. Элементы управления в web-приложениях по своим возможностям несколько уступают стандартным элементам Windows, что объясняется ограниченными возможностям HTML, но в целом процесс разработки почти не отличается от программирования традиционных приложений.

Простой пример этой технологии приведен в проекте ch14SimpleWebApp.

ВНИМАНИЕ

Все примеры web-приложений и web-служб находятся в отдельном каталоге Webs в архиве примеров. Каждый подкаталог представляет виртуальный каталог на сервере. Чтобы установить эти примеры, создайте на сервере новые виртуальные подкаталоги и скопируйте в них необходимые файлы. Возможно, самое простое решение — создать для каждого проекта новое web-приложение или web-службу, а затем скопировать файлы примеров поверх файлов нового проекта (я успешно пользовался этим способом). Я отказался от написания сценариев или программ установки, чтобы не повредить нормальной работе вашей системы, поскольку на момент написания книги все продукты .NET существовали лишь в предварительных версиях.

Простое web-приложение

Создайте web-приложение и разместите на форме две кнопки: `Webform Button` и `HTML Button` (эти типы элементов расположены на разных вкладках панели инструментов).

Щелкните на элементе `HTML Button` правой кнопкой мыши и выберите в контекстном меню команду запуска на сервере.

В результате будут созданы два файла. Файл WebForm1.aspx является файлом ASP .NET и загружается в браузере при помощи web-приложения. Файл WebForm1.aspx содержит код обработки страницы в VB .NET.

Помните, в самом начале главы я жаловался на то, что в файлах ASP HTML-код причудливо смешивается с командами клиентских и серверных сценариев? В ASP .NET такая возможность сохраняется, однако пользователи Visual Studio без особого труда отделят интерфейсный код HTML от серверного кода, сосредоточенного в отдельном файле. Редактирование файла ASPX, содержащего визуальные компоненты, обычно производится перетаскиванием элементов на поверхности дизайнера и изменением значений их свойств. Возможно, вам вообще никогда не придется вручную редактировать файлы ASPX на уровне исходного текста.

Одно из великих преимуществ Visual Studio перед «ручным» программированием заключается в том, что программист может не думать о тегах и сгенерированном коде HTML и поручить Visual Studio всю черновую работу по связыванию событий и свойств с элементами.

Файл Webform1.aspx приведен в листинге 14.4.

Листинг 14.4. Файл Webform1.aspx

```
<%@ Page Language="vb" AutoEventWireup="false" Codebehind="WebForm1.aspx.vb"
Inherits="ch14SimpleWebApp.WebForm1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
  <HEAD>
    <title></title>
    <meta name="GENERATOR" content="Microsoft Visual Studio.NET 7.0">
    <meta name="CODE_LANGUAGE" content="Visual Basic 7.0">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body>
    <form id="Form1" method="post" runat="server">
      <P>
        <INPUT type="button" value="HtmlButton"
          id="HTMLButton"
          name="Button1" runat="server">
      </P>
      <asp:Button id="WebControlButton" runat="server"
        Text="WebControlButton"></asp:Button>
    </form>
  </body>
</HTML>
```

Как видно из листинга, файл ASPX в основном состоит из обычного кода HTML и нескольких серверных тегов с префиксом asp.

В листинге 14.5 приведен модуль Webform1.aspx.vb.

Листинг 14.5. Файл Webform1.aspx.vb

```
Public Class WebForm1
  Inherits System.Web.UI.Page
  Protected WithEvents HTMLButton As _
    System.Web.UI.HtmlControls.HtmlInputButton
  Protected WithEvents WebControlButton As _
```

продолжение

Листинг 14.5 (продолжение)

```
    System.Web.UI.WebControls.Button

#Region " Web Form Designer Generated Code "

  ' Необходимо для работы дизайнера web-форм.
  <System.Diagnostics.DebuggerStepThrough()> Private Sub _
  InitializeComponent()

  End Sub

  Protected Sub Page_Init(ByVal Sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Init
    'ВНИМАНИЕ: вызов метода необходим
    ' для работы дизайнера web-форм.
    ' Не изменяйте его в редакторе!
    InitializeComponent()
  End Sub

#End Region

  Private Sub Page_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
    ' Здесь размещается пользовательский код
    ' инициализации страницы
  End Sub

  Private Sub HTMLButton_ServerClick(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles HtmlButton.ServerClick
    Page.Response.Write("HTML Button was clicked")
  End Sub

  Private Sub WebControlButton_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles WebControlButton.Click
    Page.Response.Write("Web control button was clicked")
  End Sub
End Class
```

Обратите внимание: код практически неотличим от аналогичного кода VB .NET для работы с формой приложения Windows. Главное различие заключается в том, что в качестве базового выбран класс `System.Web.UI.Page` вместо `System.Windows.Forms.Form`. В остальном определения элементов выглядят практически так же.

Когда браузер запрашивает web-страницу, ASP .NET создает класс `WebForm1`. ASP .NET вызывает методы `Page_Init` и `Page_Load` и все остальные методы, которые вы переопределяете в своей реализации, инициирует события на основании полученной информации, задает все свойства объектов (в частности, используемых элементов) и обеспечивает сохранение этих свойств, чтобы они оставались действительными на протяжении всего сеанса даже при запросе других страниц.

Как это делается?

Не знаю, да это и не важно. Просто маленькое волшебство ASP .NET, благодаря которому .NET так упрощает программирование web-приложений.

Одной из положительных сторон web-элементов является то, что они могут разрабатываться с учетом используемого браузера или генерировать базовый, платфор-

менно-независимый HTML-код — даже при использовании серверных технологий Microsoft вы сможете сгенерировать код, совместимый с любым браузером.

В начале этого раздела я предложил разместить в дизайнере ASP .NET две кнопки из разных групп панели инструментов (HTML и Web Forms). Эти кнопки представляют два разных типа элементов, используемых в приложениях ASP .NET. В документации отмечен ряд различий между HTML-элементами и web-элементами. В частности, HTML-элементы непосредственно отображаются на теги HTML, а web-элементы обеспечивают более высокий уровень абстракции и обладают расширенными возможностями. Но главное различие проявляется при попытке открыть страницу aspx в традиционном редакторе web-страниц вроде Microsoft FrontPage[1].

Вы увидите только кнопку HTML.

Дело в том, что теги HTML описывают стандартные элементы, а для определения web-элементов используются специальные теги, не поддерживаемые обычными редакторами HTML (в нашем примере — тег `asp:Button`). Редактор web-форм Visual Studio распознает эти теги и соответствующим образом воспроизводит web-элементы на форме.

Если вы предпочитаете работать во внешней программе с собственным редактором HTML, вероятно, ваш выбор будет в основном ограничиваться HTML-элементами. При использовании web-элементов теги придется вводить вручную. Кроме того, положение web-элементов должно быть согласовано с программой, чтобы она могла правильно учитывать его при выводе.

Мы подошли к очень важному аспекту программирования web-приложений в Visual Studio.

Web-приложения Visual Studio разрабатывались для удобства программиста. При помощи Visual Studio вы сможете быстро создавать сложные web-приложения с минимальными затратами времени (при условии, что вы освоили общие навыки .NET-программирования, а это отнюдь не простая задача). Однако многие web-дизайнеры слабо разбираются в программировании, искренне презирают FrontPage и предпочитают работать в продуктах других фирм, например в Dream-Weaver. Вряд ли они захотят переносить свои разработки в редактор web-страниц Visual Studio — функциональный, но внешне не впечатляющий. На момент написания книги ни одна независимая фирма, занимающаяся созданием редакторов HTML, не объявила об интеграции своего продукта с Visual Studio.

Интересно, как будут развиваться события. Захотят ли web-дизайнеры осваивать Visual Studio? Привыкнут ли разработчики web-приложений к отделению пользовательского интерфейса от программного кода? Сейчас еще рано судить об оптимальном пути интеграции этих двух подходов.

Web-приложение BillingInfo

В каталоге CH14WebBilling находится web-приложение WebBilling. Допустим, вымышленная компания из предыдущего примера хочет, чтобы ее сотрудники регистрировали свои затраты времени в Интернете при помощи web-браузера.

[1] Чтобы открыть файл Webform1.aspx в FrontPage или другом редакторе web-страниц, следует переименовать его в Webform1.asp.

На рис. 14.2 показана структура формы в окне Visual Studio Webform Designer. Информация вводится в трех текстовых полях. Рядом находится элемент `RangeValidator`, который проверяет, что введенное число часов задано в интервале от 0,1 до 200 (сообщение, показанное на экране, отображается лишь при вводе недопустимой величины).

В двух элементах `Label` выводятся два введенных значения, последнее и предпоследнее, тем самым демонстрируется возможность хранения дополнительных данных в объекте `Session`. Элемент `DataGrid` содержит текущее содержимое базы данных, но в отличие от Windows-версии этот элемент не позволяет редактировать записи непосредственно в элементе.

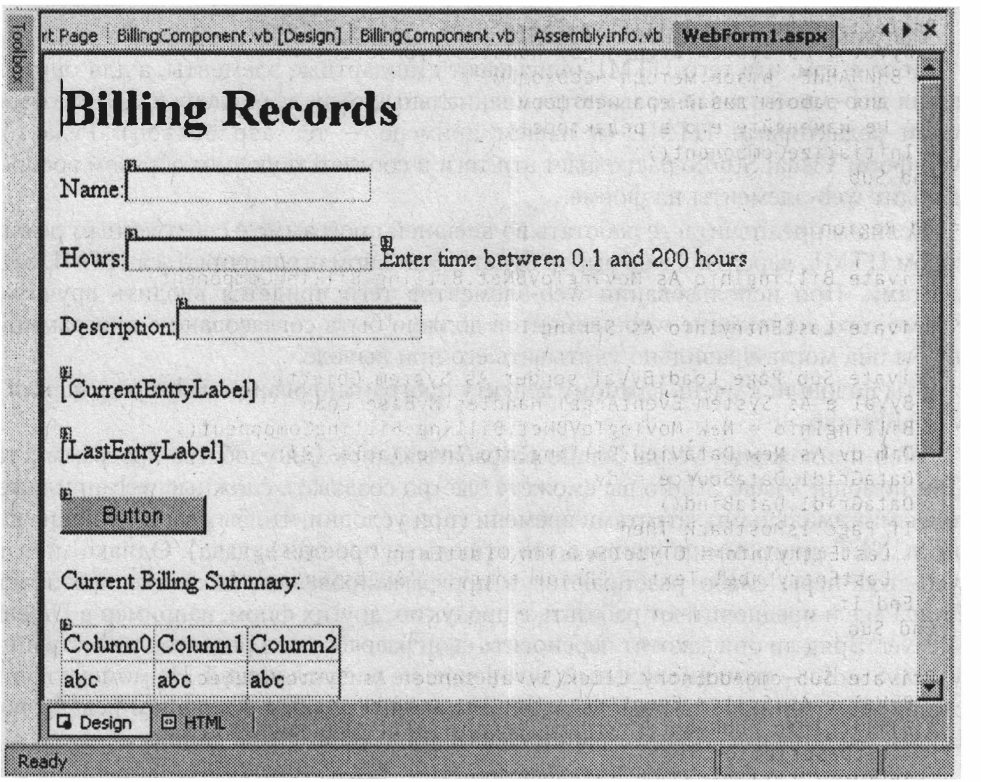


Рис. 14.2. Форма WebBilling в режиме конструирования

В листинге 14.6 показан вспомогательный код Visual Basic для файла WebBilling.aspx.

Листинг 14.6. Код модуля VB в приложении WebBilling

```
Public Class WebForm1
  Inherits System.Web.UI.Page
  Protected WithEvents txtName As System.Web.UI.WebControls.TextBox
  Protected WithEvents txtHours As System.Web.UI.WebControls.TextBox
  Protected WithEvents txtDescription As System.Web.UI.WebControls.TextBox
  Protected WithEvents DataGrid1 As System.Web.UI.WebControls.DataGrid
  Protected WithEvents LastEntryLabel As System.Web.UI.WebControls.Label
```

```
  Protected WithEvents CurrentEntryLabel As _
    System.Web.UI.WebControls.Label
  Protected WithEvents RangeValidator1 As _
    System.Web.UI.WebControls.RangeValidator
  Protected WithEvents cmdAddEntry As System.Web.UI.WebControls.Button

#Region " Web Form Designer Generated Code "

  ' Необходимо для работы дизайнера web-форм.
  <System.Diagnostics.DebuggerStepThrough()> Private _
    Sub InitializeComponent()

  End Sub

  Protected Sub Page_Init(ByVal Sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Init
    'ВНИМАНИЕ: вызов метода необходим
    ' для работы дизайнера web-форм.
    ' Не изменяйте его в редакторе!
    InitializeComponent()
  End Sub

#End Region

  Private BillingInfo As MovingToVBNet.Billing.BillingComponent

  Private LastEntryInfo As String

  Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    BillingInfo = New MovingToVBNet.Billing.BillingComponent()
    Dim dv As New DataView(BillingInfo.Info.Tables(0))
    DataGrid1.DataSource = dv
    DataGrid1.DataBind()
    If Page.IsPostBack Then
      LastEntryInfo = CType(Session("LastEntry"), String)
      LastEntryLabel.Text = "Prior entry: " & LastEntryInfo
    End If
  End Sub

  Private Sub cmdAddEntry_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdAddEntry.Click
    BillingInfo.AddRecord(txtName.Text, CInt(txtHours.Text), _
      txtDescription.Text)
    Session("LastEntry") = txtName.Text & ": " & txtHours.Text & _
      " hours - " & txtDescription.Text
    CurrentEntryLabel.Text = "Current entry: " & txtName.Text & _
      ": " & txtHours.Text & " hours - " & txtDescription.Text
    txtHours.Text = ""
    txtDescription.Text = ""

    ' Обновить таблицу
    Dim dv As New DataView(BillingInfo.Info.Tables(0))
    DataGrid1.DataSource = dv
    DataGrid1.DataBind()

  End Sub
  Protected Overrides Sub Render(ByVal writer As _
    System.Web.UI.HtmlTextWriter)
```

продолжение ⇙

Листинг 14.6 (продолжение)

```
    MyBase.Render (writer)
    writer.WriteLine ("Here is some extra text ")
  End Sub

  Private Sub WebForm1_Unload(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Unload
    BillingInfo.Dispose()
  End Sub

End Class
```

Приведенный код обладает рядом интересных особенностей.

- Объекты элементов управления определяются точно так же, как в традиционных приложениях Windows.
- При обработке события `Page_Load` создается экземпляр класса BillingComponent. Программа создает объект ADO `DataView` для таблицы `BillingTable` и назначает его источником данных для элемента `DataGrid`. Связывание элемента с данными осуществляется вызовом метода `DataBind`.
- Свойство `Page.IsPostBack` позволяет узнать, была ли страница загружена в первый раз или повторно.
- Сеансовые переменные (`Session`) используются для хранения информации уровня сеанса для одной или нескольких страниц приложения.
- Переопределение метода `Render` позволяет сгенерировать для страницы дополнительный код HTML.

В рассмотренной ситуации web-приложение открывает замечательные возможности: чтобы работники нашей вымышленной компании могли сохранить информацию о своем рабочем времени, им достаточно простого доступа к Web. Но и это решение далеко не исчерпывает всех возможностей, предоставляемых .NET.

Решение с использованием web-службы

Предположим, наша компания решила также реализовать операции с базой данных с использованием web-службы. Web-служба представляет собой средство предоставления доступа к объекту через Web. В отличие от web-приложений, web-службы не обладают пользовательским интерфейсом — у них есть только методы и свойства.

Пример WebBillingService в каталоге CH14WebBillingService показывает, как предоставить доступ к методам исходного компонента BillingComponent через web-службу.

Класс WebBillingService приведен в листинге 14.7. Он выглядит настолько тривиально, что даже не требует комментариев. Единственное его отличие от стандартных компонентов Windows заключается в том, что он объявлен производным от класса `System.Web.Services.WebService`, а с некоторыми из его методов связывается атрибут `WebMethod()`.

Листинг 14.7. Класс WebBillingService

```
Imports System.Web.Services

Public Class WebBillingService
  Inherits System.Web.Services.WebService

#Region " Web Services Designer Generated Code "

  Public Sub New()
    MyBase.New()

    ' Необходимо для работы дизайнера web-служб.
    InitializeComponent()

    ' Дальнейшая инициализация выполняется
    ' после вызова InitializeComponent().

  End Sub

  ' Необходимо для работы дизайнера web-служб.
  Private components As System.ComponentModel.Container

  ' ВНИМАНИЕ: следующий фрагмент необходим
  ' для работы дизайнера web-служб.
  ' Для его модификации следует использовать
  ' дизайнер web-служб.
  ' Не изменяйте его в редакторе!
  <System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
    components = New System.ComponentModel.Container()
  End Sub

  Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
    ' ВНИМАНИЕ: следующий фрагмент необходим
    ' для работы дизайнера web-служб.
  ' Не изменяйте его в редакторе!
  End Sub

#End Region

  '
  Dim BillInfo As MovingToVBNet.Billing.BillingComponent

  <WebMethod()> Public Sub AddBillingRecord(ByVal Name As String, ByVal hours
As Double, ByVal Description As String)
    BillInfo = New MovingToVBNet.Billing.BillingComponent()
    BillInfo.AddRecord(Name, hours, Description)
    BillInfo.Dispose()
  End Sub

  <WebMethod()> Public Function GetBillingInfo() As DataSet
    BillInfo = New MovingToVBNet.Billing.BillingComponent()
    Return (BillInfo.Info)
  End Function

End Class
```

При запуске этой службы в среде Visual Studio выводится разнообразная информация о web-службе, в том числе списки всех методов и свойств и примерный

синтаксис их вызова с использованием SOAP, команд HTTP Get и Post. Более того, вы даже сможете обращаться к методам и свойствам, если они достаточно просты для вызова простой командой HTTP Get.

Поскольку для транспортировки обращений используется протокол HTTP, службы могут работать на любом web-сервере на базе IIS. Поскольку большинство брандмауэров (firewalls) разрешает прохождение запросов HTTP (обязательное условие для всех общедоступных web-серверов), тем самым решается одна из самых больших проблем из области удаленного доступа.

Этой информации достаточно для использования web-службы из любого приложения или компонента, способного генерировать запросы HTTP. Если вы работаете с web-службой из приложения Visual Studio, задача становится еще проще.

Проектирование распределенных приложений

«Клиент-сервер», «трехуровневая архитектура», «распределенные приложения» — все эти термины описывают стандартные архитектурные решения из области сетевого программирования. Вместо того чтобы дополнять этот список новыми терминами, Microsoft .NET скорее предлагает взглянуть на распределенные приложения с новой точки зрения. Все существующие решения предполагают, что в вашем решении будет воплощена одна конкретная архитектура, а это требование накладывает немалую ответственность на проектировщика, поскольку ошибки неминуемо приводят к снижению быстродействия, потерям времени и денег. .NET предлагает другой подход. Вместо того чтобы выбирать конкретную архитектуру и придерживаться ее, вы создаете компоненты и используете их в разных сочетаниях. В некоторых ситуациях это позволяет найти не одно, а несколько оптимальных решений.

В приведенном выше примере было принято ключевое решение — создать компонент BillingComponent, обеспечивающий подключение к базе данных. При наличии готового компонента написать «толстого» клиента для локальной сети совсем несложно. Так же просто и создать web-приложение для удаленного доступа через браузер.

Пример BillingWinAppWeb показывает, как интегрировать web-службу с «толстым клиентом» простым включением web-ссылки в проект. В результате все объекты и их члены немедленно становятся доступными для приложения Windows, словно компонент установлен на локальном компьютере. Вся черная работа по сериализации объектов и маршалингу при обращениях к методам и свойствам web-службы и обратно выполняется автоматически. Круг замыкается: web-службы позволяют создавать удаленные приложения «толстых клиентов» без сложностей DCOM или проблем с настройкой брандмауэров.

Но и это не все!

Рассмотрим некоторые возможности, ставшие доступными благодаря существованию web-служб.

- Приложение для PDA (Personal Digital Assistant) или сотового телефона, позволяющее оперативно ввести информацию и отправить ее посредством вызова метода web-службы на базе простого запроса HTTP.

- Макрос Word, фиксирующий время открытия документа. Перед закрытием документа макрос предлагает ввести описание и сохраняет продолжительность работы над документом в базе данных, используя web-службу.
- Программа для сотового телефона, регистрирующая все звонки и при помощи web-службы сохраняющая номер и продолжительность каждого разговора в базе данных.
- Web-форму из описанного выше web-приложения легко превратить в web-элемент. Размещение этого элемента на других web-страницах позволит работникам вводить данные о затратах времени даже при работе с другими частями корпоративного web-сайта.
- Вы можете создать элемент на базе формы Windows и использовать его в качестве «толстого клиента» в браузере (по аналогии с использованием элементов ActiveX в COM). Элемент предоставляет расширенный пользовательский интерфейс в браузере, но при этом он будет использовать ту же web-службу.

Иначе говоря, web-служба позволяет в значительной степени автоматизировать утомительный и неточный процесс сбора статистики о затратах рабочего времени[1].

В настоящее время web-сайты в основном предоставляют информацию в форме, доступной для человека. По мнению Microsoft, в будущем Web превратится в сеть компьютеров, предоставляющих в распоряжение пользователя объекты с определенной функциональностью.

Сейчас трудно сказать, насколько эти представления соответствуют действительности. Конечно, во многих случаях такой подход вполне оправдан. Например, такая компания, как Federal Express, могла бы создать специальную web-службу для отслеживания заказов, чтобы другие web-сайты могли реализовать отслеживание самостоятельно, не отсылая своих посетителей на web-сайт Federal Express. Впрочем, пока трудно сказать, приживутся ли такие модели в других сферах бизнеса и насколько распространенными они станут.

Независимо от того, превратятся ли общедоступные web-службы в модное направление программирования, они наверняка будут использоваться во многих серьезных программах в качестве основы для построения распределенных приложений.

Итак, настоящий подход к Интернет-программированию в .NET формулируется следующим образом: вместо того чтобы выбирать в своем решении один вариант архитектуры, вы можете выбрать все варианты сразу — при этом web-службы играют роль «клея», обеспечивающего взаимодействие разных компонентов.

При разработке решений на базе .NET основное внимание должно уделяться не конкретной архитектуре, а разбиению на компоненты. Если вам удастся определить универсальный набор компонентов, вы будете пользоваться значительной свободой при выборе их места в практической реализации. Например, если тестирование выявит недостаточную пропускную способность канала между базой данных и бизнес-объектом, вы сможете легко переместить компонент на сервер или опробовать другие каналы удаленного доступа.

---

[1] Я не уверен, что работников сильно обрадует «Большой брат», наблюдающий за всем, что они делают. Помните: то, что вы *можете* что-то сделать, вовсе не означает, что это *нужно* делать.

Поддержка Winsock

Средства интернет-программирования в .NET не ограничиваются высокоуровневыми web-приложениями и web-службами. В .NET предусмотрен удобный класс для работы непосредственно с Winsock.

В листинге 14.8 показан пример приложения, предназначенного для опроса (probing) портов. Подобные программы часто используются хакерами и экспертами в области безопасности для поиска уязвимых мест в защите системы.

