*Уиллу, Хантеру и Баффи, терпеливо ждавшим, когда их папа закончит свои главы.*

*Б. Дж.*

*Моим родителям, Элу и Джэн, моему брату Брайану и всем моим друзьям.*

*К. С.*

*Джулии и Максу — пришло время вместе вписать новые главы в нашу жизнь.*

*М. Я.*

**Brian Johnson**
**Craig Skibo**
**Marc Young**

# Inside Microsoft® VISUAL STUDIO .NET

Microsoft Press

Брайан Джонсон
Крэйг **Скибо**
Марк Янг

# Основы Microsoft® VISUAL STUDIO® .NET 2003

**Москва** 2003

РУССКИ РЕДАКЦИЯ

**УДК 004.43**
**ББК 32.973.26-018**
**Ш53**

**Microsoft Corporation**

Ш53 Основы Microsoft Visual Studio .NET 2003 /Пер. с англ. - М.: Издательско-торговый дом «Русская Редакция», 2003. — 464 стр.: ил.

**ISBN 5-7502-0225-9**

В книге описаны тонкости работы с последней версией Microsoft Visual Studio .NET. Читатель познакомится с возможностями программного управления решениями, проектами, редакторами, дизайнерами и другими компонентами этой среды разработки. Каждая глава снабжена примерами макросов и надстроек, которые позволят увеличить продуктивность разработчиков всех видов ПО на любых языках программирования, поддерживаемых Visual Studio .NET 2003.

Книга состоит из 15 глав, приложения и предметного указателя.

**УДК 004.43**
**ББК 32.973.26-018**

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

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

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

© **Оригинальное** издание на английском языке, Microsoft Corporation, **2003**

© Перевод на русский язык, Microsoft Corporation, 2003

© Оформление и **подготовка к** изданию, издательско-торговый дом **«Русская** Редакция», **2003**

ISBN 0-7356-1874-7 (англ.)

ISBN 5-7502-0225-9

# Оглавление

**Благодарности** XII

**Введение** XIII

Для кого эта книга ... XIII
Структура книги ... XIV
Сопроводительный материал ... XIV
Системные требования ... XV
Контакт с авторами ... XV
Служба поддержки Microsoft Press ... XV

ЧАСТЬ I
VISUAL STUDIO .NET КАК ИНСТРУМЕНТ РАЗРАБОТКИ 1

**Глава 1 Эволюция Visual Studio .NET** 2

Путь к Visual Studio .NET ... 2
Разработка для .NET ... 3
Расширения для разработки неуправляемого кода ... 5
Новые опции компилятора ... 5
Обновленные библиотеки классов ... 7
Атрибуты C++ ... 7
Соответствие стандартам ... 7
Новая интегрированная среда разработки (IDE) ... 8
Возможности Visual Studio .NET ... 8
Редакторы, дизайнеры и окна инструментов ... 8
Пути к файлам в Visual Studio .NET ... 16
Возможности расширения Visual Studio .NET ... 19
Макросы ... 19
Надстройки ... 19
Мастера ... 20
Программа интеграции с Visual Studio (VSIP) ... 21
Что дальше ... 21

**Глава 2 Управление проектами в Visual Studio .NET** 22

Обзор решений и проектов ... 22
Решения ... 24
Элементы решения и дополнительные файлы ... 24
Свойства решений ... 26
Файл решения и файл параметров пользователя ... 27
Проекты ... 31
Элементы проекта ... 31
Свойства проекта ... 32
Исходные файлы проекта ... 40
Зависимости проекта ... 45
Компоновка проектов и решений ... 46
Что дальше ... 47

**Глава 3 Редактор Visual Studio .NET 48**

Документы в IDE 48
Все о тексте 49
Набор текста и быстрые клавиши 51
Философия отступов и форматирования кода 56
Синтаксис в цвете, нумерация строк и структуризация (или о чем говорит компилятор) 59
Синтаксис в цвете 59
Нумерация строк 62
Структуризация 64
Справка 65
Автозавершение 66
Динамическая справка 68
Командное окно 69
Поиск, замена и регулярные выражения 71
Групповые символы 72
Регулярные выражения 73
Поиск из командного окна 75
Инкрементный поиск 76
Что дальше 77

**Глава 4 Макросы Visual Studio .NET 78**

Макросы: скотч для Visual Studio .NET 78
Запись макросов 79
Команды макросов 81
Редактирование макроса в Macros IDE 82
Простой макрос 85
Работа с макросами 86
Управление документами и текстом 86
Работа с окнами 88
Использование сборок в ваших макросах 90
События макросов 91
Совместное использование макросов 94
Экспорт модулей и проектов 94
Преобразование макроса в надстройку 95
Что дальше 97

**ЧАСТЬ II**
**РАСШИРЕНИЕ VISUAL STUDIO .NET 99**

**Глава 5 Мастер надстроек и объектная модель 100**

Мастер надстроек 100
Работа с Add-in Wizard 101
Проект надстройки 102
Установка и загрузка надстройки 107
Отладка надстройки 109
Объектная модель автоматизации 111
Объекты автоматизации 111
Принципы объектной модели 111

События автоматизации ... 112
Что дальше ... 120

**Глава 6 Архитектура надстроек 121**

Создание надстройки с нуля ... 121
Компиляция надстройки Basic ... 122
Регистрация надстройки Basic как COM-объекта ... 123
Регистрация надстройки Basic в Visual Studio .NET ... 125
События надстроек ... 127
Последовательность событий надстроек ... 127
Жизненный цикл надстройки ... 128
Интерфейс IDTExtensibility2 ... 130
Пространство имен EnvDTE ... 130
OnConnection ... 131
OnStartupComplete ... 133
OnAddInsUpdate ... 133
OnBeginShutdown ... 137
OnDisconnection ... 137
Реестр и именованные значения надстройки ... 138
Разделы CommandPreload и PreloadAddinScate ... 138
LoadBehavior и CommandLineSafe ... 140
SatelliteDLLPath и SatelliteDLLName ... 142
FriendlyName и Description ... 144
AboutBoxDetails и AboutBoxIcon ... 144
Что дальше ... 146

**Глава 7 Команды 147**

Что такое команда? ... 147
Поиск команд ... 148
Имена команд ... 149
Выполнение команд ... 150
Создание команд запуска макросов ... 151
Создание команд для надстроек ... 151
Обработка команд ... 153
Состояние команды ... 154
Как найти обработчик команды надстройки ... 155
Пользовательский интерфейс команды ... 156
Объектная модель панели команд ... 156
Главная панель команд ... 157
Добавление новой панели команд ... 158
Нестандартные изображения ... 160
Восстановление потерянной команды и ее пользовательского интерфейса ... 161
devenv /setup ... 161
Нестандартная регистрация ... 162
Удаление команды ... 165
Параметры команд ... 168
Привязка к клавишам ... 169
Что дальше ... 173

**Глава 8 Программное управление решениями и проектами 174**

Работа с решениями 174
Создание, загрузка и выгрузка решений 175
Перебор проектов 176
Захват событий решения 177
Надстройки для решений 182
Работа с элементами проектов 184
Перебор элементов проекта 184
Добавление и удаление элементов проекта 187
Работа с объектами проектов, специфичными для отдельных языков 189
Проекты VSProject 189
Работа со вспомогательными типами проектов Visual Studio .NET 195
Проект Miscellaneous Files 195
Проект Solution Items 197
Внемодельные проекты 197
События проектов и элементов проекта 198
Управление конфигурациями компоновки 200
Управление параметрами решения 200
Управление параметрами проекта 207
События компоновки 210
Сохранение информации проектов и решений в сеансах IDE 212
Что дальше 213

**Глава 9 Мастера Visual Studio .NET 214**

Общий обзор 214
Типы мастеров 214
Создание объекта мастера 215
Создание .vsz-файла 218
Где сохранить .vsz-файл 219
Создание шаблона мастера 221
Использование файлов шаблонов 222
Имена файлов решения и мастер нового проекта 223
Замены 224
Необрабатываемые шаблоны для мастеров добавления нового элемента 227
Нестандартные мастера 227
Программный запуск нестандартного мастера 228
Связывание нестандартных мастеров 229
Вспомогательная библиотека мастера 231
Переменные мастера 234
Вспомогательные методы мастера 235
Завершение примера WizardBuilder 236
Что дальше 237

**Глава 10 Программирование пользовательского интерфейса 238**

Основные сведения об окнах 238
Набор Windows 238
Свойство Object 241
Основное окно 241

Окна-проводники и объект UIHierarchy ... 243
Дерево объекта UIHierarchy ... 243
Объект UIHierarchy ... 244
Объект UIHierarchyItems ... 246
Объект UIHierarchyItem ... 246
Окно Toolbox ... 247
Вкладки и элементы ... 247
Добавление элементов в Toolbox ... 248
**Окно Task List** ... 251
Элементы списка задач ... 251
Добавление новых задач ... 252
Объект TaskItem ... 256
События **Task List** ... 257
Маркеры комментариев ... 259
Окно Output ... 262
Области окна Output ... 262
Окно дизайнера форм ... 264
Интерфейс IDesignerHost ... 264
Маршалинг ... 264
Добавление элемента к форме ... 265
Поиск существующих элементов управления ... 266
Пример компоновки формы ... 266
Создание нестандартных окон инструментов ... 267
Согласователи ... 270
Настройка вкладок в нестандартном инструментальном окне ... 273
Установка объекта Selection ... 273
Диалоговое окно Options ... 274
Изменение существующих параметров ... 274
Создание нестандартных параметров ... 278
Что дальше ... 281

**Глава 11 Объекты и события редактирования текста 282**

Окна редактора ... 282
Объект Window ... 283
Объекты TextWindow и HTMLWindow ... 284
Объект TextPane ... 285
Документы ... 288
Объект Document ... 289
Объект TexcDocument ... 292
Объекты точек ... 292
TextPoint ... 293
VirtualPoint ... 293
EditPoint ... 295
Объект TextSelection ... 296
Сравнение TextSelection и EditPoint ... 297
Контексты отмены ... 298
Автоматические контексты отмены ... 299
Создание контекстов отмены ... 299
Взаимосвязь стеков ... 302

Событие LineChanged ... 304
Несколько обработчиков события LineChanged ... 308
Что дальше ... 309

**Глава 12 Модель кода 310**

Обнаружение кода ... 310
Беглый просмотр исходного текста ... 310
Получение CodeElement из объекта точки ... 321
Генерация кода ... 322
Построение исходного файла ... 322
Что дальше ... 326

**ЧАСТЬ III**
**РАЗВЕРТЫВАНИЕ, СПРАВКА И РАСШИРЕННЫЕ ПРОЕКТЫ 327**

**Глава 13 Разработка проектов установки 328**

История Microsoft Windows Installer (MSI) ... 328
Создание пользовательского проекта установки ... 329
Редактор File System ... 330
Редактор Registry ... 334
Редактор File Types ... 339
Редактор User Interface ... 342
Нестандартные действия ... 348
Редактор Launch Conditions ... 353
Подгружаемые модули ... 357
Установка программ .NET ... 358
Что дальше ... 364

**Глава 14 Справочная система Visual Studio .NET 365**

Работа со справочной системой ... 365
Окна справочной системы ... 366
Параметры поиска и указателя ... 367
Уточнение результатов поиска ... 368
Создание собственных фильтров в справочной системе ... 369
Поиск из окна Command Window ... 370
Настройка окна Dynamic Help ... 371
Использование сервиса XML Help Provider ... 372
Создание собственных файлов справки ... 376
Регистрация набора справки ... 381
Что дальше ... 383

**Глава 15 Усложненные проекты в Visual Studio .NET 384**

Visual Studio .NET из командной строки ... 384
Компоновка проектов и решений из командной строки ... 384
Установка параметров графического пользовательского интерфейса из командной строки ... 388
Параметры VSIP ... 390
Управление исходным кодом с помощью Visual SourceSafe ... 390
Установка VSS ... 390

Размещение файлов в системе контроля исходных кодов ... 390
Работа с файлами при контроле исходных кодов ... 393
Тщательное планирование решений ... 398
Не нарушайте компоновку ... 398
Что дальше ... 399

**Приложение Справочник по модели кода 400**

Объекты модели кода ... 400
FileCodeModel и CodeModel ... 400
CodeElement ... 404
Специализированные объекты модели кода ... 406
Генерация кода ... 419
Общие параметры ... 419
Основные Add-методы ... 420
Прочие Add-методы ... 423
Методы для удаления ... 426

**Предметный указатель 428**

**Об авторах 441**

# Благодарности

Нам нужно поблагодарить очень многих за помощь в работе над этой книгой.

Прежде всего спасибо нашему выпускающему редактору Даниель Берд Воуллер. Она познакомила Брайана и Марка с Биллом Чайлзом, который в свою очередь познакомил нас с Крэйгом, что было поворотным моментом в создании этой книги. Билл неутомимо снабжал нас нужной информацией, и за это мы ему особенно благодарны.

Мы также признательны Сэлли Стикни. Она превосходный редактор, но, что еще важнее для нас, авторов, — она выдающийся учитель. Джек Бедри, технический редактор, помог сделать из этой книги законченный продукт. Ине Чанг, нашему литературному редактору, мы очень благодарны за то, что она превратила нашу местами бессвязную писанину в грамматически корректный, понятный текст. Тэсс Макмиллан сделала Web-страницу этой книги, а Тим Ким протестировал установочную программу примеров. Мы также благодарим Майкла Клопфера за художественное оформление и Джейсона Фиша за нашу фотографию.

Кен Харди оказал нам неоценимую техническую поддержку. Калпана Санграйка и Натали Уэллс помогли нам выделить наиболее интересные для пользователя возможности Microsoft Visual Studio .NET. Мы также благодарим за помощь Грэга Десикко, Киппера Йорка, Кэнни Керр и Сэма Генри.

И, наконец, Брайан и Марк благодарят Мэри Дейонг, воодушевившую их на работу над книгой. Брайан благодарит свою жену Кэтрин, а Крэйг — своих родителей, брата и друзей за помощь и поддержку. Марк благодарит Big Bang за то, что все это стало возможным.

# Введение

В конце 2000 года Марк Янг и я (Брайан Джонсон) были на презентации Microsoft Visual Studio .NET для компьютерных писателей и редакторов со всего мира. Мы были потрясены, увидев, насколько просто писать Windows- и Web-приложения с помощью этого инструмента. Глядя на демонстрацию, я отчетливо понял, что этот продукт заслуживает того, чтобы о нем написать. Я сделал несколько заметок и набросал план предполагаемой книги, вверху которого значилось: «Inside Visual Studio .NET».

Год спустя меня все еще не покидала эта идея, но то то, то другое не давало мне начать ее осуществление. Однажды Марк спросил меня, не хочу ли я что-нибудь написать, и я показал ему собранный материал для книги о Visual Studio .NET. Я сказал ему, что мне интересно написать про объектную модель автоматизации Visual Studio .NET, про интегрированную среду разработки (IDE) и макросы. Марк взял мои материалы и почти все ночи напролет в течение трех или четырех месяцев писал надстройки и изучал устройство IDE.

Весной 2002 года мы связались с Дэниэль Берд Воуллер, которая познакомила нас с разработчиками Visual Studio .NET. Первым из них был Билл Чайлз, менеджер программы, ответственный за многие функции, которые мы собирались описать.

Встреча с Биллом Чайлзом была, пожалуй, самой замечательной вещью, которая с нами случилась, потому что он познакомил нас с Крэйгом Скибо. Крэйг — разработчик, написавший большую часть API автоматизации, о котором говорится в книге. В одном из наших первых разговоров Крэйг сказал, что и сам работает над книгой об объектной модели автоматизации. На самом деле у него было уже четыре готовых главы, но он не был уверен, что найдет время, чтобы закончить свою книгу.

Мы прочитали главы, написанные Крэйгом, и обнаружили, что это именно то, что нам нужно. Мы предложили ему стать соавтором книги «Основы Microsoft Visual Studio .NET 2003». Он согласился, что нас с Марком очень обрадовало.

Мы стремились написать книгу, которая помогла бы разработчикам использовать Visual Studio .NET 2003 наиболее эффективно, настраивать и расширять IDE и, наконец, смогла бы их убедить, что переход к Visual Studio .NET 2003 и .NET Framework приведет к огромному увеличению продуктивности их работы.

## Для кого эта книга

Эта книга адресована программистам, которых интересуют подробности работы Visual Studio .NET. Работая над ней, мы использовали Visual Studio .NET 2003, но практически все написанное применимо и к Visual Studio .NET 2002.

Эта книга не научит вас программировать, но поможет разобраться в принципах организации решений и проектов, применяемых для построения прило-

жений в Visual Studio .NET. Если вы новичок, изучающий какой-то язык программирования, первые главы этой книги очень помогут вам освоить IDE.

Если вы интересуетесь расширением и настройкой среды разработки, эта книга для вас.

## Структура книги

Книга разбита на три части.

- В первой части мы рассматриваем общие возможности Visual Studio .NET, позволяющие вести разработку наиболее продуктивно. Если вы хорошо ориентируетесь в IDE, возможно, вам будет достаточно лишь просмотреть эту часть, хотя мы приготовили много сюрпризов, которые будут интересны большинству читателей. Вы узнаете здесь об истории IDE, управлении решениями и проектами, редакторе кода, а еще познакомитесь с макросами.
- Во второй части описаны надстройки и объектная модель автоматизации Visual Studio .NET. Вы познакомитесь с API расширения, архитектурой надстроек и команд, программным управлением решениями и проектами, узнаете о создании мастеров Visual Studio .NET, программировании пользовательского интерфейса Visual Studio .NET, управлении текстовым редактором и использовании модели кода.
- В третьей части мы собрали материал, который редко встречается в других книгах по программированию. Здесь говорится о создании установочных проектов и построении собственной справочной системы, а также о работе с Visual Studio .NET 2003 из командной строки и использовании системы контроля исходных кодов Microsoft Visua, SourceSafe.

В приложении приведен справочный материал по модели кода, обсуждаемой в главе 12.

## Сопроводительный материал

Сопроводительный материал к этой книге доступен в Web: *http://www.microsoft.com/mspress/books/6425.asp* Щелкните ссылку Companion Content в панели More Information для перехода на страницу с сопроводительным материалом. С этой страницы вы сможете скачать установщик файлов примеров. Установщик написан с использованием Visual Studio .NET. Установщик помещает код утилит в папку Program Files\Microsoft Visual Studio .NET 2003\ Common7\IDE\PublicAssemblies, код примеров — в My Documents\Microsoft Press\InsideVSNET, а макросы записываются в каталог макросов Visual Studio .NET 2003. Вы можете удалить эти утилиты и примеры, выбрав Add Or Remove Programs (Установка и удаление программ) в Control Panel (Панель управления). На Web-странице сопроводительного материала подробно описаны компоненты, устанавливаемые на вашу машину.

**Примечание** Мы писали примеры Е расчете на Visual Studio .NET 2003, но большинство проектов будет работать, если вы создадите новые файлы проектов и импортируете исходный код в Visual Studio .NET 2002. Никаких обещаний относительно работы в Visual Studio .NET 2002 мы дать не можем, но с большинством примеров у вас не должно быть особых про-

блем. Мы рекомендуем перейти на новую версию, если у вас есть такая возможность.

## Системные требования

Вы можете использовать любую систему, на которой работает Visual Studio .NET 2003. Мы использовали Visual Studio .NET 2003 Professional Edition, так что, если у вас имеется эта версия, ее будет достаточно. Если вы работаете с Visual C# .NET, Visual Basic .NET или Visual C++ ,NET, все макросы, описанные в этой книге, будут работать, но, поскольку мы писали их на всех трех перечисленных языках, вам потребуются соответствующие компиляторы.

## Контакт с авторами

Комментарии к этой книге отправляйте по адресу *brianjjo@botmail.com*. Мы вряд ли сможем ответить на все технические вопросы, связанные с разработкой ваших надстроек, но будем рады узнать ваши соображения относительно будущих изданий этой книги. Информация об изменениях в исходных кодах содержится на сайте *http://groups.msn.com/insidevsnet*. Сведения об ошибках доступны в службе поддержки Microsoft Press (см. ниже).

Если у вас появятся технические вопросы об использовании Visual Studio .NET, написании макросов или создании надстроек, лучше всего обратиться к группам новостей Microsoft: *news://msnews.microsoft.com*. Все группы Visual Studio .NET начинаются с *microsoft.public.vsnet*. Кроме того, группа, связанная с надстройками, есть в Yahoo: *http://groups.yahoo.com/group/vsnetaddin*. Мы регулярно просматриваем эти группы.

## Служба поддержки Microsoft Press

Мы приложили все усилия, чтобы обеспечить точность сведений, изложенных в книге и содержащихся в файлах примеров. Поправки к этой книге предоставляются Microsoft Press через World Wide Web по адресу:

http://www.microsoft.com/mspress/support/

Чтобы подключиться к базе знаний Microsoft Press, откройте страницу:

*http://www.microsoft.com/mspress/support/search.asp*

Кроме контактов с авторами, пожалуйста, присылайте комментарии, вопросы и предложения, касающиеся этой книги, в Microsoft Press

по обычной почте:

Microsoft Press
Attn: Inside Microsoft Visual Studio .NET 2003 Editor
One Microsoft Way
Redmond, WA 98052-6399

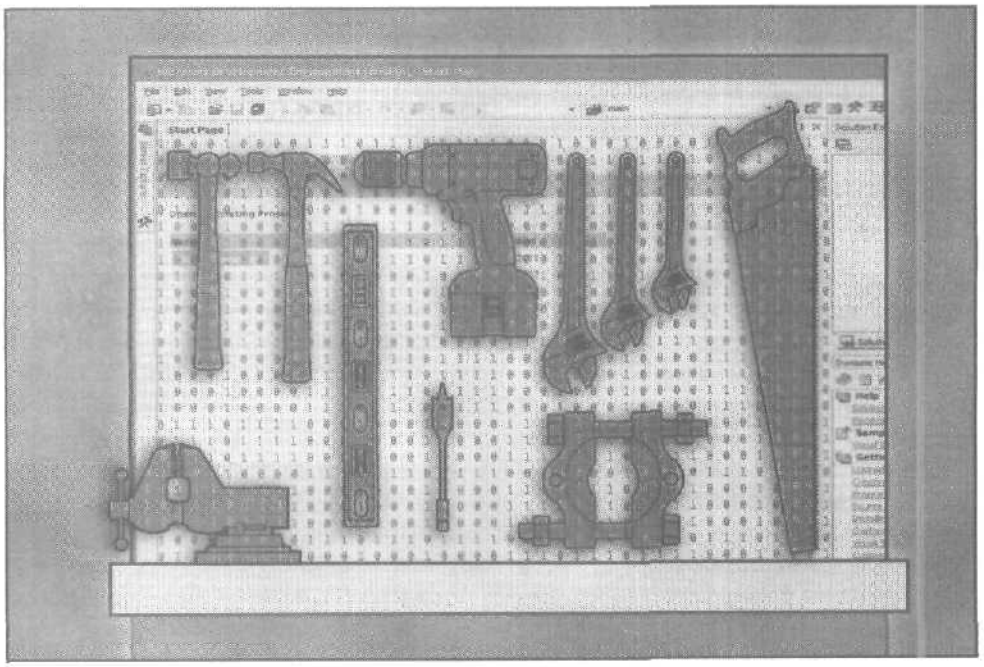
или по электронной почте:

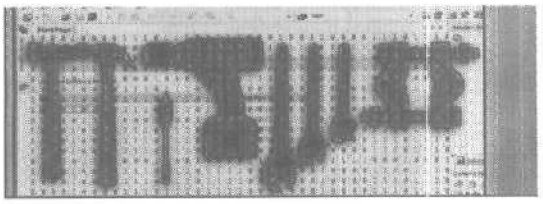
MSPINPUT@MICROSOFT.COM

Пожалуйста, обратите внимание на то, что по этим адресам не предоставляется техническая поддержка. Информацию о технической поддержке Microsoft Visual Studio .NET 2003 см. на *http://msdn.microsoft.com/vstudio/*

ЧАСТЬ I

# VISUAL STUDIO .NET КАК ИНСТРУМЕНТ РАЗРАБОТКИ





ГЛАВА

# 1

# Эволюция Visual Studio .NET

**Цикл** разработки продуктов фирмы Microsoft составляет примерно два года. Visual Studio .NET, чей выпуск состоялся Б феврале 2002 года, создавали более трех лет. Причиной такой задержки разработки была .NET — технология Microsoft для управляемых приложений и Web-сервисов XML, также реализованная в 2002 году. Visual Studio .NET 2003 появилась примерно через год после .NET. Она была обновлена специально для совместного выпуска с Microsoft .NET Server 2003- Оба продукта используют версию 1.1 .NET Framework.

В этой главе дан краткий обзор .NET и разработки в Visual Studio .NET неуправляемого кода. Затем мы познакомимся с особенностями интегрированной среды разработки (IDE). Наконец, мы обсудим те возможности расширения, которые делают Visual Studio ,NET исключительно ценным инструментом для программистов.

## Путь к Visual Studio .NET

Развитие инструментов разработки от Microsoft всегда ориентировалось на новые технологии. Выпуск нового инструмента часто совпадал по времени с представлением и реализацией новой технологии. В 1993 г., когда текущей версией Microsoft C/C++ была «семерка», фирма представила Visual C++ 1.0. Microsoft C/C++ был просто усовершенствованным MS DOS-продуктом. Но усовершенствованным настолько, что многие программисты сопротивлялись переходу на Visual C++. Однако в конце концов IDE, полностью базирующаяся на Windows, с интегрированными редактором, отладчиком, компоновщиком и браузером исходного текста сделали переход на Visual C++ неизбежным. Представление Microsoft Foundation Classes (MFC) 2,0 также положительно отразилось на росте популярности Visual C++. Visual C++ позволял быстро создавать Win32- и MFC-приложения для Windows, не связываясь с MS DOS. Оглядываясь назад, можно сказать, что использовать

Windows-приложение для создания Windows-приложений было неплохой идеей. До этого логика была не вполне прозрачной,

Технология .NET, сделавшая возможным создание исключительно эффективных управляемых приложений и Web-сервисов XML, появилась спустя девять лет. На этот момент Visual Studio, впервые представленная в 1998 г., насчитывала уже шестую версию. Этот инструмент, представляющий собой комбинацию нескольких продуктов (Visual C++, Visual Basic, Visual InterDev и Visual J++), стал одним из самых популярных средств разработки. Visual Studio позволяет создавать приложения для целого ряда платформ: Win32, MFC, COM, ActiveX, Active Template Library (ATL), Java. DirectX и для Web. Если что-либо можно сделать в Windows, это наверняка можно сделать с помощью Visual Studio 6. Visual Studio 6 по отношению к Visual Studio .NET является тем же, чем был Microsoft C/C++ по отношению к Visual C++ 1.0. По прошествии лет легко понять, почему разработчики перешли от инструментов на базе MS DOS к инструментам на базе Windows. Поскольку инфраструктура Web-сервисов XML до сих пор в процессе создания, не совсем очевидно, зачем нужно переходить к .NET и Visual Studio .NET.