Листинг 14.8. Приложение Probe

```
Imports System.Net
Public Class ProbeForm
  Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

  Public Sub New()
    MyBase.New()

    ' Необходимо для работы дизайнера форм Windows.
    InitializeComponent()

    ' Дальнейшая инициализация выполняется
    ' после вызова InitializeComponent().

  End Sub

  ' Форма переопределяет Dispose для очистки списка компонентов.
  Public Overloads Overrides Sub Dispose()
    MyBase.Dispose()
    If Not (components Is Nothing) Then
      components.Dispose()
    End If
  End Sub
  ' Разрешить доступ к элементам пользовательского интерфейса
  ' из другого потока.
  ' Windows обеспечивает автоматический маршалинг.
  Friend WithEvents listBox1 As System.Windows.Forms.ListBox
  Friend WithEvents txtIP As System.Windows.Forms.TextBox
  Friend WithEvents cmdProbe As System.Windows.Forms.Button
  Friend WithEvents label1 As System.Windows.Forms.Label
  Friend WithEvents statusBar1 As System.Windows.Forms.StatusBar

  ' Необходимо для работы дизайнера форм Windows.
  Private components As System.ComponentModel.Container

  ' ВНИМАНИЕ: следующий фрагмент необходим
  ' для дизайнера форм Windows.
  ' Для его модификации следует использовать
  ' дизайнер форм Windows.
  ' Не изменяйте его в редакторе!
  <System.Diagnostics.DebuggerStepThrough()> Private Sub _
  InitializeComponent()
    Me.txtIP = New System.Windows.Forms.TextBox()
    Me.cmdProbe = New System.Windows.Forms.Button()
    Me.label1 = New System.Windows.Forms.Label()
```

```
    Me.listBox1 = New System.Windows.Forms.ListBox()
    Me.statusBar1 = New System.Windows.Forms.StatusBar()
    Me.SuspendLayout()
    '
    'txtI
    '
    Me.txtIP.Location = New System.Drawing.Point(88, 16)
    Me.txtIP.Name = "txtIP"
    Me.txtIP.Size = New System.Drawing.Size(184, 20)
    Me.txtIP.TabIndex = 1
    Me.txtIP.Text = ""
    '
    'cmdProbe
    '
    Me.cmdProbe.Location = New System.Drawing.Point(112, 48)
    Me.cmdProbe.Name = "cmdProbe"
    Me.cmdProbe.Size = New System.Drawing.Size(64, 24)
    Me.cmdProbe.TabIndex = 3
    Me.cmdProbe.Text = "Probe"
    '
    'label1
    '
    Me.label1.Location = New System.Drawing.Point(32, 16)
    Me.label1.Name = "label1"
    Me.label1.Size = New System.Drawing.Size(56, 23)
    Me.label1.TabIndex = 2
    Me.label1.Text = "IP:"
    Me.label1.TextAlign = System.Drawing.ContentAlignment.MiddleRight
    '
    'listBox1
    '
    Me.listBox1.Location = New System.Drawing.Point(16, 88)
    Me.listBox1.Name = "listBox1"
    Me.listBox1.Size = New System.Drawing.Size(256, 160)
    Me.listBox1.TabIndex = 0
    '
    'statusBar1
    '
    Me.statusBar1.Location = New System.Drawing.Point(0, 253)
    Me.statusBar1.Name = "statusBar1"
    Me.statusBar1.Size = New System.Drawing.Size(292, 20)
    Me.statusBar1.TabIndex = 5
    Me.statusBar1.Text = "statusBar1"
    '
    'ProbeForm
    '
    Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
    Me.ClientSize = New System.Drawing.Size(292, 273)
    Me.Controls.AddRange(New System.Windows.Forms.Control() _
    {Me.statusBar1, Me.cmdProbe, Me.label1, Me.txtIP, Me.listBox1})
    Me.Name = "ProbeForm"
    Me.Text = "port prober"
    Me.ResumeLayout (False)

  End Sub

#End Region
  Private ProbingThread As Threading.Thread
  Private ProberClass As SocketProber
```

продолжение ⇩

Листинг 14.8 (продолжение)

```
  Private Sub cmdProbe_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdProbe.Click
    If cmdProbe().Text = "Stop" Then
      ProberClass.StopTheThread = True
      ProbingThread.Join()
      cmdProbe().Text = "Probe"
      ProbingThread = Nothing
      ProberClass = Nothing
      Exit Sub
    End If
    ProberClass = New SocketProber()
    ProberClass.TheForm = Me
    ProbingThread = New System.Threading.Thread(New _
    Threading.ThreadStart(AddressOf ProberClass.SubThreadEntry))
    cmdProbe().Text = "Stop"
    listBox1().Items.Clear()

    ProbingThread.Start()
  End Sub

  Friend Function GetIP() As String
    Return txtIP.Text
  End Function

  Friend Sub SetStatus(ByVal s As String)
    Me.statusBar1.Text = s
  End Sub

  Friend Sub AddToList(ByVal s As String)
    listBox1.Items.Add (s)
  End Sub
End Class

Delegate Function fNoParams() As String
Delegate Sub fSetString(ByVal s As String)

Class SocketProber
  Friend TheForm As ProbeForm
  Friend StopTheThread As Boolean
  Public Sub SubThreadEntry()
    Dim s As Sockets.Socket
    Dim ip As IPAddress
    Dim ep As IPEndPoint
    Dim textfp As fNoParams = AddressOf TheForm.GetIP

    ip = IPAddress.Parse(CStr(TheForm.Invoke(textfp)))
    Dim portnumber As Integer
    For portnumber = 1 To &H7FFF
      Dim fpstat As fSetString = AddressOf TheForm.SetStatus
      TheForm.Invoke(fpstat, New String() {"Checking port: " & _
        Str(portnumber)})
      ep = New IPEndPoint(ip, portnumber)
      s = New Sockets.Socket(Sockets.AddressFamily.InterNetwork, _
        Sockets.SocketType.Stream, Sockets.ProtocolType.Tcp)
      Try
        s.Connect (ep)
        Dim plist As fSetString = AddressOf TheForm.AddToList
```

```
        TheForm.Invoke(plist, New String() {"Connected to port " & _
          Str(portnumber)})
      Catch ex As Exception
        debug.WriteLine("Failed port " & Str(portnumber) & _
          " - " & ex.Message)
      End Try

      s.Close()
      If StopTheThread Then Exit Sub ' Запрос на завершение
    Next
  End Sub

End Class
```

В данном примере опрос портов выполняется отдельным программным потоком, чтобы процесс опроса не влиял на работу пользовательского интерфейса. Обратите внимание на пару интересных особенностей этого приложения.

- Очень удобный метод `ipaddress.Parse` преобразует IP-адрес из текстовой формы в структуру `IPAddress`.
- Вместо прямых обращений к форме фонового потока используется метод `Invoke`. Вспомните, о чем говорилось в главе 13 — формы небезопасны по отношению к потокам.

Взгляд со стороны

Иногда меня спрашивают, сколько авторов на меня работает. Уверяю вас, я действительно сам пишу свои книги и даже готовлю все иллюстрации (хотя для того, чтобы они хорошо смотрелись, обычно приходится прибегать к услугам специалистов по компьютерной графике).

Тем не менее я охотно прислушиваюсь к чужому мнению. Этой книге очень повезло с техническим редактором: Скотт Стабберт работает в Microsoft и участвует в разработке обучающих материалов и примеров программ для .NET (всего того, что Microsoft обычно демонстрирует на презентациях или семинарах). Он особенно хорошо разбирается в аспектах .NET, связанных с web-программированием, поэтому я попросил его сказать несколько слов по поводу роли .NET в Web. Привожу его мнение почти дословно.

«...web-элементы очень, очень важны. Чтобы лучше понять их роль, необходимо вернуться к выходу VB3. Почему Visual Basic стал таким популярным языком? По нескольким причинам, но самой главной из них были элементы VBX. На рынке каждую неделю появлялись всевозможные новинки, созданные независимыми фирмами, и программисты с нетерпением ждали очередного выпуска „Visual Basic Programmers Journal“. Некоторые разработки были просто замечательными — „сногсшибательными“, как любит выражаться наш друг мистер Джобс. Другие... мягко говоря, не блистали качеством. Люди покупали хорошее, а плохое отправлялось на помойку[1].

[1] Не могу удержаться от замечания (это снова Дэн) — моя компания Desaware выпустила свой первый продукт в эпоху VB1. Хотя природная скромность не позволяет мне расхваливать качество своих разработок, факт налицо — мы продолжаем работать!

А теперь перенесемся на целую эпоху (по крайней мере, так кажется) — с середины в конец 90-х годов, когда появились такие инструменты, как Visual InterDev, FrontPage и т. д. Эти инструменты пишут код за вас. Более того, этот код работает — но попробуйте изменить его и заставить делать то, что хотите *вы*. Автоматически сгенерированный код никогда не работает в точности так, как было задумано. Автоматизированное построение web-сайтов чем-то напоминает нарезку печенья по готовой форме: получается быстро и удобно, но за это приходится расплачиваться потерей индивидуальности. Профессионалы обычно пишут весь код вручную, поскольку вспомогательные программы не удовлетворяют их повышенным требованиям. Очень часто сгенерированный код трудно редактировать и сопровождать, а иногда он и вовсе оказывается бесполезным. Впрочем, я отвлекся.

Итак, web-элементы являются интернет-аналогом элементов VBX/OCX. Появятся новые „сногсшибательные" разработки — возможно, среди их авторов окажетесь и вы. Конечно, будут и неудачи. Со временем качество элементов будет оцениваться по качеству сгенерированного ими кода HTML/DHTML и клиентских сценариев. Хорошо написанный web-элемент будет изменять свои выходные данные в зависимости от типа браузера или, скажем, при запросе страницы с сотового телефона через протокол WAP (Wireless Application Protocol).

Архитектура web-элемента способствует развитию унифицированного кода, адаптируемого для конкретного случая программистом, использующим элемент. Проблема «штампованного» HTML-кода решается при помощи шаблонов. Кстати говоря, перед нами еще один уровень отделения интерфейса от реализации. Разработчик элемента определяет его функциональность, а программист, использующий элемент, определяет специфические особенности генерируемого HTML-кода.

Классы web-элементов (как и всех остальных элементов .NET) можно взять за основу для определения ваших собственных элементов и изменить их поведение в случае надобности.

Ожидается, что web-элементы, как и родственные им элементы VBX/OCX, сформируют развитый рынок для продуктов независимых фирм. Несомненно, в таком развитии событий заинтересованы все компании и особенно программисты VB, которым было бы интересно заняться web-приложениями.

В настоящее время во многих компаниях идет лихорадочная работа над элементами, которые должны быть готовы к выходу окончательной версии .NET, и *вы* как представитель сообщества программистов VB от этого только выиграете. Рынок вознаградит хорошие разработки и накажет плохие (во всяком случае, мы на это надеемся), а с каждым новым элементом ASP .NET Framework станет еще мощнее и удобнее в использовании».

Спасибо, Скотт.

Честно говоря, я не уверен в его правоте. Конечно, я согласен с тем, что возможности web-элементов колоссальны, но говорить о рыночных перспективах пока рано — хотя я не сомневаюсь, что многие разработчики компонентов предложат свои web-элементы и некоторые из них будут пользоваться успехом. Конечно, компания Desaware давно отказалась от разработки общих элементов пользовательского интерфейса — мы предпочитаем заниматься более творческими и менее очевидными вещами. Но кто знает? Может, и мне суждено написать пару-тройку web-элементов. Поживем — увидим...

Итоги

В этой главе в основном рассматриваются архитектурные решения и концепции, а не конкретные реализации. Вы узнали, что XML и XSL в действительности чрезвычайно просты, а вся шумиха вокруг них объясняется не столько их сложностью, сколько простотой и элегантностью. Также было показано, что правильное разбиение на компоненты позволяет создавать по-настоящему распределенные приложения. В этой главе были продемонстрированы разные способы использования компонентов, от традиционных «толстых клиентов», вся функциональность которых сосредоточена в одной системе, до web-приложений и web-служб. Идеология распределенных приложений в сочетании с логически согласованной архитектурой .NET обеспечивает необходимую гибкость для интеграции новых систем и технологий по мере их появления — в том числе и разработанных за пределами Microsoft.

COM Interop и использование функций Win32 API

15

Жил да был на свете программист. За долгие годы он написал много программ, и все они хорошо работали. Однажды ему явился великий дух ОС и представил новый способ программирования. По словам духа, программист должен был написать еще больше программ, и работать они должны были еще лучше.

Но бедный программист посмотрел на новую систему и понял, что переработка готовых программ займет много лет (и вообще есть занятия поинтереснее). И еще он понял, что в новой системе кое-что сделать попросту невозможно, и это «кое-что» придется делать по-старому.

Но великий дух ОС дал программисту волшебную лампу. Стоило программисту потереть лампу, как появился синий джинн, распевавший песенки на музыку из диснеевских мультиков. Джинн произнес заклинание, и все программы сами собой адаптировались для новой ОС. А еще джинн подарил программисту волшебное кольцо, которое генерировало новые классы для всего, что не поддерживалось новой системой.

И после этого программист жил долго и счастливо.

Но такое случается только в сказках.

К сожалению, синие джинны водятся только в мультиках Диснея, поэтому Microsoft не сможет обеспечить автоматический перенос в .NET всех существующих приложений и компонентов на базе COM. А все волшебные кольца были переданы для съемок фильма «Властелин колец»[1]. Так что какой бы огромной ни была библиотека классов .NET, всегда найдутся задачи, для которых в ней не предусмотрено готовых решений. К счастью, разработчики Microsoft приложили немало усилий к тому, чтобы новые программы .NET могли работать с компонентами COM и вызывать базовые функции Win32 API. Первая задача решается при помощи средств COM Interop (сокращение от Interoperability), а для решения второй используется механизм Platform Invoke (P-Invoke). Большинство классов и атрибутов, упоминаемых в этой главе, находится в пространстве имен `System.Runtime.InteropServices`.

---

[1] К которому я не имею ни малейшего отношения, хотя и жду с нетерпением.

Это два разных механизма, однако на практике они довольно тесно связаны, поскольку в обоих случаях .NET Framework приходится работать с небезопасным кодом[1].

COM Interop

COM Interop представляет собой часть .NET Framework, отвечающую за взаимодействие существующих компонентов/приложений COM с компонентами/приложениями .NET. Говоря о взаимодействии с COM, необходимо учитывать два ключевых обстоятельства.

- Компоненты .NET непосредственно работают только с другими компонентами .NET. Все компоненты .NET находятся в памяти, находящейся под управлением CLR, и уничтожаются сборщиком мусора после того, как на них перестают существовать ссылки в переменных корневого уровня.
- Компоненты COM непосредственно работают только с другими компонентами COM. Их работа основана на низкоуровневой реализации, в которой задействованы таблицы виртуальных указателей. Компоненты COM используют механизм подсчета ссылок — появление новой ссылки на объект COM должно приводить к увеличению счетчика ссылок. При освобождении ссылки значение счетчика должно уменьшаться. Когда счетчик ссылок уменьшается до нуля, объект уничтожается.

Система COM Interop выполняет функции шлюза между компонентами .NET и компонентами COM. Благодаря ей компоненты COM с точки зрения .NET кажутся компонентами .NET, а компоненты .NET с точки зрения COM кажутся компонентами COM (рис. 15.1).

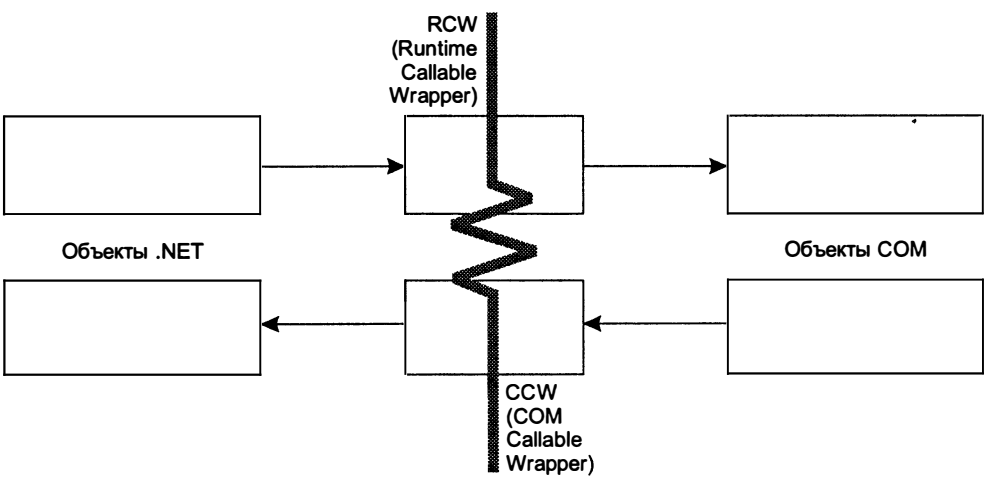


Рис. 15.1. Схема работы COM Interop

[1] То есть кодом, не находящимся под непосредственным контролем .NET Framework.

Когда объект .NET хочет обратиться к объекту COM, он использует RCW (Runtime Callable Wrapper) — сборку .NET, благодаря которой объект COM выглядит как обычный объект .NET.

Когда объект COM желает обратиться к объекту .NET, среда создает объект CCW (COM Callable Wrapper), который выглядит и работает как обычный объект COM.

COM Interop работает в обоих направлениях. Очевидно, приоритетным направлением станет использование объектов COM объектами .NET — в настоящее время существует огромное количество объектов COM, а объектов .NET совсем мало. Конечно, новые объекты .NET должны работать с COM, но часто ли будут создаваться объекты .NET, которые должны использоваться существующими компонентами COM? Подозреваю, что это будет очень редким событием. Возможно, исключением окажутся транзакционные компоненты, которые должны работать за пределами процесса (для большинства таких компонентов базовым классом будет класс `System.EnterpriseServices.ServicedComponent`). Тем не менее возможность обращения к объектам .NET из объектов COM не стоит недооценивать, поскольку приложениям .NET нередко придется предоставлять объекты, предназначенные для использования в системах на базе COM.

Я не собираюсь излагать в этой главе все тонкости работы COM Interop. Это не нужно. Хотя в документации COM Interop чрезвычайно подробно расписаны все возможные ситуации, для большинства читателей интерес представляют только сценарии, удовлетворяющие двум критериям:

- везде, где это возможно, вы используете Visual Studio вместо вспомогательного инструментария;
- практически все используемые объекты COM относятся к той категории, с которой вы привыкли работать, — то есть являются VB6-совместимыми. Это означает, что все используемые объекты совместимы с Automation и поддерживают двойственный интерфейс (непосредственный и основанный на `IDispatch`)[1].

Остальные случаи будут игнорироваться, поскольку описанный сценарий действует в 95 % (а то и больше) ситуаций, с которыми вы столкнетесь.

Начнем с направления «от .NET к COM».

Использование объектов COM в .NET

Использование объектов COM в Visual Basic .NET не вызывает ни малейших проблем. Достаточно выполнить следующие действия.

1. Убедитесь в том, что используемый объект COM зарегистрирован в системе.
2. Выберите команду **Add ▸ References** в меню **Project** или в окне **Solution Explorer**.
3. Перейдите на вкладку COM и выберите нужный объект.

[1] К сожалению, даже краткое изложение основ COM выходит за рамки этой книги. Впрочем, даже если вы незнакомы с COM, попробуйте прочитать эту главу, не обращая внимания на непонятные термины «Automation» и «IDispatch».

При добавлении ссылки на компонент COM в Visual Studio вам будет предложено построить «первичную сборку» (RCW) для компонента на основании библиотеки типов компонента. Это происходит лишь в том случае, если сборка RCW для данного компонента COM еще не была зарегистрирована в глобальном кэше сборок. Если ваш ответ будет положительным, Visual Studio создает сборку RCW и размещает ее в каталоге, в котором с ней будет работать ваш проект[1].

Проект VB6COM содержит простую библиотеку ActiveX DLL для VB6, определяемую следующим образом:

```
' Пример обращения к компоненту COM из .NET
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Option Explicit

Private Declare Function GetCurrentThreadId Lib "kernel32" () As Long

Public Function TimesTwo(ByVal x As Long) As Long
  TimesTwo = x * 2
End Function

Public Function TrimString(ByVal s As String) As String
  TrimString = Trim(s)
End Function

Public Function GetThisThreadId() As Long
  GetThisThreadId = GetCurrentThreadId()
End Function
```

Чтобы компонент dwComFromNet.dll стал доступным для VB .NET, его необходимо зарегистрировать. Для этого можно построить DLL заново или воспользоваться утилитой RegSvr32.

Пример обращения к сборке RCW после ее создания показан в приложении UsesCOM1 (листинг 15.1).

Листинг 15.1. Проект UsesCOM1[2]

```
' Пример обращения к компоненту COM из .NET
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Imports System.Threading
Module Module1
  ' В имя компонента, используемое в .NET, включается
  ' номер версии библиотеки типов.
  Dim ComObject As New dwComFromDotNet_8_0.SampleClass1()

  Sub FromAlternateThread()
    Dim tid As Long
    tid = ComObject.GetThisThreadId()
    Console.WriteLine ("From other thread, TID = " & Str(tid))
  End Sub


  Sub Main()
```

продолжение ☞

[1] Дальнейшее описание относится только к локальным сборкам. Если вы хотите создать сборку RCW для размещения в глобальном кэше сборок, воспользуйтесь утилитой TLBImp, описанной в документации. Вопросы установки и размещения компонентов более подробно рассматриваются в главе 16.

[2] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Листинг 15.1 (продолжение)

```
    Dim newThread As New Thread(AddressOf FromAlternateThread)
    Dim tid As Long
    newThread.Start()
    tid = ComObject.GetThisThreadId()
    Console.WriteLine ("From main thread, TID = " & Str(tid))
    Console.WriteLine (ComObject.TimesTwo(5))
    Console.WriteLine (ComObject.TrimString("
to trim"))
    newThread.Join()

    Console.ReadLine()

  End Sub

End Module
```

Из приведенного листинга видно, что объект COM создается тем же способом, как и объект .NET. При запуске программы будет получен следующий результат[1]:

```
From main thread, TID = 1692
10
to trim
From other thread, TID = 1692
```

Обратите внимание на интересную подробность: .NET Framework обеспечивает потоковую безопасность для объекта COM. Хотя в программе UsesCOM1 методы объекта вызываются из двух разных потоков, оба вызова автоматически передаются потоку объекта.

Обработка ошибок

Объекты COM сообщают об ошибках при помощи 32-разрядных кодов `HRESULT`. При получении кода `HRESULT`, соответствующего признаку ошибки, RCW автоматически инициирует исключение. Коды `HRESULT` отображаются на ближайшие типы исключений, а их исходные значения сохраняются в объектах `Exception`.

Освобождение объектов

Для каждого объекта COM, используемого в программе, сборка RCW создает промежуточный объект (proxy). Следовательно, программа .NET имеет дело с обычным объектом .NET, который передает обращения к методам и свойствам объекту COM. Промежуточный объект содержит ссылку на объект COM и освобождает его, когда тот становится ненужным. Но когда это происходит?

Как вы знаете, объекты .NET уничтожаются сборщиком мусора с недетерминированным завершением. Таким образом, при использовании объектов COM в .NET отсутствие ссылок на промежуточный объект еще не гарантирует его освобождения. Если объект COM удерживает некоторые ресурсы, возможно, вы предпочтете принудительно освободить эти ресурсы перед освобождением по-

[1] Естественно, идентификаторы потоков будут другими. Строка «From other thread» может находиться в разных местах в зависимости от взаимодействия между потоками.

следней ссылки на объект. Задача решается методом `System.Runtime.InteropServices.Marshal.ReleaseComObject`, уменьшающим счетчик ссылок объекта COM. Класс `Marshal` также содержит метод `AddRef` для увеличения счетчика, поэтому при желании вы фактически можете самостоятельно реализовать механизм подсчета ссылок для объекта. Впрочем, будьте осторожны: неверное использование этих методов может привести к утечке памяти или возникновению исключений.

Контроль версии

В предыдущих главах говорилось о том, что в .NET проблема «кошмара DLL» решается привязкой к конкретным версиям зависимых сборок. К сожалению, на COM эта особенность не распространяется, поэтому при обращениях к объектам COM из .NET теоретически возможны все проблемы, хорошо знакомые программистам VB. Не забывайте обеспечивать контроль версии и совместимость даже в том маловероятном случае, если ваши объекты COM будут использоваться исключительно из сборок .NET.

Позднее связывание

На уровне COM Interop может выполняться позднее связывание объектов COM. Класс `Type` содержит методы `GetTypeFromProgID` и `GetTypeFromCLSID`, позволяющие получить для объекта COM объект `Type`, используемый при позднем связывании (см. главу 11).

Передача структур и других типов параметров

Многие программисты VB не привыкли к тому, что объекты ActiveX DLL могут предоставлять для доступа извне структуры (пользовательские типы VB6). Дело в том, что данная возможность появилась только в VB6 и несовместима со старыми версиями VB (и другими платформами); кроме того, она слишком ненадежна[1]. Оказывается, передача структур в неуправляемый код — процедура весьма сложная. Вообще передача параметров и типов данных COM в .NET обходится без особых сложностей (за исключением типа `Variant`, преобразуемого в тип .NET `Object`). Задача усложняется при использовании интерфейсов, несовместимых с Automation, но, как говорилось выше, в данной главе эта тема не рассматривается, поскольку программисты VB очень редко сталкиваются с ней.

Проблемы передачи структур и других параметров будут подробно рассмотрены ниже, при обсуждении вызова функций Win32 API в .NET. В отличие от COM Interop, где процесс передачи в основном определялся библиотеками типов импорта/экспорта, при вызове функций Win32 API вам придется действовать самостоятельно. Разобравшись в принципах передачи параметров при вызове Win32 API, вы без особых трудностей разберетесь в любых ситуациях, возникающих при передаче параметров в COM.

---

[1] Я несколько раз обжегся на использовании открытых структур в VB6. Возможно, эта проблема была исправлена в последующих обновлениях, но у меня не было желания возвращаться к работе с ними

Дополнительные замечания

Visual Studio прикладывает основательные усилия к тому, чтобы средства COM Interop хорошо работали в .NET. Предусмотрена даже такая возможность, как использование существующих элементов ActiveX в контейнерах .NET. Программисты C++, привыкшие к созданию более сложных объектов и непосредственной работе с IDL (Interface Description Language), найдут в спецификации Interop дополнительные сведения о различных типах параметров и других атрибутах, определяемых в файлах IDL. Тем не менее это один из тех случаев, когда поддержка в VB6 только подмножества COM, совместимого с Automation, играет положительную роль, поскольку у объектов COM, созданных в VB6, не будет проблем с .NET-совместимостью[1].

Использование объектов .NET в COM

Создание объектов .NET, хорошо работающих в COM, — задача более сложная. Впрочем, речь идет не о реальных трудностях. Просто, работая в этой области, необходимо принимать во внимание некоторые дополнительные факторы.

Некоторые из этих факторов упоминаются в документации .NET как «необязательные». Вместо того чтобы перечислять все возможные варианты, я покажу, как правильно организовать внешний доступ к объектам из VB .NET. Данное решение применимо к объектам, которые совместимы с Automation и могут безопасно использоваться в VB6 и COM+. VB .NET также дает возможность создавать объекты, несовместимые с Automation, несовместимые с VB6 и не поддерживающие контроля версии средствами COM. Если вы захотите узнать, как создавать объекты этих типов, обращайтесь к документации Microsoft по VB.

Чтобы понять все требования и логические обоснования того подхода, который я собираюсь продемонстрировать, необходимо кое-что знать о том, как в VB6 организован внешний доступ к объектам COM.

Каждый объект VB6 поддерживает два интерфейса[2]. Первый — непосредственный интерфейс класса. Согласно правилам имя этого интерфейса начинается с символа подчеркивания, и он используется при раннем связывании. Второй интерфейс, `IDispatch`, обеспечивает позднее связывание средствами Automation. В листинге 15.2 приведена библиотека типов для файла dwComFromDotNet.dll (см. листинг 15.1).

ПРИМЕЧАНИЕ

Значения UUID и ID, используемые в вашем случае, будут отличаться от приведенных в листинге 15.2.

[1] На момент издания книги еще неизвестно, до какой степени .NET будет поддерживать управление элементами ActiveX, созданными в VB6. Помните, что элементы ActiveX значительно сложнее компонентов ActiveX DLL и ActiveX EXE, поэтому при работе с ними в среде .NET с большей вероятностью возникают разного рода непредвиденные обстоятельства. Microsoft потратила огромные усилия на обеспечение взаимодействия, но если в этой области возникнут какие-либо проблемы, скорее всего, они будут связаны с элементами ActiveX.

[2] На самом деле больше, но эти два интерфейса решают стандартные задачи COM (такие, как обработка событий) и не реализуют методы, определяемые программистом.

Листинг 15.2. Библиотека типов для файла dwComFromDotNet.dll (VB6 ActiveX DLL)

```
// Файл .IDL (сгенерирован OLE/COM Object Viewer)
//
// typelib filename: dwConFromDotNet.dll

[
  uuid(7CFFA0A5-388D-48DC-8C3F-A3F7206CFC86),
  version(6.0),
  helpstring("MovingToVB .NET: Example of calling COM component from .NET")
]
library dwComFromDotNet
{
  // TLib: // TLib: OLE Automation : {00020430-0000-C000-00000000046}
  importlib("stdole2.tlb");

  // Опережающее объявление всех типов, определенных в библиотеке
  interface _SampleClass1;

  [
    odl,
    uuid(F75159CA-A859-4B9A-10596313D0E1),
    version(1.0),
    hidden,
    dual,
    nonextensible,
    oleautomation
  ]
  interface _SampleClass1 : IDispatch {
    [id(0x60030000)]
    HRESULT TimesTwo(
        [in] long x,
        [out, retval] long* );
    [id(0x60030001)]
    HRESULT TrimString(
        [in] BSTR s,
        [out, retval] BSTR* );
    [id(0x60030002)]
    HRESULT GetThisThreadId([out, retval] long* );
  };
  [
    uuid(47B953CE-7E05-4408-95AA-2A79D739F237),
    version(1.0)
  ]
  coclass SampleClass1 {
    [default] interface _SampleClass1;
  };
};
```

Что мы видим в этом листинге?

- В нем определяется интерфейс с именем `_SampleClass1`, определяющий методы и свойства класса. В качестве признака успеха или неудачи методы возвращают код `HRESULT`.
- Интерфейс помечен атрибутами `dual` и `oleautomation`. Это означает, что он совместим с Automation, а методы интерфейса могут вызываться как напрямую, так и через `IDispatch`. На это также указывает тот факт, что интерфейс объявлен производным от `IDispatch`.

- Интерфейсу присвоен UUID — универсально-уникальный идентификатор (universally unique identifier); также встречаются термины IID (interface identifier), GUID (globally unique identifier) и CLSID (class identifier). Все эти термины относятся к 16-байтовой величине, которая заведомо однозначно идентифицирует объект, интерфейс или класс. Термин изменяется в зависимости от контекста использования, но он всегда означает одно и то же — уникальный 16-байтовый идентификатор.
- С каждым методом интерфейса связан диспетчерский идентификатор (dispid) — число, используемое для идентификации метода при обращении к нему через Automation. Этот идентификатор задается атрибутом `id(…)`.
- Секция `coclass` идентифицирует сам объект. Указанный идентификатор UUID определяет идентификатор класса (CLSID), под которым регистрируется объект.

В этой книге я уже несколько раз упоминал о «кошмаре DLL». Теперь давайте взглянем на него с точки зрения COM. Чтобы новая версия компонента dwComFromDotNet.dll сохранила совместимость с существующей версией, должны выполняться следующие правила.

1. Все UUID нового компонента должны совпадать с UUID существующего компонента[1].
2. Все имена методов и параметров нового компонента должны совпадать с аналогичными именами существующего компонента (в интерфейс можно добавить новые методы, однако это не рекомендуется: вам придется работать с несколькими версиями библиотеки типов, что приведет к усложнению задачи).
3. Методы и свойства нового компонента должны сохранить общее функциональное поведение методов и свойств существующего компонента.

Создание компонента CalledViaCOM (первая попытка)

Чтобы обеспечить возможность обращения к компоненту .NET из COM, можно установить флажок Register for COM Interop в диалоговом окне параметров проекта. Однако в этом случае компонент будет поддерживать только позднее связывание. VB .NET не создает для класса двойственный интерфейс, допускающий как раннее, так и позднее связывание.

Правильный подход к созданию компонента, используемого приложениями COM, заключается в явном определении интерфейса класса (листинг 15.3).

Листинг 15.3. Явное определение интерфейса

```
Imports System.Runtime.InteropServices
Public Interface _CallFromCOM
  Function TimesTwo(ByVal i As Integer) As Integer

End Interface
Public Class CallFromCOM
```

[1] Я не буду отвлекаться на контроль версии библиотек типов, для нашей темы это несущественно.

```
  Implements _CallFromCOM

  Public Function TimesTwo(ByVal i As Integer) As Integer Implements _
  _CallFromCOM.TimesTwo
    Return i * 2
  End Function
  <ComRegisterFunction()> Public Shared Sub _
  OnRegistration(ByVal T As Type)
    MsgBox("I'm being registered!!! :" & T.FullName)
  End Sub
End Class
```

Обращает на себя внимание функция `OnRegistration`. При помощи атрибута `ComRegisterFunction` она сообщает о том, что должна вызываться при регистрации компонента. Также существует парный атрибут `ComUnregisterFunction` для пометки функций, вызываемых при удалении регистрационных данных компонента.

Компилятору VB .NET также необходимо сообщить о том, что компонент будет использоваться объектами COM. Для этого в файле AssemblyInfo.vb устанавливается атрибут `ComVisible` (листинг 15.4).

Листинг 15.4. Файл AssemblyInfo.vb для сборки, доступной для объектов COM

```
Imports System.Reflection
Imports System.Runtime.InteropServices


' Следующая группа атрибутов содержит общую информацию о сборке.
' Изменение значений этих атрибутов приводит к изменению
' информации, связанной со сборкой.

' Значения атрибутов сборки

<Assembly: AssemblyTitle("CalledViaCOM")>
' Выводится в диалоговом окне ссылок VB6!!
<Assembly: AssemblyDescription("MovingToVB.NET Example called via COM")>
<Assembly: AssemblyCompany("Desaware Inc.")>
<Assembly: AssemblyProduct("MovingToVB.NET")>
<Assembly: AssemblyCopyright("Copyright ©2001 by Desaware Inc.")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>

' Идентификатор библиотеки типа при обращении к проекту из COM

<Assembly: Guid("6431f4c9-b2f9-40fb-9420-301b96e2fc8e")>

' Информация о версии сборки состоит из следующих
' четырех величин:
'
' основная версия;
' дополнительная версия;
' ревизия;
' номер построения.
'
' Вы можете задать значения всех атрибутов
' или задать номера построения и ревизии по умолчанию.
' Для этого используется знак '*', как показано ниже.
```

продолжение ⇙

Листинг 15.4 (продолжение)

```
<Assembly: AssemblyVersion("1.0.0.*")>

<Assembly: ComVisible(True)>
<Assembly: ClassInterface(ClassInterfaceType.None)>
```

Если атрибут `ComVisible` равен `True`, все открытые объекты и члены классов будут доступны при регистрации сборки для использования в COM. Тем не менее атрибут `ComVisible` позволяет управлять видимостью отдельных элементов сборки. Например, если установить атрибут `ComVisible(False)` для метода класса, этот метод нельзя будет вызвать из COM. Данная возможность будет продемонстрирована в примере CalledViaCOM2.

Если присвоить атрибуту `ClassInterfaceType` значение `None`, то при экспортировании библиотеки типов интерфейс `IDispatch` не будет сгенерирован как интерфейс по умолчанию, что позволит назначить для всех объектов сборки интерфейс, определенный вами. Не беспокойтесь, возможность позднего связывания при этом не теряется, поскольку определяемый вами интерфейс по умолчанию будет двойственным.

Создание и регистрация компонентов

Зарегистрировать компонент VB .NET для использования в COM очень просто. После создания библиотеки классов (единственная разновидность компонентов .NET, доступных из COM) установите флажок Register for COM Interop на вкладке Configuration Properties Build диалогового окна Project Properties. При этом VB .NET сообщит вам о том, что из COM доступны лишь проекты с сильными именами, и предложит создать сильное имя для сборки. Согласитесь с этим предложением (сильные имена рассматриваются в главе 16).

После построения сборка регистрируется для обращений со стороны объектов COM. В процессе регистрации выполняются следующие действия.

- Имя объекта (в нашем случае `CalledViaCOM.CallFromCOM`) заносится в реестр с ключом `HKEY_LOCAL_MACHINE/Software/CLASSES`. Также в реестре сохраняется CLSID объекта.
- CLSID заносится в реестр с ключом `HKEY_LOCAL_MACHINE/Software/CLASSES/CLSID`. Также создается подключ `InProcServer32` со следующей дополнительной информацией.
  - Параметр по умолчанию определяет библиотеку DLL, реализующую объект. Как ни странно, это библиотека .NET mscoree.dll! «Обертка» CCW создается исполнительной средой .NET, поэтому считается, что объект реализуется именно этой библиотекой.
  - Параметр `Assembly` содержит сильное имя сборки с версией и открытым ключом, однозначно идентифицирующим сборку (см. главу 16).
  - Параметр `CodeBase` определяет местонахождение файла DLL сборки. Таким образом, сборка не обязана находиться в глобальном кэше или в каталоге использующего компонента.
  - Параметр `RuntimeVersion` содержит версию исполнительной среды .NET, необходимую для работы сборки.

- Также существуют параметры для имени класса (каким оно представляется приложению COM) и потоковой модели.

- Для сборки создается библиотека типов, которая регистрируется с ключом HKEY_LOCAL_MACHINE/Software/CLASSES/TypeLib.

В .NET Framework входит утилита TlbExp, позволяющая вручную экспортировать библиотеку типов, и утилита RegAsm, регистрирующая сборку для использования в COM. Возможно, вы будете использовать RegAsm для регистрации сборок при их установке (например, утилита RegAsm позволяет создать .REG-файл, упрощающий регистрацию распространяемых вами сборок), но большинство разработчиков в процессе работы использует встроенные средства регистрации Visual Studio.

Проект VB6NetTest в каталоге проекта CalledViaCom демонстрирует вызов методов объекта .NET с поздним и ранним связыванием.

```
Private Sub cmdLate_Click()
  Dim c As Object
  Set c = CreateObject("CalledViaCOM.CallFromCOM")
  MsgBox c.TimesTwo(5), vbInformation, "Result from .NET component"
End Sub

Private Sub cmdEarly_Click()
  Dim c As New CallFromCOM
  MsgBox c.TimesTwo(5), vbInformation, "Result from .NET component"
End Sub
```

Построение компонента

Проведите следующий эксперимент.

1. Постройте программу VB6NetTest и убедитесь в том, что она работает.
2. Закройте VB6.
3. Постройте сборку CalledViaCom заново.
4. Попробуйте запустить исполняемый файл VB6NetTest (не запуская среды VB6!).

Вы увидите, что позднее связывание по-прежнему работает, но вызовы с ранним связыванием завершаются неудачей. Почему это происходит?

Сравните библиотеку типов, полученную при *первой* регистрации сборки (листинг 15.5) с библиотекой типов, полученной при *следующей* регистрации (листинг 15.6).

ПРИМЕЧАНИЕ

Значения UUID и ID, используемые в вашем случае, будут отличаться от приведенных в листингах 15.5 и 15.6.

Листинг 15.5. Исходная библиотека типов для сборки CalledViaCom

```
// Файл .IDL (сгенерирован OLE/COM Object Viewer)
//
// typelib filename: CalledViaCOM.tlb

[
  uuid(6431A4C9-B2F9-40FB-9420-301B96E2FC8E),
```

продолжение

Листинг 15.5 (продолжение)

```
  version(1.0),
  helpstring("MovingToVB.NET Example called via COM")
]
library CalledViaCOM
{
  // TLib: // TLib: Common Language Runtime Library :
  {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
  importlib("mscorlib.tlb");

  // Опережающее объявление всех типов, определенных в библиотеке
  interface _CallFromCOM;

  [
    odl,
    uuid(340729AC-2E20-3909-A94A-15EF27ED5F04),
    version(1.0),
    dual,
    oleautomation,
      custom({0F21F359-AB84-41E8-9A78-36D110€6D2F9},
      "CalldViaCOM._CallFromCOM")
  ]
  interface _CallFromCOM : IDispatch {
    [id(0x60020000)]
    HRESULT TimesTwo(
        [in] long i,
        [out, retval] long* pRetVal);
  };
  [
    uuid(541F4403-04F3-39B8-83AE-35AD26964015),
    version(1.0),
      custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
      "CalldViaCOM._CallFromCOM")

  ]
  coclass CallFromCOM {
    interface _Object;
    [default] interface _CallFromCOM;
  };
};
```

Листинг 15.6. Библиотека типов сборки CalledViaCom после повторного построения

```
// Файл .IDL (сгенерирован OLE/COM Object Viewer)
//
// typelib filename: CalledViaCOM.tlb

[
  uuid(6431A4C9-B2F9-40FB-9420-301B96E2FC8E),
  version(1.0),
  helpstring("MovingToVB.NET Example called via COM")
]
library CalledViaCOM
{
  // TLib: // TLib: Common Language Runtime Library :
  {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
  importlib("mscorlib.tlb");
  // TLib: OLE Automation: {00020430-0000-0000-C000-000000000046}
  importlib("stdole2.tlb");
```

```
  // Опережающее объявление всех типов, определенных в библиотеке
  interface _CallFromCOM;

  [
    odl,
    uuid(340729AC-2E20-3909-A94A-15EF27ED5F04),
    version(1.0),
    dual,
    oleautomation,
      custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
      "CalldViaCOM._CallFromCOM")
  ]
  interface _CallFromCOM : IDispatch {
    [id(0x60020000)]
    HRESULT TimesTwo(
        [in] long i,
        [out, retval] long* pRetVal);
  };
  [
    uuid(D54C0514-33B9-351F-BF11-923397721F88),
    version(1.0),
      custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
      "CalldViaCOM._CallFromCOM")

  ]
  coclass CallFromCOM {
    interface _Object;
    [default] interface _CallFromCOM;
  };
};
```

Как видно из листинга 15.6, итоговая библиотека типов очень похожа на исходную. Тем не менее идентификатор UUID объекта `CalledViaCOM.CallFromCOM` изменился.

Идентификатор UUID библиотеки типов остался прежним только потому, что Visual Studio .NET по умолчанию включает UUID библиотеки типов в атрибуты сборки, для чего используется строка вида:

```
<Assembly: Guid("6431f4c9-b2f9-40fb-9420-301b96e2fc8e")>
```

Значения UUID библиотек типов в листингах 15.5 и 15.6 совпадают со значением, заданным атрибутом `Assembly:GUID`, хотя обычно каждой сборке присваивается уникальное значение GUID.

Интересно заметить, что UUID интерфейса изменяется лишь при фактической модификации интерфейса. Напрашивается предположение, что VB .NET старается по возможности сохранять идентификаторы интерфейсов, осуществляя внутреннюю проверку совместимости.

Для позднего связывания достаточно имен объекта и метода, поэтому вполне логично, что вызовы с поздним связыванием продолжают работать и после того, как сборка была построена заново. Однако все внутренние идентификаторы объектов, интерфейсов и т. д. могут при этом измениться, что приведет к сбоям при вызовах с ранним связыванием.

К счастью, эта проблема легко решается.

Создание компонента CalledViaCOM — вторая попытка

Фокус заключается в том, чтобы самостоятельно назначить идентификаторы UUID и Dispid разным составляющим сборки.

ВНИМАНИЕ

Приведенные ниже значения UUID выбраны только для пояснения материала. Не используйте эти числа в своих приложениях.

В файле Assemblyinfo.vb проекта CallFromCom2 UUID библиотеки типов задается следующим образом:

```
<Assembly: Guid("c392d911-7806-43e2-a61d-ad3cd3e2a8f3")>
```

В листинге 15.7 приведен обновленный файл Class1.vb.

Листинг 15.7. Файл Class1.vb для сборки CallFromCom2

```
Imports System.Runtime.InteropServices
<Guid("fda97dca-bb0b-4987-961b-8383741cfa8f")> Interface _CallFromCOM2
  <DispId(1)> Function TimesTwo(ByVal i As Integer) As Integer
  <DispId(2)> Function BadWay(ByVal i As Integer) As Object
End Interface

<Guid("72d7e45a-3b76-4a12-bb7e-c096cd97709a")> Public Class CallFromCOM2
  Implements _CallFromCOM2

  <DispId(1)> Public Function TimesTwo(ByVal i As Integer) As Integer _
  Implements _CallFromCOM2.TimesTwo
    Return i * 2
  End Function

  <DispId(2)> Public Function BadWay(ByVal i As Integer) As Object _
  Implements _CallFromCOM2.BadWay
    Return i * 2
  End Function

  <ComVisible(False), DispId(3)> Public Function TimesThree(ByVal i _
  As Integer) As Integer
    Return i * 3
  End Function

  <ComRegisterFunction()> Public Shared Sub OnRegistration(ByVal T _
  As Type)
    MsgBox("I'm being registered!!! :" & T.FullName)
  End Sub
End Class
```

Атрибут Guid, указанный перед классом, определяет UUID объекта. Значения UUID интерфейса и Dispid задаются внутри сборки.

Для получения величин, используемых в качестве значений атрибута Guid, можно воспользоваться утилитой uuidgen.exe из каталога утилит Visual Studio[1].

А теперь попробуйте выполнить перечисленные действия.

---

[1] По умолчанию используется каталог <диск>:\Program Files\Microsoft Visual Studio.NET\Common7\Tools.

1. Постройте программу VB6NetTest из каталога проекта CalledViaCOM2 и убедитесь в том, что она работает.
2. Закройте VB6.
3. Постройте сборку CalledViaCom2 заново.
4. Попробуйте запустить исполняемый файл VB6NetTest.

На этот раз программа нормально работает.

Описанный подход фактически обеспечивает совместимость сборки на двоичном уровне.

Впрочем, в этом решении кроется один подвох.

ВНИМАНИЕ

Обеспечивая двоичную совместимость, проследите за тем, чтобы все было сделано правильно. В отличие от VB6 VB .NET не обеспечивает двоичной совместимости и не предупреждает о ее нарушении!

Если изменение, вносимое в интерфейс, приводит к нарушению двоичной совместимости, последствия будут очень серьезными, вплоть до ошибок защиты памяти во всех приложениях, использующих компонент.

Дело даже не в дефектах или ограниченности VB .NET, а в самой природе COM. VB .NET предоставляет в ваше распоряжение средства, позволяющие обеспечить двоичную совместимость так, как программисты C++ делали в течение многих лет, однако программистам VB эта область незнакома. Как говорилось выше, все эти трудности возникают лишь при создании компонентов, предназначенных для использования в COM.

Впрочем, проблемы раннего связывания можно решить и другим, более простым способом, а именно — ограничиться поздним связыванием. Это делается так.

- Не включайте строку `<Assembly: ClassInterface(ClassInterfaceType.None)>` в файл AssemblyInfo.vb.
- Не создавайте отдельный интерфейс для вашего класса.
- Не устанавливайте атрибуты GUID и Dispid.

В этом случае Visual Studio .NET экспортирует библиотеку типов, которая всегда использует позднее связывание. Впрочем, при этом необходимо следить за тем, чтобы параметры методов не изменялись между версиями, иначе вы столкнетесь с ошибками стадии выполнения.

Учитывая, что большинство программистов VB не будет создавать компоненты .NET для использования в COM, такая стратегия вполне приемлема.

Дополнительные обстоятельства

При обращении к объектам .NET в COM следует учитывать ряд дополнительных обстоятельств.

- При обращении к объектам из COM используются только конструкторы по умолчанию. Параметризованные конструкторы игнорируются.
- Общие члены классов в COM недоступны.

- Только первые две части номера версии сборки преобразуются в номер версии библиотеки типов.
- Идентификатор программы (ProgID) объекта .NET в COM состоит из названия пространства имен, объединенного с названием объекта. Значение может быть изменено при помощи атрибута `ProgID`.
- Параметры и возвращаемые значения, определенные с типом `As Object`, преобразуются в тип COM `Variant`. Использовать их не рекомендуется.
- Объект, освобожденный объектом COM, уничтожается сборщиком мусора по стандартным правилам, как и все остальные объекты .NET.
- Проблемы передачи структур, которые не рассматривались при описании использования компонентов COM в .NET, действуют и в этом направлении. Данная тема рассматривается в разделе «Использование функций Win32 API».

Программистам, занимающимся решением нетривиальных или нестандартных задач в COM, следует изучить документацию .NET Framework. Разработчики транзакционных компонентов найдут в ней дополнительные сведения о том, как эти приемы следует применять к компонентам, используемым в COM+. Впрочем, для большинства программистов Visual Basic приведенной информации будет вполне достаточно.

Использование функций Win32 API

Возвращаясь к истории Visual Basic, мы видим, что в эпоху Visual Basic 1 при написании сколько-нибудь профессионального приложения практически неизбежно приходилось пользоваться Win32 API. По мере развития VB программисты продолжали часто обращаться к функциям Win32 API. Одни при помощи функций API получали доступ к средствам операционной системы, недоступным на уровне VB. Другие использовали API в целях оптимизации, например при выполнении сложных графических операций. Даже по мере того, как компания Microsoft расширяла функциональные возможности компонентов и оболочек (wrappers) COM, функции Win32 API занимали ключевое место в инструментарии любого серьезного программиста VB.

Эти времена прошли.

Поймите меня правильно: некоторые возможности не поддерживаются классами .NET Framework, поэтому вы должны уметь вызывать функции API. Тем не менее пользоваться ими следует как можно реже.

В этой главе изложена большая часть того, что необходимо знать о вызове функций Win32 API в VB .NET. Существует несколько причин, по которым рекомендуется избегать вызова функций Win32 API в VB .NET.

- Для вызова функций Win32 API программа должна иметь право выполнения неуправляемого кода (безопасность в .NET рассматривается в следующей главе). Пока достаточно сказать, что при вызове функции Win32 API сборке предоставляется максимально возможный уровень привилегий. Это затрудняет распространение сборки среди получателей, не пользующихся абсолютным доверием.

- Вызовы функций Win32 API в меньшей степени защищены от ошибок. Поскольку они выполняются в неуправляемом коде, вы можете столкнуться с ошибками защиты памяти, утечкой памяти и ресурсов и всех типичных ошибок, хорошо знакомых программистам, работающие в API на любых языках.
- Функциями Win32 API труднее пользоваться, чем эквивалентными средствами .NET Framework.

А если вам все-таки приходится использовать функции Win32 API, инкапсулируйте их в классах вместо непосредственного вызова в программе. Также рассмотрите возможность выделения всех классов, использующих неуправляемый код, в отдельные сборки. Если вы твердо убеждены в том, что злонамеренное использование ваших объектов невозможно, сообщите .NET об их безопасности при помощи метода `Assert`. Проблем с распространением это не решит (поскольку сам вызов `Assert` требует высокой степени доверия), но после установки сборка сможет безопасно использоваться менее надежным кодом. Я понимаю, что все это выглядит крайне запутанно, поскольку вопросы безопасности в .NET еще не рассматривались, но не огорчайтесь — ситуация прояснится в следующей главе.

Эволюция команды Declare

При любых взаимодействиях с неуправляемым кодом объект .NET должен знать:

- как найти файл (DLL или EXE), содержащий программный код компонента, и создать экземпляр компонента в случае необходимости;
- как найти нужный код (имя метода или точку входа) в компоненте;
- как организовать передачу параметров в неуправляемый компонент.