Для ответа на этот вопрос стоит обсудить, что делает .NET притягательной для разработчиков технологией и почему именно Visual Studio .NET идеально подходит для разработок под это окружение.

## Разработка для .NET

Когда-то все мы разрабатывали программы в MS DOS и, глядя на экран с Windows 3.1, понимали, как должна на самом деле выглядеть программа: много красок, симпатичные шрифты, оконная система с высоким разрешением, которая берет на себя заботу о большинстве отвратительных мелочей, столь отягощавших разработку под MS DOS. Немного труднее увидеть преимущества .NET, так как в данном случае отличия заключаются в архитектуре и не имеют броского визуального выражения, однако для разработчиков большинство преимуществ .NET столь же значимо, как преимущества Windows по сравнению с MS-DOS.

Одна из целей .NET Framework — упростить разработку и развертывание приложений в Интернете. Это распространяется на приложения, выполняющиеся локально, удаленно или вообще рассредоточенные по Интернету. Упрощение достигается благодаря общеязыковой исполняющей среде (common language runtime, CLR), которая управляет ходом выполнения приложения и доступна для приложений, написанных на любом из языков, ориентированных на эту исполняющую среду. Функциональность, поддерживаемая CLR, предоставляется таким языкам посредством библиотеки классов .NET Framework. На рис. 1-1 показано, как CLR связана с .NET Framework.

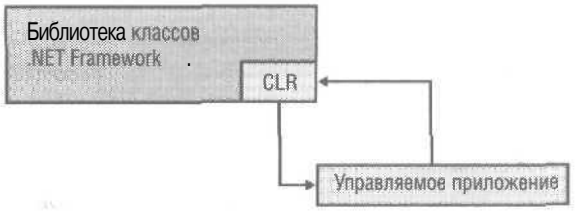



**Рис. 1-1.** *Библиотека классов .NET Framework и CLR — два главных компонента NET Framework*

Общеязыковая спецификация (Common Language Specification, CLS) определяет, что .NET-ориентированный язык должен обеспечивать для системы. Общая система типов (common type system, CTS) гарантирует, что любые типы, созданные языком, поддерживающим CLS, могут быть использованы любым другим CLS-совместимым языком.

Языки, ориентированные на CLR, компилируются в команды Microsoft Intermediate Language (MSIL или просто IL). Результатом компиляции приложений являются файлы PE (portable executable) или DLL, т. е. для пользователя откомпилированные приложения выглядят так же, как Windows-приложения. При запуске приложений код MSIL из этих файлов компилируются «на лету» на локальном компьютере в машинные команды и выполняется. Все это означает, что программа на любом CLS-совместимом языке, ориентированном на среду CLR, для этой среды абсолютно неотличима от программы на другом подобном языке и может действовать совершенно равноправно. То есть программа на Visual Basic .NET будет обладать той же базовой функциональностью, что и программа на Visual C# или даже на управляемом C++.

**Совет** Подробнее об архитектуре .NET см. книгу Джефри Рихтера «Applied Microsoft .NET Framework Programming» (Microsoft Press, 2002)[1].

Управляемая среда CLR обеспечивает ряд заметных преимуществ. Она позволяет исключить конфликты между разными версиями — известный порок DLL. Она гарантирует обеспечение безопасности при выполнении кода программы. И, наконец, для пее разработан API, которым могут пользоваться как Windows-, так и Web-приложения.

Не стоит думать о .NET Framework как о монолитной виртуальной машине, требующей постоянного ухода. .NET Framework создает среду, которую могут задействовать и неуправляемые компоненты. Неуправляемые компоненты (например, Internet Explorer или исполняющая среда ASP.NET) загружают CLR и выполняют управляемый код. Управляемая среда CLR предоставляет сервис сбора мусора и несколько уровней безопасности.

Для корпоративных разработчиков CLR решает огромную проблему. Зачастую программисты на Visual Basic, C/C++ и COBOL отделены друг от друга. Они сталкиваются с необходимостью функционального взаимодействия, но в ряде случаев работают просто как отдельные команды в общем пространстве. В учреждениях, ориентированных на CLR, процесс разработки становится более управляемым. Одна и та же библиотека классов .NET Framework доступна всем языкам. Общая система типов в библиотеке классов гарантирует простоту совместного применения компонентов всеми .NET-языками (рис. 1-2). Эти компоненты даже могут использоваться как Web-сервисы XML.

Программы для выполнения в среде CLR можно создавать на большинстве используемых сегодня языках программирования. Visual Studio .NET поддерживает такие замечательные языки, как Visual C++, Visual Basic .NET, Visual J# .NET и Visual

[1] *Рихтер Дж.* «Программирование на платформе Microsoft .NET Framework». М.: «Русская Редакция», 2002. — *Прим.перев.*

C# . Вы сами можете добавить поддержку для большого количества языков, поставляемых сторонними производителями ПО, включая COBOL, Eiffel, Python и Perl.

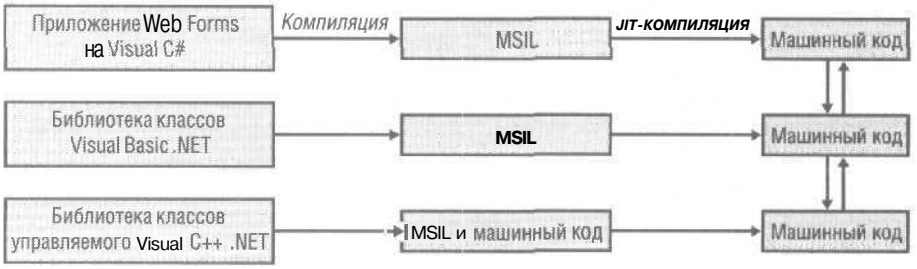



**Рис. 1-2.** *.NET позволяет создавать для управляемой среды программы на разных языках, эффективно и безопасно взаимодействующие между собой*

# Расширения для разработки неуправляемого кода

Для программистов Visual C++, разрабатывающих неуправляемый код, Visual Studio .NET предлагает расширения, обеспечивающие их дополнительными возможностями. Особенности, связанные с IDE, мы обсудим немного позже. А сейчас познакомимся с возможностями седьмой версии компилятора и библиотеки классов.

## Новые опции компилятора

Visual C++ .NET, являющийся реализацией C++ в Visual Studio .NET, позволяет C++-программистам писать устойчивый, безопасный и эффективный код. Многие из усовершенствований были представлены в версии 7.1 компилятора. Microsoft добавила в Visual C++ .NET 17 новых переключателей компилятора и 12 новых переключателей компоновщика, что даже без учета новых возможностей IDE и .NET является для некоторых разработчиков веским основанием для перехода на этот продукт.

#### *Оптимизация кода:* /GL

Переключатель */GL* приводит к оптимизации всех программ проекта. Результатом такой оптимизации обычно является некоторое повышение производительности, однако, прежде чем задействовать этот переключатель, надо внимательно прочитать предостерегающие замечания в соответствующем разделе документации.

Главное, что делает переключатель */GL,* — оптимизирует программу в целом на основании информации из каждого модуля. Такой тип оптимизации позволяет использовать регистры и за пределами границ отдельных функций. Он также позволяет генерировать встраиваемые (inline) функции, даже определенные в отдельных модулях.

---

Примечание Команда Microsoft, разрабатывающая компилятор, имела возможность протестировать такие переключатели на реальных программах. Так, при применении переключателя */GL* для оптимизации ядра игровых

программ, используемого Ensemble Studios в «Age of Empires», был получен прирост производительности в 10%.

### Проверка переполнения буфера: /GS

Почти к любому современному приложению потенциально можно обратиться через Интернет. Чаще всего приложения взламывают через Интернет, используя переполнение буфера. При этом хакер обращается к памяти за границами пространства, которое программист посчитал нужным отвести под буфер. Хакеру нужен лишь небольшой участок памяти для размещения там кода, который затем может вызвать любую функцию, доступную взломанной программе. Переключатель компилятора */GS* способен помочь избежать таких неприятностей, добавляя в модули при компиляции код для проверки безопасности.

Защитный код требует некоторого пространства в стеке непосредственно перед адресом возврата функции. При входе в функцию в это место помещается cookie защиты. При выходе из функции проверяется, был ли этот cookie изменен. Если да, вызывается подпрограмма обработки нарушения безопасности. По умолчанию в этом случае пользователь уведомляется о потенциальном нарушении, и процесс обработки заканчивается. Данную стандартную обработку позволяет изменить функция *_set_security_error_handler*.

**Примечание** В качестве технического редактора второго издания «Writing Secure Code»[1] я столкнулся с небольшой проблемой, подбирая примеры, демонстрирующие переполнение буфера: я не смог воспроизвести в Visual Studio .NET требуемое поведение примеров. Как можно предположить, проверки, по умолчанию устанавливаемые в версии 7 компилятора, блокировали взлом, использующий переполнение буфера. Становится ясно, что даже без дополнительных усилий код, компилируемый Visual Studio .NET, более безопасен.

### Контроль над ошибками времени выполнения: /RTCn

Контроль над ошибками периода выполнения призван помочь в поиске проблем, которые обычно трудно обнаружить. Переключатели */RTCn* помогают найти ошибки, связанные с разрушением стека (*/RTCs*), с использованием неинициализированных переменных (*/RTCs*) и с потерей данных при присвоении больших значений переменным, неспособным их вместить, а также с выходом за границы массива (*/RTCc*).

Подобно переключателю */GS,* контроль за ошибками периода выполнения приводит к внедрению при компиляции дополнительного кода. Переключатели */RTCn* нельзя применять совместно с переключателем /O, попытка это сделать вызовет ошибку. Переключатели */RTCn* имеют комплексную поддержку в отладчике Visual Studio .NET. Если во время выполнения будут обнаружены условия возникновения ошибки, по умолчанию приложение будет прервано и вызван отладчик.

[1] *Ховард М., Лебланд Д.* «Защищенный код». М.: «Русская Редакция», 2003. — *Прим. перев.*

Любой из рассмотренных выше переключателей можно установить в IDE Visual Studio .NET в диалоговом окне со страницами свойств (Property Pages), доступном из Solution Explorer: стоит лишь щелкнуть правой кнопкой проект, для которого надо установить переключатели, и выбрать Properties. Об этом диалоговом окне см. главу 2.

## Обновленные библиотеки классов

Visual C/C++ .NET позволяет разработчикам применять привычные им библиотеки классов. Microsoft взяла обязательства поддерживать в Visual Studio .NET в обозримом будущем разработчиков, использующих библиотеки классов *с* неуправляемым кодом. Эта поддержка включает обновление этих библиотек для последующих версий Windows.

**Примечание** Microsoft продолжает двигаться к управляемому коду, и довольно быстро. Мы рекомендуем использовать .NET Framework, так как эта среда со временем станет «родным» API для Windows-приложений,

Большинство библиотек классов, поставляемых с Visual Studio .NET, имеет номер версии, соответствующий номеру версии Visual Studio .NET, т. е. сейчас это версия 7. К этим библиотекам относится MFC 7, поддерживающая Windows XP, Windows 2000 и ATL 7. Среди новшеств этой библиотеки — диалоговые окна DHTML, позволяющие создавать удобные диалоговые окна для MFC-приложений. И, наконец, ATL и MFC теперь совместно используют ряд общих классов, например *CString*, что позволяет ATL-программистам задействовать функциональность MFC, не загружая всю библиотеку MFC.

Новая особенность ATL — появление сервера ATL. ATL-сервер служит для создания высокопроизводительных Web-приложений на основе ISAPI, а также для создания Web-сервисов XML. ATL-сервер можно применять для создания решений с неуправляемыми Web-сервисами XML или для интеграции существующего ATL-кода в новые .NET-решения.

К прочим библиотекам классов, которых коснулось обновление, относятся Standard Template Library (STL) и C-библиотека периода выполнения (CRT).

### Атрибуты C++

Атрибуты C++ позволяют программистам добавлять к исходным файлам C++ атрибуты в стиле языка определения интерфейса (Interface Definition Language, IDL). Эти атрибуты отличаются от тех, что можно добавлять к управляемому коду Атрибуты, обеспечивающие определенную функциональность ATL, добавляемую компилятором при компоновке приложения, позволят избежать излишнего к ди-рования вручную.

## Соответствие стандартам

Разработчики Visual C++ обеспечили почти полное соответствие стандартам ANSI/ ISO C++. Реализованный в Visual C++ .NET 2003 компилятор C++ совместим с ними более чем на 98%.

### Новая интегрированная среда разработки (IDE)

Переход к .NET потребовал создания набора инструментов, облегчающих разработку программ для разных сред. Visual Studio .NET предоставляет ряд усовершенствований как для разработчиков, пишущих код для Windows, так и для Web-разработчиков, мигрирующих к .NET.

Во-первых, в Visual Studio .NET унифицирована IDE для большинства языков, имеющихся в Visual Studio 6. Теперь разработчикам проще переходить с языка на язык в пределах одной IDE. Разработчики, использующие разные языки, в Visual Studio .NET могут работать гораздо согласованнее и эффективнее.

Во-вторых, все языки, поставляемые с Visual Studio .NET, могут работать с CLR. Причем Visual Basic .NET и Visual C# ориентированы только на CLR. Visual C++ позволяет создавать как управляемый код для CLR, так и неуправляемый код для Windows. Поскольку любой код, предназначенный для .NET, первоначально компилируется в MSIL, а затем уже «на лету» компилируется в машинные команды, CLR действует одинаково независимо от того, с чем вы работаете: с Visual C# или с Visual Basic .NET. В некоторых языках доступ к функциям за пределами CLR проще, однако по сути CLR-ориентированные языки функционально идентичны. Разработчикам, специализирующимися на том или ином языке, знаком «языковый шовинизм». С появлением .NET подобные вещи забываются, и разработчик, отказавшийся от какого-то языка, может увидеть его в совершенно новом свете.

И, наконец, Visual Studio .NET обеспечивает редактирование и расширение, что делает эту IDE первоклассным инструментом независимо от целевой платформы. Механизм макросов управляемого кода и IDE значительно упрощают запись и выполнение макросов. Усовершенствованный API расширения открывает доступ к тем частям IDE, которые никогда не были доступны разработчикам, использующим Visual Studio.

## Возможности Visual Studio .NET

Visual Studio .NET, безусловно, является большим и сложным продуктом. Термины, используемые для описания IDE, помогут вам лучше понять, как использовать данный инструмент, и, что важнее, помогут тем, кто намерен расширять функциональность IDE посредством макросов и надстроек.

### Редакторы, дизайнеры и окна инструментов

Окна в IDE Visual Studio ,NET делятся на две основные группы. *Окна документов* обычно представлены вкладками в центре IDE и содержат редакторы, дизайнеры, Web-страницы или разделы справки. *Окна инструментов* — это окна IDE, которые предоставляют программистам обслуживающие функции: Solution Explorer, Class View, Properties и др. Окна инструментов отличаются от редакторов и дизайнеров тем, что находятся по бокам IDE (рис. 1-3).

---

**Примечание** API расширения, встроенный в Visual Studio .NET, позволяет программистам создавать окна инструментов для их использования в пакетах языков, установленных F. IDE. API также открывает доступ к редакто-

рам и дизайнерам IDE, однако модель расширения не позволяет создавать новые типы окон документов. Для этого служит программа интеграции Visual Studio (Visual Studio Integration Program, VSIP).

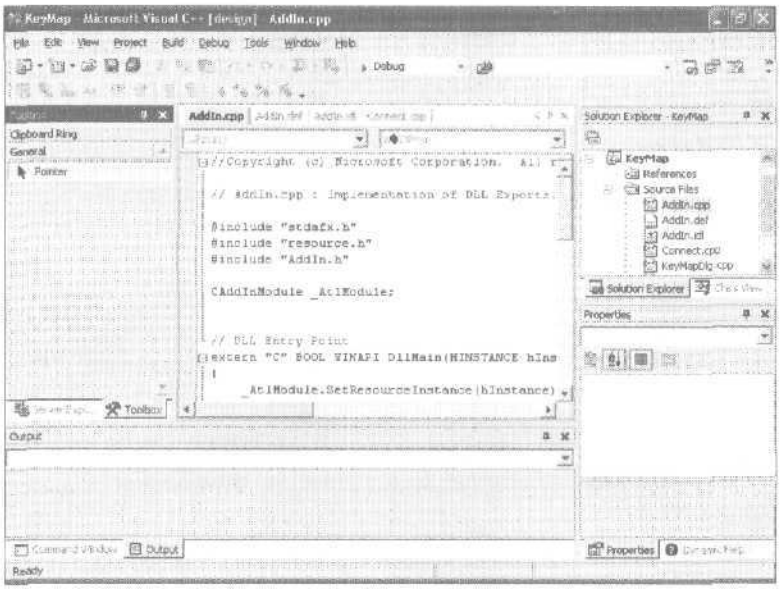


Рис. 1-3. *Типичный вид решения в Visual Studio .NET*

**Примечание** Все управляемые языки, поставляемые с Visual Studio .NET 2003, обеспечены дизайнером в IDE.

### Стартовая страница

При первом запуске Visual Studio .NET появляется стартовая страница с открытой вкладкой My Profile (рис. 1-4). Профиль определяет предпочтения в настройке окон, клавиатуры и справки, свойственные определенному типу программиста. Вкладка My Profile позволяет выбрать профиль и применить соответствующие характеристики для настройки IDE. Эта страница поможет установить параметры окон и клавиатуры, позволяющие легко перейти от используемого вами языка Visual Studio 6 к Visual Studio .NET. Если вы мигрируете к .NET от Visual Studio 6, стоит потратить несколько минут, чтобы просмотреть доступные для выбора профили,

Совет Можно применить предлагаемый по умолчанию и используемый в большинстве книг по Visual Studio .NET профиль Visual Studio Developer, в котором окна инструментов настроены логично и эффективно. Вы, возможно, станете тратить меньше времени на эксперименты с настройкой при повторных установках системы и при работе на разных машинах.

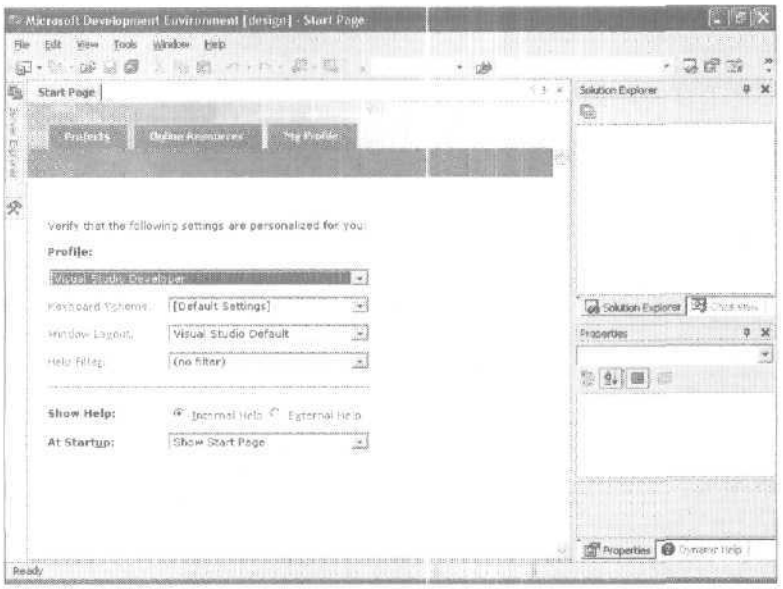



**Рис. 1-4.** *Установка подходящего начального профиля на стартовой странице Visual Studio .NET*

Кроме изменения параметров окон выбранный профиль определяет набор клавиш быстрого доступа к командам IDE. Большинство клавиш быстрого доступа Visual Basic 6 и Visual InterDev относится к соответствующим профилям. Если вы программист C++ и привыкли к комбинациям клавиш Visual C++ 6, то, вероятно, заметите небольшие изменения, способные повлиять на вашу работу. Так, в профиле Visual Studio Developer комбинация Alt+7 для вызова окна дизассемблера заменена на Ctrl+Alt+D. Если вы заметите, что привычные комбинации не делают в IDE того, что вы ожидали, проверьте, какой профиль вы используете и отрегулируйте настройку схемы клавиатуры (Keyboard Scheme). См. об этом главу 3-

**Практикум: использование** нестандартного **профиля в макросе**

Изучая возможности расширения Visual Studio .NET, в своих экспериментах с API расширения вы будете использовать IDE, предназначенную для работы с макросами (Macros IDE). Подробно мы обсудим макросы в главе 4, а сейчас мы потратим буквально минуту на рассмотрение возможностей макросов. Нажмите Alt+F11, *чтобы* открыть IDE макросов. Обратите внимание на окно Project Explorer, относящееся к IDE макросов. Если вы установили примеры, поставляемые с этой книгой, вы увидите там макрос-проект InsideVSNET, Найдите в нем модуль Samples.

Макрос *AutoHideToggle*, который мы собираемся запустить, изменяет состояние окон инструментов Visual Studio .NET. Это делается командой *WindowAutoHideAll* главного меню Visual Studio .NET.

Проблема в том, что у этой команды нет обратной — *WindowAutoUnhideAll*. Поэтому приходится сбрасывать параметры окна щелчком кнопки Reset

Window layout диалогового окна Options, заново выбирать свой профиль на вкладке My Profile стартовой страницы иди вручную открывать нужное окно инструмента. Наш маленький макрос берет эти заботы на себя, действуя в истинно хакерском стиле.

Нам нужно сохранить свое тщательно подобранное размещение окон, поэтому мы создадим временный профиль, в котором сохранятся текущие позиции всех открытых окон инструментов. Сделаем это с помощью объекта *DTE.WindowConfigurations*. Макрос затем применит этот профиль и вызовет команду *Window.AutoHideAll*. Чтобы возвратиться на исходные позиции, мы проверим, является ли временная конфигурация активной. Если да, то опять применим саой профиль, что приведет к загрузке всех окон инструментов на свои места,

**ВОТ ЭТОТ КОД:**

```
Sub AutoHideToggle()
    Static aTempAs tfindowConfiguratlon * Nothing ;
    Dim cmdobj As Command
     ' Создать переменную для хранения команды AutoHideAll.
    cmdobj * DTE.Commands.Item("Window.AutoHideAll")
    With DTE.WindowConfigurations
        ' Проверить, используется ли сейчас наша временная конфигурация.
        ' Если нет, создать ее и сохранить на будущее.
        If aTemp Is Nothing Then
            ' Если мы не находимся в конфигурации aTemp, создаем
            ' новую aTemp и применяем ее, чтобы она сохранилась.
            aTemp = .Add("aTemp")
          . aTemp.Apply()
            ' Вызвать AutoHideAll.
            DTE.Commands.Raise(cmdobj.Guid, cmdobj.ID, Nothing, Nothing)
        Else
            ' Повторное выполнение макроса, вернуть окна обратно в то
            ' положение, в котором они были при первом выполнении макроса.
            aTemp.Apply()
            aTemp = Nothing
        End If
    End With
End Sub
```

Макрос работает, но слишком медленно (но все равно это лучше, чем делать то же вручную). Он несколько раз загружает профиль конфигурации (при каждом вызове метода *Apply)*. В 4 главе мы увидим, как ускорить его работу, обернув в надстройку. В главе 3 приведено несколько способов открыть доступ к этому макросу из меню и клавишами быстрого доступа.

Настройка Help Filter позволяет фильтровать информацию, которую справочная система Visual Studio .NET по умолчанию предоставляет во время работы. C++-программист, специализирующийся на MFC или ATL, сэкономит время и несколько тактов процессора, установив фильтр, пропускающий только интересующие его основные разделы справки. Настройка фильтра справки отражается на комбини-

рованном списке Filtered By окна Search, делая его значением по умолчанию то, что вы задали на вкладке My Profile. В главе 14 объясняется, как создавать собственные нестандартные фильтры справки.

Последний параметр страницы My Profile задает вид Visual Studio .NET при запуске. Установив для At Startup значение Show Start Page (по умолчанию), при загрузке IDE вы получите открытую вкладку Get Started стартовой страницы, предлагающей ряд замечательных возможностей. Помните: этот параметр доступен в каждом отдельном профиле, и, если вы хотите изменить начальный вид IDE, надо устанавливать его индивидуально.

### Редактор

Разработчики базового редактора Visual Studio .NET стремились создать редактор кода, который стал в один ряд с лучшими на сегодняшний день коммерческими и свободно распространяемыми образцами. Они добились своей цели во многом благодаря исключительно мощной модели расширения, новому средству записи макрокоманд и специализированной IDE для работы с макросами (Macros IDE). Модель расширения Visual Studio .NET — главная тема этой книги, поскольку именно ее мы используем для настройки и расширения функциональности IDE.

К числу прочих новых возможностей редактора относятся структуризация, нумерация строк и средства поиска и замены (см. главу *У).*

### Дизайнеры

Visual Studio .NET предлагает четыре главных типа дизайнеров: дизайнеры Windows Forms, позволяющие визуально создавать приложения Windows Forms; дизайнеры Web Forms, помогающие создавать приложения Web Forms по принципу WYSIWYG (What You See Is What You Get); дизайнер компонентов, позволяющий создавать компоненты для серверных решений масштаба предприятия, и XML-дизайнер, облегчающий программистам работу с файлами определения схемы XML (XML Schema Definition, XSD).

В Visual Studio .NET 2003 все языки обеспечены дизайнерами для создания .NET-приложений. Это означает, что вы можете создавать приложения Windows Forms и Web Forms, применяя для этого тот же язык, что и для программирования важнейших алгоритмов. В прошлом разработчики часто создавали пользовательский интерфейс своих приложений с помощью визуальных инструментов, например Visual Basic, а логику писали на Visual C++. Благодаря способу взаимодействия .NET-сборок вы по-прежнему вольны применять разные языки для работы с формами и для написания библиотек, но теперь в этом нет нужды.