При работе с объектами COM основная часть информации, необходимой для решения этих трех задач, сосредоточена в библиотеке типов. При вызове функций API или DLL все необходимые сведения должны определяться в команде `Declare`.

Вы должны хорошо понимать, чем компонент DLL, предоставляющий свою функциональность через COM, отличается от обычных компонентов с традиционным экспортированием функций. Доступ к функциональности DLL может быть организован тремя способами.

1. Реализация в виде сборки .NET. Вы загружаете сборку и работаете с ее объектами и методами при помощи .NET CLR.
2. Реализация в виде объектов COM. Вы требуете у DLL создать объект заданного типа, после чего вызываете методы этого объекта.
3. Экспортирование функций (исходный способ предоставления доступа к функциональным возможностям DLL). Win32 API состоит из тысяч экспортируемых функций, реализованных базовыми DLL из состава Windows.

Некоторые программисты VB не видят принципиальных различий между этими подходами. Сборки Visual Basic .NET и C# не могут экспортировать функции в традиционном смысле этого слова, они могут лишь предоставлять доступ к объектам и их методам через .NET CLR. В Visual Basic 6 ActiveX DLL не могут

экспортировать функции без помощи компонентов независимых фирм (наподобие Desaware SpyWorks): на уровне языка такая возможность не поддерживается.

Тем не менее Visual Basic .NET, C# с Visual Basic 6 могут работать с функциями, экспортируемыми DLL. Когда речь заходит о вызове функций Win32 API, мы по-прежнему говорим о вызове экспортируемых функций, реализованных в DLL операционной системы.

В соответствии с документацией .NET синтаксис команды `Declare` выглядит так:

```
[Public | Private | Protected | Friend | Protected Friend] Declare [Ansi | _
Unicode | Auto] [Sub] имя Lib "имя_библиотеки" [Alias "псевдоним"] _
[([аргументы])]
```

Или:

```
[Public | Private | Protected | Friend | Protected Friend] Declare [Ansi | _
Unicode | Auto] [Function] имя Lib "имя_библиотеки" [Alias "псевдоним"] _
[([аргументы])] [As тип]
```

Первая часть объявления (`[Public | Private | Protected | Friend | Protected Friend]`) просто описывает область видимости, в которой может использоваться функция.

Никогда не используйте объявления с ключевым словом `Public`: невозможно найти разумные причины для того, чтобы внешний код вызывал функции DLL через вашу сборку.

В объявлениях всегда указывается наименьшая область видимости из всех возможных. Если объявление должно использоваться во всей сборке, используйте атрибут `Friend`. Тем не менее оптимальное решение заключается в инкапсуляции всех вызовов API в классах, что не только уменьшает вероятность возникновения ошибок, связанных с вызовом функций API, но и позволяет лучше контролировать параметры безопасности сборки (см. главу 16).

Секция `[Ansi | Unicode | Auto]` передает CLR информацию о том, как должны обрабатываться строки.

Windows NT/2000/CP основаны на технологии NT, которая (как и VB6 и VB .NET) использует во внутреннем представлении строковых данных кодировку Unicode. Однако Windows 95/98/ME основаны на устаревшей технологии[1] и используют внутреннюю кодировку ANSI. Чтобы в системах на базе Unicode могли использоваться как ANSI-, так и Unicode-программы, в этих системах экспортируются две версии большинства функций API, использующих строки. Например, функция API `GetWindowText` в действительности экспортируется дважды с именами `GetWindowTextA` (ANSI) и `GetWindowTextW` (Unicode).

В Visual Basic 6 всегда используются точки входа ANSI, чтобы создаваемые приложения были совместимы с операционными системами на базе ANSI и Unicode. Обычно в поле `Alias` (см. ниже) задается точка входа с суффиксом `A`. В результате в Unicode-системах происходит нечто странное: при вызове функции API VB6 сначала преобразует все строковые параметры в ANSI, затем вызывает ANSI-функцию DLL операционной системы, а затем преобразует строки в Unicode для нормальной работы операционной системы!

---

[1] Хотите — верьте, хотите — нет, но эти системы все еще содержат 16-разрядный код.

Visual Basic .NET позволяет обойтись без многократных преобразований и выбрать нужную точку входа. Если вы хотите использовать точку входа ANSI, укажите ключевое слово `Ansi` и имя точки входа ANSI в секции `Alias`. Чтобы использовать точку входа Unicode, укажите ключевое слово `Unicode` и имя соответствующей точки входа в секции `Alias`. Если функция не использует строки, эти три параметра указывать не обязательно — достаточно указать точное имя точки входа в секции имени или псевдонима команды `Declare`.

Но в большинстве случаев указывается параметр `Auto`. В этом случае CLR автоматически выполняет ряд действий.

- CLR проверяет, существует ли заданная точка входа.
- Если точка входа не найдена, CLR присоединяет к имени суффикс, соответствующий текущей операционной системе (`W` в Unicode-системах, `A` в ANSI-системах), и ищет полученное имя.
- После обнаружения точки входа все строки автоматически преобразуются в соответствии с ее типом.

Параметр `Auto` нормально работает для большинства функций API. В некоторых функциях (например, в функциях OLE) кодировка выбирается независимо от операционной системы, поэтому функция имеет всего одну точку входа. В этом случае кодировка выбирается в соответствии с документацией.

Псевдоним (`Alias`) задается в тех случаях, когда в приложении используется имя, отличное от имени точки входа, например, если имя экспортируемой функции API совпадает с ключевым словом языка Visual Basic.

Список аргументов рассматривается ниже.

По сравнению с Visual Basic 6 в команду `Declare` был внесен ряд усовершенствований.

- Visual Basic .NET требует объявлять тип возвращаемого значения, благодаря чему ликвидируется одна из самых распространенных ошибок VB6, когда программист забывает указать тип возвращаемого значения, что приводит к ошибке «Bad DLL Calling Convention», так как функция API возвращает 32-разрядное число, а VB6 рассчитывает получить `Variant`.
- В абсолютном большинстве функций API используется традиционная схема передачи параметров (конвенция PASCAL). По умолчанию она применяется и в функциях, объявленных командой `Declare`. Тем не менее при помощи атрибута `Calling-Convention` в команде `Declare` можно выбрать схему передачи параметров C (Cdecl). Обычно это приходится делать при работе DLL независимых фирм, когда разработчики случайно оставили схему передачи параметров C.
- Visual Basic .NET в текущей бета-версии распознает ошибки передачи параметров DLL менее эффективно, чем Visual Basic 6.
- Visual Basic .NET не поддерживает параметры `As Any`.

Три главных правила, о которых необходимо помнить при вызове функций API из VB .NET

Даже если вы совсем ничего не поймете в этом разделе, запомните три главных правила.

Даже в VB .NET вызовы функций API могут быть опасными

Ошибка при объявлении функции может привести к исключениям защиты памяти.

Помните о ключевом слове ByVal

Вызовы функций API нетерпимы к ошибкам (в частности, к отсутствию ключевого слова `ByVal` там, где оно необходимо, или к указанию `ByVal` при передаче переменной по ссылке).

Вместо Long следует использовать тип Integer

В VB .NET тип данных `Long` является 64-разрядным. В приложениях VB .NET использование типа `Long` вместо `Integer` часто приводит к снижению быстродействия, а в объявлениях API — вызывает ошибки и даже исключения защиты памяти.

Подсистема P-Invoke

Как ни странно, многие программисты VB6 громко жалуются на то, что разработчики Microsoft убрали из VB .NET «скрытые» операторы `VarPtr`, `StrPtr` и `ObjPtr`. Но они не понимают, что эти операторы стали ненужными. Их функциональные возможности не исчезли, а просто переместились в .NET Framework. В сущности, весь процесс вызова функций Win32 API вообще не является частью языка VB .NET — команда `Declare` всего лишь инкапсулирует подсистему .NET Framework, которая называется P-Invoke (сокращение от Platform Invocation).

Управление работой подсистемы P-Invoke осуществляется при помощи методов пространства имен `System.Runtime.InteropServices`. Из-за этого нередко возникает путаница, поскольку не всегда понятно, какие объекты и методы этого пространства имен относятся к COM Interop, какие — к P-Invoke, а какие применимы в обеих областях. Ситуация усложняется тем, что атрибуты, управляющие передачей параметров при вызове функций Win32 API, также управляют передачей параметров при вызове методов COM!

Поэтому передача параметров не рассматривалась в части этой главы, посвященной COM, — они ничем не отличаются от параметров, передаваемых при вызове функций Win32 API. А поскольку COM Interop весьма разумно действует по умолчанию, приведенный ниже материал с большей вероятностью пригодится при вызове функций Win32 API, нежели в COM Interop.

Становится понятно, почему мы ничего не теряем с исчезновением `VarPtr`, `StrPtr` и `ObjPtr`. Все «хакерские» приемы, основанные на использовании этих функций, значительно проще и надежнее реализуются средствами P-Invoke. Подумайте: все объекты .NET, использующие возможности операционной системы, от форм Windows до Winsock, должны использовать P-Invoke. Неудивительно,

что P-Invoke не только справляется со всеми задачами, связанными с API, но и весьма стабильно работает.

Конечно, из этого вовсе не следует, что в P-Invoke легко разобраться.

Атрибуты передачи параметров

Существует ряд атрибутов, которые могут указываться перед параметрами (и возвращаемым значением) в объявлениях функций API. Для правильного вызова сложных функций API необходимо хорошо понимать смысл этих атрибутов.

- `MarshalAs`. Атрибут указывает, как CLR следует передавать данные (например, должна ли строка передаваться в виде указателя на строку, завершенную нуль-символом, или в строковом формате OLE `BSTR`).
- `UnmanagedType`. Используется в сочетании с атрибутом `MarshalAs` для описания передачи параметров.
- `Marshal`. Объект содержит большое количество общих методов для выполнения исключительно разнообразных задач, связанных с управлением памятью. Например, вы можете выделить блок неуправляемый памяти и вручную передавать данные в память с использованием атрибутов передачи параметров.
- `GCHandle`. Атрибут позволяет зафиксировать местонахождение объекта в управляемой памяти и получить указатель, по которому можно обращаться из неуправляемой памяти. На первый взгляд такая возможность выглядит весьма заманчиво, но на практике почти не используется.

Осваиваем P-Invoke

В процессе работы над этой главой я столкнулся с двумя проблемами.

- Поскольку в Win32 API входит более 9000 функций, я не смогу ни предвидеть все нюансы каждой функции, ни изложить этот материал в ограниченном пространстве.
- Мне неизвестна ваша квалификация. Если бы я написал эту главу для новичков в области Win32 API, только на описание базовых концепций потребовалось бы не менее сотни страниц.

Итак, я буду исходить из двух предположений. Во-первых, у вас уже имеется некоторый опыт вызова функций API в Visual Basic — достаточный, чтобы я мог сказать «как в VB6» и быть уверенным в том, что в дальнейшем вы разберетесь. Во-вторых, вы готовы читать документацию и заниматься самостоятельными исследованиями[1].

Руководствуясь этими предположениями, я сначала покажу, как узнать точные значения параметров, переданные при вызове функции API. Я сам пользовался этим приемом, выясняя, как различные атрибуты влияют на передачу параметров.

Проект VBInterface

Откройте решение из каталога `VBInterface` (не обращайте внимания на файлы `.cpp`). В результате открываются два проекта: VBInterface и VBInterfaceTest.

[1] А если не готовы, я могу провести консультации в частном порядке, только это обойдется недешево.

Программа VB .NET VBInterfaceTest демонстрирует вызов функций DLL с разными параметрами. Программа VBInterface представляет собой библиотеку DLL, экспортирующую функции. Библиотека написана на C++ и состоит из неуправляемого кода.

Оба проекта загружаются одновременно; проект VBInterfaceTest выбирается в качестве стартового. Для совместной отладки проектов остается лишь вызвать диалоговое окно свойств проекта VBInterfaceTest, выбрать команду Configuration Properties ▸ Debugging на левой панели и убедиться в том, что флажок Unmanaged Code Debugging установлен.

Начнем с простого примера.

Функция `ReceivesShortDLL` определяется в файле VBInterface.cpp следующим образом:

```
STDAPI_(short) ReceivesShort(short x)
{
  _itoa(x, tbuf, 10); /* Разместить во временном буфере */
  MessageBox(GetFocus(),(LPSTR)tbuf,(LPSTR)"ReceivesShort", MB_OK);
  return(x);
}
```

Функция выводит окно сообщения с полученной 16-разрядной величиной и возвращает это же значение. Объявление в файле VBInterface выглядит так:

```
Public Declare Auto Function ReceivesShort Lib _
"..\..\VBInterface\Debug\VBInterface.dll" (ByVal s As Short) As Short
```

Пример вызова в функции `Numbers` программы VBInterfaceTest:

```
i = ReceivesShort(4);
Console.WriteLine(i)
```

При выполнении этого кода число 4 будет выведено в окне сообщения и на консоли.

Попробуйте установить точку прерывания в одной из этих команд, а затем продолжить выполнение в пошаговом режиме. Вы увидите, что управление передается прямо в программу C++. Находясь в программе C++, вы сможете просмотреть значения полученных параметров.

Итак, вы познакомились с первым приемом: *если у вас возникли трудности с правильным объявлением и передачей параметров при вызове функций API, создайте в файле* VBInterface.cpp *функцию, объявление которой точно соответствует объявлению вызываемой функции API на языке C, и попробуйте вызвать эту функцию из программы VB.* Это позволит вам узнать точные значения параметров, полученные функцией API.

Секреты передачи параметров

На простом уровне передача большинства типов данных обходится без проблем. Если вы знакомы с вызовом функций API в VB6, единственное, о чем следует помнить (не считая особого подхода к структурам, о котором будет сказано ниже), — это замена типа `Long` типом `Integer`. В табл. 15.1 перечислены основные типы параметров Win32 API (согласно документации WIn32 API) и эквивалентные определения в команде `Declare` VB .NET.

Таблица 15.1. Передача простых параметров функциям API

Тип параметра	Объявление
BYTE	ByVal As Byte
CHAR, Char	ByVal As Char
SHORT, USHORT, WORD	ByVal As Short
int, INT, long, DWORD, ULONG и т. д.	ByVal As Integer
LPBYTE	ByVal As Byte
LPCHAR	ByVal As Char
LPSTR	ByVal As String
LPSHORT, LPWORD	ByVal As Short
LPDWORD, LPLONG	ByVal As Integer
LPxxxPROC (указатель на функцию)	ByVal As Delegate (тип делегата) См. проект Delegates в главе 10

Внимательнее с типом Long!

Как я уже говорил, самой распространенной ошибкой программистов VB .NET при работе с Win32 API будет использование 64-разрядного типа `Long` для 32-разрядных параметров. Проект VBInterface показывает, что происходит при ошибочной передаче данных типа `Long`.

Файл VBInterface.cpp:

```
STDAPI_(__int64) ReceivesLong(__int64 y)
{
  DumpValues((LPVOID)&y, 8);
  return(y+0x1000200030004000);
}
```

Модуль Module1.vb проекта VBInterfaceTest:

```
Public Declare Auto Function ReceivesLong Lib _
"..\..\VBInterface\Debug\VBInterface.dll" (ByVal a As Long) As Long
  l = ReceivesLong(4);
  Console.WriteLine(Hex$(1))
```

Результат в окне сообщения: 00 40 00 30 00 20 00 10.

Возвращаемое значение: 2000400060008000.

Данные в окне сообщения выглядят несколько странно. Вывод шестнадцатеричного дампа памяти осуществляется функцией `DumpValues`. В PC память организована таким образом, что младшая часть числового значения расположена по младшему адресу памяти. Таким образом, байтовая последовательность 00 40 соответствует числу &H4000.

Так или иначе, данный пример еще раз подчеркивает необходимость использования типа `Integer` вместо `Long` в объявлениях API. Конечно, при передаче типа .NET `Long` (64-разрядного) функции Win32 API, рассчитывающей на получение типа `long` C++ (32-разрядный), может произойти серьезная ошибка.

Передача строк

При вызове функций Win32 API, получающих строковые параметры, строки практически всегда объявляются в виде `ByVal As String`.

Вспомните, что говорилось в главе 9 о возможности модификации объектов, переданных по значению, и о неизменности строк. VB .NET идет на небольшое жульничество. Хотя в объявлении указано ключевое слово `ByVal`, VB .NET присваивает переменной типа `String`, переданной в качестве параметра, новое значение, совпадающее со значением строки после возврата из функции.

Иначе говоря, перед нами один из тех случаев, когда синтаксис VB .NET совместим с синтаксисом VB6. Мне это кажется довольно странным, поскольку в VB6 этот синтаксис выглядел нелогично. Странно, что этот дефект не был исправлен в процессе чистки языка. Впрочем, на передачу строковых параметров все равно можно повлиять при помощи атрибутов.

Начнем с рассмотрения функций DLL, не изменяющих строк.

Функции `ReceivesANSIString` и `ReceivesUnicodeString` предназначены для получения и вывода строк в кодировках ANSI и Unicode. Функция `ReceivesAutoString` выводит дамп полученного буфера и наглядно показывает, какой формат используется в конкретной операционной системе.

```
/* Используется для большинства функций API. VBcalls.
  VB передает строку, завершенную нуль-символом.
*/
STDAPI_(VOID) ReceivesANSIString(LPSTR tptr)
{
    MessageBox(GetFocus(), (LPSTR)tptr, (LPSTR)"ReceivesANSIString", MB_OK);
}
STDAPI_(VOID) ReceivesUnicodeString(LPWSTR tptr)
{
    MessageBoxW(GetFocus(), tptr, L"ReceivesUnicodeString", MB_OK);
}

STDAPI_(VOID) ReceivesAutoString(LPSTR tptr, int count)
{
    DumpValues(tptr, count);
}
```

В листинге 15.8 приведены объявления и примеры вызовов этих функций в проекте VBInterfaceTest.

Листинг 15.8. Примеры использования строковых функций в проекте VBInterfaceTest

```
Public Declare Ansi Sub ReceivesANSIString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" (ByVal s As String)
Public Declare Unicode Sub ReceivesUnicodeString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" (ByVal s As String)
Public Declare Auto Sub ReceivesAutoString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" (ByVal s As String, _
ByVal chars As Integer)
Public Declare Sub ReceivesNoInfoString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" _
Alias "ReceivesAutoString" (ByVal s As String, ByVal chars As Integer)

  Dim s As String = "Test string"
  Dim s2 As String
  Console.WriteLine ("Strings examples")
  ReceivesANSIString (s)
  ReceivesUnicodeString (s)
  s2 = s
  ReceivesAutoString(s, Len(s))
```

```
If Not s2 Is s Then
  Console.WriteLine _
  ("s and s2 are no longer the same after after ReceivesAutoString")
End If
ReceivesNoInfoString(s, Len(s))
```

Проанализируем полученный результат.

Функция `ReceivesAnsiString` правильно получает и выводит строку «Test String». То же самое можно сказать и о функции `ReceivesUnicodeString`. Это доказывает, что атрибуты `Ansi` и `Unicode` обеспечивают передачу строковых параметров в заданном формате.

Функция `ReceivesAutoString` выводит последовательность 54 00 65 00 73 00 74 00 20 00 73. Несомненно, перед нами строка Unicode (поскольку программа выполнялась в Windows 2000, а эта система использует кодировку Unicode). Последовательность усечена, поскольку количество байт вдвое больше количества символов в строке, но и приведенный фрагмент однозначно показывает, что перед нами строка Unicode.

Любопытная подробность: хотя строка передавалась с ключевым словом `ByVal`, на экране появляется сообщение о том, что строка изменилась. В данном случае поведение VB .NET не соответствует общепринятой трактовке атрибута `ByVal`.

При отсутствии заданной кодировки выводится последовательность 54 65 73 74 20 73 74 72 69 6e 67. Это наглядно доказывает, что по умолчанию в VB .NET используется кодировка ANSI (и это вполне логично для сохранения совместимости с VB6). Тем не менее атрибут кодировки всегда следует задавать явно.

В следующем примере (листинг 15.9) функция модифицирует переданные строки. Приведенный фрагмент взят из файла VBInterface.cpp.

Листинг 15.9. Примеры использования строковых функций в проекте VBInterfaceTest (продолжение)

```
/* Модификация строки (при условии, что программа не выходит
 за границы выделенного буфера */

STDAPI_(VOID) ChangesStringA(LPSTR tptr)
{
  if (*tptr) *tptr = 'A';
}

STDAPI_(VOID) ChangesStringW(LPWSTR tptr)
{
  if(*tptr) *tptr = 'W';
}

STDAPI_(VOID) ChangesByRefStringW(LPWSTR *ptr)
{
  lstrcpyW(*ptr, L"New String");
}

STDAPI_(VOID) ChangesByRefStringA(LPSTR *ptr)
{
  lstrcpyA(*ptr, "New String");
}
```

продолжение

Листинг 15.9 (продолжение)

```
STDAPI_(VOID) ChangesBSTRString(BSTR *sptr)
{
  if(!sptr) return; // Условие никогда не выполняется (перестраховка)
  if(*sptr) SysFreeString(*sptr);
  *sptr = SysAllocString(L"Any Length Ok");
}


/* Возвращение строки при вызове функции DLL. */
STDAPI_(BSTR) ReturnsVBString()
{
  return(SysAllocString(L"Here's a return string"));
}
```

В проекте VBInterfaceTest используется следующий код:

```
Public Declare Auto Sub ChangesString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" (ByVal s As String)
Public Declare Auto Sub ChangesByRefString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" (ByRef s As String)
Public Declare Unicode Sub ChangesBSTRString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" _
(<MarshalAs(UnmanagedType.BStr)> ByRef s As String)
Public Declare Unicode Function ReturnsVBString Lib _
"..\..\VBInterface\Debug\VBInterface.dll" () As
<MarshalAs(UnmanagedType.BStr)> String

  s2 = s
  ChangesString (s)
  If Not s2 Is s Then
    Console.WriteLine ("s and s2 are no longer the same after ChangesString")
  End If
  Console.WriteLine ("Changed String: " & s)
  ChangesByRefString (s)
  Console.WriteLine ("Changed ByRef String: " & s)
  ChangesBSTRString (s)
  Console.WriteLine ("Changed BSTR String: " & s)
  s = ReturnsVBString()
  Console.WriteLine ("Returned String: " & s)
```

Результат выглядит следующим образом:

```
s and s2 are no longer the same after ChangesString
ChangedString: West string
Changed ByRef String: New String
Changed BSTR String: Any Length Ok
Returned String: Here's a return string
```

Функция `ChangesString` изменяет один символ в строке, переданной по значению. Как и в VB6, необходимо действовать очень осторожно, чтобы не изменить данные за концом строки. Если длина строки, заданная при инициализации, недостаточна для хранения возвращаемых данных, вызов почти наверняка приведет к исключению защиты памяти.

У функции `ChangesString` есть одна любопытная особенность: она наглядно показывает, как работает атрибут `Auto` при объявлении функции. В объявлении `ChangesString` не указан псевдоним `ChangesStringA` или `ChangesStringW`, но результат доказывает, что в DLL была найдена точка входа `ChangesStringW`

(предполагается, что программа работает в операционной системе, использующей кодировку Unicode).

При передаче строки с ключевым словом ByRef функция получает указатель на переменную, содержащую указатель на строковый буфер. *Не пытайтесь* присваивать значение этой переменной, поскольку это лишь приведет к сбою в программе. Правило о предварительном выделении буфера достаточного размера действует и в этом случае. Данный тип параметра (LPSTR) используется лишь небольшим подмножеством функций Win32 API. В таких случаях можно объявить параметр ByRef As Integer, а затем при помощи объекта Marshal получить строку по указателю, полученному при вызове функции API[1]. Кстати, в этом отношении VB .NET отличается от VB6, где для параметров ByRef As String передается указатель на BSTR (строковый тип OLE).

Тип BSTR может передаваться при вызове функций API, хотя в общих функциях Win32 API он не используется (только в функциях подсистемы OLE). Эта возможность продемонстрирована в функциях ChangesBStrString и ReturnsVBString. Обратите внимание на атрибут MarshalAs(UnmanagedType.BSTR): он сообщает CLR о том, что переданные строки относятся к типу BSTR. Преимущество типа BSTR заключается в том, что он позволяет изменять длину строки или определить возвращаемую строку с произвольной длиной.

Хотя тип BSTR не применяется при вызовах функций Win32 API, он широко используется во внутренних операциях COM Interop.

Передача массивов

С передачей массивов нередко возникают трудности. В листинге 15.10 приведен фрагмент файла VBInterface.cpp. Функция ReceivesShortArray получает указатель на short (первое число в последовательности), а функция ReceivesShortRefArray получает указатель на переменную, содержащую адрес массива. Обе функции увеличивают значения первых двух элементов массива (показывая, что данные были скопированы в исходный массив). Функция ReturnsSafeArray показывает, как функция DLL может полностью переопределить переданный массив.