### Окна инструментов

Окна инструментов (не являющиеся окнами документов) — это окна IDE, обеспечивающие вас информацией и служебными функциями во время работы. IDE имеет массу разных окон инструментов, к которым позволяют обратиться клавиши быстрого доступа, окно Command и команды меню. Ниже приведены наиболее часто используемые в Visual Studio .NET окна инструментов и сочетания клавиш, ускоряющие из вызов.

### Окно Solution Explorer (Ctrl+Alt+L)

Solution Explorer (рис. 1-5) — вероятно, самое важное окно инструментов в Visual Studio .NET. Почти вся работа, выполняемая программистом в Visual Studio .NET, крутится вокруг решения. Решение — это набор проектов, каждый из которых является набором файлов. Solution Explorer обеспечит доступ к файлам ваших проектов. Именно здесь вы будете добавлять к проектам новые классы и файлы и даже новые проекты, укрупняющие решение.

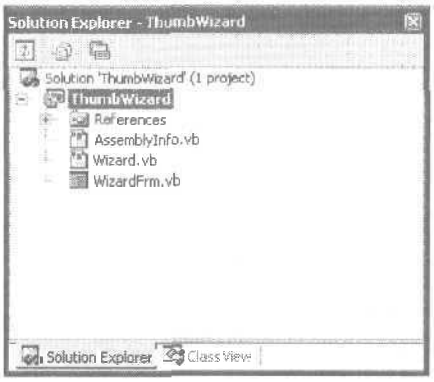



**Рис. 1-5.** *Управляемый проект в Solution Explorer*

### Окно Class View (Ctrl+Shift+C)

Окно Class View (рис. 1-6) позволяет увидеть иерархию классов решения. Если ведется работа над большими проектами, удобнее перемещаться по решениям с помощью Class View, а не с помощью Solution Explorer.

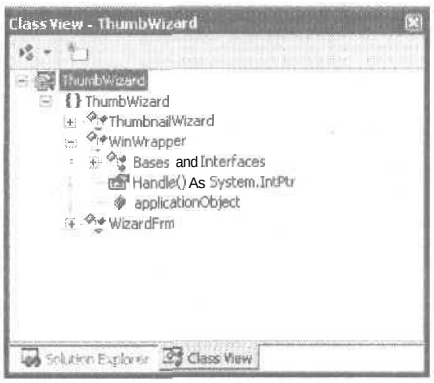



**Рис. 1-6.** *Окно Class View предоставляет альтернативное представление объектов решения*

### Окно Properties (F4)

Это окно позволяет узнать или установить свойства элементов пользовательского интерфейса, добавляемых к приложениям Windows Forms и Web Forms. В этом окне можно также установить свойства решений, проектов и файлов, предвари-

тельно выделив их в Solution Explorer. Рис. 1-7 показывает окно Properties для установочного проекта, являющегося частью решения,

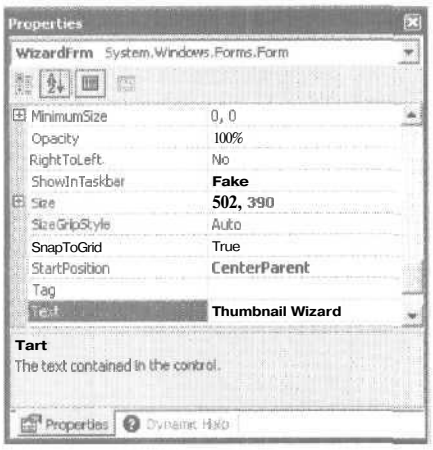



**Рис.** 1-7. *Окно Properties позволяет установить свойства компонентов, проектов и решений*

#### **Окно** Server Explorer (Ctrl+Alt+S)

Окно Server Explorer служит для доступа к источникам данных и информации на вашем локальном компьютере и на удаленном сервере. Из этого окна можно подключаться к данным, получать показания счетчиков производительности и информацию из журнала событий, а также управлять службами системы. Даже при локальном использовании этот инструмент способен экономить ваше время, позволяя легко запускать и останавливать системные службы и работать с системными журналами. На рис. 1-8 в Server Explorer доступны два компьютера; локальный, на котором работает Visual Studio .NET, и удаленный тестовый сервер.

Примечание Для доступа к системной информации сервера нужен соответствующий уровень полномочий.

#### **Окно Toolbox** (Ctrl+Alt+X)

В основном окно Toolbox в Visual Studio .NET используется для размещения элементов управления, добавляемых к форме в приложениях Windows Forms и Web Forms. Но это окно позволяет также размещать там часто используемые фрагменты своего кода или фрагменты справочных файлов и Web-страниц, которые потребовалось сохранить при их чтении. Clipboard Ring позволяет возвращаться к ранее сохраненным фрагментам текста и использовать их. Можно добавлять к этому окну свои вкладки и тем самым упорядочивать элементы управления и разрабатываемый код. На рис. 1-9 показана такая нестандартная вкладка окна Toolbox, добавленная ознакомительным курсом по Visual Studio .NET (находится на http://msdn.microsoft.com/vstudio/productinfo/tour.asp),




**Рис. 1-8.** *Окно Server Explorer обеспечивает дистанционный доступ к компьютерам, с которыми вы работаете*

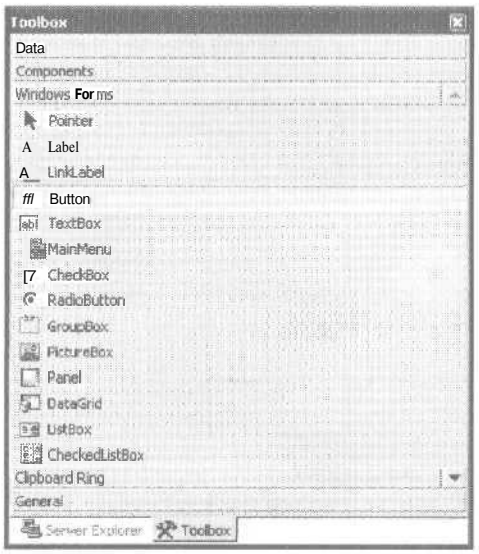



**Рис. 1-9.** *Окно Toolbox открывает доступ к элементам управления и фрагментам кода*

### Окно **Command Window (Ctrl+Alt+A)**

Новая возможность Visual Studio .NET — окно Command Window — объединяет лучшие качества окна Immediate из Visual Basic с мощью командной строки (см. главу 2). Это окно служит для ввода и исполнения *именованных команд* (named commands) прямо в Visual Studio .NET. Именованная команда — это по сути лю-

бая команда IDE, которую вы можете запустить с помощью меню, кнопок панели инструментов или быстрых клавиш Многим из именованных команд Visual Studio .NET по умолчанию не сопоставлены комбинации клавиш или пункты меню Единственный способ до них добраться, не назначая им быстрых клавиш и не добавляя на панель инструментов, — ввести их прямо в окне Command Window.

Command Window имеет два режима работы. В командном режиме (Command) окно действует как командная строка (рис. 1-10). В режиме непосредственного выполнения (Immediate) можно выполнять операторы, менять значения переменных и печатать эти значения, а также вычислять выражения. Чтобы перейти из командного режима в режим непосредственного выполнения, нужно ввести **immed**. Чтобы перейти из режима непосредственного выполнения в командный, нужно ввести **>cmd.**

**Примечание** В командном режиме приглашение «>» всегда присутствует в строке, где вводится команда. В режиме непосредственного выполнения приглашение не отображается.

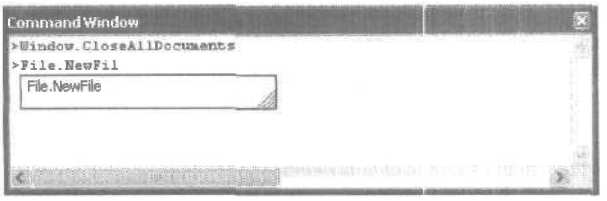



**Рис 1-10.** *Окно Command Window в Visual Studio .NET предоставляет легкий доступ к именованным командам IDE*

#### Окно Macro Explorer (Alt+F8)

Окно Macro Explorer (рис. 1-11) позволяет посмотреть проекты макросов, загруженные в данный момент в IDE. Помните: чтобы макросы были доступны для использования или редактирования в IDE макросов, нужно предварительно загрузить проект макросов.

При записи временного макроса (Ctrl+Shift+R) этот макрос становится доступен через Macro Explorer в MyMacros\Recording и Module\TemporaryMacro. Макрос можно переименовать, чтобы сохранить его, или скопировать в другой модуль при помощи IDE макросов, О записи макросов см. главу 4.

IDE включает еще ряд важных окон, подробный разговор о которых — в следующих главах. К ним относятся окна отладки, окно справки и окно Object Browser.

## Пути к файлам в Visual Studio .NET

Ниже мы поговорим немного о том, где Visual Studio .NET размещает важные для него файлы. Подробности на эту тему мы будем узнавать на протяжении всей книги, сейчас же познакомимся с тем, где находятся файлы, связанные с расширением IDE и автоматизацией. По умолчанию Visual Studio .NET 2003 устанавливается в каталог \Program Files\Microsoft Visual Studio .NET 2003. Большинство каталогов, о которых говорится в этом разделе являются подкаталогами этого каталога (кроме случаев, где приводится полный путь).

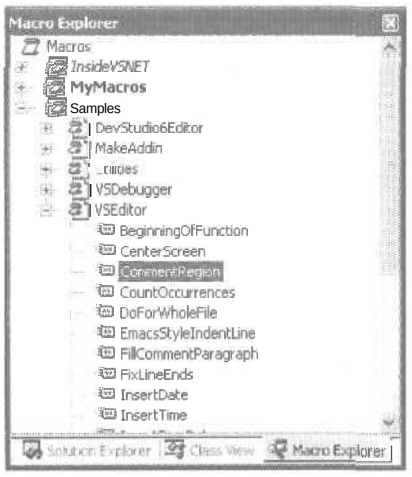



**Рис. 1-11.** *Окно Macro Explorer позволяет работать со всеми доступными в данное время макросами*

При установке Visual Studio .NET в подкаталог SDK\v1.1 устанавливается .NET Framework SDK. В этом подкаталоге можно найти все инструменты .NET Framework, а также примеры кода, и, если вы хотите изучить .NET, это хорошая стартовая площадка. Начать знакомство с .NET Framework SDK следует с файла StartHere.htm в каталоге SDK\v1.1.

Каждый из языков, поставляемых с Visual Studio .NET, имеет свой подкаталог, содержащий шаблоны решений и проектов для соответствующих типов проектов. Эти каталоги имеют понятные имена. Файлы, связанные с C++, находятся в каталоге VC7, файлы C# — в каталоге VC#, а файлы Visual Basic .NET — в Vb7. Вы будете использовать эти каталоги для создания и добавления нестандартных проектов к разным языкам в IDE.

**Совет** В каталоге IDE вы можете заметить файл Samptes.vsmacros. Пример макроса для Visual Studio .NET, запускаемого в IDE, в действительности хранится в каталоге My Documents\Visual Studio Project\VSMacros. Версия в каталоге IDE — копия. Можно редактировать файл Samples.vsmacros, находящийся в каталоге My Documents\Visual Studio Project\VSMacros, но версию из каталога IDE трогать не следует. Если когда-нибудь макрос перестанет работать, можно будет восстановить первоначальную работоспособную версию макроса из файла Samples.vsmacros, находящегося в каталоге IDE,

Выполняемый файл IDE называется Devenv.exe. Он находится в подкаталоге Common7\IDE. Каталог IDE включает ряд подкаталогов, которые мы будем использовать на протяжении всей книги. К ним относятся каталоги PublicAssemblies и PrivateAssemblies, предназначенные для добавления нестандартных сборок, доступных для макросов в IDE. Каталог HTML служит для настройки стартовой страницы. Шаблоны для проектов макросов находятся в каталогах MacroProjectItems и MacroProjects. Шаблоны общих элементов (не связанные с определенным типом проекта) хранятся в каталогах NewFileItems и NewScriptItems.

### Как добавить в меню Tools ссылку на каталог IDE

Если производится достаточно большой объем работ по расширению, может появиться желание добавить в меню Tools ссылку на каталог IDE. Для этого сделайте так.

1. Нажмите Ctrl+Alt+A, чтобы открыть Command Window, и введите **Tools.ExternalTools**. Это приведет к открытию диалогового окна External Tools.
2. Щелкните Add, чтобы добавить в меню новый инструмент, и введите в качестве заголовка IDE **Folder**.
3. В текстовом поле Command введите **Explorer.exe**.
4. В текстовом поле Arguments добавьте путь к каталогу IDE. (Обычно это C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE).
5. Щелкните OK.

Если все прошло успешно, при выборе в меню Tools пункта IDE Folder, откроется каталог IDE, Далее в книге мы еще не раз будем использовать возможности External Tools для создания ссылок, экономящих время.

Если вы собираетесь запускать компоновку из командной строки или просто любите работать в режиме командной строки, вам потребуется при запуске Cmd.exe устанавливать значения переменных окружения для Visual Studio .NET. Это можно сделать следующим образом. Во-первых, просто открыть меню Пуск и выбрать пункт Visual Studio .NET Command Prompt в папке Tools, которая находится в папке Microsoft Visual Studio .NET 2003.

Примечание Советуем добавить ссылку на Visual Studio .NET Command Prompt прямо в меню Пуск, чтобы иметь быстрый доступ к вашей основной командной строке.

Переменные окружения Visual Studio задаются в файле vsvars32.bat, находящемся в подкаталоге Common7\Tools. Если нужен доступ к этим переменным из всех экземпляров Cmd.exe, запущенных на компьютере, можно добавить C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools к вашему системному пути. (Другой вариант: скопировать этот файл в каталог, указанный в вашем пути.) Затем, находясь в любом экземпляре командной строки, достаточно ввести vsvars32, и вы получите рабочее окружение Visual Studio .NET.

Можно пойти дальше и создать в меню Tools пункт Command. Новый пункт создается в диалоговом окне External Tools: щелкните Add, введите название для нового пункта (Command Prompt) и введите cmd.exe в поле Command. При желании можно установить $(ProjectDir) в поле Initial Directory (начальный каталог). Установив в качестве начального каталога каталог своего проекта, вы будете открывать командную строку для этого каталога. Таким образом, будет очень удобно работать с файлами проекта из командной строки.

И, наконец, рассмотрим добавление пути Common7\IDE к системным переменным. Полный путь выглядит так C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE. Это сделает Devenv.exe доступным из любого экземпляра командной строки в вашей системе. Данный путь добавляется командой vsvars32.bat, но иногда нужен доступ только к Devenv.exe.

Совет Не забывайте о Devenv как альтернативе Блокноту для редактирования файлов при компоновке в режиме командной строки. Хотя вы не получите доступа к средствам работы с компоновщиком и проектом без решения, вы сможете использовать свои нестандартные инструменты и макросы.

# Возможности расширения Visual Studio .NET

Visual Studio .NET создана с использованием модели расширения, разработанной для Visual C++ 5. В Visual Studio .NET объект DTE (Development Tools Extensibility) находится на верхнем уровне модели автоматизации, куда входит около 200 объектов.

DTE API могут использовать разработчики, создающие макросы, мастера и надстройки. Еще большая функциональность предоставляется разработчикам коммерческих языков, участвующим в программе Visual Studio Integration Program (VSIP). (Мы обсудим ее позднее.)

Ниже описаны механизмы автоматизации, доступные разработчикам, настраивающим Visual Studio .NET.

## Макросы

Средства поддержки макросов в Visual Studio .NET обеспечивают программистов простым доступом к возможностям, открываемым посредством API автоматизации. Эти средства включают окно инструментов Macro Explorer, исключительно мощные средства записи макросов и собственную IDE макросов, которая к тому же сама является расширяемой при использовании объектной модели DTE. В книге мы будем использовать макросы, иллюстрирующие концепции, относящиеся к расширяемости. В главе 4 подробно описан процесс создания и редактирования макросов при помощи IDE макросов.

Макросы в Visual Studio .NET пишутся на языке Visual Basic .NET. Но, поскольку средства поддержки макросов используют все возможности .NET, программисты макросов имеют доступ к .NET Framework в целом, а также к нестандартным сборкам, разработанным на любом другом .NET-языке.

Чтобы открыть IDE макросов, находясь в Visual Studio .NET, нажмите Alt+F11. IDE макросов откроется в новом окне. Компоновка этой среды очень похожа на компоновку самой Visual Studio .NET. Фактически большинство возможностей, доступных вам как программисту Visual Studio .NET, доступны вам и как программисту макросов.

На рис. 1-12 — IDE макросов в своей стандартной компоновке, включающей окно Project Explorer, в котором показаны все загруженные на данный момент проекты макросов.

## Надстройки

Надстройки позволяют разработчикам создавать расширения к IDE Visual Studio .NET и к IDE макросов. Вообще компилированные настройки обеспечивают лучшую производительность, чем макросы Visual Studio .NET. Кроме того, функциональность, создаваемая надстройками, бесшовно интегрируется в среду разработки,

Независимые поставщики ПО и отдельные программисты могут расширять IDE с помощью надстроек, при этом использование надстроек выглядит так же, как использование встроенных частей IDE.

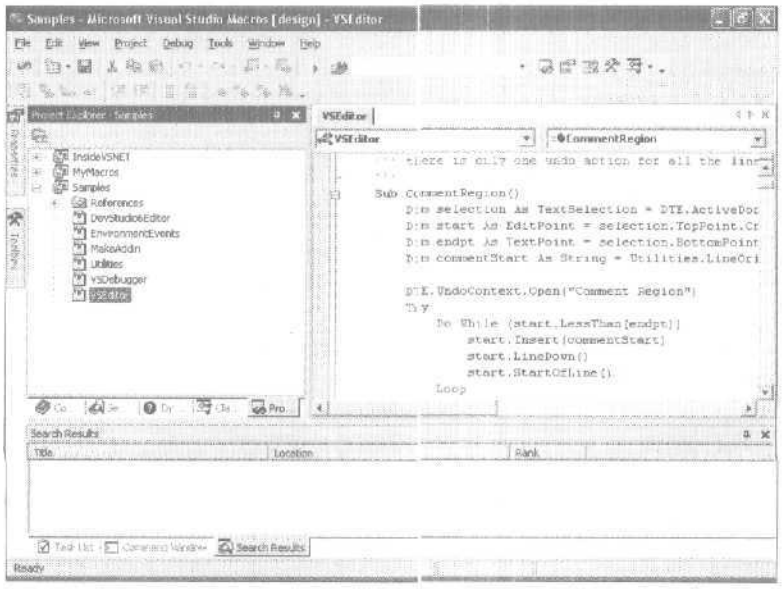



**Рис. 1-12.** *IDE макросов Visual Studio .NET*

Надстройки можно писать на любом .NET-языке, их также можно создать как COM-компоненты на неуправляемом Visual C++. Надстройки должны реализовывать интерфейс *IDTExtensibility2*. Большинство примеров надстроек в книге написаны на Visual C#. Примеры для книги, устанавливаемые из Web, написаны и на C#, и на Visual Basic .NET.

Microsoft создала ряд примеров надстроек, позволяющих изучать объектную модель расширения или просто добавлять функциональность к своей IDE в Visual Studio .NET. Эти примеры расположены по адресу http:// msdn.microsoft.com/vstudio/downloads/automation.asp. Некоторые из них мы используем в начальных главах книги, чтобы добавить к Visual Studio .NET новые возможности. Начиная с главы 5 мы откроем все детали, необходимые для создания собственных надстроек.

## Мастера

Мастера Visual Studio .NET похожи на надстройки, но создаются они при помощи интерфейса *ITDWizard* и предназначены для того, чтобы шаг за шагом вести пользователя по пути решения определенной задачи. Они позволяют достичь самых разных целей. Мастера проектов помогают начать работу над проектом определенного типа. Прочие мастера IDE, например мастер MFC Event Handler Wizard (рис. 1-13), помогают добавить код к существующему проекту.

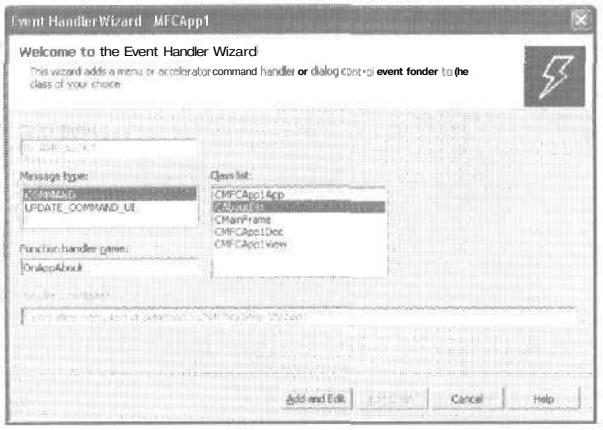



Рис. 1-13. *Мастер Event Handler Wizard помогает добавить код к существующим проектам*

## Программа интеграции с Visual Studio (VSIP)

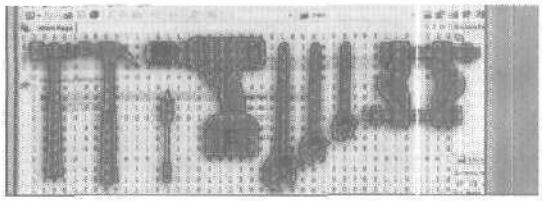
В этой книге мы не станем много обсуждать VSIP, поскольку это специализированная программа с существенными лицензионными отчислениями. Эта программа позволяет лицензировать API, не являющееся частью API расширения, обсуждаемого в этой книге. Разработчики — участники программы могут создавать собственные редакторы и дизайнеры, чтобы интегрировать новые .NET-языки в IDE.

К компаниям, изготавливающим специальные интегрированные продукты, относятся ActiveState (производитель Visual Perl .NET и Visual Python .NET), Compuware (производитель DevPartner Profiler) и Fujitsu (производитель NetCOBOL для .NET).

О VSIP см. по адресу http://msdn.microsoft.com/vstudio/vsip/.

# Что дальше

В главе 2 мы продолжим обсуждение Visual Studio .NET, сосредоточившись на средствах IDE для управления проектами. В этой главе будет приоткрыто одно из самых значительных преимуществ Visual Studio ,NET — возможность соединения в одном решении проектов на разных языках программирования.



ГЛАВА

# 2

# Управление проектами в Visual Studio .NET

Почти все, что приходится делать в Microsoft Visual Studio .NET, вращается вокруг решений (solutions) и проектов (projects). Что это такое, вы узнаете из этой главы. Мы также расскажем, как управлять проектами в Visual Studio .NET и как организовать их, чтобы они максимально задействовали возможности IDE.

## Обзор решений и проектов

Visual Studio .NET помогает управлять программными проектами, организуя их в структуры из проектов и решений (групп проектов), а также задавая ссылки на сборки и компоненты вне этой структуры. Такая организация, допускающая ссылки, позволяет многократно использовать код, обращаться к связанным проектам, сборкам, COM-компонентам и исходному коду. Простейший способ повторно использовать .NET-код — задать ссылку на сборку в проекте/решении.

**Важно!** Visual Studio .NET организует программные проекты на двух концептуальных уровнях. *Решения* содержат как проекты, так и элементы, не являющиеся проектами. *Проекты* содержат исходные файлы, которые компилируются в исполняемые файлы и сборки.

Самый важный инструмент управления проектом в Visual Studio .NET — Solution Explorer (Ctrl+Alt+L) (рис. 2-1). Solution Explorer (проводник решений) — это окно, отображающее структуру проекта в виде дерева, что обеспечивает доступ ко всем проектам и файлам, являющимся частями открытого в данный момент решения. Visual Studio .NET позволяет работать только с одним решением, но, если требуется параллельно работать с несколькими решениями, можно запустить несколько экземпляров Visual Studio.




**Рис. 2-1.** *Решения являются контейнерами для проектов и элементов решения*

Большинство новых проектов создается на основе шаблонов, разработанных интеграторами языков (разработчиками языков, встраиваемых в Visual Studio), Так, Visual Studio .NET 2003 поставляется с поддержкой Microsoft Visual Basic .NET, Visual C#, Visual J# и Visual C++ .NET. Для каждого из этих языков есть несколько типов проектов, один из которых выбирает программист, создавая новый проект, По умолчанию этот проект создается как часть нового решения. Можно добавлять проекты к существующему решению.

Для приложений Windows Forms и неуправляемых Windows-приложений файлы решения хранятся по умолчанию в той же папке, что и проект. Для приложений Web Forms файлы решения обычно хранятся в папке Visual Studio Projects, вложенной в папку My Documents, и указывают на Web-сервер, на котором размещено данное Web-приложение,

Проект может быть членом многих решений. Поскольку реорганизовать проект в Visual Studio .NET довольно просто, можно спокойно создавать проекты в решении по умолчанию. Позднее проекты можно перемещать и добавлять к новым решениям.

### Практикум: поиграем с решениями

Лучший способ освоить решения и проекты — поиграть с ними. Допустим, вы пишете книгу, у вас есть несколько примеров, и вы хотите, чтобы читатель видел все примеры каждой главы в одном решении.

Вначале создайте папку Chapter02, которая будет содержать папки разных проектов, входящих в решение. Затем создайте проект Visual C# Windows Application и назовите его Sample01.csproj. Сохраните проект и назовите решение Sample01.sln. Сохраните решение в той же папке, что и проект Sample01.

*см. след. стр.*

Перейдите в Solution Explorer, выделите файл решения, в меню File выберите пункт Save Sample01.sln As и сохраните файл в папке Chapter02, назвав его Chapter02.sln.

К этому моменту вы получили два решения; одно в папке Chapter02, другое — в папке Sample01. Текущим открытым файлом является Chapter02.sln. Добавим в решение другой проект. Нажав Ctrl+Alt+A, введите **File.AddNewProject** в окне Command Window. Появится диалоговое окно Add New Project Создайте проект C# Windows Application и назовите его Sample02. Теперь у вас есть файл Chapter02 и два проекта. Закройте решение Chapter02 и щелкните Yes в появившейся подсказке.

В папке Sample02 теперь нет файла решения для этого проекта. Не страшно — просто откройте файл Sample02.csproj. Вы можете работать только с этим проектом и сохранить при желании новое решение в папке Sample02. Вы можете закрыть этот проект и просмотреть содержимое папки Chapter02 — и опять вы увидите два проекта. Итак, проект может быть членом нескольких решений, Это важно понять для планирования разработки проектов в команде. Эти концепции лежат в основе повторного использования кода в Visual Studio .NET.

# Решения

В Visual Studio .NET решение — это тонкая обертка одного или нескольких проектов. Каждый проект в Visual Studio .NET по умолчанию является частью решения, и даже если вы открываете для редактирования только файл проекта, в какой-то момент сеанса редактирования вам будет предложено сохранить для него новый файл решения. Концепция решений важна, поскольку многое из того, что можно делать в Visual Studio .NET, связано с доступом к функциональности, предоставляемой разными проектами.

## Элементы решения и дополнительные файлы

Кроме проектов, решения могут содержать элементы решений и дополнительные файлы. К элементам решения относятся HTML-файлы, растровые картинки, значки, XML-файлы, шаблоны и схемы. Дополнительные файлы могут показаться загадочными. Чтобы задействовать такие файлы, надо сделать видимой папку Miscellaneous Files в Solution Explorer, установив флажок Show Miscellaneous Files In Solution Explorer на вкладке Documents в папке Environment диалогового окна Options (рис. 2-2). Помните: вы не увидите папку Miscellaneous Files, пока не откроете в IDE элемент, не являющийся проектом, используя *File.OpenFile*.

Дополнительные файлы являются файлами, открываемыми в IDE с целью получения дополнительных данных, например, просмотреть код, который не хотелось бы делать частью проекта. Открытие такого файла в IDE без импортирования его в решение автоматически помещает этот файл в папку Miscellaneous Files. Такие связанные файлы собираются в папке Miscellaneous Files, входящей в решение.

Помните: на любой файл, открываемый в Visual Studio .NET, создается ссылка в папке Miscellaneous Files. В этой папке ваши элементы хранятся между сеанса-

ми, если установить значение Miscellaneous Files Project Saves Last равным, скажем, 3 элементам. Это значит, что можно открывать спецификации, календарные планы, заметки и иметь быстрый доступ к этим файлам каждый раз при открытии проекта (рис. 2-3).

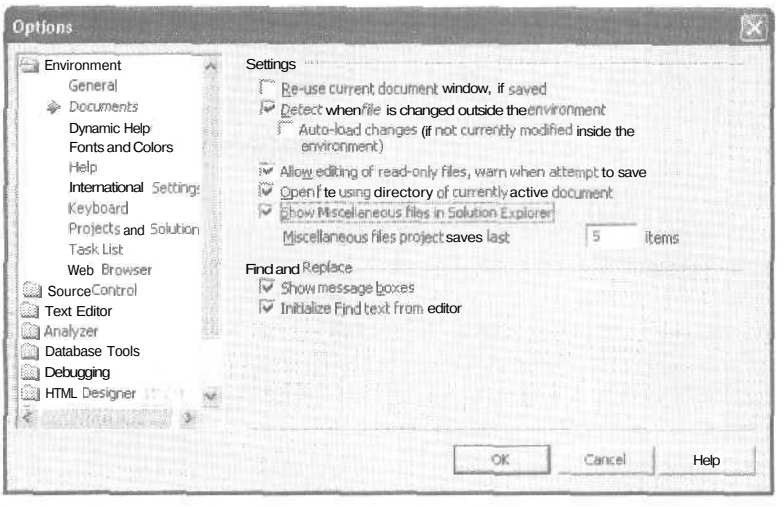



**Рис.** 2-2. *Вы можете разрешить отображение папки Miscellaneous Files в диалоговом окне Options*

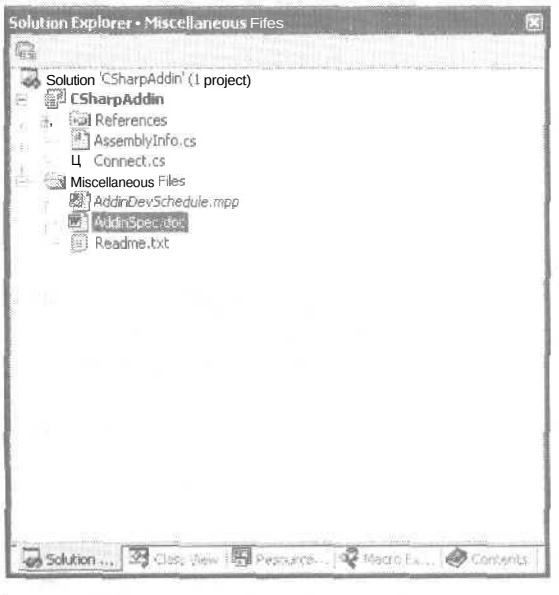
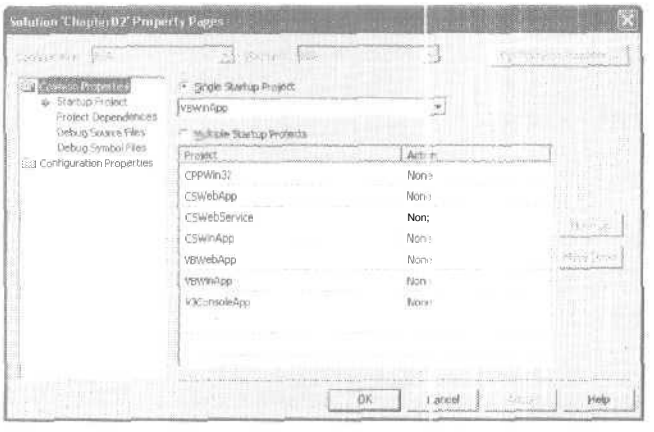



**Рис.** 2-3. *В папке Miscellaneous Files можно хранить ссылки на документы, связанные с проектами*

## Свойства решений

Диалоговое окно Solution Property Pages дает простой доступ к параметрам, применяемым ко всему решению. Параметры, которыми можно управлять, включают задание начального проекта или проектов в решении, расположение файлов и символьных представлений, используемых при отладке, и конфигурационные параметры, применяемые к разным проектам в решении.

В окно Solution Property Pages (рис. 2-4) можно попасть, выбрав имя решения в Solution Explorer, нажав Ctrl+Alt+A и введя **Project.Properties** в появившемся окне Command Window или нажав правую кнопку мыши и выбрав из контекстного меню пункт Properties.




**Рис,** 2-4. *Диалоговое окно Solution Property Pages предоставляет доступ к свойствам решения*

### Общие свойства

Выбор папки Common Properties (общие свойства) в левом окошке дает доступ к нескольким параметрам. Первый — это Startup Project (начальный проект). В решениях с несколькими проектами можно выбрать проект, запускаемый при выполнении решения из меню Debug. Чаще всего эта возможность устанавливается «на лету» при щелчке правой кнопкой имени проекта в Solution Explorer с последующим выбором из контекстного меню пункта Set As Startup,

Включение переключателя Multiple Startup Projects позволит запускать более одного проекта при выборе пунктов Start или Start Without Debugging из меню Debug. Выбор этого параметра позволяет установить поведение каждого проекта при вызове *DebugStart* или *DebugStartWithoutDebugging*Для каждого проекта можно задать Start, Start Without Debugging или None. Кнопки Move Up и Move Down справа от списка проектов позволяют установить порядок, в котором программы начнут выполняться.

В некоторых сценариях может быть полезен параллельный запуск нескольких проектов. Так, можно протестировать взаимодействие процессов различных сборок в решении. Можно использовать второй проект для построения профиля или

оснастки. Еще один вариант — запуск утилиты, управляющей другой сборкой, с целью автоматического тестирования.

Другой параметр в папке Common Properties — это Project Dependencies. Когда в одном решении некоторые сборки зависят от других, порядок построения проектов в решении критически важен. Параметры Project Dependencies позволяют указать, какие проекты построить раньше других, чтобы все решение выполнялось, как надо.

Последние два параметра папки Common Properties позволяют установить пути к исходным и символьным файлам приложения. Эти параметры позволяют добраться до исходного кода библиотек, не являющихся частью проекта, на которые, однако, проект ссылается. При отладке проекта, ссылающегося на отладочную версию .NET-сборки, Visual Studio .NET обычно имеет возможность найти исходник сборки при условии его доступности. Если же исходник хранится в другом месте, можно задать положение исходных и символьных файлов, необходимых отладчику, что и позволит вести отладку.

### Свойства конфигураций

Решения могут иметь несколько конфигураций, что позволяет получить быстрый доступ к предварительной установке параметров, связанных с решением. Конфигурации Debug и Release доступны в новых проектах по умолчанию, но можно создать и собственные конфигурации, используя Configuration Manager, к которому позволяет добраться диалоговое окно Solution Property Pages или меню Build (*Build.ConfigurationManager*).

Visual Studio .NET предлагает два типа конфигураций: решения и проекта. Первый служит для настройки параметров компоновки отдельных решений. Так, можно создать и сохранить решение, позволяющее выбирать конфигурацию для каждого проекта в решении.

Второй мы подробно обсудим чуть ниже, а пока рассмотрим, как разные проекты могут быть связаны в решении. Конфигурация проекта позволяет изменить специальные характеристики, включая оптимизацию кода, отладочные переключатели и даже расположение компилируемых файлов проекта. Если в одном решении пять проектов и для них заданы разные значения параметров, следует задать и собственную конфигурацию решения, позволяющую сохранить и управлять разными сценариями компоновки.

## Файл решения и файл параметров пользователя

Исходный файл решения (.sln) — простой текстовый документ, описывающий решение, — содержит ссылки на проекты, содержащиеся в решении. Он также содержит сведения о версии формата самого файла решения. Файлы решений, созданных в Visual Studio .NET 2002, имеют подпись *Microsoft Visual Studio Solution File, Format Version 7.00*, а файлы решений, создаваемых или конвертированных в Visual Studio .NET 2003, — *Microsoft Visual Studio Solution File, Format Version 8.00*.

**Важно!** После конвертирования файла решения в формат Visual Studio .NET 2003 его нельзя будет открыть в Visual Studio .NET 2002,

Файл .sln содержит также сведения о конфигурациях, установленных для решения, а также информацию о том, как организованы конфигурации проектов в конфигурациях решений. Следующий код описывает организацию нескольких типов проектов этой главы (листинг 2-1).

**Листинг 2-1.** Пример файла решения

**Chapter02.sln**

```
Microsoft Visual Studio Solution File, Format Version 7.00
Project("{F184B08F-C81C-45F6-A57F-5ABD9991F28F}") * "VBWinApp",
    "VBWinApp\VBWinApp.vbproj",
    "{D7AFF922-38D5-461C-A07B-859080BFCFBF}"
EndProject
Project("{F184B08F-C81C-45F6-A57F-5ABD9991F28F}") = "VBWebApp",
    "http://localhost/VBWebApp/VBWebApp.vbproj",
    "{D654F4FC-7144-457A-9D73-A149ECD0DB40}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "CSWinApp",
    "CSWinApp\CSWinApp.csproj",
    "{9D585FB1-AA56-4227-AB14-23F16E4F07E4}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "CSWebApp",
    "http://localhost/CSWebApp/CSWebApp.csproj",
    "{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}"
EndProject
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") ж "CPPWin32",
    "CPPWin32\CPPWin32.vcproj",
    "{912ECF9E-5ABA-4E85-8955-2B0DC464C377}"
EndProject
Project("{E6FDF86B-F3D1-11D4-8576-0002A516ECE8}") - "VJConsoleApp",
    "VJConsoleApp\VJConsoleApp.vjsproj",
    "{2022B3FD-6AFB-4912-8687-4B09257A48A1}"
End Project
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "CSWebService",
    "http://localhost/CSWebService/CSWebService.csproj",
    "{22A37BFE-687D-44E6-9C0C-711735BC5018}"
EndProject
Global
    GlobalSection(SolutionConfiguration) - preSolution
        ConfigName.0 = Debug
        ConfigName.1 = Debug1
        ConfigName.2 = PostCMD
        ConfigName.3 * PostExplorer
        ConfigName.4 = Release
    EndGlobalSection
    GlobalSection(ProjectDependencies) = postSolution
    EndGlobalSection
    GlobalSection(ProjectConfiguration) - postSolution
    {D7AFF922-38D5-461C-A07B-859080BFCFBF}.Debug.ActiveCfg = Debug|.NET
```

```
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.Debug.Build.0 = Debug|.NET
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.Debug1.ActiveCfg = Debug|.NET
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.Debug1.Build.0 = Debug|.NET
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.PostCMD.ActiveCfg * ftelease|.NET
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.PostCMD.Build.0 = Release|.NET .
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.PostExplorer.ActiveCfg =
   Release|.NET
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.PostExplorer.Build.0 =
   Release|.NET
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.Release.ActiveCfg *
   Release|.NET
{D7AFF922-38D5-461C-A07B-859080BFCFBF}.Release.Build.0 = Release|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.Debug.ActiveCfg = Debug|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.Debug.Build.0 - Debug|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.Debug1.ActiveCfg - Debug|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.Debug1.Build.0 = Debug|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.PostCMD.ActiveCfg = Release|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.PostCMD.Build.0 = Release|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.PostExplorer.ActiveCfg =
   Release|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.PostExplorer.Build.0 =
   Release|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.Release.ActiveCfg = Release|.NET
{D654F4FC-7144-457A-9D73-A149ECD0DB40}.Release.Build.0 = Release|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.Debug.ActiveCfg = Debug|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.Debug.Build.0 = Debug|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.Debug1.ActiveCfg * Debug1|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.Debug1.Build.0 = Debug1|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.PostCMD.ActiveCfg = Debug1|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.PostCMD.Build.0 = Debug1|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.PostExplorer.ActiveCfg =
   Debug1|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.PostExplorer.Build.0 *
   Debug1|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.Release.ActiveCfg = Release|.NET
{9D585FB1-AA56-4227-AB14-23F16E4F07E4}.Release.Build.0 = Release|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.Debug.ActiveCfg = Debug|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.Debug.Build.0 - Debug|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.Debug1.ActiveCfg * Debug|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.Debug1.Build.0 = Debug|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.PostCMD.ActiveCfg = Release|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.PostCMD.Build.0 = Release|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.PostExplorer.ActiveCfg =
   Release|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.PostExplorer.Build.0 -
   Release|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.Release.ActiveCfg = Release|.NET
{AB66D7EB-DB6F-45E8-AE7D-B972D2A652F1}.Release.Build.0 = Release|.NET
{912ECF9E-5ABA-4E85-8955-2B0DC464C377}.Debug.ActiveCfg * Debug|Win32
{912ECF9E-5ABA-4E85-8955-2B0DC464C377}.Debug.Build.0 = Debug|Win32
```

*см. след. стр.*

```
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.Debug1.ActiveCfg = Debug|Win32
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.Debug1.Build.0 = Debug|Win32
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.PostCMD.ActiveCfg =
       PostCMD|Win32
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.PostCMD.Build.0 = PostCMD|Win32
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.PostExplorer.ActiveCfg =
       PostExplorer|Win32
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.PostExplorer.Build.0 -
       PostExplorer|Win32
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.Release,ActiveCfg =
       Release|Win32
    {912ECF9E-5ABA-4E85-8955-2B0DC464C377}.Release.Build.0 = Release|Win32
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.Debug.ActiveCfg * Debug|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.Debug.Build.0 = Debug|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.Debug1.ActiveCfg = Debug|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.Debug1.Build.0 = Debug|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.PostCMD.ActiveCfg = Release|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.PostCMD.Build.0 = Release|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.PostExplorer.ActiveCfg =
       Release|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.PostExplorer.Build.0 =
       Release|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.Release.ActiveCfg = Release|.NET
    {2022B3FD-6AFB-4912-8687-4B09257A48A1}.Release.Build.0 = Release|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.Debug.ActiveCfg = Debug|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.Debug.Build.0 = Debug|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.Debug1.ActiveCfg = Debug|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.Debug1.Build.S = Debug|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.PostCMD.ActiveCfg = Release|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.PostCMD.Build.0 = Release|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.PostExplorer.ActiveCfg -
       Release |.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.PostExplorer.Build.0 =
       Release|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.Release.ActiveCfg = Release|.NET
    {22A37BFE-687D-44E6-9C0C-711735BC5018}.Release.Build.0  = fieleasef.NET
  EndGlobalSection
  GlobalSection(ExtensibilityGlobals) = postSolution
  EndGlobalSection
  GlobalSection(ExtensibilityAdd-ns) * postSolution
  EndGlobalSection
EndGlobal
```

Как видите, в файле решения, в которое включались или добавлялись элементы решения, информации об этих элементах нет. Элементы решения рассматриваются как элементы пользователя, так что ссылки на файлы, задающие элементы решения, хранятся в файле параметров пользователя (.suo). При передаче папки, содержащей файлы .sln и .suo, другому пользователю или на другую машину большая часть информации, содержащейся в файле .suo, бесполезна и будет проигнорирована.

Некоторые важные элементы, хранимые в файле .suo, возможно, захочется передать другому пользователю. Так, точки прерывания, установленные в решении, являются задачами, добавленными в окне Task List. Чтобы сделать эту информацию доступной людям, совместно работающими над решением, надо позаботиться о сохранении файла .suo вместе с файлом ,sln. Если же в этом необходимости нет, файл .suo лучше удалить, так как он может содержать конфиденциальную информацию, такую как сетевые пути и даже ваш псевдоним e-mail.

# Проекты

Проекты являются вторым типом контейнеров, используемых в Visual Studio .NET. Они позволяют управлять исходными файлами, связанными с индивидуальными сборками, Web-узлами и службами, а также приложениями. Как и в случае решений, основное средство для управления проектами в Visual Studio .NET — Solution Explorer.

## Элементы проекта

Проекты в Visual Studio .NET состоят главным образом из файловых элементов. Эти элементы могут быть ссылками на файлы или исходными файлами, хранящимися в той же папке, что и файл проекта. Является элемент ссылкой или фактическим файлом, зависит от типа проекта. Файлы, связанные с проектами Visual C++, являются ссылками, отображаемыми в Solution Explorer. Так случилось, что эти файлы обычно находятся в той же папке, что и проекты. Удаление ссылки на файл в проекте Visual C++ не обязательно приводит к удалению файла, открываемого по этой ссылке. Это просто приятное исключение, но, если вам случалось перемещать проект Visual C++ и обнаруживать отсутствие файла проекта, этот файл мог находиться вне папки проекта. Проекты Windows Forms могут состоять из смеси ссылок и действительных файловых элементов. Проекты Web Forms обычно содержат действительные файлы в папке, структура которой задается файлом проекта. Вот возможные отношения между типом проекта и файловыми элементами в Visual Studio .NET 2003 (табл. 2-1):

**Табл. 2-1. Элементы проекта в Visual Studio .NET 2003**

| Тип проекта | Элементы |
|---|---|
| Visual C++ | Ссылки на элементы |
| Visual Basic .NET (Web) | Элементы в папке проекта |
| Visual C# .NET (Web) | Элементы в папке проекта |
| Visual Basic .NET | Ссылки и действительные элементы |
| Visual C* .NET | Ссылки и действительные элементы |
| Visual J# | Ссылки и действительные элементы |

Если взглянуть в Solution Explorer на приложение Windows Forms, написанное на Visual C# или Visual Basic .NET, то, используя *Project.ShowAllFiles* вы увидите смесь из элементов проекта и элементов файловой структуры. Файлы, являющиеся частью проекта, будут появляться в нормальном виде. Файлы, не входящие в проект, но расположенные в папке проекта, будут показаны серым цветом. Команда *Show-*

*AilFiles* доступна из меню Project и по нажатию кнопки на панели инструментов в Solution Explorer. Дополнительно можно будет увидеть файлы, которые по умолчанию спрятаны от пользователя. К ним относятся некоторые виды конфигурационных файлов и файлы с фоновым кодом (code-behind), которые применяются в приложениях ASP.NET. На рис. 2-5 большинство из показанных в проекте составляющих являются элементами проекта.




**Рис. 2-5.** *Ссылки и файлы в решении Visual C#*

Кроме элементов, проект хранит метаданные о конфигурации, связанной с проектом, включая сведения о конфигурации, заданные в IDE, а также об отладке и сборке. Природа этих данных зависит от типа проекта. Все языки .NET компилируются в промежуточный язык MSIL (Microsoft Intermediate Language), но сами компиляторы написаны разными командами разработчиков, и потому доступные параметры варьируются от языка к языку

## Свойства проекта

Параметры проекта задаются в диалговом окне Property Pages. (В Solution Explorer щелкните правой кнопкой имя проекта и выберите Properties в контекстном меню.) Property Pages во многом подобно диалоговому окну Solution Property Pages, но имеет значительно больше изменяемых параметров проекта. Большинство из них — это параметры, которые иначе следовало бы задать в командной строке при компиляции проекта; они соответствуют определенным параметрам командной строки.

Разговор о большинстве параметров компилятора поведем с учетом четырех основных языков, поставляемых вместе с Visual Studio .NET, и различных типов проектов. Вот когда конфигурация проекта в Visual Studio .NET становится полезной! Создавая пользовательские конфигурации проекта, можно испытать разные типы сборки проекта и сохранить эти конфигурации для дальнейшего применения и ссылок на них.

### Сохранение пользовательской конфигурации

Диалоговое окно Configuration Manager вызывается щелчком кнопки Configuration Manager в диалоговом окне Property Pages или вводом команды **Build.ConfigurationManager** в окне Command Window. Чтобы создать новую конфигурацию проекта, щелкните на строке с именем желаемого проекта кнопку с выпадающим списком, расположенную в столбце Configuration таблицы Project Contexts, и выберите New из предложенного списка. Откроется диалоговое окно New Project Configuration (рис. 2-6).

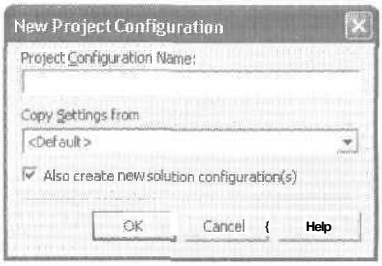



**Рис.** 2-6. *Диалоговое окно New Project Configuration*

Задайте имя новой конфигурации и начальные параметры, копируя их из существующих конфигураций, выбрав нужную конфигурацию из выпадающего списка в окне Copy Settings From. Как и в диалоговом окне New Solution Configuration, можно автоматически создать соответствующую проекту новую конфигурацию для решения, установив флажок Also Create New Solution Configuration(s).

Полагаю, вы готовы поиграть с параметрами проекта. Просто создайте новую тестовую конфигурацию, с которой можно будет экспериментировать, а для двух существующих по умолчанию конфигураций оставьте все параметры по умолчанию. Теперь рассмотрим типы проектов и некоторые из параметров, которые можно задать в каждой из конфигураций.

### Проекты управляемых приложений

Управляемые приложения, такие как Visual Basic .NET, Visual C# и Visual J#, используют похожие раскладки в диалоговом окне Property Pages. В зависимости от того, какое приложение создается — Windows Forms или Web Forms, доступны разные папки.

### Общие свойства

На рис. 2-7 показана страница General, расположенная в папке Common Properties для приложения Visual Basic .NET Windows Forms. Эта страница отличается от страницы General в папке Configuration Properties. Заметьте: поле со списком Configuration, поле со списком Platform, кнопка Configuration Manager недоступны, когда открываются элементы из папки Common Properties. Эти параметры не зависят от конфигурации, и их невозможно сохранить отдельно для каждого типа новой компоновки. Параметры в этой папке задают такие свойства, как Assembly Name, Output Type, и Namespace. Другие параметры Common Properties включают Page Layout, позволяющий указать тип раскладки (Flow или Grid) для приложений Web Forms и Windows Forms: Target Schema, позволяющий указать тип и версию

Web-браузера, с которым приложение совместимо, и язык сценариев, используемый приложением.

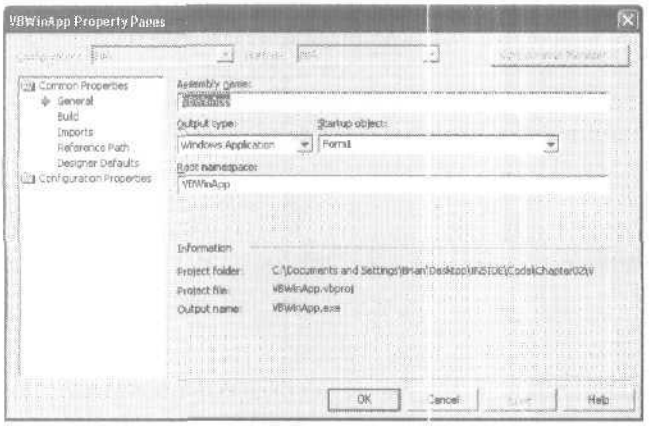



**Рис.** 2-7. *Страница General в папке General для приложения Visual Basic .NET Windows Forms*

Противоположным по стилю язляется диалоговое окно, демонстрирующее страницу General для приложения Visual C# Web Forms (рис. 2-8). Для настройки параметров на странице Visual Basic .NET служат элементы управления, на странице Visual C# они задаются в таблице.

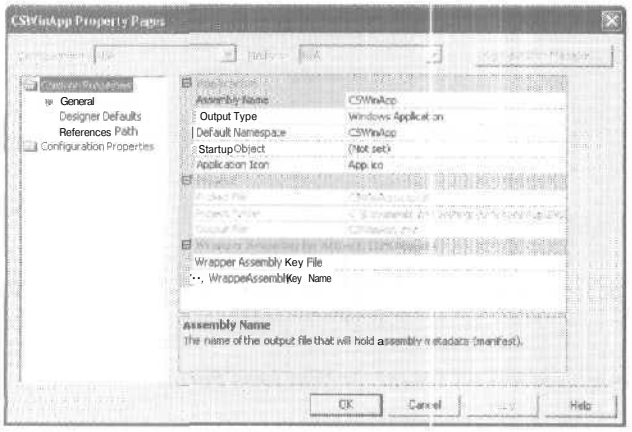



**Рис. 2-8.** *Страница General для приложения Visual C# Windows Forms*

Эти два типа стилей страницы свойств постоянны для каждого языка. И хотя эти типы стилей визуально отличаются, они обеспечивают задание одних и тех же параметров.