Листинг 15.10. Операции с массивами в программе C++

```
/* Целочисленный массив целых чисел - следите за тем,
  чтобы не нарушить границы массива!
  Обратите внимание на специальную схему передачи параметров
  в примере VB - для строк она не подходит.
*/

STDAPI_(VOID) ReceivesShortArray(short FAR *iptr)
{
  wsprintf((LPSTR)tbuf, (LPSTR)"1st 4 entries are %d %d %d %d",
    *(iptr), *(iptr+1), *(iptr+2), *(iptr+3));
  MessageBox(GetFocus(), (LPSTR)tbuf, (LPSTR)"ReceivesShortArray", MB_OK);
  (*iptr)++; // Увеличить элемент массива, чтобы проверить
             // передачу адресом и узнать, выполняется ли
             // операция с временной копией.
```

продолжение

[1] Методы Marshal.PtrToStringAnsi, Marshal.PtrToStringAuto и Marshal.PtrToStringUni позволяют получить объект .NET типа String по у азателю на блок неуправляемой памяти. Эти методы работают аналогично методу Marshal.PtrToStructure, встречающемуся ниже в этой главе.

Листинг 15.10 (продолжение)

```
  (*(iptr+1))++;
}

short newArrayBuffer[4] = { 5, 4, 3, 2};

STDAPI_(VOID) ReceivesShortRefArray(short FAR **piptr)
{
  short *iptr;
  iptr = *piptr;
  wsprintf((LPSTR)tbuf, (LPSTR)"1st 4 entries are %d %d %d %d",
    *(iptr), *(iptr+1), *(iptr+2), *(iptr+3));
  MessageBox(GetFocus(), (LPSTR)tbuf, (LPSTR)"ReceivesShortArray", MB_OK);
  (*iptr)++; // Увеличить элемент массива, чтобы проверить
       // передачу по ссылке и узнать, выполняется ли
       // операция с временной копией.
  (*(iptr+1))++;
  *piptr = newArrayBuffer;
}

STDAPI_(VOID) ReturnsSafeArray(SAFEARRAY **psa)
{
  short *pdata;
  pdata = (short *)((*psa)->pvData);
  wsprintf((LPSTR)tbuf, (LPSTR)"Array of %d dimensions, %ld bytes per
element\n First int entry is %d", (*psa)->cDims, (*psa)->cbElements, *pdata);

    MessageBox(GetFocus(), (LPSTR)tbuf, (LPSTR)"ReturnsArray", MB_OK);
    SAFEARRAYBOUND bounds[1];
    long l;
    bounds[0].lLbound = 0;
    bounds[0].cElements = 4;
    *psa = SafeArrayCreate(VT_I2, 1, bounds);
    short storeval;
    for(l = 0; l<4; l++) {
  storeval = (short)l * 5;
  SafeArrayPutElement(*psa, &l, &storeval);
  }
}
```

Начнем с `ReceivesShortArray`. Эта функция ожидает получить указатель на массив 16-разрядных чисел типа `Short`. При вызове используются два разных варианта объявления. В одном случае передается первый элемент массива, а во втором — весь массив по значению (листинг 15.11).

Листинг 15.11. Передача массивов в проекте VBInterfaceTest

```
Public Declare Sub ReceivesShortArray1 Lib _
  "..\..\VBInterface\Debug\VBInterface.dll" Alias _
  "ReceivesShortArray" (ByRef i As Short)
Public Declare Sub ReceivesShortArray2 Lib _
  "..\..\VBInterface\Debug\VBInterface.dll" Alias _
  "ReceivesShortArray" (ByVal i() As Short)
  Dim i() As Short = {1, 2, 3, 4}
  Dim x As Integer

  ReceivesShortArray1 (i(0))
  For x = 0 To 3
    Console.Write (Str(i(x)) & ", ")
```

```
Next x
console.WriteLine()

ReceivesShortArray2 (i)
For x = 0 To 3
  Console.Write (Str(i(x)) & ", ")
Next x
console.WriteLine()
```

При первом вызове `ReceivesShortArray` в окне сообщения выводится последовательность «1, 2, 3, 4». Следовательно, хотя при вызове указывается только первый элемент массива, VB .NET фактически передает в неуправляемую память весь массив. Вероятно, эта возможность была добавлена в VB .NET для поддержки способа, часто используемого программистами VB6 при передаче массивов функциям API. При возвращении из функции выводится последовательность «2, 3, 3, 4», из чего можно сделать вывод, что измененные элементы массива успешно возвращены в массив .NET.

То же самое происходит и при передаче по значению, из чего можно сделать вывод об идентичности этих вызовов.

Функция `ReceivesShortRefArray` правильно получает массив (с элементами 3, 4, 3, 4), но по возвращении оказывается, что верхняя граница массива равна 0! Что произошло? Interop не может сделать обоснованных предположений относительно длины модифицированного массива, поэтому она копирует только первый элемент! Впрочем, как показано ниже, в программе можно определить массив фиксированной длины, чтобы система знала размер массива при возвращении из функции.

```
Public Declare Sub ReceivesShortRefArray Lib _
  "..\..\VBInterface\Debug\VBInterface.dll" (ByRef i() As Short)

ReceivesShortRefArray (i)
Console.WriteLine ("Array bound is now: " & UBound(i))
console.WriteLine()
```

Функция `ReturnsSafeArray` демонстрирует передачу массива по ссылке в виде типа OLE `SAFEARRAY`.

```
Public Declare Sub ReturnsSafeArray Lib _
"..\..\VBInterface\Debug\VBInterface.dll" _
(<MarshalAs(UnmanagedType.SafeArray)> ByRef i() As Short)

  ReturnsSafeArray (i)
  For x = 0 To UBound(i)
    Console.Write (Str(i(x)) & ", ")
  Next
  console.WriteLine()
```

Тип `SAFEARRAY`, как и упоминавшийся выше тип `BSTR`, не применяется при вызове функций Win32 API (возможно, кроме функций OLE API), но широко используется во внутренней работе COM Interop.

Структуры

Передача параметров-структур и модификация их содержимого при вызове функций Win32 API — очень распространенное явление. Следовательно, очень

важно не только правильно организовать передачу структур в неуправляемую память, но и возврат полученных данных.

Тем не менее задача усложняется рядом обстоятельств.

- Структуры .NET не похожи на те структуры, к которым вы привыкли. Например, поля в них могут храниться в памяти в произвольном порядке.
- В .NET не поддерживаются строки фиксированной длины — неотъемлемая часть многих структур Win32 API.
- В .NET не поддерживаются массивы фиксированной длины, также задействованные во многих структурах Win32 API.

Таким образом, при передаче структур необходимо передать информацию о расположении полей в памяти и о том, как должны передаваться поля строк и массивов.

Расположение полей структуры в памяти

Прежде всего, следует запомнить, что все функции API рассчитывают на определенный порядок следования полей структуры в памяти. Впрочем, это и так понятно. Тем не менее функции API не всегда подчиняются одним и тем же правилам выравнивания. В большинстве функций используется упаковка по границам байтов, то есть все поля следуют непосредственно друг за другом без заполнителей. В VB6 структуры передаются с естественным выравниванием, то есть каждое поле выравнивается по границе, кратной его размеру. Байтовые поля могут находиться где угодно, 16-разрядные поля выравниваются по границам четных байтов, а 32-разрядные поля выравниваются по границе 4 байт. Рассмотрим следующую структуру C++:

```
typedef struct usertypestruct {
  BYTE a;      // Соответствует типу VB Byte
  short b;
  long c;
  BYTE d[4];
  char e[16];
} usertype;
```

При естественном выравнивании ANSI-версия этой структуры представляет собой следующую последовательность байтов (буквы обозначают байты соответствующих полей, а нули — заполнители, вставленные компилятором):

```
a 0 b b c c c c d d d d e e e e e e e e e e e e e e e e = 28 байт
```

При выравнивании по границе байтов массив выглядит так:

```
a b b c c c c d d d d e e e e e e e e e e e e e e e e = 27 байт
```

В проекте VBInterface компилятор настроен на выравнивание по границе байтов, поэтому в данном случае VB6 не сможет правильно передать структуру функции DLL![1] К счастью, в большинстве функции API поддерживается явное включение заполнителей. Даже несмотря на то, что такие функции используют упаковку по границе байтов, они правильно работают с VB6.

[1] В таких случаях помогает компонент для упаковки/распаковки пользовательских типов, входящий в пакет Desaware SpyWorks.

В VB .NET тип упаковки полей определяется атрибутом `StructLayout` (листинг 15.12).

Листинг 15.12. Атрибут StructLayout

```
<StructLayout(LayoutKind.Sequential, Pack:=1)> Public Structure GoodStruct
  Public A As Byte
  Public B As Short
  Public C As Integer
  <MarshalAs(UnmanagedType.ByValArray, SizeConst:=4)> Public D() As Byte
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=16)> Public E As String

  Public Sub InitStruct()
    A = 1
    B = 2
    C = 3
    ReDim D(3)
    D(0) = 4
    D(1) = 5
    D(2) = 6
    D(3) = 7
    E = "16 char string "   ' С нуль-символом
  End Sub
End Structure
```

Практически всегда используется последовательное расположение полей в структурах (`LayoutKind.Sequential`) и упаковка по границе байтов. Более того, атрибут `StructLayout` позволяет задать расположение полей в памяти с указанием точного смещения каждого поля от начала структуры. Проблемы с выравниванием уходят в прошлое!

Байтовые массивы и строки в .NET определяются в виде обычных массивов и объектов `String`. Массив должен инициализироваться с правильной длиной в методе `Structure` (для структур конструктор по умолчанию не переопределяется, поэтому не забудьте вызвать этот метод перед использованием структуры). Массив передается как при передаче по ссылке (как было показано из приведенного выше примера с массивами, именно этот способ передачи массивов функциям API является правильным). Указание размера массива обеспечивает возможность передачи массива фиксированного размера в обоих направлениях.

Для строки указан тип передачи `ByValTStr` — тип `TSTR` относится к строке, тип которой зависит от используемой операционной системы. Другими словами, в Unicode-системах строка представляется 32 байтами в кодировке Unicode. Большинство строк в структурах, используемых функциями API, фактически являются строками `TSTR`, тип которых зависит от использованной точки входа. Применение типа `ByValTStr` в структурах фактически эквивалентно использованию атрибута `Auto` в команде `Declare`.

Определение функции `ReceivesUserType` для точки входа ANSI выглядит следующим образом:

```
/* Передача по ссылке */
STDAPI_(VOID) ReceivesUserType(usertype FAR *u)
{
    DumpValues(u, 30);
  wsprintf((LPSTR)tbuf,
    (LPSTR)"usertype contains %d %d %d (%d %d %d %d) %s",
```

```
    u->a, u->b, u->c, u->d[0], u->d[1], u->d[2], u->d[3], u->e);
    MessageBox(GetFocus(), (LPSTR)tbuf, (LPSTR)"ReceivesUserType1", MB_OK);
    if(u->c == 3) {
    lstrcpy(u->e, "New Data");
    u->d[0] = 99;
    }
}
```

При вызове функция выводит данные пользовательского типа в следующем виде:

```
01 02 00 03 00 00 00 04 05 06 07 20 36 20 63 68 61 72 20 73 74 72 69 6e 67 20
```

Перед нами структура в формате с упаковкой полей по границе байтов.

Функция DLL копирует текст «New Data» в строковое поле. Вывод строки после возвращения из функции показывает, что модифицированные данные были успешно переданы в структуру. Также передается и измененное содержимое массива, о чем свидетельствует значение 99 в консольном окне. •

Нетривиальные вызовы функций Win32 API

Если в начале этой главы вы хотя бы в общих чертах представляли, как функции API вызываются в VB6, то для большинства случаев изложенного материала вполне достаточно (если учесть, что из-за масштабов библиотеки классов .NET необходимость в вызове функций API возникает редко).

В завершение этой главы я хочу представить пару примеров нетривиальных вызовов функций Win32 API.

Получение информации о соединениях удаленного доступа при помощи функции API RasNumEntries

Функция API RasNumEntries заполняет буфер массивом структур `RASENTRYNAME`, содержащих информацию о соединениях удаленного доступа (создайте хотя бы пару соединений, прежде чем запускать эту программу).

ПРИМЕЧАНИЕ

> Для понимания примеров RasEntries и RasGetEntry, описанных в этом разделе, требуется хорошее знание общих концепций программирования (указатели, расположение данных в памяти и т. д.) и нетривиальных приемов использования API в VB6. Я привожу эти примеры для опытных читателей, хотя разобраться в них полностью смогут далеко не все.

Структура `RASENTRYNAME` определяется следующим образом[1]:

```
typedef struct _RASENTRYNAME {
  DWORD dwSize;
  TCHAR szEntryName[RAS_MaxEntryName + 1];
} RASENTRYNAME;
```

Значение `RAS_MaxEntryName` равно 256.

В листинге 15.13 приведено определение структуры `RASENTRYNAME` из проекта RasEntries.

[1] Для простоты из определения исключены новые поля, появившиеся в Windows 2000. Версия структуры определяется значением поля `dwSize`.

Листинг 15.13. Структура RASENTRYNAME

```
' Приложение RasEntries
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Imports System.Runtime.InteropServices
Module Module1
  ' 256 символов в szEntryName, буфер на 257 символов
  <StructLayout(LayoutKind.Sequential, Pack:=4, CharSet:=Charset.Auto)> _
  Structure RASENTRYNAME
    Public dwSize As Integer
    <MarshalAs(UnmanagedType.ByValTStr, sizeConst:=257)> Public _
    szEntryName As String
    Public Sub Init()
      dwSize = Marshal.SizeOf(Me)
    End Sub
  End Structure
```

Если заглянуть в заголовочный файл rasapi.h, вы найдете в нем строку `#include <pshpack4.h>` — признак того, что поля этой структуры выравниваются по границе 4 байт вместо 1.

В листинге 15.13 также продемонстрировано использование автоматического выбора кодировки для структур. Атрибут `Charset.Auto` аналогичен атрибуту `Auto` команды `Declare`, но относится к строкам внутри структур.

Поле `szEntryName` передается в виде строки фиксированной длины, состоящей из 257 символов.

Поле `dwSize` должно содержать размер структуры в байтах. Хотя его значение можно вычислить вручную, объект `Marshal` содержит общий метод `SizeOf`, возвращающий размер структуры при передаче в неуправляемую память. Этот метод значительно надежнее функции VB6 `LenB`, и с его помощью можно проверять правильность задания атрибутов полей структуры. Инициализировать поле `szEntryName` не обязательно, поскольку атрибут `MarshalAs` определяет размер строки и обеспечивает передачу данных нужной длины.

Объявление функции API `RasEnumEntries` в документации MSDN выглядит так:

```
DWORD RasEnumEntries (
  LPCTSTR reserved,       // Зарезервировано, должно быть равно NULL
  LPTCSTR lpszPhonebook, // Указатель на полное имя файла
                          // телефонной книги
  LPRASENTRYNAME lprasentryname,
           // Буфер, заполняемый данными
           // из телефонной книги
  LPDWORD lpcb,        // Размер буфера в байтах
  LPDWORD lpcEntries  // Количество структур, записанных в буфер
);
```

Мы воспользуемся следующей командой `Declare`:

```
Public Declare Auto Function RasEnumEntries Lib _
"rasapi32.dll" (ByVal reserved As Integer, ByVal lpszPhoneBook _
As String, ByVal rasentries As IntPtr, ByRef lpcb As Integer, _
ByRef lpcEntries As Integer) As Integer
```

В параметре `rasentries` передается адрес блока неуправляемой памяти. Ниже показано, как создать этот блок и выполнить с ним необходимые операции.

Функция RasEnumEntries вызывается дважды: при первом вызове она возвращает количество элементов и размер массива, а при втором — непосредственные данные.

Программа вычисляет размер одной структуры RASENTRYNAME и выделяет в неуправляемой памяти блок для ее хранения. Выделение памяти осуществляется методом Marshal.AllocHGlobal.

```
Sub Main()
  Dim res As Integer
  Dim cb, cbentries As Integer
  Dim idx As Integer
  Dim iptr As IntPtr
  Dim SizePerStruct As Integer
  SizePerStruct = Marshal.SizeOf(GetType(RASENTRYNAME))
  ' Получить размер структуры
  iptr = Marshal.AllocHGlobal(SizePerStruct)
```

Затем мы создаем одну структуру RASENTRYNAME, инициализируем ее и передаем в буфер методом Marshal.StructureToPtr. После вызова метода RasEnumEntries данные возвращаются в структурную переменную методом Marshal.PtrToStructure. Обратите внимание на освобождение буфера в неуправляемой памяти методом Marshal.FreeHGlobal.

```
Dim rasentries(0) As RASENTRYNAME
rasentries(0).Init()
cb = rasentries(0).dwSize
cbentries = 1
Marshal.StructureToPtr(rasentries(0), iptr, False)

' При первом вызове возвращается количество записей.
res = RasEnumEntries(0, Nothing, iptr, cb, cbentries)

rasentries(0) = CType(Marshal.PtrToStructure(iptr, _
  GetType(RASENTRYNAME)), RASENTRYNAME)
Marshal.FreeHGlobal (iptr)
```

Если функция возвращает код 603, значит, в телефонной книге остались другие записи. В этом случае мы переобъявляем массив rasentries с новым размером, достаточным для хранения всех записей телефонной книги, а затем вручную копируем все записи в заново выделенный буфер нужного размера. Следующий фрагмент также демонстрирует инициализацию указателей на неуправляемую память в конструкторе IntPtr:

```
  If res = 603 Then
    ReDim rasentries(cbentries - 1)
    cb = 0
    iptr = Marshal.AllocHGlobal(cbentries * SizePerStruct)

    For idx = 0 To cbentries - 1
      rasentries(idx).Init()
      Marshal.StructureToPtr(rasentries(idx), _
        New IntPtr(iptr.ToInt32 + cb), False)
      cb = cb + rasentries(idx).dwSize
    Next
    res = RasEnumEntries(0, Nothing, iptr, cb, cbentries)
  End If
```

```
    If res = 0 Then
      cb = 0
      For idx = 0 To cbentries - 1
        rasentries(idx) = CType(Marshal.PtrToStructure(New _
          IntPtr(iptr.ToInt32 + cb), GetType(RASENTRYNAME)), _
          RASENTRYNAME)
        cb = cb + rasentries(idx).dwSize
        console.WriteLine (rasentries(idx).szEntryName)
      Next
    End If
    Marshal.FreeHGlobal (iptr)

    console.ReadLine()
  End Sub

End Module
```

Итак, средства Visual Basic .NET позволяют выделять блоки в неуправляемой памяти и копировать данные в неуправляемую память и обратно. Более того, VB .NET значительно мощнее VB6, поскольку ваши возможности не ограничиваются простым копированием данных функцией API `RtlMoveMemory`, часто использовавшейся в подобных ситуациях в VB. Методы `Marshal.StructureToPtr` и `Marshal.PtrToStructure` копируют данные с учетом выравнивания и позволяют управлять передачей каждого поля структуры[1].

Копирование данных в буфер и обратно играет важную роль, поскольку функции API иногда возвращают буферы переменной длины.

Получение информации об одном соединении функцией API RasGetEntryProperties

Описание конкретной записи телефонной книги хранится в структуре `RASENTRY` (листинг 15.14), содержащей большое количество полей[2].

Листинг 15.14. Структура RASENTRY (C++)

```
typedef struct tagRASENTRY {
  DWORD  dwSize;
  DWORD  dwfOptions;
  //
  // Местонахождение/телефон
  //
  DWORD  dwCountryID;
  DWORD  dwCountryCode;
  TCHAR  szAreaCode[ RAS_MaxAreaCode + 1 ];
  TCHAR  szLocalPhoneNumber[ RAS_MaxPhoneNumber + 1 ];
  DWORD  dwAlternateOffset;
  //
```

продолжение ⇙

[1] Вероятно, вас интересует, почему я не передаю сразу весь массив структур RASENTRYNAME? Просто мне не удалось заставить работать это решение. Передача данных массива функции API проходила нормально, но с передачей в обратном направлении возникли проблемы. Система P-Invoke умеет передавать массивы простых типов, но на момент написания книги в документации ничего не говорилось о передаче массивов структур. Пока трудно сказать, что это такое — норма или ошибка в бета-версии.

[2] Как и прежде, мы не используем расширенную версию этой структуры с новыми полями, появившимися в Windows 2000.

Листинг 15.14 (продолжение)

```
  // PPP/Ip
  //
  RASIPADDR  ipaddr;
  RASIPADDR  ipaddrDns;
  RASIPADDR  ipaddrDnsAlt;
  RASIPADDR  ipaddrWins;
  RASIPADDR  ipaddrWinsAlt;
  //
  // Сетевой протокол
  //
  DWORD  dwFrameSize;
  DWORD  dwfNetProtocols;
  DWORD  dwFramingProtocol;
  //
  // Сценарии
  //
  TCHAR  szScript[ MAX_PATH ];
  //
  // Автодозвон
  //
  TCHAR  szAutodialDll[ MAX_PATH ];
  TCHAR  szAutodialFunc[ MAX_PATH ];
  //
  // Устройство
  //
  TCHAR  szDeviceType[ RAS_MaxDeviceType + 1 ];
  TCHAR  szDeviceName[ RAS_MaxDeviceName + 1 ];
  //
  // X.25
  //
  TCHAR  szX25PadType[ RAS_MaxPadType + 1 ];
  TCHAR  szX25Address[ RAS_MaxX25Address + 1 ];
  TCHAR  szX25Facilities[ RAS_MaxFacilities + 1 ];
  TCHAR  szX25UserData[ RAS_MaxUserData + 1 ];
  DWORD  dwChannels;
  //
  // Зарезервированные поля
  //
  DWORD  dwReserved1;
  DWORD  dwReserved2;
} RASENTRY;
```

Основные трудности преобразования этой структуры в VB .NET связаны с правильным указанием длин всех строк. Объявление, приведенное в листинге 15.15, имеет много общего с предыдущим примером (см. листинг 15.13).

Листинг 15.15. Структура RASENTRY (VB .NET)

```
' Приложение RasGetEntry
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Imports System.Runtime.InteropServices

Module Module1

<StructLayout(LayoutKind.Sequential, Pack:=4, CharSet:=charset.Auto)>
Structure RASENTRY
  Public dwSize As Integer
  Public dwfOptions As Integer
```

```
  Public dwCountryID As Integer
  Public dwCountryCode As Integer
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=11)> Public _
    szAreayCode As String '11 символов
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=129)> Public _
    szLocalPhoneNumber As String
  Public dwAlternateOffset As Integer
  Public ipaddr As Integer
  Public ipaddrDns As Integer
  Public ipaddrDnsAlt As Integer
  Public ipaddrWins As Integer
  Public ipaddrWinsAlt As Integer

  Public dwFrameSize As Integer
  Public dwfNetProtocols As Integer
  Public dwFramingProtocol As Integer

  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=260)> Public _
    szScript As String

  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=260)> Public _
    szAutodialDll As String
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=260)> Public _
    szAutodialFunc As String

  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=17)> Public _
    szDeviceType As String
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=129)> Public _
    szDeviceName As String

  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=33)> Public _
    szX25PadType As String
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=201)> Public _
    szX25Address As String
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=201)> Public _
    szFacilities As String
  <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=201)> Public _
    szUserData As String
  Public dwChannels As Integer

  Public dwReserved1 As Integer
  Public dwReserved2 As Integer
End Structure
```

Функция RasGetEntryProperties определяется в документации MSDN следующим образом:

```
DWORD RasGetEntryProperties(
  LPCTSTR lpszPhonebook,     // Указатель на полное имя файла
                             // телефонной книги
  LPCTSTR lpszEntry,         // Указатель на имя записи
  LPRASENTRY lpRasEntry,     // Буфер, заполняемый данными записи
  LPDWORD lpdwEntryInfoSize, // Размер буфера lpRasEntry
                             // в байтах
  LPBYTE lpbDeviceInfo,      // Буфер, заполняемый параметрами
                             // конфигурации для конкретного устройства
  LPDWORD lpdwDeviceInfoSize // Размер буфера lpbDeviceInfo
                             // в байтах
```

В программе для функции RasGetEntryProperties создаются два объявления. Одно объявление получает в качестве параметра указатель на одну структуру RASENTRY, а другое — указатель на блок неуправляемой памяти. Вскоре вы увидите, почему это необходимо. Объявления выглядят так:

```
Public Declare Auto Function RasGetEntryProperties Lib _
"rasapi32.dll" (ByVal lpszPhoneBook As String, ByVal lpszEntry _
As String, ByRef lpRasEntry As RASENTRY, ByRef lpdwEntryInfoSize _
As Integer, ByVal devinfo As Integer, ByVal devinfosize _
As Integer) As Integer

Public Declare Auto Function RasGetEntryProperties2 Lib _
"rasapi32.dll" Alias "RasGetEntryProperties" (ByVal _
lpszPhoneBook As String, ByVal lpszEntry As String, ByVal _
lpRasEntry As IntPtr, ByRef lpdwEntryInfoSize As Integer, _
ByVal devinfo As Integer, ByVal devinfosize As Integer) As Integer
```

В документации Win32 сказано, что вместо неиспользуемых параметров devinfo можно передать Null. В нашем примере оба параметра объявлены с типом ByVal As Integer, и при вызове функции им присваиваются нули.