### Свойства конфигурации

Папка Configuration Properties проекта — это то место, где можно поиграть с параметрами и сохранить их для отдельного типа компоновки (build). Создать и сохранить новые типы компоновки довольно просто почти для любого типа про-

екта в Visual Studio .NET, затем можно скомпилировать их в IDE или из командной строки (см. об этом главу 15).

В этом разделе рассмотрим несколько важных параметров Configuration Properties, задаваемых для проектов Visual C# в диалоговом окне Property Pages. Эти же параметры можно задать и в диалоговом окне Property Pages для проектов Visual Basic .NET, но отдадим предпочтение Visual C#, поскольку избранный здесь стиль таблицы делает страницы свойств более выразительными.

На рис. 2-9 вы видите страницу Build папки Configuration Properties для проекта Visual C#. Любые из этих параметров можно сохранить в пользовательском типе компоновки. Одним из самых полезных параметров конфигурации в нем — Output Path. Положением выходных файлов компоновки для конфигурации Visual C# Debug по умолчанию является папка \bin\Debug\. Компоновка конфигурации Relea.se по умолчанию размещена в \bin\Release\. При создании пользовательского типа компоновки предлагается один из этих путей в зависимости от того, какой тип компоновки выбран при инициализации параметров. Если создавать пользовательский тип компоновки, имеет смысл разместить выходные файлы в новой папке, что позволит сравнивать выходные сборки. В случаях, подобных этому, разумно задавать новый путь компоновки, соответствующий ее имени. Так, если компоновка названа DebugOverflow (что указывает на включение проверки переполнения для этого типа компоновки), можно изменить путь вывода на \bin\DebugOverflow.

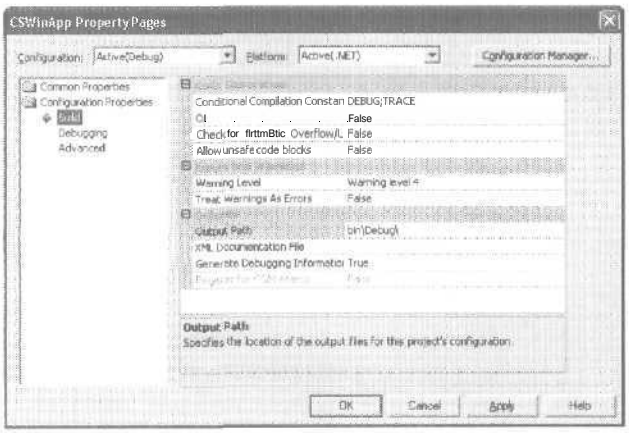



Рис. 2-9. *Страница Build для приложения Visual C# Windows Forms*

Страница Debugging (рис. 2-10) может быть особенно полезна при построении проектов библиотек классов, Web Forms и Web-сервисов XML. Можно поиграть с параметрами на этой странице, но для целей нашего обсуждения особенно полезен Start URL. Используя разные типы компоновки, можно задавать особые URL, указывающие на Web-сервисы XML, которые надо протестировать Для тестирования библиотек можно использовать параметр Start Application. Это позволяет легко отлаживать сервисы и библиотеки для нескольких тестовых приложений.

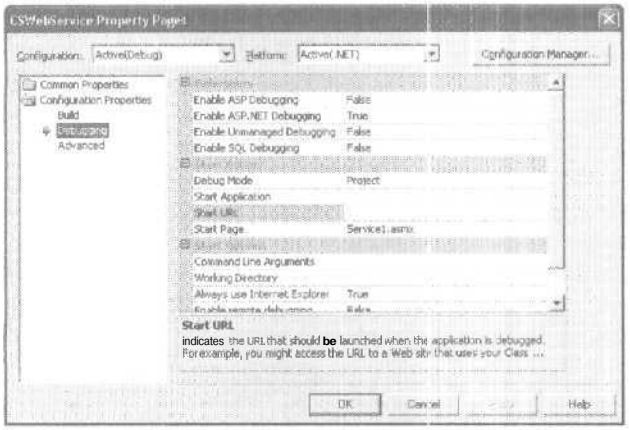



Рис. 2-10. *Страница Debugging для приложения Web Forms*

### Параметры для командной строки

В своих компоновках я часто использую параметр Command Line Arguments. Создавая собственный тип компоновки и задавая параметры командной строки для него, я экономлю массу времени. Этот параметр пришелся весьма кстати, когда я был техническим редактором книги по безопасности и мне нужно было протестировать несколько строк в сценарии с переполнением буфера. Создав три или четыре конфигурации компоновки, я мог быстро отладить и протестировать приложение, используя различные аргументы командной строки. Еще важнее, что у меня была возможность создать эти типы компоновки, перейти на время к другой работе и возвратиться к проекту по безопасности, войдя прямо в него, поскольку я сохранил все мои тестовые сценарии как пользовательские компоновки.

Страница Advanced (рис, 2-11) имеет не менее полезные параметры, особенно для тестеров. На странице Visual C# Advanced в папке Configuration Properties можно указать смещение для DLL, что может пригодиться при отладке. Можно задать размер раздела для выходного файла. Обычно в Windows его значение равно 4 Кб, но в .NET можно указать 512, 1024, 2048, 4096, 8192 или 16384. Параметр Incremental Build важен, если вы используете параметр */doc* для генерации документации из комментариев в исходном коде. Параметр */doc* игнорируется, когда задана инкрементная компоновка, так что, возможно, вы захотите создать пользовательский тип компоновки и установить для него свойство Incremental Build в False, что гарантирует обновление документации при отладке.

### Проекты Visual C++

Диалоговое окно Property Pages для проектов Visual C++ имеет массу параметров, так как при компиляции и компоновке доступно много возможностей. Параметры пользовательских компоновок, *о* которых мы говорили выше, применимы и к Visual C++. Пользовательские параметры для неуправляемых проектов весьма удобны, особенно при тестировании и обучении.

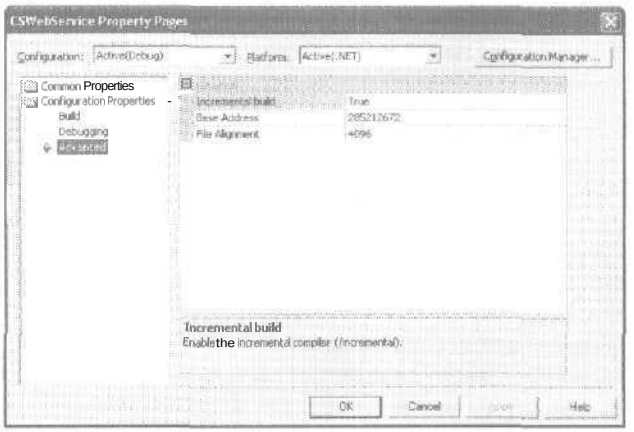



**Рис. 2-11.** *Страница Advanced для приложения Web Forms*

На рис. 2-12 вы видите диалоговое окно Property Pages в проектах Visual C++ Win32: в левом окне со списком папок корневая папка единственная, Все в ней можно настроить и сохранить в уникальной конфигурации проекта.

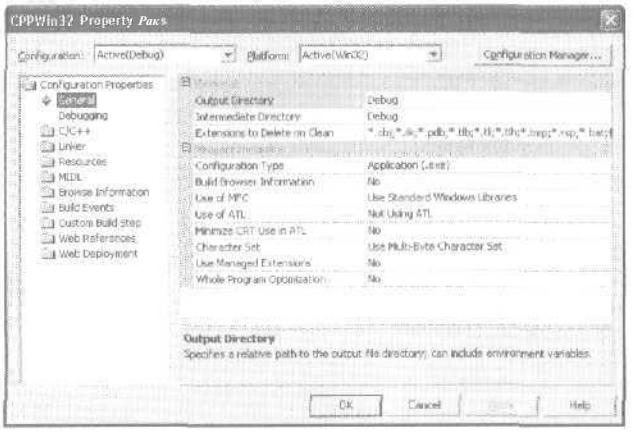



**Рис. 2-12.** *Настройка конфигурации в Visual C++*

Диалоговое окно Property Pages для проектов Visual C++ имеет несколько подпапок, вложенных в папку Configuration Properties. Табл. 2-2 содержит список этих папок и общие свойства, которые можно установить в каждой из них,

**Табл. 2-2. Подпапки Configuration Properties в Visual C++**

| Подпапка | Свойства |
|---|---|
| C/C++ | Параметры компилятора, препроцессорные определения, пути к некоторым выходным файлам, параметры командной строки. |
| Linker | Параметры компоновки, параметры отладки и параметры компоновки из командной строки. |
| Resources | Имя и путь файла ресурсов, региональные стандарты, командная строка компилятора ресурсов. |

*см. след. стр.*

Табл. 2-2. *(окончание)*

| Подпапка | **Свойства** |
|---|---|
| MIDL | Параметры компилятора Microsoft Interface Definition Language (MIDL), пути выходных файлов и командная строка компилятора. |
| Browse Information | Параметры, связанные с BSCMAKE (файлами обозревателя). |
| Build Events | Команды, которые можно выполнять во время процесса компоновки. |
| Custom Build Step | Свойства конфигурирования дополнительных задач при компоновке файла или проекта; так. можно передать входной файл инструментальному средству, возвращающему выходной файл. |
| Web References | Свойства, определяющие способ создания прокси-класса Web-сервиса, на который есть ссылка в проекте. |
| Web Deployment | Указывает, как средства развертывания через Web будут устанавливать ваше приложение. |

Для опытных программистов на Visual C++ большая часть этих параметров понятна. Папка Web References представляет интерес, так как позволяет в Visual Studio .NET ссылаться на Web-сервисы XML, и IDE создаст подходящий прокси-класс, что упрощает доступ к функциональности этого сервиса.

Папка Build Events позволяет выполнять несколько интересных вещей при настройке пользовательских компоновок. На рис. 2-13 показана страница Post-Build Event. При работе с несколькими проектами и компоновками запускать приложения и сценарии во время компоновки можно из папки Build Events. В нашем случае добавляется вызов Regsvr32.exe из командной строки для Post-Build Event, После того как проект скомпонован в этой конфигурации, целевой файл будет зарегистрирован в Windows.

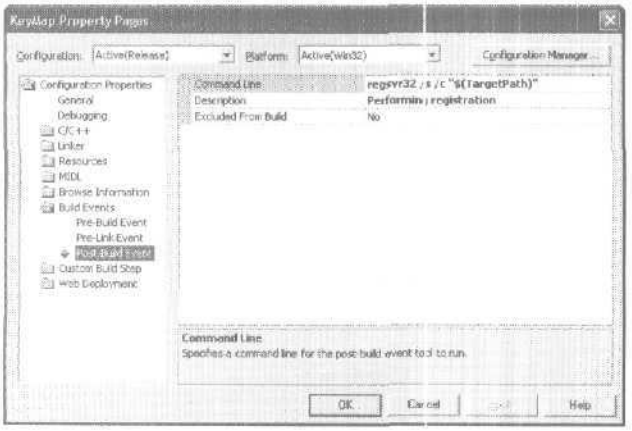



Рис. 2-13. *События компоновки позволяют выполнять приложения во время компоновки*

### Практикум: добавление события КОМПОНОВКИ

Это простенький, но полезный пример. Скажем, вы разбираетесь с гарогра: - мой на C и хотите открыть папку, в которую помещается вновь скомпонованный файл, Вы можете добавить новый инструмент в меню Tools в Visual Studio .NET, что позволит открыть в Windows Explorer папку с компоновкой текущего проекта. (Вам надо добавить такое средство, поскольку так проще всего получить нужную папку.) Но давайте будем по-настоящему ленивы и создадим посткомпоновочное событие, которое и запустит Explorer.exe и откроет папку, где размещена компоновка.

1. Создайте пользовательскую компоновку и назовите ее PostExplorer.
2. Добавьте в Post-Build Event следующую строку в поле Command Line:

```
Explorer.exe $(TargetDir)
```

3. Постройте проект заново,

Бы увидите, что компоновка нормально завершится. По завершении компоновки вы должны увидеть, что Windows Explorer открыл вашу целевую папку. Теперь его можно запустить двойным щелчком exe-файла, проверить файлы, созданные при компоновке, и прочитать файл BuildLog.htm.

Диалоговое окно Property Pages допускает применение нескольких макросов (табл. 2-3).

**Табл. 2-3. Макросы, используемые в Property Pages**

| Макрос | Описание |
|---|---|
| *$(ConfigurationName)* | Текущая конфигурация проекта |
| *$(DevEnvDir)* | Папка установки Visual Studio .NET |
| *$(FrameworkDir)* | Папка установки .NET Framework |
| *$(FrameworkSDKDir)* | Папка установки .NET Framework SDK |
| *$ (FrameworkVersion)* | Номер версии .NET Framework |
| *$(Inherit)* | Порядок наследуемых свойств командной строки, созданной Visual Studio .NET |
| *$(InputDir)* | Папка с исходным файлом, эквивалентна папке с файлом проекта |
| *$(InputExt)* | Расширение исходного файла |
| *$(InputFileName)* | Имя исходного файла (имя + расширение) |
| *$(InputName)* | Имя исходного файла |
| *$(InputPath)* | Полный путь к исходному файлу |
| *$(IntDir)* | Промежуточный путь к исходному файлу |
| *$(NoInherit)* | Отменяет наследование свойств |
| *$(OutDir)* | Папка с выходным файлом |
| *$(PlatformName)* | Имя платформы проекта, обычно Win32 или .NET |
| *$(ProjectDir)* | Папка, содержащая файлы проекта |
| *$(ProjectExt)* | Расширение (.vcproj) |

*см. след. стр.*

**Табл. 2-3.** *(окончание)*

| Макрос | Описание |
|---|---|
| *$(ProjectFileName)* | Полное имя проекта (CPPWin32.vcproj) |
| *$(ProjectName)* | Имя проекта (CPPWin32) |
| *$(ProjectPath)* | Полный путь к файлу проекта |
| *$(RemoteMachine)* | Удаленная машина (для удаленной отладки) |
| *$(SolutionDir)* | Папка, содержащая файл решений |
| *$(SolutionExt)* | Расширение файла решений (.sln) |
| *$(SolutionFileName)* | Полное имя решения (Chapter02.sln) |
| *$(SolutionName)* | Имя решения (Chapter02) |
| *$(SolutionPath)* | Полный путь к файлу решений |
| *$(TargetDir)* | Папка с результатами компоновки проекта |
| *$(TargetExt)* | Расширение результата компоновки (.exe) |
| *$(TargetFileName)* | Полное имя файла результата (CPPWin32.exe) |
| *$(TargetName)* | Имя файла результата (CPPWin32) |
| *$(TargetPath)* | Полный путь результата |
| *$(VCInstallDir)* | Папка установки Visual C++ .NET |
| *$(VSInstallDir)* | Папка установки Visual Studio .NET |

## Исходные файлы проекта

У исходных файлов проекта различные расширения, определяемые языком, на котором проект создается (табл. 2-4). При добавлении новых языков, разрабатываемых в соответствии с программой Visual Studio Integration Program (VSIP) поставщики добавляют и новые типы проектов.

**Табл. 2-4. Типы проектов и их расширения**

| Тип проекта | Расширение |
|---|---|
| Visual Basic .NET | .vbproj |
| Visual C# | .csproj |
| Visual C++ | .vcproj |
| Visual J# | .vjproj |
| Deployment | .vdproj |

CSWinApp.csproj — файл проекта Visual C# в Visual Studio .NET (листинг 2-2) — содержит многие из параметров, обсуждавшихся в этой главе.

**Листинг 2-2. Пример файла проекта**

**CSWinApp.csproj**

```
<VisualStudioProject>
    <CSHARP
        ProjectType = "Local"
        ProductVersion = "7.10.2215"
        SchemaVersion = "2.0"
```

```
    ProjectGuid = "{9DC32270-2155-414F-9BE5-C593ADE47FFD}"
>
    <Build>
        <Settings
            ApplicationIcon = "App.ico"
            AssemblyKeyContainerName = ""
            AssemblyName = "CSWinApp"
            AssemblyOriginatorKeyFile = ""
            DefaultClientScript = "JScript"
            DefaultHTMLPageLayout = "Grid"
            DefaultTargetSchema = "IE50"
            DelaySign = "false"
            OutputType = "WinExe"
            PreBuildEvent = ""
            PostBuildEvent = ""
            RootNamespace = "CSWinApp"
            RunPostBuildEvent = "OnBuildSuccess"
            StartupObject = ""
        >
            <Config
                Name = "Debug"
                AllowUnsafeBlocks = "false"
                BaseAddress = "285212672"
                CheckForOverflowUnderflow = "false"
                ConfigurationOverrideFile = ""
                DefineConstants = "DEBUG;TRACE"
                DocumentationFile = ""
                DebugSymbols = "true"
                FileAlignment = "4096"
                IncrementalBuild = "false"
                NoStdLib = "false"
                NoWarn = ""
                Optimize = "false"
                OutputPath = "bin\Debug\"
                RegisterForComInterop = "false"
                RemoveIntegerChecks = "false"
                TreatWarningsAsErrors = "false"
                WarningLevel = "4"
            />
            <Config
                Name = "Release"
                AllowUnsafeBlocks = "false"
                BaseAddress = "285212672"
                CheckForOverflowUnderflow = "false"
                ConfigurationOverrideFile = ""
                DefineConstants = "TRACE"
                DocumentationFile = ""
                DebugSymbols = "false"
                FileAlignment = "4096"
                IncrementalBuild = "false"
```

*см. след. стр.*

```
                NoStdLib - "false"
                NoWarn = ""
                Optimize - "true"
                OutputPath = "bin\Release\"
                flegisterForConilnterop = "false"
                RemoveIntegerChecks * "false"
                TreatWarningsAsErrors = "false"
                WarningLevel - "4"
            />
            <Config
                Name = "Debug1"
                AllowUnsafeBlocks = "false"
                BaseAddress = "285212672"
                CheckForOverflowUnderflow = "true"
                ConfigurationOverrideFile = ""
                DefineConstants = "DEBUG;TRACE"
                DocumentationFile = ""
                DebugSymbols - "true"
                FileAlignment = "4096"
                IncrementalBuild = "false"
                NoStdLib = "false"
                NoWarn = ""
                Optimize - "false"
                OutputPath = 'bin\Debug\"
                fiegisterForCofrrInterop-"false"
                RemoveIntegerChecks * "false"
                TreatWarningsAsErrors - "true"
                WarningLevel = "3"
            />
        </Settings>
        <References>
            <Reference
                Name = "System"
                AssemblyName * "System"
                HintPath =
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\System.dll"
            />
            <Reference
                Name - "System.Data"
                AssemblyName = "System.Data"
                HintPath =
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\System.Data.dll"
            A
            <Reference
                Name = "System.Drawing"
                AssemblyName = "System.Drawing"
                HintPath =
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\System.Drawing.dll"
            />
            <Reference
```

```
                    Name = "System.Windows.Forms"
                    AssemblyName = "System.Windows, Forms"
                    HintPath =
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\System.Windows. Forms.dll"
                />
                <Reference
                    Name = "System.XML"
                    AssemblyName = "System. Xml"
                    HintPath =
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\System.XML.dll"
                />
            </References>
        </Build>
        <Files>
            <Include>
                <File
                    fielPath = "App.ico"
                    BuildAction = "Content"
                />
                <File
                    RelPath = "AssemblyInfo.cs"
                    SubType = "Code"
                    BuildAction = "Compile"
                />      .  . ..".
                <File
                    RelPath = "Form1.cs"
                    SubType = "Form"
                    BuildAction = "Compile"
                />
                <File
                    RelPath = "Form1.resx"
                    DependentUpon = "Form1.cs"
                    BuildAction * "EmbeddedResource"
                />
            </Include>
        </Files>
    </CSHARP>
</VisualStudioProject>
```

Проекты Visual Basic .NET, Visual C#, Visual J# содержат также файлы с параметрами пользователя: *ProjectNameProjectExt*.user.

Файл параметров пользователя Visual Basic .NET имеет расширение .vbproj.user, является XML-файлом и содержат сведения о создаваемых вами пользовательских компоновках. Файл параметров пользователя проекта Visual C# в листинге 2-3 соответствует файлу проекта .csproj из листинга 2-2. Здесь есть пользовательская конфигурация, добавленная к проекту в процессе разработки, — Debug 1. В отличие от двоичного файла .suo файлы .user встроены в пользовательскую компоновку и должны обычно храниться вместе с проектом.

**Листинг 2-3.** Пример файла параметров пользователя

**CSWinApp.csproj.user**

```
<VisualStudioProject>
    <CSHARP LastOpenVersion = "T. 10.2215" >
        <Build>
            <Settings ReferencePath = "" >
                <Config
                    Name = "Debug"
                    EnableASPDebugging = "false"
                    EnableASPXDebugging - "false"
                    EnableUnmanagedDebugging * "false"
                    EnableSQLServerDebugging = "false"
                    RemoteDebugEnabled = "false"
                    RemoteDebugMachine = ""
                    StartAction = "Project"
                    StartArguments = ""
                    StartPage = ""
                    StartProgram = ""
                    StartURL = ""
                    StartWorkingDirectory = ""
                    StartWithIE = "true"
                />
                <Config
                    Name - "Release"
                    EnableASPDebugging = "false"
                    EnableASPXDebugging = "false"
                    EnableUnmanagedDebugging * "false"
                    EnableSQLServerDebugging = "false"
                    RemoteBebugEnabled = "false"
                    RemoteDebugMachine = ""
                    StartAction = "Project"
                    StartArguments = ""
                    StartPage = ""
                    StartProgram = ""
                    StartURL = ""
                    StartWorkingDirectory = ""
                    StartWithIE - "false"
                />
                <Config
                    Name = "Debug1"
                    EnableASPDebugging = "false"
                    EnableASPXDebugging = "false"
                    EnableUnmanagedDebugging * "false"
                    EnableSQLServerDebugging = "false"
                    RemoteDebugEnabled - "false"
                    RemoteDebugMachine = ""
                    StartAction = "Project"
                    StartArguments = ""
```

```
                StartPage * *"
                StartProgram * ""
                StartURL = -
                StartWorkingDirectory = ""
                StartWithIE * "true"
            />
        </Settings>
    </Build>
    <OtherProjectSettings
        CopyProjectDestinationFolder = ""
        CopyProjectUncPath = ""
        CopyProjectOption = "0"
        ProjectView = "ProjectFiles"
        ProjectTrust = "0"
    />
  </CSHARP>
</VisualStudioProject>
```

## Зависимости проекта

Solution Explorer поможет управлять зависимостями между проектами в сложных решениях, содержащих несколько сборок. Solution Explorer позволяет без "руда добавлять к проектам файлы, проекты и Web-ссылки. Для решений с зависимостями между проектами вы, возможно, захотите использовать ссылки на проекты.

Для добавления ссылки на проект откройте диалоговое окно Add Reference, доступ к которому можно получить, выделив имя проекта в Solution Explorer и едав команду **Project.AddReference** в окне Command Window, На вкладке Projects можно будет видеть список проектов (рис. 2-14).

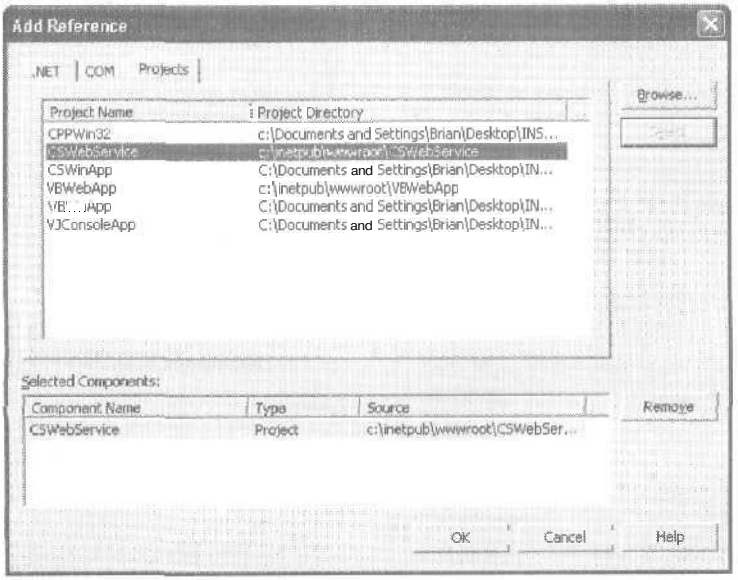



**Рис.** 2-14. *Добавление к проекту ссылки на другой проект*

После добавления ссылки на проект функциональность добавленного проекта становится доступной и в проекте, к которому добавлена ссылка. В этот момент становится важным порядок компоновки, так как сборка, на которую идет ссылка, должна быть построена до того, как на нее сошлется какой-либо проект. Для управления подобными зависимостями в Visual Studio .NET имеется диалоговое окно Project Dependencies (*Projects.ProjectDependencies*)(рис, 2-15), где можно указать зависимость, что изменит порядок сборки затронутых проектов в решении. Диалоговое окно автоматически обновляется с учетом полученной информации при добавлении зависимости между проектами. (Если попытаться добавить в качестве зависимости ссылку на сам построенный проект, Visual Studio .NET ничего не изменит.) Как только зависимость будет указана, Web-сервис XML, на который идет ссылка, добавляется в вершину списка компоновки на странице Build Order.

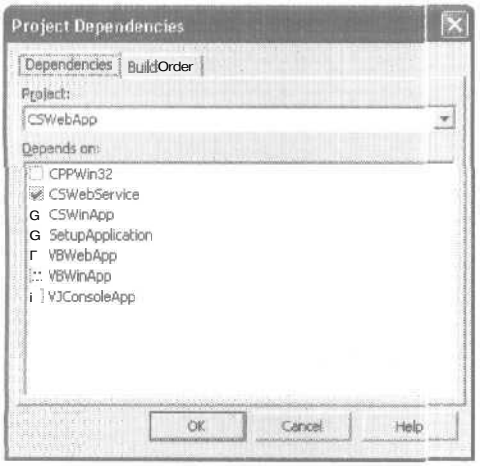



**Рис 2-15.** *Настройка зависимостей компоновки проекта*

## Компоновка проектов и решений

Как только в решении заданы проекты, конфигурации пользовательских компоновок и ссылки, можно приступить к работе над сценариями компоновки для разных конфигураций. Указать в решении компонуемые проекты можно в диалоговом окне Configuration Manager (рис. 2-16). Можно исключить проекты, доставляющие проблемы, или оставить только те компоновки, на которых в данный момент нужно сосредоточиться.