Присвойте константе PhoneBookEntryToGet имя записи своей телефонной книги по умолчанию (вряд ли в ней найдется запись с именем DesawareModem). Программа инициализирует поле dwSize правильной длиной, присваивает то же значение переменной bufsize и вызывает функцию RasGetEntryProperties.

```
Const PhoneBookEntryToGet As String = "DesawareModem"

Sub Main()
  Dim res As Integer
  Dim re As RASENTRY
  Dim bufsize As Integer
  re.dwSize = Marshal.SizeOf(re)
  bufsize = re.dwSize
  res = RasGetEntryProperties(Nothing, PhoneBookEntryToGet, re, bufsize, 0, 0)
  If res = 623 Then
    Console.WriteLine ("Can't find specified dial-up entry")
  End If
```

Почему вызов может завершиться неудачей? Прежде всего — если в телефонной книге отсутствует запись с указанным именем. Столкнувшись с ошибкой 623, проверьте, не забыли ли вы присвоить константе PhoneBookeEntryToGet имя соединения удаленного доступа в вашей системе.

Но даже если имя соединения задано верно, ошибки все равно возможны, поскольку функция RasGetEntryProperties может возвращать дополнительную информацию об устройстве, выходящую за границы структуры. Подобные ситуации очень часто встречаются при вызове нетривиальных функций API. Иногда в этом дополнительном пространстве хранятся строковые данные, на которые ссылаются указатели в структуре (эти данные также легко извлекаются средствами P-Invoke). Универсальное решение — выделить блок памяти длины, ожидаемой функцией, скопировать структуру в блок, вызвать функцию API и затем скопировать данные обратно, как показано ниже:

```
  If res = 603 Then
    Dim iptr As IntPtr
    iptr = Marshal.AllocHGlobal(bufsize)
```

```
      Marshal.StructureToPtr(re, iptr, False)
      res = RasGetEntryProperties2(Nothing, PhoneBookEntryToGet, _
        iptr, bufsize, 0, 0)
      re = CType(Marshal.PtrToStructure(iptr, GetType(RASENTRY)), _
        RASENTRY)
      Marshal.FreeHGlobal (iptr)
    End If
    Console.WriteLine (re.szLocalPhoneNumber)
    Console.ReadLine()
  End Sub

End Module
```

Итоги

Платформа .NET упрощает программирование, но переход на эту платформу может оказаться долгим — очень долгим. Готовые компоненты не переключатся на нее сами по себе. То же самое можно сказать и о других технологиях Microsoft — таких, как Transaction Server и Microsoft Message Queue (как бы они не назывались после очередного переименования). Следовательно, взаимодействие с COM и базовыми средствами операционной системы является важнейшей частью .NET Framework.

В этой главе было показано, как просто использовать компоненты COM в сборках .NET. Visual Studio выполняет за вас большую часть работы. С использованием компонентов .NET в программах COM дело обстоит чуть сложнее, но если ограничиться Automation-совместимыми интерфейсами (такими, как используются в VB6), вам удастся избежать практически всех затруднений, возникающих при работе с другими типами данных. Предоставляя доступ к сборкам .NET из COM, необходимо в первую очередь позаботиться о контроле версии, в противном случае может возникнуть знакомая ситуация «кошмара DLL».

Мы подошли к концу одной из моих любимых тем — использования функций API в Visual Basic .NET. В этой области VB .NET превосходит VB6 по гибкости и разнообразию возможностей. Многие функции API в VB .NET вызываются так же просто, как в VB6, однако при вызове сложных функций приходится использовать нетривиальные приемы (кстати говоря, в C# это делается ничуть не проще). Главное, о чем должен помнить программист VB6 при переходе на VB .NET, — что все параметры типа `Long` заменяются на `Integer`, а параметры типа `Integer` заменяются на `Short`.

Дальнейшие перспективы 16

Книга называется «Переход на VB .NET: стратегии, концепции, код». Мы познакомились со стратегией перехода на VB .NET. Мы изучили ключевые концепции, которыми должен владеть каждый разработчик VB .NET. Мы рассмотрели не только изменения в синтаксисе языка VB .NET, но и примеры программного кода, демонстрирующие основные принципы использования важнейших пространств имен .NET.

Библиотека .NET Framework огромна. Хотя в книге рассматривается лишь малая ее часть, надеюсь, описанные базовые концепции помогут вам взяться за ее самостоятельное изучение.

Остается рассмотреть несколько дополнительных тем, которые являются неотъемлемой частью .NET, хотя и влияют на относительно небольшую часть процесса разработки.

Сначала мы в последний раз вернемся к «кошмару DLL», а затем обсудим проблемы безопасности в .NET. Книга завершается несколькими мелкими темами, для которых не нашлось места в предыдущих главах.

Хочу особо подчеркнуть, что эта глава чрезвычайно важна. Приведенный материал расположен в конце книги вовсе не потому, что ему уделяется второстепенное значение. Более того, контроль версии и безопасность одними из первых оказались в списке тем, когда я продумывал общую структуру книги, — без их хорошего понимания невозможно нормально программировать в .NET. Впрочем, при наличии хороших познаний в области .NET разобраться в них значительно проще — а если вы дочитали до этой страницы, считайте, что такие познания у вас уже есть.

Контроль версии в .NET

Из главы 11 вы узнали, что в .NET-языках проблемы с конфликтами компонентов, часто возникающие в приложениях COM, решаются посредством JIT-компиляции. Умение JIT-компилятора связывать имена методов зависимых сборок при их построении и обнаруживать несовместимые изменения в процессе JIT-компиляции решает целый класс проблем с совместимостью DLL, которые мы ласково называем «кошмаром DLL».

Однако «кошмар DLL» — штука упрямая, и JIT-компиляция не учитывает некоторые возможные ситуации.

ВНИМАНИЕ

В контексте этой главы все, что говорится о зависимости приложения от компонентов или сборок, относится в равной степени и к зависимости последних от других компонентов или сборок.

Кошмарные сценарии

Предположим, у вас имеется распределенное приложение A, использующее COM DLL с именем D версии 1.0 (Dv1.0). Затем появляется приложение B, которое использует новую версию COM DLL D версии 2.0 (Dv2.0).

К сожалению, хотя разработчики Dv2.0 постарались организовать контроль версии DLL при помощи режима двоичной совместимости VB6, а внутренние GUID всех методов и параметров остались полностью совместимыми, вкралась небольшая ошибка. В Dv1.0 поддерживалось свойство `Value`, которое в случае присваивания ему отрицательной величины автоматически обнулялось, но не возвращало информации об ошибке (поскольку данное свойство не могло принимать отрицательные значения). Было решено, что это неправильно, и в Dv2.0 ошибка была исправлена: при попытке задания отрицательного значения свойство `Value` инициировало ошибку.

По недосмотру программиста в приложении A свойству `Value` задавалось значение –1. На работу приложения это не влияло, поскольку библиотека Dv1.0 просто обнуляла свойство, и дальше все шло нормально.

Приложение B, распространяемое с Dv2.0, прекрасно работало. Но после установки в системе Dv2.0 в приложении A происходил сбой. Там, где раньше некорректное изменение свойства `Value` обходилось без проблем, возникала ошибка времени выполнения, которую разработчики приложения A не потрудились перехватить...

Перед вами вполне реальная история[1].

Подобные проблемы возникают в Windows из-за того, что в общем случае в системе может присутствовать только одна версия компонента. После установки нового компонента в каталоге System или его регистрации (в зависимости от типа компонента — традиционная библиотека DLL, экспортирующая функции, или COM DLL, предоставляющая доступ к объектам) этот компонент будет возвращаться любому приложению, от которого поступил запрос.

---

[1] Компонент назывался «Animated Button» и входил в подмножество продукта Desaware Custom Control Factory, лицензированное компанией Microsoft для включения в Visual Basic. Приложением B был сам Visual Basic. В процессе тестирования специалисты Microsoft обнаружили ошибку со значением свойства `Value` и потребовали, чтобы при передаче отрицательной величины инициировалась ошибка. Я предупредил, что это вызовет серьезные проблемы с совместимостью, но они настаивали, что это явная ошибка и что все, кто использует значение –1 в своей программе, просто исправит свой код. Поскольку счета оплачивались в Microsoft, я внес изменения, и, конечно, приложение A перестало работать — в нем свойству `Value` элемента Animated Button по ошибке задавалось значение –1. Приложением A была одна из ранних версий Microsoft Encarta. До сих пор пакет Custom Control Factory (откуда был взят элемент Animated Button) принимает значение –1 для свойства `Value` и спокойно обнуляет его. В компании Desaware *серьезно* относятся к проблемам совместимости. Лучше добавить в элемент новое свойство, чем вносить исправления, принципиально изменяющие поведение нашего компонента.

Возникает целая серия потенциальных проблем. Приведу лишь несколько примеров.

- Новая версия компонента устанавливается поверх старой версии, но при этом она не обладает обратной совместимостью.
- Старая версия компонента устанавливается поверх новой, а работа приложения зависит от возможностей, присутствующих только в новой версии.
- В системе регистрируется старая или новая версия компонента COM таким образом, что, несмотря на присутствие правильной версии, ваше приложение загружает другой компонент по данным реестра.

Короче говоря, вы оказываетесь в «кошмаре DLL».

Как видите, даже при обеспечении идеальной совместимости на двоичном уровне всегда остается возможность функциональных несоответствий. Пока в системе работает единственная версия компонента, «кошмар DLL» теоретически нельзя исключить.

В Windows 98 и 2000 появились две особенности, в определенной степени исправлявшие положение дел. В этих операционных системах поддерживается так называемое параллельное выполнение (side-by-side execution) и механизм подстановки DLL, позволяющий приложению загрузить DLL из локального каталога вместо каталога System. Если создать в каталоге файл с расширением .local, приложение могло загрузить локальную копию компонента даже при наличии в системе другой версии. Параллельное выполнение требует учета специальных факторов при написании компонента.

К сожалению, на практике этим решениям присущи некоторые недостатки.

- Поскольку при проектировании большинства программных компонентов возможность параллельного выполнения не учитывалась, невозможно гарантировать нормальную работу этих компонентов при подстановке DLL.
- Эти механизмы не работают в версиях Windows, предшествующих Windows 98 и Windows 2000, однако фирмы-разработчики вряд ли захотят игнорировать эти операционные системы как платформу для создания коммерческих компонентов.
- Компоненты VB6 могут использоваться при подстановке DLL, но поскольку программист не может управлять регистрацией компонентов VB6, создание компонентов для параллельного выполнения в VB6 невозможно.

Вряд ли есть смысл тратить значительные усилия на разработку компонентов COM с поддержкой параллельного выполнения, поскольку эта проблема (как вы вскоре убедитесь) легко решается в .NET.

Параноидальные сценарии

Недавно я общался по электронной почте с одним программистом. Бедняга опасается, что кто-нибудь дизассемблирует его приложение, и хочет знать, нельзя ли зашифровать один исполняемый файл внутри другого и восстанавливать внутренний файл при запуске внешнего. При этом требуется полностью заблокировать отладку программы и даже исключить ее из списка процессов[1].

[1] Нет, я не знаю, можно это сделать или нет.

Интересно, какая программа может требовать такого уровня защиты?[1]

Но любой параноик способен представить себе следующий сценарий.

Приложение A использует COM DLL Dv1 — очень надежный компонент, неспособный повредить системе. Некий злоумышленник вынашивает коварный план. Он создает копию Dv1 и дизассемблирует ее. Ему не нужно разбираться в функциональности компонента, так как большинство функций вообще не изменяется и воссоздается повторным ассемблированием. Но в результате модификации некоторые ключевые методы начинают вытворять ужасные вещи: стирать жесткий диск, рассылать испорченную библиотеку по списку адресов электронной почты, хранящихся в системе и т. д. Затем злоумышленник строит DLL, присваивает ей то же имя (назовем ее Dv1bad) и увеличивает номер ревизии до 1.01, чтобы библиотека устанавливалась поверх любой существующей DLL с тем же именем. Dv1bad обладает теми же данными библиотеки типов, теми же свойствами версии и т. д. Наконец, злоумышленник распространяет испорченную DLL с приложением B, которое внешне выглядит совершенно невинно.

Подобный прием называется *фальсификацией* (spoofing). Он встречается довольно редко, но может приводить к очень серьезным последствиям — особенно если фальсифицируется системный файл.

К сожалению, в Windows не предусмотрено встроенных средств для решения подобных проблем. Обычно фальсифицированная программа рассматривается как вирус, а ее сигнатура включается в базы данных антивирусных программ. К сожалению, в системах, где эта программа была запущена, такие меры уже не помогут.

Контроль версии в .NET

В Microsoft .NET поддерживаются два разных механизма контроля версии в зависимости от того, какое имя было выбрано для сборки.

Но позвольте, как выбор имени связан с версией приложения?

Оказывается, между ними существует прямая зависимость.

В .NET сборкам могут присваиваться простые или сильные (strong) имена. Простое имя представляет собой имя сборки, указанное в параметрах проекта, и обычно совпадает с именем файла сборки. Сильное имя содержит простое имя, номер версии, культуру, открытый ключ и сигнатуру сборки.

Во всех сборках, которые мы создавали до настоящего момента, использовались простые имена. Это были простые программы, написанные для учебных целей, поэтому я не беспокоился о контроле версии или зависимостях. Фальсификация меня тоже не страшит — в конце концов, у вас имеются все исходные тексты.

Однако в большинстве сборок, предназначенных для последующего распространения, следует использовать сильные имена, что объясняется следующими причинами.

- Простые имена предназначены для случаев, когда все сборки загружаются из локального каталога приложения.

[1] Нет, я не спрашивал.

- Сборки с простыми именами не поддерживают контроля версии. Осуществляя связывание со сборкой, обладающей простым именем, CLR загружает из текущего каталога любую найденную сборку с заданным именем, не проверяя номер версии.
- Простые имена не защищают от фальсификации.

Отсутствие сборки с простым именем будет обнаружено исполнительной средой .NET. При нарушении совместимости между версиями зависимой сборки с простым именем во время выполнения программы произойдет ошибка и будет инициировано исключение — по крайней мере, это не приводит к ошибкам защиты памяти. Конечно, если исключение не будет перехвачено, все кончится аварийным завершением программы.

Проекты UnsignedApp1 и UnsignedDLL1 написаны специально для того, чтобы вы могли поэкспериментировать с зависимостями простых сборок. Измените версию UnsignedDLL1 и запустите UnsignedApp1. Также попробуйте изменить параметры функции `Class1.MyVersion` в UnsignedDLL1 и посмотрите, что произойдет при запуске UnsignedApp1.

Другими словами, хотя при использовании простых имен .NET защитит от системных сбоев, «кошмар DLL» по-прежнему остается очень серьезной проблемой.

Существуют ли ситуации, в которых предпочтительно использовать простые имена? Я могу предложить только два варианта.

1. Если проблемы контроля версии или зависимости для вас несущественны (например, при написании примеров для книги или статьи).
2. Если вы полностью контролируете содержимое каталога. От параллельного выполнения выигрывают даже сборки с простыми именами — это естественное следствие того факта, что .NET сначала пытается загрузить сборку с простым именем из локального каталога приложения. Но даже в этом случае сильные имена не имеют никаких отрицательных сторон, если не считать некоторых дополнительных хлопот в процессе построения (почти полностью автоматизированного в Visual Studio).

Сильные имена рекомендуется использовать во всех сборках, предназначенных для практического использования (даже внутри организации).

Сильные имена

Попробуйте создать простое приложение VB6, в котором функция `CreateObject` создает объект с именем `Project1.Class1`.

Если ваш компьютер ранее использовался для программирования на VB6, эта операция завершится успешно. Вы получите объект для последнего созданного ActiveX EXE или DLL.

В COM нет ничего, что помешало бы двум программистом создать одноименные объекты. Проблема конфликтов имен в COM решается при помощи GUID — глобально-уникальных идентификаторов, которые являются «истинными» именами объектов, классов и интерфейсов.

В Microsoft .NET избран другой подход: сборка идентифицируется объединением различных информационных элементов. К числу таких элементов относятся:

- простое имя сборки;
- номер версии;
- культура, то есть локальный контекст сборки (необязательно);
- открытый ключ автора сборки;
- хэш-код сборки.

Сочетание этих элементов однозначно идентифицирует сборку. По отдельности эти элементы служат разным целям.

- Простое имя представляет название сборки в виде, нормально воспринимаемом человеком.
- Номер версии определяет точную версию сборки, что позволяет приложению запросить и загрузить конкретную разновидность сборки.
- Локальный контекст позволяет различать варианты сборки, локализованные для разных языков или по другим причинам поддерживающим разные локальные контексты.
- По открытому ключу можно обнаружить возможную фальсификацию и узнать, что сборка была создана не ее исходным автором, а кем-то другим.
- Хэш-код позволяет обнаружить модификации сборки.

Использование сильных имен исключает опасность «кошмара DLL» для компонентов .NET[1].

Сильные имена и контроль версии

Вспомним, как приложение загружает компоненты COM.

1. Приложение знает имя или GUID загружаемого объекта.
2. Приложение ищет в реестре данные о местонахождении объекта.
3. Приложение загружает объект.

Поскольку в реестре для каждого компонента хранится всего одна запись, не существует механизма идентификации, поиска и загрузки разных версий одного компонента[2].

Компоненты .NET вообще не используют реестр. Общие сборки хранятся в глобальном кэше сборок (global assembly cache, GAC), причем для хранения используются только длинные имена. Другими словами, GAC может содержать столько разных версий одной сборки, сколько вам захочется создать.

Процесс загрузки компонента .NET в приложении .NET называется связыванием (binding). Он принципиально отличается от аналогичного процесса COM[3].

---

[1] Впрочем, сильные имена не препятствуют возникновению «кошмара DLL» при использовании компонентов COM средствами COM Interop.

[2] Не считая редко используемого параллельного выполнения на базе {\lang1033 COM}, о котором говорилось выше.

[3] Следующее описание несколько упрощено. За подробным описанием процесса загрузки обращайтесь к разделу «How the Runtime Locates Assemblies» электронной документации.

- Приложение вычисляет сильное имя загружаемой сборки на основании данных приложения, разработчика и системных конфигурационных файлов (см. ниже).
- Если сборка была загружена ранее, приложение выполняет связывание с загруженной сборкой.
- CLR проверяет, присутствует ли сборка в GAC. Если сборка будет обнаружена, она загружается из GAC.
- CLR ищет компонент в нескольких местах на основании содержимого конфигурационных файлов. К числу таких мест относится каталог, заданный параметром `Codebase` (если параметр указан), каталог приложения и прочие подкаталоги, указанные в конфигурационных файлах.
- Связывание проводится по сильному имени. Поскольку сильное имя содержит номер версии, это означает, что каждое приложение располагает точным списком зависимых сборок и по умолчанию работает только с ними лишь при точном совпадении номера версии.

Пример использования сильных имен

Проекты StrongApp1 и StrongDLL1 предназначены для экспериментов с контролем версии.

Проект StrongDLL1 содержит единственный класс с методом, возвращающим полное имя версии для компонента:

```
Imports System.Reflection
Public Class Class1
  Public Function MyVersion() As String
    Return Reflection.Assembly.GetExecutingAssembly.FullName()
  End Function

End Class
```

Точный номер версии задается в файле Assemblyinfo.vb при помощи атрибута:

```
<Assembly: AssemblyVersion ("1.0.0.1")>
```

В соответствии с конфигурацией проекта StrongDLL1 при создании сильного имени используется ключевой файл, заданный в параметрах проекта (на вкладке Sharing). Ключевой файл TestKeys.snk создается утилитой sn.exe с параметром `-k` (хотя его также можно создать при помощи Visual Studio). Файл TestKeys.snk находится в каталоге CH16 и используется несколькими проектами этой главы. Не используйте этот файл в своих приложениях[1].

Обычно доступа к ключевому файлу у вас не будет: большинство организаций предпочитает хранить свои закрытые ключи на гибком диске или защищенном сервере и не распространяет их среди разработчиков. Чтобы из-за этого у вас не возникали затруднения, установите флажок Delay Sign. При этом в сборку внедряется открытый ключ. В результате приложения могут ссылаться на сборку, а в исполняемом файле резервируется место для последующего включения полного

[1] Ключевой файл был сгенерирован специально для примеров, включенных в эту главу. Не используйте его для каких-либо целей, кроме экспериментов с этими программами.

имени. Вы можете при помощи программы sn.exe извлечь открытый ключ и распространить его среди разработчиков. Перед выпуском окончательной версии программа sn.exe заново «подписывает» сборку закрытым ключом и вставляет в нее сильное имя.

Ссылка на сборку StrongDLL1 содержится в проекте StrongApp1. Это консольное приложение выводит данные о версии сборки, вызывая ее открытую функцию MyVersion.

```
Imports strongDLL1
Module Module1

  Sub Main()
    Dim c As New strongDLL1.Class1()
    Console.WriteLine (c.MyVersion)
    Console.ReadLine()
  End Sub

End Module
```

При запуске программы выводится следующий результат:

```
strongDLL1, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=4139a2c451ef76d7
```

Поле PublicKeyToken содержит хэшированную версию открытого ключа (вскоре мы вернемся к этой теме). Если дизассемблировать приложение StrongApp1, в нем обнаруживается следующий фрагмент:

```
.assembly extern strongDLL1
{
  .publickeytoken = (41 39 A2 C4 51 EF 76 D7)
  // A9..Q.v
  .ver 1:0:0:0
}
```

Приложение будет связываться только с версией 1.0.0.0 библиотеки strongdll1 с заданным открытым ключом. Таким образом, по умолчанию для работы каждой сборки с сильным именем необходимо точное совпадение версий всех зависимых сборок. Тем самым предотвращаются несовместимости, обусловленные расхождением версий.

Разумеется, подобный подход затрудняет обновление компонентов.

Обновление компонентов с сильными именами

На первый взгляд может показаться, что при любом распространении обновленных компонентов приходится заново строить и распространять все приложения, использующие данный компонент (поскольку в противном случае приложения будут ссылаться на старую версию компонента). Ликвидируя «кошмар DLL», мы создаем «кошмар с распространением».

К счастью, в .NET предусмотрен механизм управления связыванием, позволяющий загружать обновленные версии зависимых сборок. В работе этого механизма используется не один и не два, а целых три конфигурационных файла в формате XML. Конфигурационные файлы управляют безопасностью, контролем версии и удаленным доступом к приложению.

Первый из этих файлов (конфигурационный файл приложения, application configuration file) определяет стандартную конфигурацию приложения. Конфигурационный файл публикации (publisher configuration file) добавляется при обновлении приложения для переопределения стандартных параметров, в том числе и относящихся к контролю версии. Также существует и конфигурационный файл компьютера (machine configuration file), который создается системным администратором и управляет поведением всех сборок в системе (с переопределением всех параметров двух предыдущих файлов).

Имя конфигурационного файла приложения строится по правилу «имя приложения + суффикс .config» — например, Myapp.exe.config. Имя конфигурационного файла публикации строится в формате `политика.осн.доп.сборка.dll`, где `осн` и `доп` — основной и дополнительный номера версии `сборки`, к которой применяется заданная `политика`. Для версий 2.1.0.0–2.1.x.x сборки strongdll1 политика определяется файлом с именем policy.2.1.strongdll1.dll. Файлы политики применяются только к сборкам, хранящимся в GAC. За инструкциями по созданию файлов политики обращайтесь к электронной документации. Системная конфигурация обычно задается при помощи Microsoft Management Console.

Все конфигурационные файлы заключаются в теги верхнего уровня `<Configuration>` (то есть файл начинается с тега `<Configuration>` и завершается тегом `</Configuration>`).

Рассмотрим пример конфигурационного файла.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
      <assemblyIdentity name="strongDLL1"
      publicKeyToken="4139a2c451ef76d7" culture="" />
        <bindingRedirect oldVersion="1.0.0.0-1.0.0.1"
        newVersion="1.0.0.2" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Для каждой зависимой сборки, поведение которой требуется изменить, в конфигурационном файле создается блок `<dependentAssembly>`. В него включается тег `<assemblyIdentity>`, теги которого используются для идентификации сборки по сильному имени (имя, версия, локальный контекст и открытый ключ).

Тег `<codebase>` показывает, откуда следует загружать сборку, если она отсутствует в глобальном кэше сборок или в локальном каталоге.

Тег `<bindingRedirect>` предписывает исполнительной среде загрузить обновленную сборку. В данном примере приложения, построенные с версиями 1.0.0.0–1.0.0.1 сборки strongDLL1, загружают версию 1.0.0.2. Предполагается, что перед заданием этого параметра вы тщательно проверили совместимость обновленной сборки.

Тег `<publisherPolicy>` (отсутствующий в приведенном примере) указывает, следует ли использовать конфигурационный файл публикации, который представляет мнение публикующей стороны о том, с какой версией следует выполнить связывание. Иначе говоря, даже если разработчик компонента выпустит

обновленную сборку с файлом политики, разрешающим автоматическое обновление, у вас есть возможность переопределить настройки этого файла и заставить приложение работать со старой версией компонента.