Есть и более мощное средство пакетной компоновки сложных конфигураций в один проход. Диалоговое окно Batch Build (*Project.BatchBuild*)доступно не для всех типов решений — его можно использовать, если проект не содержит приложений Web (рис. 2-17).

Диалоговое окно Batch Build позволяет выполнять несколько важных действий, включая компоновку, перекомпоновку и чистку проектов. Щелчок кнопки Build инициализирует инкрементную компоновку проектов, если они настроены для такой компоновки. Кнопка Rebuild инициирует Rebuild All для всех выбранных проектов. По нажатию кнопки Clean удаляются файлы, представляющие резуль-

тат компоновки, так что можно запустить чистку или разделение проектов без ненужного сбора всего в одну кучу. Заметьте: в Visual Studio .NET команда Clean фактически ничего не делает с проектами Visual Basic .NET и Visual C#. Команды разработчиков Visual Basic .NET и Visual C# решили, что этот вид функциональности не является необходимым для управляемых проектов и поэтому решили обойтись без чистки.

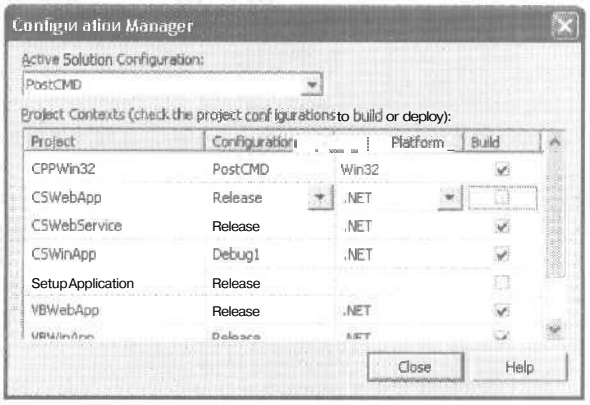



**Рис. 2-16.** *Определение проектов, которые нужно построить в данной конфигурации компоновки*

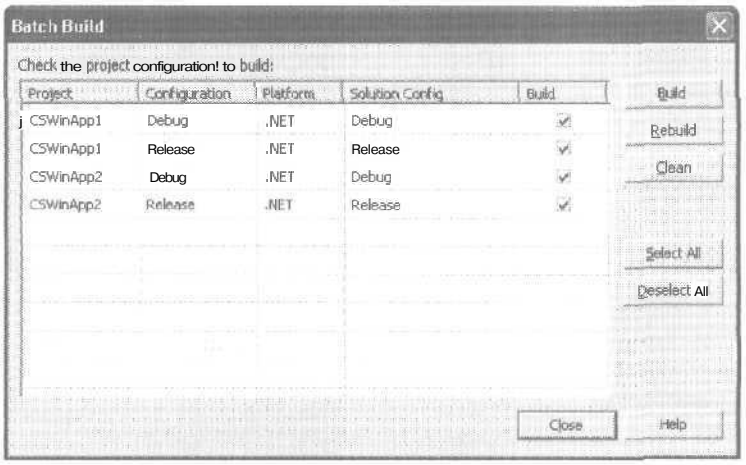
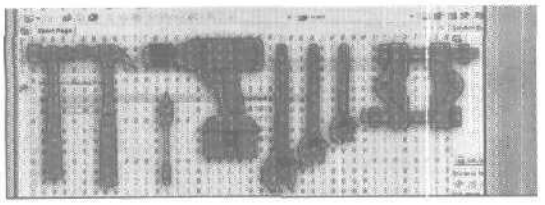



Рис. 2-17. *Диалоговое окно Batch Build*

# Что дальше

Возможности управления проектами в Visual Studio .NET впечатляют, Но полезность IDE для разработчиков зиждется и на возможностях редактора в IDE. В главе 3 мы обсудим редактор исходных текстов и более эффективные методы создания кода.



ГЛАВА

# 3

# Редактор Visual Studio .NET

Редактор — сердце любой среды разработки. Программисты «живут» в своих редакторах, и редактор в Microsoft Visual Studio .NET спроектирован именно как редактор для программистов. В этой главе мы познакомимся поближе с инструментами редактирования, встроенными в Visual Studio .NET. Вы узнаете, как с помощью редактора облегчить работу, а также о некоторых особенностях интегрированной среды разработки (IDE), делающих работу в Visual Studio .NET действительно приятной.

## Документы в IDE

Вся работа в Visual Studio .NET вращается вокруг решений и проектов, включенных в решения. В этом смысле Visual Studio .NET — это инструмент управления проектами. Обычно вам приходится управлять исходными документами, которые содержатся в проектах, и окнами инструментов, реализующими некоторые функции в рамках IDE. Для создания и редактирования документов служат редактор кода (Code Editor) и дизайнеры. Редактируемые файлы отображаются в окнах, открывающихся в центре IDE, и становятся частью панели с вкладками. Окна, содержащие эти файлы, известны как окна документов; это могут быть окна дизайнеров, редакторов, Web-браузера или справки.

В редакторе кода Visual Studio .NET версий 7.0 и 7.1 появилась масса возможностей, в том числе структуризация (outlining), форматирование кода и моя любимая — нумерация строк.

Доступ к ним осуществляется при помощи именованных команд, задаваемых в Command Window (Ctrl+Alt+A) или выбираемых из меню, или нажатием комбинаций клавиш. Освойте эти команды, и вы освоите Visual Studio .NET.

### Стыкуемые окна инструментов

Не все окна-вкладки в IDE являются окнами документов. К окнам-вкладкам в центре IDE можно добавить окно инструментов, для чего нужно выделить это окно и сбросить значение Dockable в меню Window. Окно браузера объектов (Object Browser) (Ctrl+Alt+J) по умолчанию не является стыкуемым, что делает его окном-вкладкой в IDE. Преимущество добавления окна инструментов к набору окон-вкладок в центре IDE в том, что одновременно можно отобразить большое количество информации. Можно отстыковать окно инструментов, отбуксировав его от края, к которому он пристыкован, и оставить установленным флажок Dockable, что делает окно плавающим. Эта техника особенно удобна при работе с несколькими мониторами.

## Все о тексте

Место, где вы редактируете код, в Visual Studio .NET имеет одно из двух имен в зависимости от типа редактируемого файла. При работе с файлом, сохраняемым как текст на известном Visual Studio .NET языке программирования, редактор, с которым вы работаете, называется редактором кода (Code Editor). Доступные при этом возможности зависят от того, насколько данный язык интегрирован в IDE. Если вы работаете с обычным текстовым файлом или файлом, тип которого не опознается IDE, вы работаете в редакторе текста (Text Editor), Последний обладает меньшей функциональностью, чем редактор кода, но достаточно мощен, Вы можете запускать макросы в любом редакторе, хотя, конечно, редактор кода гораздо удобнее. По большей части мы будем называть оба эти редактора редакторами кода, делая различие между ними лишь в необходимых случаях.

На рис. 3-1 показаны части редактора кода в IDE: в зависимости от используемого вами языка в названиях частей редактора кода могут быть отличия.

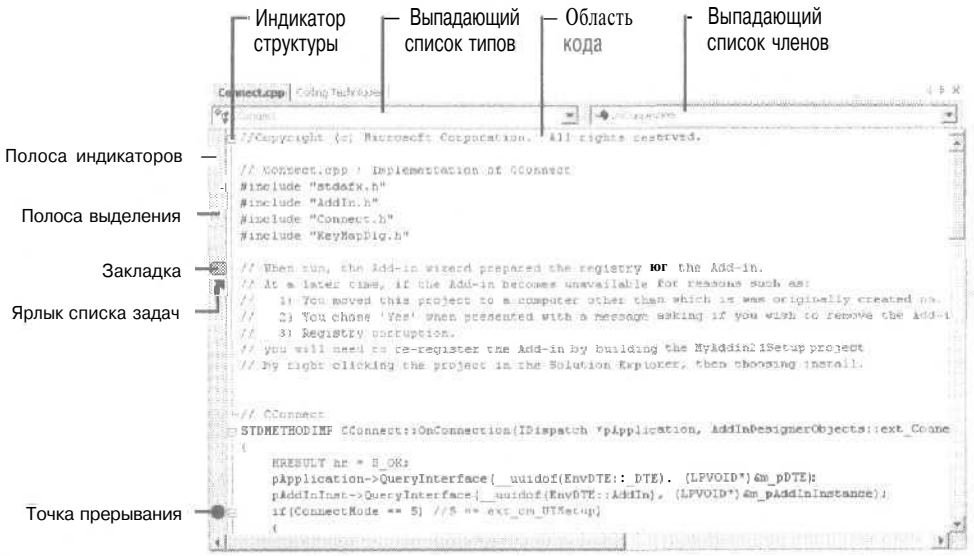



**Рис. 3-1.** *Части окна редактора кода*

В редакторе кода вы набираете код и текст. Большинство средств, показанных на рисунке, можно выключить, чтобы расширить поле обзора редактора кода. Вы можете, щелкнув над правой полосой прокрутки, перетащить границу, чтобы разделить область на две области кода, Используя несколько областей при работе с одним файлом, можно видеть разные части своего кода одновременно.

Полоса навигации (Navigation Bar) содержит два поля в верхней части редактора кода. В Visual C# это выпадающие списки типов (Types) и членов (Members). В Visual Basic .NET это списки имен классов (Class Name) и и имен методов (Method Name). Полоса навигации предназначена для быстрого перемещения к разным частям кода. В Visual Basic .NET упомянутые поля позволяют также добавлять методы в текущий исходный файл.

Тонкая вертикальная линия слева от кода — индикатор структуры. Щелкая прямоугольники с + или -, расположенные вдоль этой полосы, можно скрывать/показывать блоки кода исходного файла (см. подробнее ниже).

Между крайней правой частью индикатора структуры и полосой индикаторов (Margin Indicator Bar) находится полоса выделения. Щелчок в ней приводит к выделению смежной строки кода. Вид указателя мыши, помещенного в область выделения, меняется со стрелки, указывающей на северо-запад на стрелку, указывающую на северо-восток. Щелкая и этой области и буксируя вверх или вниз указатель, можно выделять блоки кода (рис. 3-2). Преимущество такого способа выделения в том, что по окончании выделения к каждой из выделенных строк добавляется одинаковое количество пробелов, формируя красивый и наглядный блок. (Для редакторов и писателей эта возможность неоценима.) Выделение одним щелчком целой строки способствует уменьшению количества ошибок выделения и способствует сохранению форматирования кода при копировании текста между файлами.

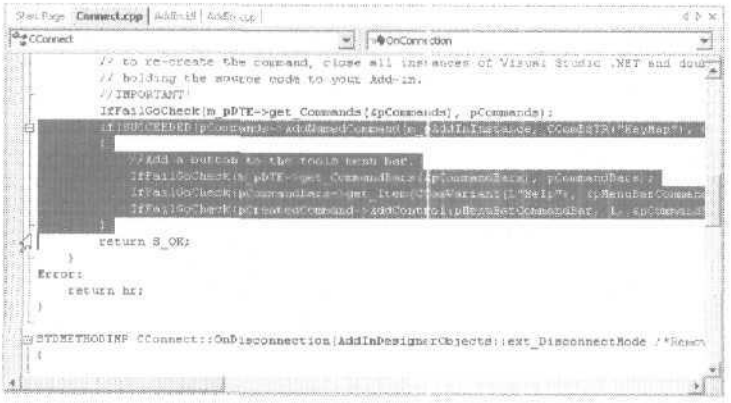



**Рис. 3-2.** *Выделение текста в полосе выделения*

Полоса индикаторов — инструмент многофункциональный, используемый для установки и удаления точек прерывания в коде, для индикации закладок и для хранения ярлыков списка задач. При отладке в этой области виден индикатор. При достижении точки прерывания индикатор точки прерывания будет содержать желтую стрелку, указывающую на текущую строку кода. По умолчанию сама стро-

ка подсвечивается желтым. При пошаговой отладке желтый индикатор показывает, в каком вы месте кода — соответствующая строка кода выделяется в редакторе.

Когда вы находитесь в режиме просмотра вкладок, в верхней части окна редактора кода видна двкладка ля всех открытых окон документов и нестыкуемых окон инструментов. Она позволяет перемещаться между несколькими исходными файлами и формами проекта. Если же вы предпочитаете окна с многооконным интерфейсом (MDI), как это было в Visual Studio 6, вы можете отключить вкладки в диалоговом окне Options.

Теперь посмотрим, что можно сделать в редакторе кода, чтобы упростить редактирование и программирование.

### Практикум: перемещение между окнами и представлениями

Давайте разомнем пальцы и посмотрим, как перемещаться между окном редактора кода (Code Editor), дизайнера (Designer), обозревателя решений (Solution Explorer) и обозревателя классов (Class View).

Допустим, мы разрабатываем код и пользовательский интерфейс. Нажмите на клавиатуре комбинацию Ctrl+Alt+L, чтобы войти в Solution Explorer. Переместитесь к файлу с кодом C# и нажмите Enter. По умолчанию откроется любая Windows-форма или Web-форма, связанная е этим файлом, в противном случае откроется исходный файл. Чтобы увидеть исходный код данного C#-файла, нажмите F7. Чтобы вернуться обратно кформе, нажмите Shift+F7.

Теперь снова щелкните Ctrl+Alt+L, чтобы вернуться в Solution Explorer, Переместитесь к другому файлу и нажмите Enter. У вас должно быть по крайней мере три вкладки, отображаемые в IDE. Для перехода между открытыми файлами нажимайте Ctrl+Tab или Ctrl+F6.

И, наконец, для перехода в Class View, нажмите Ctrl+Shift+C.

Эти комбинации клавиш позволяют перемещаться по IDE, не отрывая рук от клавиатуры.

## Набор текста и быстрые клавиши

Хороший редактор обеспечивает эффективный ввод и манипуляции с текстом. Новичку неведомы жестокие баталии, разыгрывающиеся в чатах, новостных группах и на Web-сайтах между программистами, предпочитающими один редактор другому и доказывающими преимущества любимого редактора. (Но я-то знаю, что Visual Studio .NET — Единственный Настоящий Редактор!)

Причина этих яростных споров в том, что для овладения редактором требуется время и, изучив свой редактор в совершенстве, программист не спешит переходить на новый, если только последний не окажется заметно лучше.

### Наиболее распространенные быстрые клавиши

Приложения, разработанные для Microsoft Windows, используют ряд стандартных сочетаний клавиш быстрого доступа, которые вам, вероятно известны. Эти быстрые клавиши известны как CUA-клавиши (Common User Accessibility), они были определены в работе, выполненной IBM и стандартизирующей быстрые клавиши

для ряда платформ. Преимущество стандартного набора очевидны: раз запомнив их, вы можете их использовать почти во всех Windows-приложениях, включая Microsoft Office. Эти быстрые клавиши помечены в ряде популярных клавиатур, в том числе в большинстве клавиатур фирмы Microsoft.

**Примечание** Подробнее о быстрых клавишах Windows см. «Microsoft Windows User Experience» (Microsoft Press, 1999).

Вот быстрые клавиши для работы с файлами в Visual Studio .NET (табл. 3-1):

**Табл. 3-1. Быстрые клавиши для работы с файлами**

| Команда | Комбинация | Именованная команда |
|---|---|---|
| New | Ctrl+N | *File.NewFile* |
| Open | Ctrl+O | *File.OpenFile* |
| Save | Ctrl+S | *File.SaveSelectedItems* |
| Save All | Ctrl+Shift+S | *File.SaveAll* |
| Print | Ctrl+P | *File.Print* |

При создании нового файла открывается диалоговое окно New File. Если вы привыкли при создании нового документа сразу видеть его на экране, потребуется небольшая адаптация. Выбирая при создании определенный тип файла, вы активизируете большинство возможностей, связанных с данным языком до момента закрытия файла. При создании файла можно сэкономить время, указывая в командном окне (Command Window) имя и расширение файла, который хотите создать. Так, чтобы создать файл UserMotion.cpp, нажмите Ctrl+Alt+A, чтобы открыть командное окно, и введите команду **File.NewFile UserMotion.cpp**. В разделе «Командное окно» этой главы показано, как, применяя для подобных команд псевдонимы, быстро создавать часто используемые файлы, В главе 4 объясняется, как создать макрос, создающий файлы наиболее интересующего вас типа при нажатии Ctrl+N.

Перемещение по документу при помощи последовательностей клавиш — одна из наук, которой можно научиться без книг и статей. Здесь приведен обзор клавиш для перемещения и выделения, а также быстрых клавиш (табл. 3-2). Процесс выделения включает удерживание нажатой клавиши Shift, а распространение выделения с помощью некоторой клавиши обычно требует удерживать нажатой клавишу Ctrl.

**Табл. 3-2. Комбинации клавиш для перемещения и выделения**

| Перемещение | Клавиша(и) для перемещения | Клавиши для выделения |
|---|---|---|
| Символ | стрелка-вправо | Shift+стрелка-вправо |
| | стрелка-влево | Shift+стрелка-влево |
| Слово | Ctrl+стрелка-вправо | Ctrl+Shift+стрелка-вправо |
| | Ctrl+стрелка-влево | Ctrl+Shift+стрелка-влево |
| Строка | End | Shift+End |
| | Home | Shift+Home |
| | стрелка-вниз | Shift+стрелка-вниз |
| | стрелка-вверх | Shift+стрелка-вверх |

**Табл. 3-2.** *(окончание)*

| Перемещение | Клавиша(и) для перемещения | Клавиши для выделения |
|---|---|---|
| Видимая область кода | Page Down<br>Page Up | Shift+Page Down<br>Shift+Page Up |
| Документ | Ctrl+End<br>Ctrl+Home | Ctrl+Shift+End<br>Ctrl+Shift+Home |

Выделенный фрагмент текста можно скопировать иди вырезать в буфер обмена, а затем вставить обратно в редакторе кода. Вот комбинации клавиш для редактирования (табл. 3-3):

**Табл. 3-3. Комбинации клавиш для редактирования**

| Команда | Комбинация | Именованная команда |
|---|---|---|
| Вырезать | Ctrl+X | *Edit.Cut* |
| Копировать | Ctrl+C | *Edit.Copy* |
| Вставить | Ctrl+V | *Edit.Paste* |
| Отменить | Ctrl+Z | *Edit.Undo* |
| Повторить | Ctrl+Y | *Edit.Redo* |
| Выделить текущее слово | Ctrl+W | *Edit.SelectCurrentWord* |
| Выделить все | Ctrl+A | *Edit.SelectAll* |

### Практикум: работа с кольцевым буфером обмена

- Кольцевой буфера обмена позволяет записывать и использовать в ГОЕ несколько скопированных фрагментов. Откройте панель инструментов и щелкните вкладку Clipboard Ring. Откройте какой-нибудь код и выделите любой фрагмент. Скопируйте его в буфер обмена и посмотрите на кольцевой буфер обмена. Скопируйте еще несколько фрагментов и обратите внимание, что фрагмент, выделенный последним, всегда находится наверху. Двойным щелчком элемента буфера обмена можно вставить его в файл, но гораздо интереснее использовать этот инструмент по-другому. Нажмите Ctrl+Shift+Insert и удержите нажатыми Ctrl+Shift. Это приведет к вставке текста в вершину стека. Заметьте: вставленный текст все еще выделен, Снова нажмите Insert и следующий элемент стека заменит собой выделенный текст. Нажмите Insert s третий раз, и будет вставлен фрагмент, скопированный при первой операции копирования. (Вы поняли идею.) Нажмите Insert в последний раз, и вы вернетесь к тексту, который вы выделяли последним. Так вы можете перебирать кольцевой буфера обмена в поиске нужного скопированного ранее фрагмента.

Следующие комбинации можно использовать для обмена позициями двух элементов в редакторе кода (табл. 3-4). Например, если установить курсор перед буквами *AB*, нажатие Ctrl+T приведет к изменению порядка их следования на *BA* Если курсор находится рядом или на слове «идти» в строке «идти напролом», нажатие Ctrl+Shift+T приведет к перестановке «напролом идти». Самое полезное сочетание клавиш из этой группы запускает команду *Edit.LineTranspose*. Нажатие Alt+Shift+T

меняет местами строку, на которой расположен курсор, со следующей строкой, позволяя легко сместить вниз строку кода.

**Табл. 3-4. Комбинации клавиш для перестановки**

| Команда | Комбинация | Именованная команда |
|---|---|---|
| Переставить символы | Ctrl+T | *Edit.CharTranspose* |
| Переставить слова | Ctrl+Shift+T | *Edit.WordTranspose* |
| Переставить строки | Alt+Shift+T | *Edit.LineTranspose* |

### Пользовательские комбинации клавиш

Ранее мы говорили о переключении состояния Dockable для окна, чтобы добавить окно в центр IDE, По умолчанию команде *Window.Dockable* не сопоставлено комбинации клавиш, но вы можете назначить такую комбинацию сами, чтобы облегчить чтение перегруженного данными окна.

Чтобы создать в IDE новое сочетание быстрых клавиш, откройте командное окно (Ctrl+Alt+A) и введите команду **Tools.Options**. Появится диалоговое окно Options (рис. 3-3). Щелкните в папке Environment элемент Keyboard, чтобы перейти на страницу Keyboard. Эта страница позволяет произвести ряд операций с комбинациями быстрых клавиш IDE: создать и отредактировать комбинацию, изменить схему назначения клавиш и сохранить нестандартную схему назначения. При первом добавлении в IDE пользовательского сочетания быстрых клавиш предлагается сохранить вашу схему назначения под каким-либо именем.

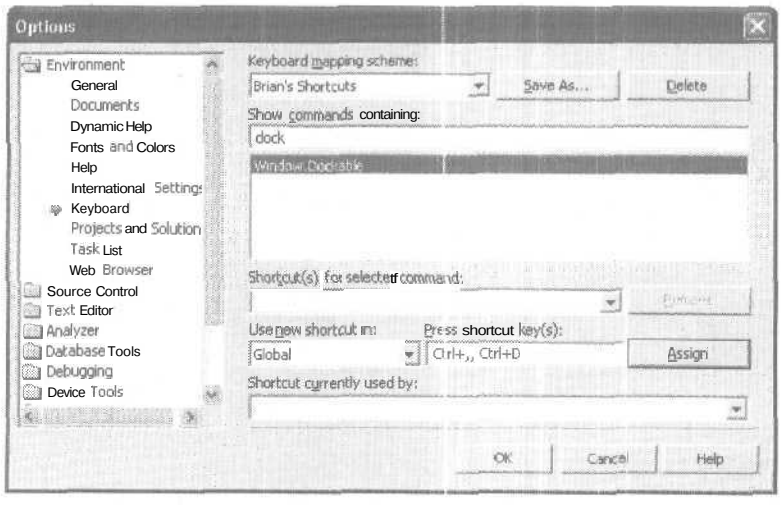



**Рис. 3-3.** *Страница Keyboard диалогового окна Options*

Чтобы отыскать команду, которой надо назначить новую комбинацию быстрых клавиш, напечатайте часть ее имени в поле Show Commands Containing. В нашем случае напечатайте **dock,** и в списке команд появится Window.Dockable.

А теперь о сложностях. В Visual Studio .NET задействованы почти все возможные сочетания клавиш. Вы можете переопределить работающее сочетание, если

уверены, что в своем изначальном значении оно вам не понадобится. Очевидно, это не самое лучшее решение. Поэтому, если вы собираетесь работать на отдельном компьютере и никогда не меняли сочетаний клавиш, можете прекратить использование сочетания клавиш для ненужной вам команды и назначить его для другой.

Visual Studio .NET позволяет задействовать связку комбинаций клавиш (chorded shortcut). Чтобы начать связку, надо нажать клавишу Ctrl вместе с еще одной клавишей. После этого IDE ожидает следующей комбинации клавиш, чтобы определить, какую команду выполнить. Я нашел, что сочетание Ctrl+, (Ctrl+запятая) в Visual Studio .NET нигде не применяется. Поэтому я могу связать все свои личные команды с этим сочетанием, используя дополнительное сочетание клавиш для уточнения того, какую именно из моих команд выполнить. Для команды *WindowDockable* я назначил последовательность Ctrl+, Ctrl+D (Ctrl+запятая, Ctrl+D), набрав эту комбинацию в поле Press Shortcut Key(s). Обязательно щелкните кнопку Assign, чтобы сохранить новое сочетание клавиш.

Пока открыто диалоговое окно Options, можно добавить сочетание быстрых клавиш для самого диалогового окна Options, напечатав в поле Show Commands Containing название **Tools.Options** и назначив ему сочетание Ctrl+, Ctrl+O Теперь для настройки своей IDE можно быстро открыть это окно, нажав нужное сочетание.

Сочетания быстрых клавиш можно присвоить именованным командам IDE, надстройкам и макрокомандам, которые вы создаете и сохраняете.

### Использование надстройки Keybindings Table

Единственный способ определить, какие быстрые клавиши задействованы в Visual Studio .NET, не прибегая к коду, — просмотреть в диалоговом окне Options назначения для каждой из команд. Это способ не из легких, и Крэйг написал замечательную небольшую надстройку, которая отображает все именованные команды и назначенные им в IDE текущие сочетания быстрых клавиш.

Эту надстройку можно найти на Web-сайте Visual Studio .NET по адресу *http://msdn.microsoft.com/vstudio/downloads/automation.asp*Поскольку она написана на C++, ее надо откомпилировать, используя Visual Studio .NET или Visual C++ .NET. Откомпилированная надстройка регистрируется в Visual Studio .NET при компоновке. Если после этого перезапустить Visual Studio .NET, IDE загрузит эту надстройку.

В Visual Studio .NET в меню Help появится новая команда KeyMap. Ее выполнение откроет диалоговое окно Keyboard Help (рис. 3-4).

Надстройка Keybindings Table позволяет просматривать перечень быстрых клавиш, назначенных именованным командам в IDE, а также список доступных команд. Элементы полученной таблицы можно копировать в буфер обмена, Достаточно выделить требуемый элемент и щелкнуть кнопку Copy. Чтобы выделить несколько элементов, нужно нажать клавишу Ctrl и щелкнуть еще один элемент списка. Элементы из буфера обмена затем можно вставить в Microsoft Excel для удобства их просмотра и сортировки.

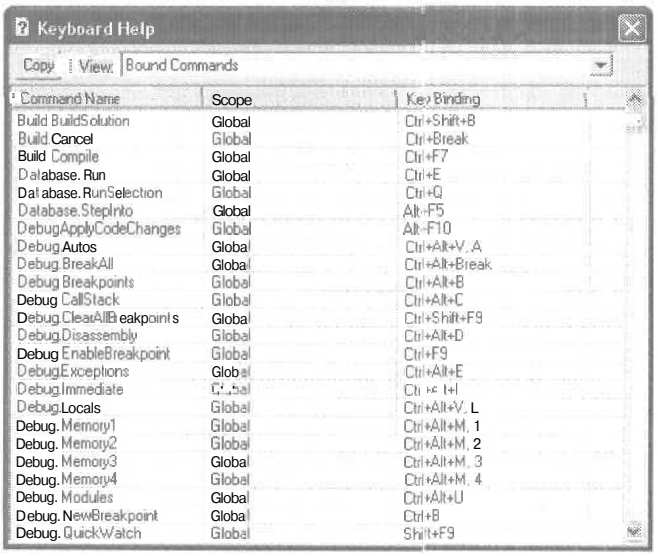



**Рис. 3-4.** *Диалоговое окно Keyboard Help*

## Философия отступов и форматирования кода

В каждой организации свои стандарты оформления кода. Параметры форматирования для каждого из языков, поддерживаемых в IDE Visual Studio, устанавливаются в диалоговом окне Options в папке Text Editor. Когда задаются параметры для всех языков (All Languages), перезаписываются параметры для каждого отдельного языка из перечисленных в папке Text Editor (рис. 3-5).

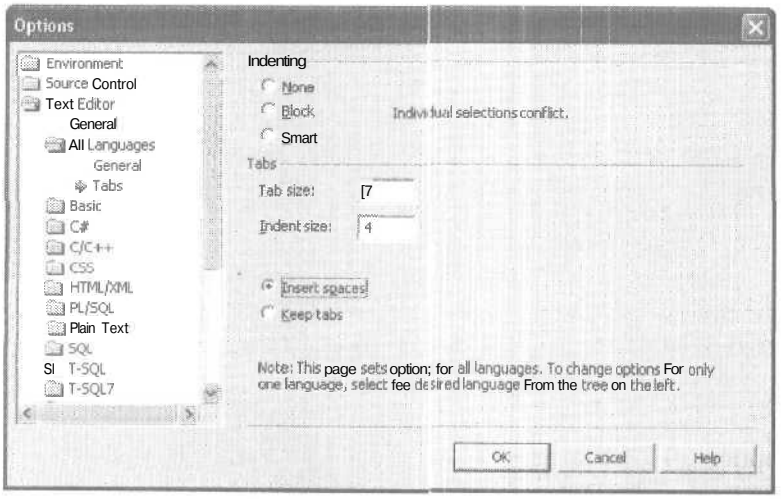



**Рис 3-5.** *Глобальная установка параметров отступов*

Как видно из рисунка, можно установить параметр Indenting (отступы) в None, Block или Smart. Конкретная реализация этих значений определяется языком и

следующим ниже параметром Tabs. Если тип отступов установлен в None, нажатие в конце строки клавиши Enter приведет к тому, что следующая строка начнется в крайней левой позиции редактора кода. Если выбрано Block, новая строка начнется с тем же отступом, что и первый символ предыдущей строки. Это самое распространенное значение, позволяющее делать отступ вручную, не вводя при этом лишних символов табуляции. Значение Smart приводит к определению величины отступа на основании контекста. Так, нажатие в C# клавиши Enter после открывающей фигурной скобки ({) автоматически сформирует нужный отступ следующей строки.

Выбор между отступом с помощью пробелов или символов табуляции — дело вкуса или внутрифирменных стандартов. Если вы предпочитаете, чтобы всюду в коде находились пробелы, а не символы табуляции, можно глобально задать соответствующую опцию при настройке IDE. Если же вы предпочитаете пробелы, помните, что по умолчанию Visual C# для задания отступов использует символы табуляции. Чтобы сделать видимыми в документе символы пустых промежутков (white space), к которым относятся пробелы и символы табуляции, выполните команду *Edit.ViewWhitespace*(Ctrl+R, Ctrl+W). На рис. 3-6 показан документ, в котором визуализированы символы пустых промежутков. Символы табуляции изображаются стрелками-вправо, а каждый пробел — точкой.

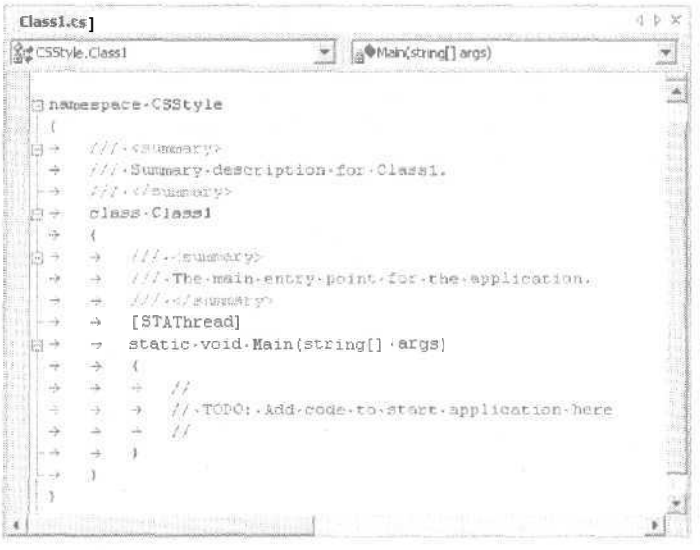



**Рис. 3-6.** *Отображение в редакторе кода символов пустых промежутков*

Чтобы преобразовать файл, использующий для отступов символы табуляции, в файл, использующий пробелы, или наоборот, выберите на странице Tab;s диалогового окна Options нужное значение параметра и, щелкнув OK, закройте диалоговое окно. Затем выделите весь код файла, нажав Ctrl+A. Нажмите Ctrl+K, Ctrl+F (*Edit.FormatSelection*),чтобы применить к выделенному новое форматирование.

Страница Formatting диалогового окна Options управляет рядом характеристик форматирования кода на панели кода (рис. 3-7). Эта страница присутствует в большинстве языков, поддерживаемых в IDE. но для языка Basic настройка осу-

ществляется на странице VB Specific. Страница на рисунке показывает параметры форматирования для C#.

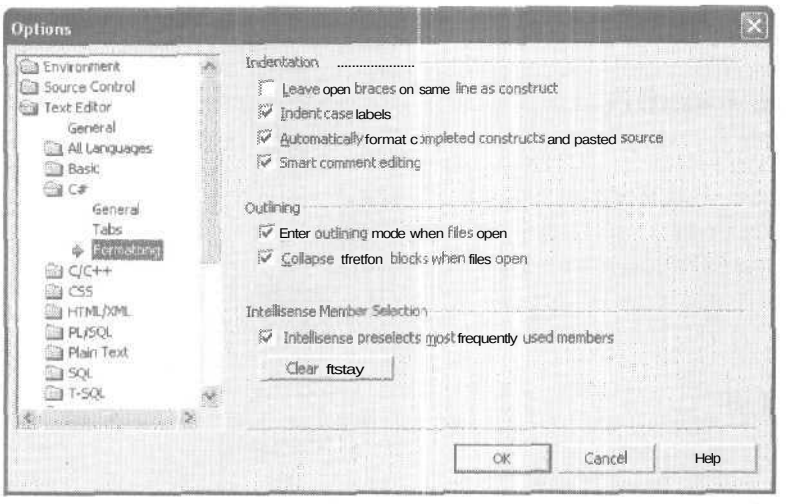



**Рис. 3-7.** *Страница Formatting для C#*

Для разных языков доступны разные параметры. Рассмотрим флажок Leave Open Braces On Same Line As Construct (рис. 3-7). В C# он по умолчанию не установлен. Это вызывает в редакторе кода перенос открывающей фигурной скобки, введенной в конце строки, на следующую строку и ее выравнивание относительно начала оператора. В итоге следующая строка выглядит так:

```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //
}
```

Установка флага Leave Open Braces On Same Line As Construct приводит к тому, что код форматируется в скошенном стиле, называемом стилем K&R, который был предложен в книге Керниган и Ритчи «Язык программирования C». В этом стиле открывающая фигурная скобка блока размещается в конце оператора:

```
static void Main(string[] args) {
    //
    // TODO: Add code to start application here
    //
I
```

Когда вы выберете стиль форматирования, который хотите применить к своему коду, можете использовать для применения стиля команду *Edit.FormatSelection*.

Вы, возможно, заметили, что спецификация на язык C# применяет смешанный стиль, напоминающий строгий стиль K&R, при котором для методов применяется скошенный стиль, а для классов — прямой. Когда вы форматируете код командой *Edit.FormatSelection*выбранный стиль применяется ко всем выделенным фраг-

ментам. Если используется команда *Edit.FormatDocument*, каждый блок документа видоизменяется согласно заданному стилю. Чтобы задать смешанный стиль, установите флажок Leave Open Braces On Same Line As Construct, но удостоверьтесь, что применяете форматирование не для всего документа.

# Синтаксис в цвете, нумерация строк и структуризация (или о чем говорит компилятор)

В зависимости от используемого языка компилятор при обработке файла удаляет форматирование и символы пустых промежутков. Однако на уровне редактора даже незначительное удобство, предоставляемое со стороны IDE, может повлиять на производительность и комфорт при работе с кодом.

## Синтаксис в цвете

Чтобы изменить цвета, применяемые текстовым редактором, нужно перейти к странице Fonts And Colors, находящейся в папке Environment диалогового окна Options. Эта страница позволяет менять параметры большинства окон, используемых в IDE (рис. 3-8): параметры принтера, окон инструментов, дизайнеров запросов и представлений.

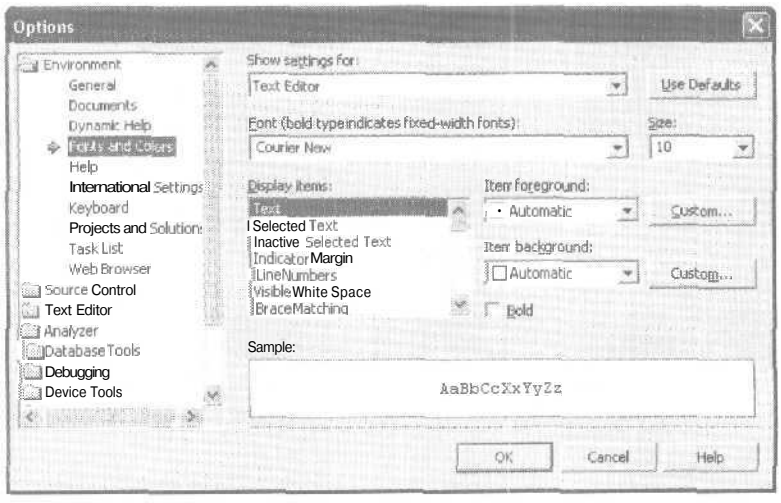



**Рис. 3-8.** *Страница шрифтов и цветов*

Чтобы настроить цвета, которыми элемент будет отображаться в окне текстового редактора, надо выделить этот элемент в списке Display Items и указать цвета переднего плана (Item Foreground) и фона (Item Background). Чтобы элемент имел полужирное начертание, установите флажок Bold. Настройка элемента Text влияет на внешний вид всего текста, отображаемого в текстовом редакторе, так что, если хотите отображать белый текст на черном фоне, настройте сначала эти цвета, а затем уже задавайте цвета остальных элементов. В окне образца вы можете увидеть, как остальные элементы с установленным для них значением Auto-

matic для опций Item Background *и* Item Foreground, смотрятся при выбранных вами цветах элемента Text. Так, на рис. 3-9 для элемента Text установлен черный фон и белый цвет текста. Щелкнув элемент Comment, вы увидите, что для комментариев цвет фона установлен Automatic, а для текста — Dark Green (темно-зеленый). Цвет текста можно изменить на зеленый, чтобы добиться лучшей читаемости этого элемента.

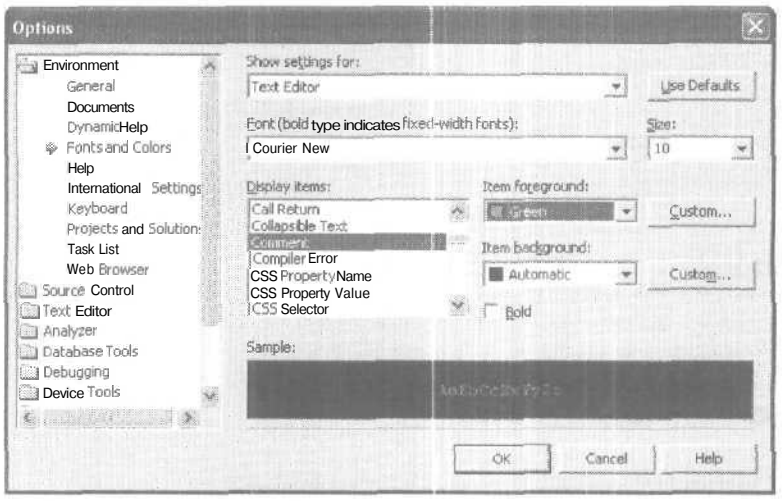



**Рис. 3-9.** *Настройка текстового редактора*

На странице Fonts And Colors можно также изменить шрифт, используемый по умолчанию в диалоговых окнах и на стартовой странице IDE. Рис, 3-Ю показывает окно инструментов после того, как значением параметра Dialogs And Tool Windows был выбран нестандартный шрифт. Чтобы вернуться к значениям, используемым в Visual Studio .NET по умолчанию, щелкните кнопку Use Defaults на странице Fonts And Colors.

Один параметр может оказаться чрезвычайно полезным, когда для печати листингов применяется стандартный черно-белый принтер или если вы изменяете большое количество параметров текстового редактора IDE. В списке Show Setting For можно выбрать Printer и задать шрифты и цвета, которые будут задействованы при печати документов. В этом случае кнопка Use заменит собой кнопку Use Defaults, позволяя применять при печати отдельный набор параметров цвета и шрифта. Если для отображения вы выбрали комбинацию «белый текст на черном фоне», вы, возможно, не захотите в таком же формате печатать на принтере. Значения параметров для принтера хранятся отдельно от тех, что использует текстовый редактор. Можно использовать при печати и собственные параметры, выбрав Text Editor Settings из списка (рис. 3-11).

Здесь мы применим значения параметров, отличные от используемых в текстовом редакторе. Настроив вывод на принтер, вы можете повысить читаемость печатаемого кода. На рис. 3-12 показана распечатка, сделанная после того, как на странице Fonts And Colors комментариям было назначено полужирное начертание. Помните, что данные параметры не влияют на отображение кода в текстовом редакторе.




**Рис.** 3-Ю. *Окно инструментов Index после изменения параметра Dialogs And Tool Windows*

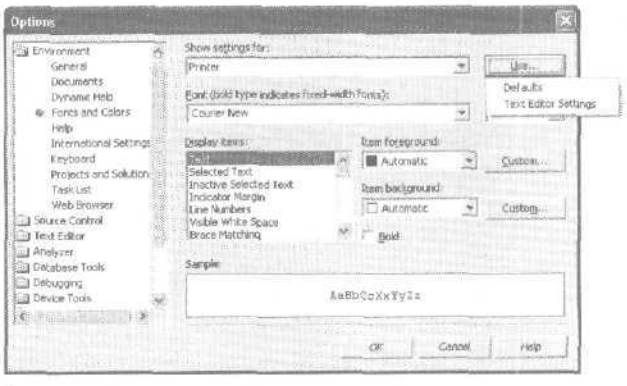



**Рис. 3-11.** *Настройка значений, используемых при печати*

```
c:\Documents and SettingsXbriamjo\Desktop\KeyMap\KeyMap\AddIn.cpp
//Copyright (c) Microsoft Corporation.  All rights reserved.

// AddIn.cpp : Implementation of DLL Exports.

#include "stdafx.h"
#include "resource.h"
#include "AddIn.h"

CAddInModule _AtlModule;
```

**Рис. 3-12.** *Распечатка исходного кода с комментариями полужирного начертания*

Если вы проделали немалую работу по настройке цветов и шрифтов в своей IDE, вам может понадобиться сохранить эти параметры, чтобы применить их на другой машине или после переустановки. Для этого используйте Regedit и экспортируйте следующий раздел реестра своего компьютера в REG-файл.

**Visual Studio .NET 2002**

```
HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\7.0\FontAndColors
```

**Visual Studio .NET 2003**

```
HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\7.1\FontAndColors
```

После сохранения данного раздела можно импортировать раздел из REG-файла через Regedit. Если после этого перезапустить IDE, импортированные параметры вступят в силу.

**Настройка по-Microsoft'овски**

Если вы бывали на конференциях, где представители Microsoft демонстрирует продукты фирмы, вы, наверное, замечали, что они не прибегают к значительным изменениям пользовательского интерфейса или окружения Windows, в котором работают, Тому есть причина. В Microsoft существует негласное правило: если вам нужно настраивать свое окружение (и вы демонстрируете это клиентам), вы исправляете недоработку, и это доказывает необходимость включения сделанных вами изменений в параметры по умолчанию. (Разумеется, временное увеличение размера шрифта в текстовом редакторе для удобства аудитории тут не в счет.) Сказанное **не** означает, что подгонка сама по себе неправильна. Просто мы всегда стараемся обеспечить пользователя наилучшим вариантом прямо «из коробки». Если что-то нужно изменить, чтобы улучшить работу, это изменение следует по возможности сделать значением по умолчанию,

## Нумерация строк

Нумерация строк для всех языков, доступных в папке Text Editor, устанавливается на странице General диалогового окна Options. Можно установить нумерацию для отдельного языка или для всех сразу. С этим параметром не связана ни одна именованная команда — его можно изменить только в диалоговом окне Options.

Чтобы изменить это значение, не открывая диалогового окна, можно запустить макрос или использовать надстройку. Среди образцов макросов, поставляемых е Visual Studio .NET, есть два предназначенных для включения/выключения нумерации строк. Можно самостоятельно настроить способ их запуска: из командного окна, нажатием комбинации быстрых клавиш, с помощью созданной команды меню или кнопкой на панели инструментов.

Макросы нумерации называются *Macros.Samples.Utilities.TurnOnLineNumbers* и *Macros.Samples.Utilities.TurnOffLineNumbers*. Набирать такие имена в командном окне даже с помощью автозавершения (IntelliSense) трудно, поэтому мы создадим для них псевдонимы — короткие имена команд, используемые вместо их настоящих имен в командном окне. Чтобы создать псевдоним для макроса *TurnOnLineNumbers*, откройте командное окно, нажав Ctrl+Alt+A, и введите:

```
>alias  lnon Macros.Samples.Utilities.TurnOnLineNumbers
```

Теперь, если ввести **lnon**, будет включена нумерация страниц (если она была выключена). Чтобы выключить нумерацию, создадим псевдоним для макроса *TurnOffLineNumbers,* напечатав:

```
>alias lnoff Macros.Samples.Utilities.TurnOffLineNumbers
```

Теперь макрос *TurnOffLineNumbers* будет запускаться и под именем *lnoff.*

Допустим, теперь нужно назначить этим макросам последовательности быстрых клавиш. Это несложно: вы просто ищете команды со словом *Numbers* на странице Keyboard папки Environment диалогового окна Options. Выделите макрос, которому назначаете быстрые клавиши, напечатайте выбранную последовательность клавиш и щелкните кнопку Assign. Для наших макросов мы выбрали такие последовательности: для *TurnOnLineNumbers*— Ctrl+„ Ctrl+N (N обозначает нумерацию), а для *TurnOffLineNumbers* — Ctrl+„ Ctrl+Shift+N. На рис. 3-13 показано, как выглядят эти последовательности клавиш, после того как мы присвоили их макросам.

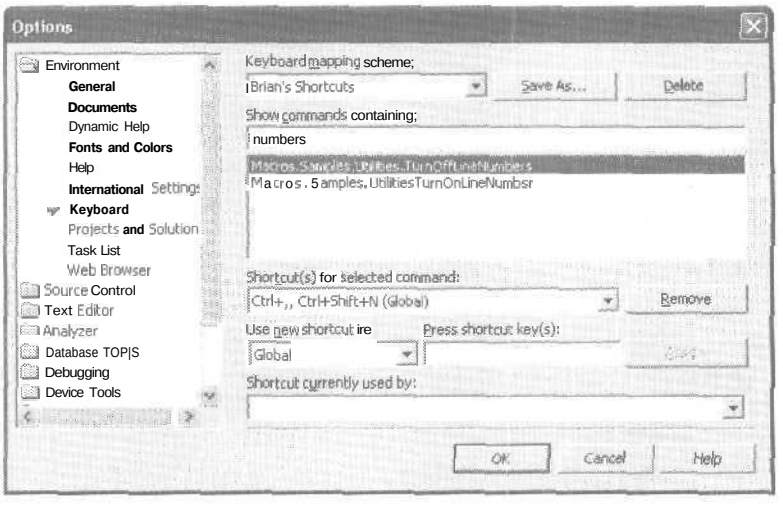



**Рис. 3-13.** *Макрос* TurnOffLineNumbers *с назначенной ему комбинацией клавиш*

Установить нумерацию строк довольно просто, но так ли часто мы будем включать/выключать нумерацию, чтобы выучить псевдонимы или сочетания быстрых клавиш, которые мы для этого создали? Может, и да. Если же нет, стоит поместить эти команды в меню. Чтобы добавить макросы в меню, щелкните правой кнопкой на панели инструментов Visual Studio .NET и выберите Customize. В диалоговом окне Customize на вкладке Commands щелкните Macros, в списке команд выберите команду Samples.Utilities.TurnOnLineNumbers и отбуксируйте ее л меню. Меню раскроется, и вы сможете поместить выбранную команду точно в нужное место.

Чтобы настроить вновь созданную команду меню, щелкните ее правой кнопкой — появится контекстное меню (диалоговое окно Customize должно оставаться открытым). Новую команду можно назвать Turn On Line Numbers. Сделайте то же самое с макросом *TurnOffLineNumbers* (рис. 3-14).

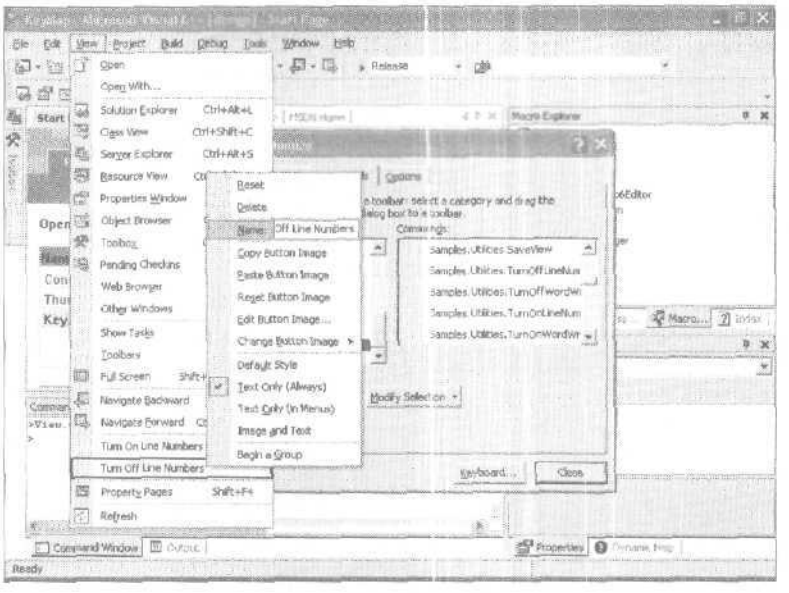



**Рис. 3-14.** *Добавление макроса* TurnOffLineNumbers *в меню*

Макрос, подобный рассмотренным, тоже можно поместить на панель инструментов и даже задать значок для новой кнопки, вызывающей макрос.

## Структуризация

Идея структуризации (outlining) заключается в группировании кода по функциональности для более удобной навигации в редакторе. Рис. 3-15 показывает файл кода, в котором структурно свернуты тела методов. В конце каждого свернутого узла виден прямоугольник с эллипсисом внутри. Если над прямоугольником задержать указатель, всплывет подсказка, показывающая часть содержимого узла.

Чтобы выключить структуризацию, нужно нажать Ctrl+M, Ctrl+P (*Edit.StopOutlining*). Это приведет к исчезновению расположенных вдоль границы редактора кода индикаторов структуры. Включается структуризация комбинацией Ctrl+M, Ctrl+O (*Edit.CollapsetoDefinitions*). Команда *CollapsetoDefinitions* включает в редакторе кода структуризацию и сворачивает каждый блок кода в файле. Нажатие Ctrl+M, Ctrl+L (*Edit.ToggleAllOutlining*) раскрывает все свернутые блоки в редакторе кода.

Помимо скрытия в текстовом редакторе логических блоков кода, можно сворачивать произвольно выделенные фрагменты. Для этого нужно выделить любой фрагмент и нажать Ctrl+M, Ctrl+H (*Edit.HideSelection*). Эта функция может оказаться полезной для сворачивания кода между двумя удаленными точками файла кода и работает для сворачивания строк в обычном текстовом файле. Чтобы развернуть свернутый выделенный фрагмент, нужно нажать Ctrl+M, Ctrl+U (*Edit.StopHidingCurrent*).

Последовательности быстрых клавиш, связанных со структуризацией в текстовом редакторе, представлена ниже (табл. 3-5).