В теге <probing>, находящемся внутри блока <assemblyBinding>, перечисляются подкаталоги приложения, в которых осуществляется поиск сборки. С помощью этого тега можно организовать логическую группировку, если в проекте задействовано большое количество сборок.

Контроль версии исполнительной среды .NET

Рассмотрим следующий пример конфигурационного файла:

```
<configuration>
  <startup>
    <requiredRuntime version="1.0.0.0" safeMode="false"/>
  </startup>
</configuration>
```

В приведенном примере указывается, что для работы приложения необходима конкретная версия исполнительной среды .NET.

Прежде я неоднократно упоминал о том, что .NET решает проблему «кошмара DLL», позволяя сборкам указывать точную версию зависимых сборок. К счастью, в Microsoft вовремя сообразили, что этот принцип относится к самой среде .NET, особенно если учесть, что Microsoft породила больше «кошмаров DLL» с несовместимыми компонентами, чем любая другая компания (не сочтите выпадом против Microsoft — это простое отражение того факта, что в Microsoft было создано больше DLL, чем где-либо еще). Более того, в системе может быть одновременно установлено несколько версий исполнительной среды .NET, которые могут одновременно работать.

У этого флага есть лишь один недостаток: он не всегда помогает с приложениями, работающими под управлением Internet Explorer (например, web-элементами), поскольку в одном процессе не могут одновременно работать две разные версии среды .NET.

Сильные имена и фальсификация

Итак, вы узнали, как сильные имена помогают избавиться от проблем контроля версии за счет включения номера версии в имя сборки. Теперь давайте посмотрим, какое место они занимают в борьбе с фальсификацией и модификацией сборок.

Как было показано выше, приложение StrongApp1 содержит образец открытого ключа сборки StrongDLL1. Образец представляет собой хэш-код реального ключа. Идея заключается в том, что сохранять в приложении весь открытый ключ не нужно, достаточно убедиться в том, что открытый ключ, содержащийся в зависимой сборке, совпадает с ключом, указанным при построении приложения. Для этого вполне можно обойтись хэш-кодом, поскольку вероятность совпадения хэш-кодов двух разных открытых ключей близка к нулю (как и вероятность сгенерировать открытый ключ по хэш-коду).

В процессе загрузки загрузчик CLR просматривает зависимую сборку и хэширует ее открытый ключ. Результат сравнивается с образцом из вызывающего при-

ложения. В случае совпадения CLR знает, что открытый ключ вызывающего приложения действителен и может использоваться для дальнейших проверок.

Природа шифрования с открытым ключом такова, что для расшифровки информации, зашифрованной с закрытым ключом, используется только открытый ключ. При пометке сборки закрытым ключом (в процессе построения или окончательного выпуска) вся сборка хэшируется, в результате чего появляется сигнатура. Сигнатура зашифровывается закрытым ключом и расшифровывается открытым ключом.

Пока закрытый ключ остается недоступным для злоумышленников, сборку не удастся незаметно изменить. Во время загрузки хэшируется вся сборка в ее текущем состоянии. Открытый ключ позволяет расшифровать сигнатуру, включенную в сборку при создании. Если новый хэш-код не совпадает с сигнатурой в файле, CLR узнает, что файл был модифицирован, и отказывается загружать его.

Сохраненную сигнатуру невозможно изменить, поскольку она зашифрована закрытым ключом. Открытый ключ невозможно изменить, поскольку образец не совпадет с хранящимся в приложении.

Итак, сильные имена успешно решают проблему фальсификации сборок.

Сильные имена и пометка кода

Тем не менее, сильные имена не позволяют точно определить, откуда поступила сборка. Допустим, вы получили компонент от компании-разработчика hcwrecords.com. На этот компонент можно ссылаться и использовать его в программе, но кто может поручиться, что этот компонент действительно поступил от hcwrecords.com? Проще говоря, сильные имена анонимны, а любой хакер может присвоить своему изделию сильное имя.

На помощь приходит механизм пометки кода, хорошо знакомый авторам элементов ActiveX, «подписывавшим» свои разработки при помощи Authenticode. Некая третья сторона предоставляет вам открытый ключ разработчика компонента. Если вы доверяете третьей стороне, то можете быть уверены в том, что компонент поступил действительно от hcwrecords.com, а не от кого-то другого, маскирующегося под них[1].

Сильные имена и пометка кода не исключают друг друга и могут использоваться в сборке одновременно. В этом случае сначала применяются сильные имена.

Конфликты в пространствах имен

При выборе имен сборок нередко возникает путаница с пространствами имен. Обратитесь к главе 10, где эта тема рассматривается более подробно. Помните о том, что пространства имен общих сборок (тех, которые вы собираетесь установить в глобальном кэше сборок) должны быть уникальными. Для этого проще всего строить иерархию пространств имен от названия вашей компании. По этой причине названия всех пространств имен .NET Framework, созданных в Microsoft, начинаются с префиксов `System` и `Microsoft`.

[1] Я не буду углубляться в подробности. Вероятно, сейчас пометка кода будет использоваться значительно реже, чем при разработке элементов ActiveX (сильные имена обеспечивают защиту от модификации, которая была одной из важнейших особенностей Authenticode).

.NET и параллельное выполнение

Исполнительная среда Microsoft .NET рассчитана на поддержку двух типов параллельного выполнения: на одном компьютере и в одном процессе. Впрочем, текущая версия среды не поддерживает параллельного выполнения в одном процессе. В документации сказано, что эта возможность появится в окончательной версии.

Иначе говоря, два разных приложения могут одновременно использовать две разные версии сборки, но один процесс пока не может загрузить две версии сборки. Сказанное подтверждается проектами StrongDLL2A, StrongDLL2B и StrongApp2.

StrongDLL2B ссылается на версию 1.0 сборки StrongDLL2A. StrongApp2 содержит ссылку как на версию 1.0.0.1 StrongDLL2A, так и на StrongDLL2B. Следовательно, для работы программы в память должны быть одновременно загружены версии 1.0.0.0 и 1.0.0.1. Поскольку это пока невозможно, загрузка завершается неудачей, если только в файле StrongApp2.exe.config не указано, что версия 1.0.0.1 может быть принята в качестве обновления 1.0.0.0.

Параллельное выполнение должно учитываться при проектировании сборок. Например, если сборка использует жестко закодированные пути к файлам или именованные мьютексы (mutexes), следует помнить, что к этим объектам возможен одновременный доступ со стороны двух разных версий вашего кода.

Безопасность

В понятие «безопасность» часто вкладывается разный смысл. Под ним понимают:

- аутентификацию пользователей (локальную или по Интернету);
- защиту компьютера или сервера от разного рода внешних нападений;
- возможность запретить выполнение некоторых приложений;
- предотвращение вреда, наносимого выполнением злонамеренных программ;
- хранение конфиденциальной информации или поддержание защищенных каналов связи.

В общем, безопасность тоже заслуживает отдельной книги (наверное, вы уже привыкли к этой фразе). С другой стороны, каждый программист хотя бы в общих чертах разбирается в этой теме. Для начала стоит предположить, что средства безопасности, поддерживаемые в VB6 на программном уровне, могут быть реализованы и в .NET при помощи таких пространств имен, как `System.Security` и `System.DirectoryServices`.

Настоящий раздел посвящен одному аспекту безопасности — предотвращению вреда от злонамеренных программ. В этой области .NET Framework содержит ряд новых, чрезвычайно интересных возможностей, с которыми стоит познакомиться поближе.

Прощайте, сбои, прощайте, вирусы?

Одна из целей .NET Framework заключалась — ни больше, ни меньше — в устранении неустранимых сбоев приложений и защите системы от вирусов и других злонамеренных программ.

Вижу, как вы недоверчиво качаете головой: «Да, конечно — уже слышали...»

Ваш скептицизм вполне оправдан. Каждый раз, когда Microsoft объявляет о новой схеме безопасности, за этим неизбежно следует поток сообщений о дефектах защиты и поспешных исправлений. А в новых операционных системах, которые работают *еще* надежнее предыдущих, все равно возникают фатальные ошибки, от которых спасает только перезагрузка (хотите вы того или нет).

Пока трудно судить о том, каким будет истинный уровень защиты .NET и будет ли эта система устойчивее прежних, но после чтения этого раздела вы согласитесь с тем, что сама архитектура безопасности в .NET наглядно доказывает: в Microsoft очень серьезно относятся к этой проблеме. По крайней мере, это большой шаг, сделанный в нужном направлении.

Указатели и сбои

Каждый, кому доводилось программировать в эпоху 16-разрядных версий Windows, хорошо помнит «прелести» ошибок UAE (Unrecoverable Application Error, неустранимая ошибка приложения), позднее превратившихся в GPF (General[1] Protection Fault, общая ошибка защиты). В наши дни сбои приложений по-прежнему случаются, о чем обычно свидетельствует ошибка «Memory Exception» или внезапное исчезновение программы, но общесистемные сбои (полное «зависание» в Windows 98/ME или «посмертный синий экран» NT/2000) встречаются реже.

Почему возникают подобные сбои?

Потому что большинство приложений пишется на языках, использующих указатели, например на C++. При работе с указателем всегда существует вероятность того, что ему по какой-либо причине будет присвоено неправильное значение. Если вы забудете инициализировать указатель, он может ссылаться в несуществующую область памяти или в сегмент программного кода вашего приложения, что приведет к непреднамеренному стиранию кода[2]. Если ошибочный указатель будет ссылаться в область данных, он может испортить структуры данных (в том числе и другие указатели). Возможны и другие варианты, например случайная запись после конца приемного буфера с порчей данных в памяти.

Да, сбои могут возникать и по другим причинам, например из-за деления на ноль. Многие приложения нередко инициируют ошибки времени выполнения (случайно или намеренно), но самые неожиданные и зловредные ошибки связаны с применением указателей.

В 32-разрядных операционных системах Microsoft защита от большинства системных сбоев обеспечивается изоляцией процессов друг от друга и от операционной системы[3]. Тем не менее для .NET подобной изоляции процессов недостаточно. Как было сказано в главе 10, в .NET основной единицей является домен приложения, а не процесс. Домен приложения может соответствовать отдельному процессу, но в системах типа ASP .NET один процесс может содержать сотни и даже тысячи доменов приложений.

---

[1] В оригинале — Global Protection Fault. — *Примеч. перев.*

[2] В зависимости от типа ОС и указателя попытки записи в программный код либо приводят к модификации программы, либо инициируют исключение защиты памяти.

[3] В Windows 95/98/ME это утверждение не совсем верно, но для наших целей сойдет.

Если бы указатель одного из этих доменов приложений мог каким-то образом испортить данные другого домена (или ASP .NET), ни о какой устойчивой работе и речи быть не могло.

Напрашивается предположение, что проблема решается исключением указателей из языков .NET (таких, как VB .NET и C#), но как было показано в главе 15, даже VB .NET может использовать переменные-указатели `Int32Ptr` для работы с неуправляемым кодом.

Решение проблемы лежит в области безопасности типов — вспомните, что говорилось о неуправляемом коде в главе 15. Обмен данными с неуправляемым кодом осуществляется через объект `Marshal`, благодаря чему данные отправляются и принимаются в виде четко определенных структур данных. Другими словами, даже копирование данных в памяти выполняется с точным указанием расположения этих данных в памяти и сильной типизацией переменных. При правильном объявлении функций или структур CLR гарантирует, что данные будут переданы правильно, без загадочных переполнений или порчи данных.

Что касается других обращений к управляемой памяти, поскольку весь доступ осуществляется через объекты с сильной типизацией (см. главу 10), вы можете обеспечить *проверяемую* (verifiable) изоляцию доменов приложений. Говорят, что приложение обладает *проверяемой безопасностью типов*. В CLR предусмотрены средства для анализа кода и проверки безопасности типов. Не каждый язык .NET создает код с проверяемой безопасностью типов (программа может быть безопасной по отношению к типам, но не проверяемой — такой код генерируется в C++), но код VB .NET всегда должен удовлетворять этому условию. Безопасность типов сборки можно проверить при помощи утилиты `Preverify.exe`. В C# у вас есть возможность выбора, но ненадежный код не дает почти никаких (или просто никаких) преимуществ.

Зато у ненадежного кода есть один серьезный недостаток: для его выполнения требуется более высокая степень доверия. Вскоре я объясню, что это значит.

Злонамеренный код и вирусы

Программный код поступает в вашу систему из разных источников. Конечно, в первую очередь это программы, устанавливаемые с компакт-дисков или гибких дисков. Когда-то этим и ограничивались все ваши хлопоты, но в наши дни появились элементы и компоненты, загружаемые с web-страниц; сценарии, работающие на web-страницах; программы, принятые с web- и FTP-сайтов; файлы, присоединенные к сообщениям электронной почты, и даже макросы, скрытые в документах текстовых редакторов и электронных таблиц.

И любой фрагмент программного кода может вызвать полный хаос в вашей системе.

Почему? Потому что любая программа (кроме некоторых сценарных языков) при получении управления могла вытворять в вашей системе все, что угодно.

Операционные системы Windows NT/2000 обладают чрезвычайно ограниченными средствами безопасности на уровне приложений. Система может запретить пользователю или группе пользователей запускать некоторое приложение, но если приложение все же будет запущено, оно может делать практически все. Некоторые сценарные языки работают в так называемой «песочнице» (sandbox) —

виртуальной машине, жестко ограничивающей их возможности. Сценарий не может причинить особого вреда системе, но и лишается возможности выполнять многие полезные задачи.

В .NET Framework принят принципиально иной подход к безопасности на уровне приложений. В дополнение к хорошо знакомой вам ролевой безопасности (распределению прав в зависимости от идентификации пользователя), в .NET появилась так называемая безопасность программного доступа[1].

Общая идея заключается в следующем: каждая сборка работает в собственном контексте безопасности, определяемом информацией о сборке и политике безопасности, установленной в системе.

Если сборка обладает проверяемой безопасностью типов, CLR может следить за тем, какой код работает и что он делает (невозможно передать управление таким образом, чтобы об этом не стало известно CLR). Таким образом, CLR точно знает, когда программа вызывает тот или иной метод пространства имен. В свою очередь, это означает, что среда может установить надежные ограничения доступа на уровне объектов или методов.

CLR также умеет проходить по стеку и проверять безопасность не только кода, работающего в настоящий момент, но и каждой сборки, из которой он был вызван. Злонамеренным программам становится труднее выполнять свою грязную работу посредством других сборок.

Краткий обзор подходит к концу. Давайте поближе познакомимся с безопасностью .NET и возможностями ее практического использования.

Сборки и политики безопасности

Первое, что необходимо знать о безопасности программного доступа: все решения из области безопасности в .NET принимаются на основании информации о сборке, имеющейся у среды.

Информация о сборке

А что .NET знает о сборке? Точнее, какая информация о сборке может храниться в системе?

- Издатель сборки (если последняя обладает цифровым сертификатом).
- Web-сайт, пытающийся выполнить сборку (при вызове через обращение к web-странице).
- Сильное имя сборки (обеспечивает уникальную идентификацию сборки).
- URL (web-сайт или FTP-сайт), с которого была принята сборка.
- Зона Интернета, из которой была принята сборка. Зоны определяются в настройках Internet Explorer — Интернет, местная интрасеть, надежные и ограниченные узлы, а также локальный компьютер.

[1] Ролевая безопасность в .NET реализована весьма гибко. Помимо идентификации на основании учетных записей пользователей и групп вы можете определять нестандартные роли. Например, можно построить базу данных пользователей и ролей, независимую от системных данных, и использовать ее для принятия решений из области безопасности. Тем не менее общие принципы ролевой безопасности не изменились, поэтому мы не будем подробно рассматривать эту тему.

- Признак хранения сборки в одном каталоге с запустившим ее приложением (или в одном из его подкаталогов).

  Рассмотрим некоторые возможные ситуации.
- Проверка компонентов ASP может быть основана на сайте, с которого поступил вызов.
- CLR точно знает, откуда поступает весь принятый код. Допустим, вы не сомневаетесь в надежности сборок, полученных из местной интрасети и от Microsoft.com. Сборки, полученные из этих источников, объявляются доверенными.
- Привилегии можно регулировать избирательно, например запретить всему коду с указанного web-сайта модификацию реестра или запись на диск.
- При установке сборки в системе .NET сохраняет все доступные сведения о ней и использует их при определении политики безопасности для данной сборки.

Привилегии

Объекты пространства имен `System.Security.Permissions` определяют уровень привилегий, то есть разрешений на доступ к защищенным ресурсам или выполнение привилегированных операций. Вы можете определить собственный объект привилегий, но эта тема выходит за рамки данной книги.

Таблица 15.1. Классы привилегий

Объект	Операция
DirecttoryServicesPermission	Объекты пространства имен System.DirectoryServices
DnsPermission	Операции DNS
EnvironmentPermission	Переменные окружения
EventLogPermission	Журнал событий
FileDialogPermission	Выбор файлов в стандартном диалоговом окне File Open
FileIOPermission	Доступ к каталогам и файлам
IsolatedStorageFilePermission	Закрытые виртуальные файловые системы
IsolatedStoragePermission	Изолированное хранилище — новая разновидность хранения данных в .NET, ассоциируемая с конкретным кодом
MessageQueuePermission	Microsoft Message Queue (MSMQ)
OleDbPermission	Операции OLE DB
PerformanceCounterPermission	Счетчики быстродействия
PrintingPermission	Операции с принтером
ReflectionPermission	Операции рефлексии в .NET
RegistryPermission	Операции с системным реестром
SecurityPermission	Различные привилегии, связанные с безопасностью программного доступа

продолжение

Таблица 15.1 (продолжение)

Объект	Операция
ServiceControllerPermission	Службы Windows
SocketPermission	Операции с сокетами
SQLClientPermission	Операции с базами данных SQL
UIPermission	Функции пользовательского интерфейса
WebPermission	Web-операции (прием или отправка)

Для управления безопасностью операций используются методы соответствующих объектов привилегий. Эта тема более подробно рассматривается ниже.

Наборы привилегий

Совокупность привилегий называется *набором* (set). Скажем, набор привилегий Интернета определяет привилегии, типично используемые для программного кода, принятого из Интернета, а набор привилегий «с полным доверием» (full trust) позволяет программе выполнять любые операции.

Помните, что набор привилегий — всего лишь средство логической группировки привилегий. Вы можете присвоить любой кодовой группе произвольный набор привилегий, определять новые наборы привилегий и даже изменять состав привилегий, входящих в стандартные наборы.

Политики

Весь код в .NET относится к одной или нескольким кодовым группам (code groups) на основании сведений, доступных для данного кода. Например, весь код относится к группе «All code», а код с цифровой сигнатурой Microsoft принадлежит к группе «Published by Microsoft». Кодовая группа даже может состоять из одной сборки, заданной по сильному имени. Кодовые группы определяются в виде иерархии, возглавляемой группой «All code».

В каждой системе могут действовать до трех политик: для всей организации, для данного компьютера и для конкретного пользователя.

Допустим, сборка, работающая в браузере, была принята с сайта BadCodeSite.bad. На основании имеющихся сведений она принадлежит к двум кодовым группам, All Code (Весь код) и Internet Zone (Зона Интернета).

В соответствии с политикой компьютера для зоны Интернета, скорее всего, будет запрещена запись на диск, обращения к локальным папкам электронной почты, обращение к реестру и выполнение Интернет-операций. Любые попытки выполнения опасных операций сборкой приведут к инициированию исключения.

Но что, если код заверен сигнатурой доверенного сайта HCWrecords.com? В этом случае он может принадлежать к группе HCWRecords. Если привилегии этой группы разрешают запись на диск или выполнение других защищенных операций, эти права будут предоставлены сборке.

Другими словами, на уровне политики используется правило «минимального ограничения». Если принадлежность кода к какой-либо кодовой группе приводит к предоставлению некоторой привилегии, данная привилегия предоставляет-

ся всей сборке. Кроме того, возможна настройка, обеспечивающая прекращение поиска на конкретной кодовой группе или уровне политики.

Фактически предоставляемые полномочия определяются минимальным уровнем по трем политикам. Например, если бы в приведенном примере политика организации запрещала бы некоторую привилегию кодовой группе HCWRecords, запрет действовал бы даже в том случае, если бы эта привилегия разрешалась в соответствии с политиками компьютера и пользователя.

Политики применяются в следующем порядке: сначала политика организации, затем политика пользователя, после чего политика компьютера.

В дополнение к этим трем группам каждая сборка также проверяется на соответствие параметрам безопасности домена приложения.

Вероятно, мои объяснения могут показаться невразумительными. Почитайте документацию Microsoft — загляните в .NET Framework SDK, раздел «Programming with the .NET Framework\Securing your Application». Там все объясняется подробнее, но ясность изложения, мягко говоря, оставляет желать лучшего. Надеюсь, из чтения этой главы вместе с документацией вы получите нормальное представление о безопасности в .NET.

А пока давайте рассмотрим конкретный пример, который покажет, как механизмы безопасности работают на практике.

Безопасность в примерах

Ничто не проясняет сложную тему так, как практический пример. Надеюсь, последующие примеры покажутся вам достаточно интересными.

Конфигурация и трассировка стека

Решение CallUnmanaged1 содержит два проекта, CallUnmanaged1 и UnmanagedClass. Определение класса UnmanagedClass выглядит так:

```
Public Class Class1
  Private Declare Auto Function GetTickCount Lib "kernel32" () As Integer
  Public Function Ticks() As Integer
    Return GetTickCount()
  End Function
End Class
```

Перед нами простой пример вызова функции API через объект-обертку. Мы знаем, что функция совершенно безвредна, но CLR этого не знает. Поскольку функция API может выполнить много опасных операций в обход .NET Framework (а следовательно, и стандартных объектов привилегий, о которых говорилось выше), выполнение неуправляемого кода требует высокого доверия.

Например, привилегию выполнения неуправляемого кода никогда не следует предоставлять программам, принятым из Интернета, если только у вас нет очень веских оснований доверять им.

В этом заключается одна из главных причин, по которым использовать функции Win32 API в VB .NET не рекомендуется: они значительно снижают мобильность ваших программ. Например, функции API не могут использоваться в элементах, которые должны запускаться в браузерах по Интернету. Скорее всего, такие попытки завершатся неудачей.

Итак, наша конкретная сборка была успешно создана и установлена в системе, а поскольку она принадлежит кодовой группе локальной системы, пользуется полным доверием. Следовательно, при вызове из приложения Windows она будет нормально работать. Но что произойдет, если вызвать ее из сборки, принятой из Интернета?

Открытая функция `Ticks` сама по себе не использует неуправляемого кода. Сможет ли сборка, не обладающая привилегий выполнения неуправляемого кода, воспользоваться функцией `Ticks` и обмануть класс `UnmanagedClass.Class1`, заставив его вызвать функцию API?

Давайте посмотрим.

Консольное приложение CallUnmanaged1 создает объект класса `Class1` и вызывает метод `Ticks`:

```
' Вызов метода класса с неуправляемым кодом
' Copyright ©2001 by Desaware Inc. All Rights Reserved
Imports UnmanagedClass
Module Module1

  Sub Main()
    Dim c As New Class1()
    Dim x As Integer
    For x = 0 To 100
      Console.WriteLine (c.Ticks)
    Next
    Console.ReadLine()
  End Sub

End Module
```

Если построить приложение CallsUnmanaged1 и запустить его, оно будет работать, поскольку это приложение тоже принадлежит к кодовой группе локальной системы. Чтобы проверить систему безопасности, необходимо изменить набор привилегий для этого приложения. Запустите программу mmc.exe (Microsoft Management Console) и добавьте в нее модуль конфигурации .NET командой Add/Remove snap-in[1].

На рис. 16.1 показано окно настройки системы безопасности с развернутой политикой компьютера (Machine). Из рисунка видно, что в политике уровня компьютера определено несколько кодовых групп и наборов привилегий, присваиваемых кодовым группам. На рисунке перечислен состав набора привилегий для Интернета.

Наша задача — включить приложение CallsUnmanaged1 в новую кодовую группу. Для этого следует щелкнуть правой кнопкой мыши в группе My_Computer_Zone (поскольку сборка является подмножеством этой группы) и определить в ней новую подгруппу.

Мастер предлагает ввести имя и описание кодовой группы. Введите любое имя по своему желанию.

---

[1] Настройка системы безопасности также выполняется утилитой caspol. Более того, системные конфигурационные файлы можно редактировать вручную в текстовом редакторе — они хранятся в простом формате XML.

Далее запрашивается условие принадлежности к этой группе. Здесь указывается тип сведений, используемых для идентификации сборок, входящих в данную группу. Укажите идентификацию по сильному имени (Strong-Name).

Затем вам будет предложено ввести открытый ключ, имя и сведения о версии для идентификации сборки. Для этого проще всего щелкнуть на кнопке Import и импортировать нужную информацию из исполняемого файла CallsUnmanaged1.exe. Не забудьте установить флажки Name и Version, иначе вы измените параметры безопасности для всех сборок, использующих ваш открытый ключ!

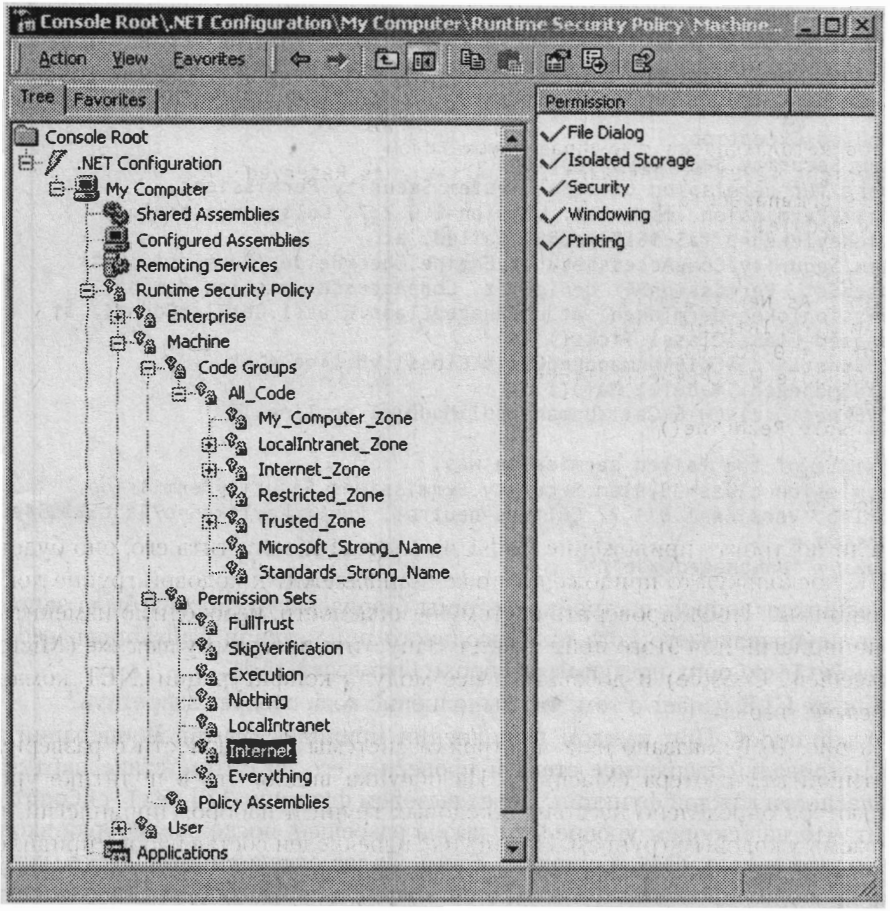


Рис. 16.1. Настройка политики безопасности в окне MMC

Далее мастер запрашивает набор привилегий, назначаемый данной группе. Выберите набор Internet.

На следующем этапе вам предоставляется возможность установить флажки Exclusive и Final level. Флажок Exclusive сообщает системе безопасности о том, что указанный набор привилегий должен применяться ко всем сборкам кодовой группы независимо от того, принадлежат ли они при этом другим группам или нет. Флажок Final level указывает на то, что проверка должна завершаться на этом уровне.

В приведенном примере установка этого флажка приведет к тому, что поиск на уровне User производиться не будет.

Установите флажок Exclusive. В данном случае это существенно, поскольку локальный код может присутствовать и в других группах, а в пределах уровня действует правило «минимального ограничения».

Щелкните правой кнопкой мыши на узле Runtime Security Policy и выберите команду Save All, чтобы сохранить внесенные изменения.

Если все было сделано верно, то с точки зрения CLR приложение CallsUnmanaged1 обладает привилегиями, типичными для приложений, принятых из Интернета.

Что же произойдет при попытке запуска этого приложения?

При запуске из командной строки будет выведена следующая информация[1]:

```
Unhandled Exception:
System.Security.SecurityException:
Request for permission of type "System.Security.Permissions.
SecurityPermission, mscorlib, Version=1.0.?.?, Culture=neutral,
PublicKeyToken=b77a5c561934e089" failed. at
System.Security.CodeAccessSecurityEngine.CheckHelper(PermissionSet
grantedSet, PermissionSet deniedSet, CodeAccessPermission demand,
PermissionToken permToken) at UnmanagedClass.Class1.GetTickCount() at
Unmanaged-Class.Class1.Ticks() in
D:\CPBknet\Src1\CH16\UnmanagedClass\Class1.vb:line 4 at
CallsUnmanaged1.Module1.Main() in
D:\CPBknet\Src1\CH16\CallsUnmanaged1\Module1.vb:line 10

The state of the failed permission was:
<IPermission class="System.Security.Permissions.SecurityPermission,
mscorlib, Version=1.0.?.?, Culture=neutral, PublicKeyToken=b77a5c561934e089"
  version = "1"
  Flags = "UnmanagedCode"/>
```

Возникает вопрос: настройка системы безопасности должна была запретить вызов неуправляемого кода из консольного приложения CallsUnmanaged1. Но вызов `GetTickCount` поступил из сборки UnmanagedCall.dll.

Как же CLR узнает о том, что выполнение кода следует запретить?

Ответ прост. При каждом выполнении привилегированной операции CLR просматривает содержимое стека и проверяет его на соответствие параметрам безопасности каждой функции. Перед вызовом функции API `GetTickCount` CLR видит, что на текущем уровне этот вызов разрешен, поскольку сборка входит в локальную группу (My_Computer_Zone). Далее следует метод `Ticks`, который тоже проходит проверку, поскольку он входит в ту же сборку. Следующая проверка относится к процедуре `Main` приложения CallsUnmanaged1. На этот раз результат оказывается отрицательным, поскольку сборка не обладает привилегиями на выполнение неуправляемого кода.

Как видите, CLR отлично справляется с идентификацией программного кода и поддержанием защиты. Параметры безопасности кода могут выбираться в зависимости от степени доверия. Более того, с точки зрения программиста появляется возможность корректного сокращения функциональности кода за счет пере-

[1] Номера версии, образцы ключей и некоторые второстепенные подробности могут отличаться.

хвата ошибок безопасности и отключением некоторых возможностей в менее защищенных средах.

А как быть в случаях, когда при выполнении заведомо безопасной операции требуется, чтобы операция могла выполняться менее защищенными сборками (при условии, что ваш компонент пользуется доверием)?

Методы Demand и Assert

В решение CallUnmanaged2 входят два проекта: CallUnmanaged2 и UnmanagedClass2.

Прежде чем переходить к экспериментам, постройте оба проекта и при помощи конфигурационной программы MMC настройте приложение CallsUnmanaged2 на использование привилегий Интернета, как это было сделано ранее для программы CallsUnmanaged1.

Класс UnmanagedClass2 определяет метод `GetTickCount`, как и в предыдущей версии, но новый вариант функции `Ticks` выглядит несколько иначе.

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

Public Class Class1
  Private Declare Auto Function GetTickCount Lib "kernel32" () As Integer

<SecurityPermission(SecurityAction.Demand, _
  Flags:=SecurityPermissionFlag.UnmanagedCode)>
Public Function Ticks() As Integer
  Return GetTickCount()
End Function
```

Атрибут `SecurityPermission` с командой `Demand` сообщает CLR, что для выполнения кода необходимы привилегии выполнения неуправляемого кода. Установка подобных требований удобна тем, что вам уже не придется перехватывать отдельные ошибки времени выполнения внутри метода. Вызывающая сторона должна обладать необходимыми привилегиями даже для простого вызова метода. Применение подобных атрибутов обеспечивает так называемую «декларативную безопасность», поскольку требования декларируются при объявлении метода.

Включение следующего фрагмента в файл assemblyinfo.vb запрещает загрузку сборки, если она не соответствует указанным минимальным требованиям к наличию привилегий.

```
<Assembly: SecurityPermission(SecurityAction.RequestMinimum, _
  Flags:=SecurityPermissionFlag.UnmanagedCode)>
```

Сборка UnmanagedClass2 содержит несколько тестовых функций, вызываемых из программы CallsUnmanaged2. Функции замеряют время выполнения 50 000 вызовов `Ticks` и возвращают результат в виде объекта `TimeSpan`. Значение в миллисекундах выводится в консольном окне.

Функция `Ticks2`, приведенная в листинге 16.1, показывает, как создать объект `SecurityPermission` для конкретной привилегии на стадии выполнения.

Листинг 16.1. Метод UnmanagedClass2.Class1.Ticks2[1]

```
Public Function Ticks2() As TimeSpan
  Dim x As Integer
  Dim l As Long
  Dim t As New DateTime()
  Dim ts As TimeSpan

  Dim sec As New SecurityPermission(SecurityPermissionFlag.UnmanagedCode)

  sec.Assert()

  t = DateTime.Now
  For x = 0 To 50000
    l += Ticks()
  Next
  ts = DateTime.Now.Subtract(t)
  CodeAccessPermission.RevertAssert()

  Try
    Ticks()
  Catch e As Security.SecurityException
    MsgBox ("Caught the security exception after revert!")
  End Try
  Return ts
End Function
```

Вызов `Assert` сообщает CLR о том, что данному коду разрешается выполнять неуправляемый код. Поскольку сборка пользуется полным доверием, ей позволено отдавать подобные указания. В процессе проверки стека CLR достигает блока, в котором вызывается метод `Assert`, прекращает дальнейшую проверку и разрешает выполнение кода.

Разумеется, к вызову метода `Assert` необходимо подходить очень осторожно. Используйте его лишь в том случае, если вы абсолютно уверены в невозможности злонамеренного использования вашего кода.

Последствия вызова `Assert` отменяются немедленно после возвращения из функции. Все вызовы `Assert`, действующие для функции, можно отменить при помощи общего метода `CodeAccessSecurity.RevertAssert`.

Приведенная выше функция также показывает, как перехватывать ошибки безопасности. Попытка вызова `Ticks` завершается неудачей, поскольку в объявлении метода указан атрибут безопасности `Demand` (а последствия предыдущего вызова `Assert` были отменены).

ПРИМЕЧАНИЕ

В зависимости от быстродействия компьютера сообщение может появиться через одну-две минуты.

Поскольку функция API `GetTickCount` всегда безопасна, запись можно сократить прямым включением вызова `Assert` в объявление функции API:

```
<SecurityPermission(SecurityAction.Assert, _
  Flags:=SecurityPermissionFlag.UnmanagedCode)> _
  Private Declare Auto Function GetTickCountSpecial Lib "kernel32" _
  Alias "GetTickCount" () As Integer
```

[1] Все исходные тексты можно найти на сайте издательства «Питер» www.piter.com. — *Примеч. ред.*

Данное объявление следует использовать лишь для вызова абсолютно безопасных функций API, а также для закрытых функций, которые вызываются абсолютно безопасным способом.

В методе `Ticks3` (листинг 6.2) продемонстрирован распространенный способ применения метода `Assert`. Вместо того чтобы вызвать `Assert`, вы сначала используете `Demand` для проверки другой привилегии, обеспечивающей по возможности более жесткие ограничения. В данном примере вместо безопасности программного доступа используется ролевая безопасность. Вызов `Demand` гарантирует, что неуправляемый код будет вызываться лишь членами локальной административной группы.

Листинг 16.2. Метод UnmanagedClass2.Class1.Ticks3

```
Public Function Ticks3() As TimeSpan
  Dim x As Integer
  Dim l As Long
  Dim t As New DateTime()
  Dim ts As TimeSpan

  Dim sec2 As New _
    SecurityPermission(SecurityPermissionFlag.ControlPrincipal)
  sec2.Assert()
  AppDomain.CurrentDomain.SetPrincipalPolicy _
    (PrincipalPolicy.WindowsPrincipal)
  Dim roleSec As New PrincipalPermission(Nothing, _
    "BUILTIN\Administrators")
  roleSec.Demand()

  AppDomain.CurrentDomain.SetPrincipalPolicy _
    (PrincipalPolicy.UnauthenticatedPrincipal)
  CodeAccessPermission.RevertAssert()

  t = DateTime.Now
  For x = 0 To 50000
    l += GetTickCountSpecial()
  Next
  ts = DateTime.Now.Subtract(t)

  Try
    GetTickCountSpecial()
  Catch e As Security.SecurityException
    MsgBox ("Caught the security exception after revert!")
  End Try
  Return ts
End Function
```

Сначала функция методом `Assert` разрешает доступ к параметрам роли пользователя для домена приложения, включая идентификационные данные, сведения о принадлежности к группам и т. д. По умолчанию эти данные недоступны, поскольку с их помощью программа может идентифицировать пользователя, выполняющего код, что создает потенциальную брешь в системе безопасности.

После предоставления этих привилегий для текущего домена приложения создается объект `PrincipalPermission` для встроенной административной группы `Administrators`. Метод `Demand` этого объекта инициирует ошибку времени выполнения в том случае, если текущий пользователь не является администратором.

Далее функция отменяет последствия вызова `Assert` и восстанавливает параметры домена приложения в более защищенном состоянии.

Наконец, функция 50 000 раз вызывает функцию `GetTickCountSpecial`. Как говорилось выше, эта функция содержит собственный вызов `Assert`.

Нетрудно заметить, что эта функция работает значительно быстрее предыдущей. Чем раньше в цепочке вызовов выполняется `Assert`, тем меньше времени тратится на перебор стека.

В класс также включен метод `Ticks4`, объявленный без `Assert`. Попытка его вызова из приложения CallUnmanaged2 завершилась бы неудачей. Метод `Ticks4` должен вызываться из проекта CallsUnmanaged2b, практически идентичного CallsUnmanaged2 за одним исключением: ему оставлены параметры безопасности по умолчанию (полное доверие). Воспользуйтесь этим проектом, чтобы получить представление о том, как проверки безопасности влияют на скорость работы приложения.

Дополнительные средства безопасности

Существует ряд дополнительных средств, используемых при обеспечении безопасности. Ниже перечислены важнейшие примеры.

- Помимо метода `Assert` существует и метод `Deny`. Используется в тех ситуациях, когда вы уверены в абсолютной безопасности своего кода и хотите гарантировать, что в нем заведомо не могут выполняться некоторые операции.
- Ограничение привилегий определенным набором даже в том случае, если уровень привилегий сборки, от которой поступило обращение, превышает необходимый минимум.
- Запрос привилегий в процессе загрузки с разграничением минимальных привилегий, необходимых для работы сборки, и необязательных привилегий, используемых в случае их доступности. Привилегии даже могут отклоняться, что помешает предоставить их вашей сборке.

Я хочу особо выделить некоторые ключевые моменты. Пожалуйста, отнеситесь к ним серьезно.

Применение `Assert` *связано с риском!*

Не используйте `Assert`, если у вас нет полной уверенности в том, что компонент не может использоваться некорректно.

Не используйте `Assert` без абсолютной необходимости. Если такая необходимость существует, сведите предоставление привилегий к минимуму при помощи `Demand`.

Используйте декларативную безопасность для корректного сокращения функциональности, если ваш код не обладает достаточным уровнем привилегий для выполнения.

Используйте минимальное количество привилегий в своих программах, чтобы повысить мобильность кода и упростить его распространение.

Разное

В этом разделе собраны все второстепенные темы, которые мне не удалось пристроить в других главах.

Дизассемблирование

Представьте, что один из ваших программистов создал изощренное решение некой задачи и включил в свой код информацию, которую вы считаете своей интеллектуальной собственностью. Программа выходит в свет, и в один прекрасный день вы получаете по электронной почте следующее сообщение.

Привет, я немножко покопался в сборке, распространяемой с вашим суперприложением Emailer 2000. Результаты весьма любопытные. Например, взглянем на алгоритм VerifyEmail из следующего листинга, полученного дизассемблированием вашего продукта:

```
// Microsoft (R) .NET Framework IL Disassembler. Version 1.0.2914.16
// Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

.namespace Disasm
{
  .class /*02000002*/ public auto ansi Emailer
    extends [mscorlib/*23000001*/]System.Object/* 01000001 */
  {
    method /*06000002*/ public instance bool
    VerifyEmail(string Email) cil managed
    {
      // Code size    20 (0x14)
      .maxstack 3
      .locals init (bool V_0)
      IL_0000: ldarg.1
      IL_0001: ldstr  "@" /* 70000001 */
      IL_0006: ldc.i4.0
      IL_0007: call   int32[Microsoft.VisualBasic/* 23000002
      */]Microsoft.VisualBasic.Strings/* 01000002 */::
      InStr(string,string,valuetype[Microsoft.VisualBasic
      /* 230000002 */]Microsoft.VisualBasic.CompareMethod
      /* 01000003 */) /* 0A000002 */
      IL_000c: ldc.i4.0
      IL_000d: ble.s  IL_0012

      IL_000f: ldc.i4.1
      IL_0010: br.s   IL_0013

      IL_0012: ldloc.0
      IL_0013: ret
    } // end of method Emailer::VerifyEmail
```

Вполне очевидно, что вся проверка сводится к проверке наличия символа @ в адресе электронной почты. Наверное, вы не читали собственной рекламы с утверждениями о том, что ваши программисты запатентовали новый алгоритм для надежной проверки адресов?

А ваша надежная функция проверки хоста

```
  .method /* 06000003 */public instance string
    GetMyHost() cil managed
```

```
  {
    // Code size    16 (0x10)
    .maxstack  1
    .locals init (string V_0)
    IL_0000: ldarg.0
    IL_0001: callvirt instance int32 Disasm.Emailer
    /* 02000002 */::Delay()/* 06000004 */
    IL_0006: pop
    IL_0007: call    string [System/* 23000003 */]System.Net.Dns
    /* 01000004 */::GetHostName()/* 0A000003 */
    IL_000c: br.s    IL_000f

    IL_000e: ldloc.0
    IL_000f: ret
  } // end of method Emailer::GetMyHost
```

всего лишь возвращает значение, полученное функцией .NET Framework System.Net. Dns.GetHostName.

Кроме того, я обнаружил в вашей программе функцию Delay:

```
  .method /*06000004*/private instance int32
  Delay() cil managed
  {
    // Code size    48 (0x30)
    .maxstack  2
    .locals init (int32 V_0,
      int64 V_1,
      int64 V_2)
    IL_0000: ldc.i8   0x1
    IL_0009: stloc.1
    IL_000a: ldloc.2
    IL_000b: ldc.i8   0x1
    IL_0014: add.ovf
    IL_0015: stloc.2
    IL_0016: ldloc.1
    IL_0017: ldc.i8   0x1
    IL_0020: add.ovf
    IL_0021: stloc.1
    IL_0022: ldloc.1
    IL_0023: ldc.i8   0xf4240
    IL_002c: ble.s  IL_000a

    IL_002e: ldloc.0
    IL_002f: ret
  } // end of method Emailer::Delay

} // end of class Emailer

} // end of namespace Disasm

//*********** DISASSEMBLY COMPLETE **********************
// WARNING: Created Win32 resource file
D:\MovingToVBNet\Source\CH16\Disasm\Disasm.res
```

Цикл? Вы организуете задержку при помощи цикла? Неужели ваши программисты никогда не слышали о таймерах с ожиданием? Честно говоря, я поражен. Пожалуй, я выложу результаты в Интернете и посмотрю, что об этом скажут другие.

Ужас.

На эту тему шли бурные дебаты. Некоторые разработчики критиковали .NET за простоту дизассемблирования (использованный в приведенном примере дизассемблер ildasm.exe входит в пакет .NET SoftwareDevelopment Kit). Разработчики коммерческих компонентов требовали, чтобы компания Microsoft по крайней мере выпустила утилиту, которая бы переименовывала переменные, параметры, методы и т. д., затрудняя тем самым дизассемблирование и анализ сборок.

Я тоже считаю, что Microsoft следует выпустить такую утилиту, но прежде чем впадать в панику, следует учесть некоторые обстоятельства.

Данная проблема присуща не только .NET, она встречается и на других платформах, построенных на базе виртуальных машин.

Сведения, используемые .NET для предотвращения конфликтов версий, представляют интерес для охотников за информацией. JIT-компилятор должен располагать полной информацией о классах, методах и параметрах.

Когда-то я работал в компании, занимавшейся анализом сбоев в полупроводниках. Некоторые клиенты пользовались нашими услугами, чтобы узнать, как работают микросхемы конкурентов. Если не жалеть денег, можно восстановить исходную схему любого устройства или программы.

На ранних порах существования Visual Basic было много разговоров об одной немецкой компании, разработавшей дизассемблер VB. В итоге эта тема забылась. Откровенно говоря, большинство компаний не боится дизассемблирования. Дело в том, что никто не сможет дизассемблировать ваш код, собрать его заново под исходным именем и заменить ранее существовавший модуль: этому помешают сильные имена, о которых говорилось выше.

Думаю, относительная простота дизассемблирования приложений .NET не вызовет особых проблем, по крайней мере у подавляющего большинства разработчиков. Я упоминаю об этом лишь для того, чтобы вы учитывали такую потенциальную возможность (и еще потому, что это мой последний шанс высказаться на спорную тему).

Установка

Время и место не позволяют мне рассмотреть проблемы установки в этой книге. Вряд ли кто-нибудь возьмется за отдельную книгу по этой теме, но она несомненно заслуживает отдельной главы — которую я, к сожалению, уже не напишу (по крайней мере в этом издании).

Впрочем, не могу не поделиться одной мыслью.

За годы эволюции система Windows прошла большой путь, становясь все более и более сложной. Программы установки тоже становились все более сложными и интеллектуальными.

С появлением .NET система Windows достигла такого совершенства, что она позволяет установить целое приложение простым копированием программы и всех используемых файлов в каталог простой командой Copy или XCopy.

Точно так же, как это было в MS-DOS...

Итоги

Глава посвящена двум важнейшим областям, в которых должен разбираться каждый программист VB .NET. Сначала мы рассмотрели контроль версий и применение сильных имен для создания сборок, привязанных к конкретным версиям зависимых компонентов, и борьбы с фальсификацией и модификацией кода. Вы узнали, как при помощи конфигурационных файлов разрешить обновление некоторых сборок.

Далее рассматривалась тема безопасности в .NET и безопасность программного доступа — весьма важное нововведение, повышающее мобильность кода без снижения уровня защиты. Вы научились пользоваться средствами системы безопасности для выполнения привилегированных операций (если ваша сборка обладает достаточными правами для подобных запросов) и корректного сокращения функциональности в ситуациях, когда программа не имеет прав на выполнение некоторых операций, используемых в работе сборки.

Однако не стоит забывать, что возможности, предоставляемые .NET для управления безопасностью на уровне методов (и даже внутри методов), могут применяться и в области ролевой безопасности и что вы можете определять роли по своему усмотрению, не ограничиваясь стандартными учетными записями пользователей и групп, определяемыми на уровне операционной системы.

Глава завершается кратким упоминанием проблем дизассемблирования и установки программ.

Заключение

Говорят, лучший способ научиться чему-то — учить других.

Наверное, это правда.

Я написал эту книгу по двум причинам. Во-первых, я четко представляю, что нужно знать нынешнему программисту VB6 для изучения .NET. Я уже несколько лет использую некоторые из новых средств (в частности, наследование и многопоточность) и не сомневаюсь, что именно в этой области у многих программистов возникнут проблемы. В этой книге я постарался поделиться своим опытом и надеюсь, что мне это удалось.

Кроме того, я знал, что работа над книгой заставит меня серьезно заняться .NET и толком разобраться в этой среде. Я из той породы программистов, которые хотят не только что-то делать, но и понимать, что при этом происходит, как и почему. Несомненно, изучение базовых концепций, на которых основан язык (в данном случае это .NET Framework), повысило мою квалификацию программиста, а долгие часы, в течение которых я разбирался с некоторыми проблемами, вызванными то ли ошибками с моей стороны, то ли дефектами самой среды или языка, дали мне желанный практический опыт работы с .NET.

Теперь этот опыт передался и вам.

Надеюсь, наше путешествие в мир .NET окажется для вас таким же познавательным, как для меня в роли вашего проводника.

Дэн Эпплман
Май 2001 г.

Алфавитный указатель

N

O

P

R

S

З

И

К

Л—М

P

C

Т

У

Ф

Ц—Я

Дан Эпплман

БИБЛИОТЕКА ПРОГРАММИСТА

Переход на VB .NET

стратегии, концепции, код

Эта книга является описанием .NET, призванным ознакомить читателя с основными идеями новой архитектуры.
Она поможет вам взглянуть на эту технологию с позиций собственных рабочих потребностей и быстро освоить те концепции, которые покажутся необычными для большинства программистов Visual Basic.
Широко известный и уважаемый в компьютерном мире автор постарался излагать материал как можно более сжато и в то же время достаточно глубоко, чтобы помочь вам в освоении тех областей архитектуры .NET, которые представляют наибольший практический интерес.

- основные аспекты .NET
- критерии необходимости перехода на .NET
- концепции Visual Basic .NET
- управление памятью
- многопоточность
- общие сведения о пространствах имен
- особенности программирования на VB в рамках новой архитектуры
- изменения в новой версии Visual Basic

и многое другое...

Уровень пользователя:
опытный/эксперт
Серия:
библиотека программиста

ISBN 5-318-00746-5

ПИТЕР®
WWW.PITER.COM

a!
Apress