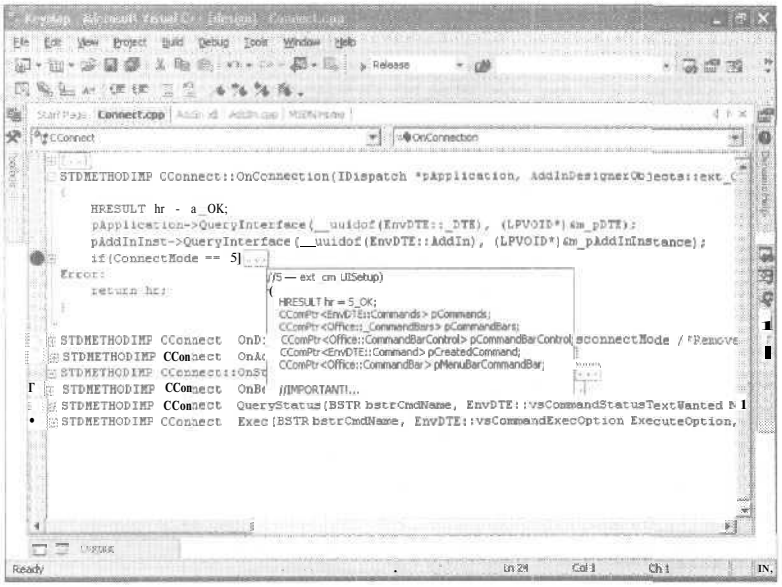


Рис. 3-15. *Когда структуризация включена, всплывающая подсказка отображает содержимое свернутого узла*

**Табл. 3-5. Быстрые клавиши для структуризации**

| Команда | Комбинация | Именованная команда |
|---|---|---|
| Выключить структуризацию | Ctrl+M, Ctrl+P | *Edit.StopOutlining* |
| Переключить структуризацию | Ctrl+M, Ctrl+L | *Edit.ToggleAllOutlining* |
| Переключить развертывание | Ctrl+M, Ctrl+M | *Edit.ToggleOutliningExpansion* |
| Скрыть выделенное | Ctrl+M, Ctrl+H | *Edit.HideSelection* |
| Прекратить скрытие выделенного | Ctrl+M, Ctrl+U | *Edit.StopHidingCurrent* |

# Справка

Справка в Visual Studio .NET предельно проста; выделяете непонятное и нажимаете F1. И почти всегда получаете то, что нужно.

### Сила F1

Несколько лет назад мне звонит приятель и говорит, что не может понять, почему Visual Basic выдает ошибку, когда он вызывает некоторую функцию (по-моему, это было в Visual Basic 4). По электронной почте он переслал мне свой проект, и я его запустил. Та же ошибка! А спустя 10 секунд я объяснил приятелю, что означало это сообщение об ошибке. «Как ты узнал?» — спросил он. Да просто нажал F1, пока окно с сообщением было открыто!

## Автозавершение

Автозавершение (IntelliSense) обеспечивает окончание ввода оператора, предлагая контекстно-чувствительный список членов, который автоматически появляется в процессе печатания кода в редакторе кода. Этот список позволяет сэкономить время при программировании с использованием малознакомого API, а также уменьшить количество ошибок, которые обычно можно отловить только при компоновке.

Автозавершение работает путем разбора кода, вводимого в редакторе, основываясь на типе текущего проекта. А значит, чтобы автозавершение работало, исходный файл должен быть частью проекта или решения. Автозавершение обычно работает автоматически для кода, являющегося частью .NET Framework и для внешних методов, ссылки на которые добавлены к проекту. Оно работает даже для Web-сервисов XML, ссылки на которые также добавлены к проекту.

При работе с поддерживаемым языком автозавершение обеспечивает четыре главных типа функциональности: завершение операторов, информацию о параметрах, завершение слов и комментарии к коду. Большинство программистов будет использовать их в автоматической форме, т. е. будет брать информацию, представляемую IDE, особо не задумываясь над тем, что было показано. И это абсолютно нормально: если это вам помогает, автозавершение делает именно то, для чего его придумали. Но его можно применять и более осмысленно, используя быстрые клавиши, связанные с отображаемой автозавершением информацией.

Подсказки, связанные с завершением операторов и информацией о параметрах, можно отключить. Тогда для их отображения придется использовать быстрые клавиши, именованные команды или кнопки панели инструментов. Чтобы отключить в редакторе кода автозавершение операторов, перейдите в папку Text Editor диалогового окна Options. Выберите язык, для которого хотите внести это изменение (или All Languages, чтобы сделать это для всех языков), откройте страницу General и в разделе Statement Completion снимите флажок Auto List Members. На этой странице можно также переключить параметр автозавершения параметров (Parameter Information).

---

**Примечание** Вы можете увеличить количество параметров, возвращаемых в списке членов, сбросив флажок Hide Advanced Members. По умолчанию в Visual Basic .NET он установлен, и вы можете пропустить ряд завершений, если оставите его установленным.

---

При включенном автозавершении операторов вы получаете информацию при вводе знака операции. Если автозавершение операторов выключено или если нужно немедленно отобразить эту информацию, нажмите Ctrl+J (рис. 3-16).

Для выбора элемента, подходящего для завершения оператора, служат клавиши-стрелки «вверх» и «вниз», а для завершения — клавиша Tab.

---

**Примечание** На странице Formatting папки C# в диалоговом окне Options в Visual Studio .NET 2003 есть параметр, позволяющий выбрать для автозавершения отдельного оператора Visual C# наиболее часто используемый член.

---

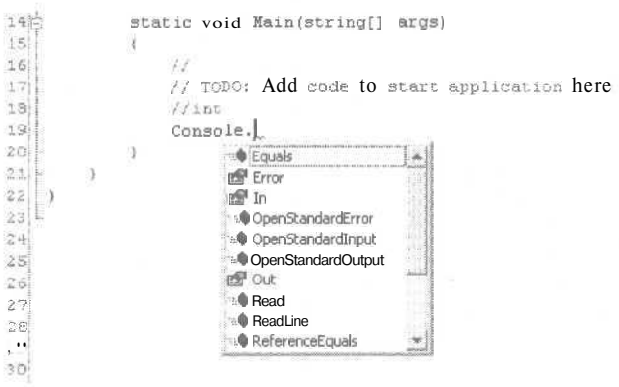



Рис. 3-16. *Принудительное завершение оператора нажатием Ctrl+J*

Чтобы включить информацию о параметрах подобно тому, как показано ни рис. 3-17, нажмите Ctrl+Shift+пробел. Вам будет предоставлен список вариантов перегруженных параметров для определенного метода. Для перемещения между вариантами и просмотра доступных параметров служат клавиши-стрелки «вверх» и «ВНИЗ».

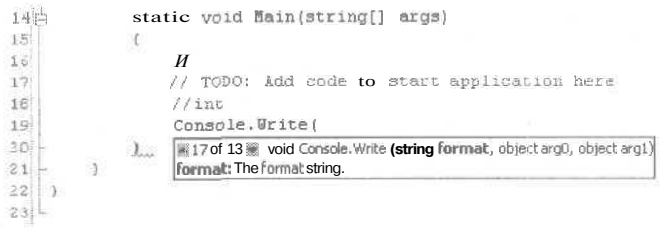



**Рис. 3-17.** *Просмотр информации о параметрах метода нажатием Ctrl+Shift+пробел*

Используя информацию о параметрах, предоставляемую выбранным элементом (наиболее вам подходящим), напечатайте параметры для своего метода. По мере ввода следующий параметр списка выделяется полужирным начертанием. Возможность сверяться с предоставленным образцом помогает вводить каждый параметр из списка без ошибок.

В автоматическом режиме вы не увидите все возможности автозавершения слов, которое позволяет напечатать несколько начальных букв оператора и получить список возможных его завершений. Это отличается от завершения операторов, при котором предлагается список членов в зависимости от контекста. Последовательность быстрых клавиш для завершения слов — Alt+стрелка-вправо, при этом также работает автозавершение операторов по нажатию Ctrl+J.

И, наконец, заметьте, что, когда в редакторе кода вы задерживаете указатель мыши над идентификатором, всплывает подсказка. Эта подсказка — часть Quick Info, еще одной возможности, связанной с автозавершением. Там хранятся объявления идентификаторов и связанные с ними комментарии к коду. Чтобы принудительно включить эту информацию, нажмите Ctrl+K, Ctrl+I. Комментарии в коду методов в Visual C# задаются в строке, непосредственно предшествующей опре-

делению метода, и начинаются символами ///. Даже если вы не собираетесь создавать документацию на свои методы, автозавершение превращает подобные комментарии *в* удобное средство помощи разработчику, которому придется использовать ваш код.

### Соответствие скобок

Еще одна возможность автозавершения — автоматический контроль за соответствием открывающих и закрывающих скобок. Контроль работает в Visual C# и Visual C++ и проявляется, когда вы печатаете в редакторе кода закрывающие скобки. Контролируются круглые (), квадратные [] и фигурные {} скобки. Проверяются также условные макровыражения *#if*, *#else* и *#endif* при печати закрывающих выражений кавычки при печати закрывающих кавычек.

Этот контроль нельзя отключить. да и зачем? Но на время набора кода его можно скрыть. Чтобы настроить способ проверки соответствия скобок, откройте страницу Fonts And Colors папки Environment диалогового окна Options. Выберите в списке Display Items элемент Brace Matching. Если вы сбросите флажок Bold и зададите параметрам Foreground и Background значение Automatic, то больше не увидите предложения подходящих скобок при наборе кода в редакторе. Естественно, вы можете изменить цвета и параметры так, чтобы еще сильнее выделить это предложение.

## Динамическая справка

Эту возможность трудно переоценить, работая с такими сложными инструментами, как Visual Studio .NET. Динамическая справка — одно из окон инструментов IDE, в котором появляется справочная информация, зависящая от того, что мы в данный момент делаем в IDE. Так, перейдя в дизайнер, вы увидите информацию о дизайнере.

Но что важнее, динамическая справка выдает информацию, относящуюся к контексту кода в IDE. Открыв окно редактора кода Visual C# и напечатав *Console,* вы увидите окно динамической справки (рис. 3-18). В этом окне приведены ссылки на разделы справки по классу *System.Console*. Это очень важно: страшно вспомнить, сколько раз я нажимал Ctrl+Alt+F3 (*Help.Search*)и после этого печатал то же самое, что уже напечатал в редакторе кода.

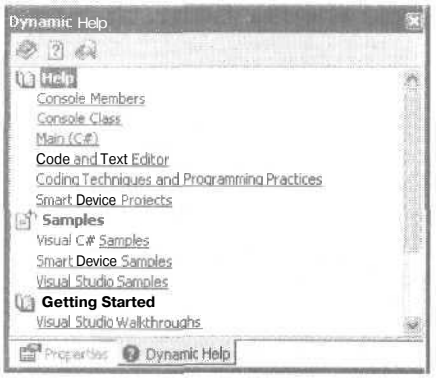



Рис. 3-18. *Окно динамической справки*

Я думаю, эта возможность редко упоминается из-за ограниченности пространства IDE. Разработчики часто открывают окно свойств (Properties), поэтому динамическая помощь редко отображается на экране. Быстрые клавиши помогут вам изменить это поведение и упростить использование данного окна.

Окно динамической помощи открывает комбинация Ctrl+Fl. Чтобы почувствовать, как это выглядит при работе с IDE, откройте проект и убедитесь, что окно свойств загораживает окно динамической помощи. Переместитесь в какое-либо место одного из файлов кода, входящих в проект, и посмотрите, как получить сведения об элементах исходного файла. Поместите курсор на какой-нибудь оператор и нажмите F1, чтобы вызвать связанную с ним справку. Вы также можете вызвать по Ctrl+Alt+F3 окно поиска и напечатать в качестве субъекта поиска элемент, справку о котором вам нужна. И, наконец, вы можете нажать Ctrl+Fl и выбрать один из вариантов, предлагаемых динамической справкой. В 90% случаев динамическая справка выводит именно нужную информацию.

## Командное окно

Если вы уже использовали модальные редакторы типа Vim (Vi improved) и привыкли печатать команды редактора в командной строке, командное окно Visual Studio .NET окажется для вас приятным сюрпризом.

Если у вас установлена надстройка Keybindings Table, вы можете убедиться, что в IDE применяется более 1100 команд. Из них только 400 по умолчанию связаны с последовательностью быстрых клавиш. Остальные доступны из меню, панели инструментов и командного окна.

Командное окно Visual Studio .NET поддерживает два режима работы. В командном режиме (Command Mode) оно выполняет именованные команды. В режиме непосредственного выполнения (Immediate Mode) оно вычисляет выражения, относящиеся к коду, с которым вы работаете. Для переключения между режимами служит пара команд: **immed** переведет окно в режим непосредственного выполнения, **>cmd** позволяет вернуться в командный режим. Некоторые вычисления выражений можно производить и в командном режиме при отладке, предваряя команду знаком вопроса (?):

```
>? i
4
>? i = 7
7
>? i + i
14
```

В режиме же непосредственного выполнения можно вводить команды, предваряя их знаком «больше» (>):

```
i
7
i + 1
14
>Edit.Find i
>immed
```

Стоит вам ввести законченную команду, как командное окно переключится в командный режим. Для возвращения в режим непосредственного выполнения надо ввести **immed.** Некоторые команды, такие как *EditFind,* открывают диалоговое окно и снова возвращают режим непосредственного выполнения.

**Совет** Команда **cls** позволяет очистить командное окно. Это можно делать в любом из двух режимов. После ее выполнения командное окно возвращается в режим, предшествовавший вводу команды.

Именованные команды в Visual Studio .NET обычно отображаются в пункты меню IDE. Поэтому, если вам нужно в командном окне вызвать именованную команду, просто напечатайте имя меню, содержащего эту команду, затем оператор «точка» (.) и ее имя. Скажем, чтобы организовать поиск через командное окно, выведите его на передний план (Ctrl+Alt+A). Чтобы открыть диалоговое окно Find, введите в командном окне **Edit.Find**. Некоторые команды (включая *Edit.Find*) могут иметь аргументы. То есть вы можете начать поиск, не общаясь с диалоговым окном.

А теперь посмотрим, как могут облегчить работу с командным окном псевдонимы. В качестве примера используем команду *EditFind.* Чтобы сделать ее действительно полезной, нужно иметь возможность быстро печатать имя команды и параметры поиска. Ввод **Edit.Find** требует немногим больше усилий, чем ожидается от такой команды. Допустим, вы работали с реактором Vi и привыкли для поиска применять обратный слэш (\). Команде *EditFind* можно сопоставить обратный СЛЭШ:

```
>alias \ Edit.Find
```

Чтобы задействовать новый псевдоним, просто вводите его так, как вводили бы настоящее имя команды. Вот как использовать псевдоним \ для поиска пары элементов в текстовом редакторе.

```
>\ main|args /regex
```

Такая реализация символа \ для поиска не очень удобна: приходится вводить пробел между этим символом и искомым элементом (или выражением). Однако это работает и может облегчить переход к новому редактору.

Очень удобны псевдонимы для разработанных макросов, так как путь макросов обычно довольно длинен (о макросах см. главу 4).

Командное окно в Visual Studio .NET открывается комбинацией Ctrl+Alt+A. Вы также можете ввести команду в поле со списком Find панели инструментов Standard, нажав Ctrl+D (*Edit.GoToFindCombo*). Обычно печать текста в поле со списком Find просто позволяет быстро найти элемент или фразу в текущем документе. Если в этом поле напечатать знак (>), оно сможет принимать команды, и в него можно вводить команды, обычно вводимые в командном окне. Поскольку и командное окно, и поле со списком Find поддерживают автозавершение, можно просто начать вводить именованную команду, чтобы увидеть возможные варианты ее завершения (рис. 3-19).

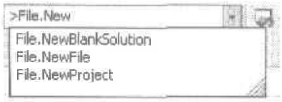



Рис. 3-19- *Автозавершение команды в поле со списком Find*

## Поиск, замена и регулярные выражения

Если вы не можете запросто отыскать или заменить текст, вы скорее всего работаете не с самым лучшим редактором. В текстовом редакторе Visual Studio .NET несколько способов поиска, в том числе поиск с помощью регулярных выражений.

Но сначала рассмотрим именованные команды и быстрые клавиши, связанные с поиском и заменой в Visual Studio .NET. Они соответствуют быстрым клавишам Common User Accessibility (CUA), а также используются в Windows и в Office.

Чтобы открыть диалоговое окно Find, нужно нажать Ctrl+F (рис. 3-20). Параметры диалогового окна Find позволяют задать проверку с учетом регистра (Match Case), поиск полного слова (Match Whole Word) и направление поиска (Search Up), Одна из интересных возможностей — поиск скрытого текста (Search Hidden Text). Если этот флажок сброшен, не осуществляется поиск в фрагментах текста, скрытых с помощью структуризации.

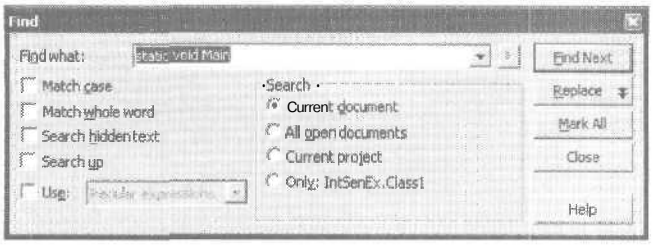



Рис. 3-20. *Диалоговое окно Find*

Диалоговое окно Find является окном инструментов, поэтому вы можете пристыковывать его в IDE и даже, выключив опцию Dockable в контекстном меню, доступном из заголовка окна, сделать его окном-вкладкой в центре IDE.

Чтобы пристыковать окно Find, достаточно отбуксировать его к одной из сторон IDE. Но предпочтительнее сделать его плавающим диалоговым окном, поскольку легко случайно пристыковать окно при попытке убрать его в сторону. Чтобы отключить стыковку, в контекстном меню диалогового окна надо выбрать пункт Floating (щелчок правой кнопкой мыши кнопки Close диалогового окна откроет связанное с ним контекстное меню). Это позволит перемещать диалоговое окно по экрану.

Диалоговое окно Replace (рис. 3-21) открывает комбинация Ctrl+H. Это окно очень похоже на окно Find, но содержит поле со списком для заменяющего текста (Replace With).

Как работать в обоих окнах, очевидно. Вы можете выбрать поиск по текущему документу, по всем открытым документам, по текущему проекту или по выделенным в редакторе фрагментам или блокам.

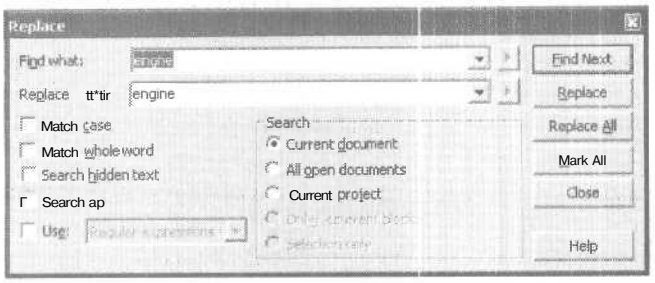



Рис. 3-21. *Диалоговое окно Replace*

Диалоговое окно Find and Replace (Ctrl+Shift+F) (рис. 3-22) и диалоговое окно Replace In Files (Ctrl+Shift+H) служат для поиска текста в пределах проекта или структуры каталогов. Кнопка «...» открывает диалоговое окно Look In, предназначенное для определения границ поиска: это может быть и заданный проект, и каталог локального компьютера. По умолчанию результат поиска Find In Files пересылается в окно Find Results.

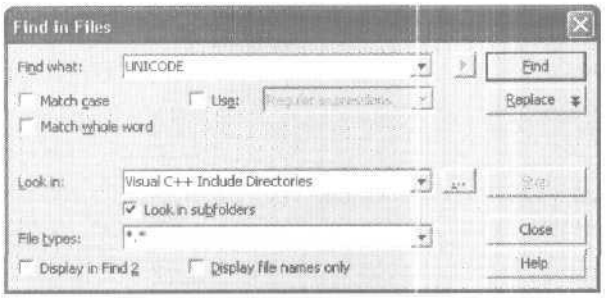



Рис. 3-22. *Диалоговое окно Find in Files*

Вот быстрые клавиши для поиска и замены в Visual Studio .NET (табл. 3-6):

**Табл. 3-6. Распространенные быстрые клавиши для поиска**

| Команда | Комбинация | Именованная команда |
|---|---|---|
| Find | Ctrl+F | *Edit.Find* |
| Replace | Ctrl+H | *Edit.Replace* |
| Find in files | Ctrl+Shift+F | *Edit.FindinFiles* |
| Replace in files | Ctrl+Shift+H | *File.ReplaceinFiles* |

## Групповые символы

Флажок Use диалогового окна Find and Replace позволяет выбрать между регулярными выражениями и групповыми символами (wildcards) для формирования строки поиска. Если вы не используете регулярных выражений, параметр Wildcards обеспечит вас большим количеством дополнительных возможностей. Групповые символы шаблона позволяют тонко настроить параметры поиска.

Посмотрим, как это работает. Если в файле, содержащем ряд возрастающих значений типа HO3XA01, H03XB02, H03XC03, H03XD04 и т. д., нужно найти те, что

удовлетворяют некоторому образцу, можно использовать групповой символ #, обозначающий любую цифру в данной позиции образца, и ?, обозначающий любой символ в данной позиции образца. Чтобы выполнить поиск, используя эти символы, установите в диалоговом окне Find флажок Use и в примыкающем к нему поле со списком выберите Wild Cards. Теперь можно ввести строку *H03X?0#* (или что-то в этом роде).

Применение групповых символов позволяет осуществлять поиск текста без учета последних символов или групп символов. Это обеспечивает символ '. Продолжая предыдущий пример, более общая строка поиска, которой удовлетворяют и все строки, найденные по предыдущему образцу, выглядит так: *H03X**. Если тело образца не слишком часто встречается в файле, подобное расширение образца вп< >лне приемлемо.

Не забывайте, что поиск с использованием ' может принести улов, существенно больший, чем вы ожидали. К примеру, если вы ищете слово *class* в редакторе кода с помощью строки поиска *cla**, одним из найденных элементов может оказаться *class Class1*. Если вы выполняете замену, вам, вероятно, такие строки не нужны. Если известен образец, описывающий элемент, который нужно найти, лучше использовать более специфичные шаблонные символы.

Для поиска строк, которые не содержат определенных символов, служит групповой символ [!]. Например, продолжая работу с нашим списком, допустим, что нужно найти все элементы, удовлетворяющие шаблону H03X?0#, кроме тех, в которых пятой буквой стоит A или C. Строка поиска *H03X[!AC]0#* позволит добиться этого результата.

Символ [ ] работает, принимая любой из символов, находящийся в этих квадратных скобках. Чтобы получить элементы, содержащие только A или D, строка поиска должна выглядеть так: *H03X[AD]0#*.

Групповые символы не столь мощное средство, как регулярные выражения, о которых мы поговорим ниже, но их проще запомнить, тем более, если регулярные выражения не приходится использовать часто. Вот список групповых символов, поддерживаемых в операциях поиска и замены в Visual Studio .NET (табл. 3-7):

**Табл. 3-7. Групповые символы, поддерживаемые в Visual Studio.NET**

| Символ | Описание |
|---|---|
| ? | Любой одиночный символ |
| # | Любая одиночная цифра |
| * | Один или более символов |
| [!] | Символов из этого списка не должно быть в данном месте в искомых элементах |
| [ ] | В этом месте должны находиться только символы из этого списка |

## Регулярные выражения

О регулярных выражениях написаны целые тома. Однако эта книга посвящена Visual Studio .NET, и мы обсудим вопрос ровно настолько, насколько требуется, чтобы начать применять регулярные выражения при поиске и замене. Visual Studio .NET поддерживает использование в операциях поиска и замены около 70 регу-

лярных выражений, но мы обсудим лишь часть из них. Чтобы инициировать использование регулярных выражений в поиске, надо установить флажок Use и выбрать Regular Expressions в поле со списком Use.

Выражение-множество [ ] похоже на упоминаемый ранее символ шаблона. Это выражение служит для включения в поиск нескольких разных символов и позволяет указать диапазон символов, помещая тире между двумя символами. Так, для поиска любых символов из диапазона от *a* до *d* с помощью регулярного выражения следует задать в поле со списком Find What строку *[a-d]*. Выражение *[a-d]b* ищет в текстовом редакторе строки *ab, bb, cb* и *db*.

Если существует возможность указать, что следует найти, должна быть возможность указать, чего искать не следует. Выражение [^] позволяет исключить ненужные варианты. Так, выражению *[^t-v]s* удовлетворяют строки *as, bs, cs, ds*, а строки *ts, us, vs* и т. п. будут отсеяны.

Если требуется найти что-то в началах строк файла, в начало строки поиска нужно поместить символ ^. Символ S обозначает поиск в конце строки. Два следующих выражения обозначают поиск в начале и в конце отдельного слова: символ < обозначает в строке поиска начало любого слова, а > — конец любого слова. Эти выражения должны находиться в строке поиска в логической последовательности. К примеру, если нужно найти в файле все вхождения слова *int,* в строке поиска можно задать *<int*. Будут найдены все слова, начинающиеся на *int*. То есть подойдут *integer* и *int*, а вот *mint* — нет. Но, если требуется найти только появления в файле *int* как отдельного слова, к строке поиска нужно добавить закрывающее выражение >.

Инструменты регулярных выражений позволяют создавать сколь угодно сложные выражения. Вы можете использовать эти типы выражений для поиска в коде по образцу, соответствующему оператору определенной формы. Так, если нужно найти C#-операторы, передающие в функцию *MessageBox.Show* лишь один аргумент — строку, выражение поиска может выглядеть так;

```
<MessageBox.Show\(:q|(<.>)\);
```

Это относительно сложное выражение, но оно включает уже известные нам элементы. Так, < — это начало слова. Слова *MessageBox.Show* — то, что мы ищем. Выражение \ обозначает, что следующий за ним символ — литерал, а не элемент регулярного выражения. Выражение *:q* обозначает строку в кавычках. То же самое можно представить выражением *((«[^»]*»)|('[^']*'))*. Выражение | после *:q* обозначает операцию «или», Поскольку строка в *MessageBoxSbow* может быть представлена строковой переменной, вы можете искать там любое отдельное слово, поместив <.> в скобках. Точка . обозначает любой символ, а скобки служат для участия в операции «или». И, наконец, проверяется наличие точки с запятой.

В этом выражении одно слабое место, оно связано с тем, как форматируются в редакторе кода вызовы методов. Если программист, написавший вызов метода, который мы ищем, поставил между концом параметра и закрывающей скобкой символ пустого промежутка (white space), поиск не сработает. Во избежание этого следует добавить выражение *:Wh* и вслед за ним выражение ', означающее любое количество экземпляров (в том числе нулевое) предшествующего выражения. Уточненное выражение выглядит так: *<MessageBoxShow\(:q|(<.>):Wh*'\)*. А если вспом-

нить о пустых промежутках перед строкой в кавычках, то уже так: *<MessageBox.Show\(Wh*:q|(<.>):Wh*\)*;А если поставить пробелы перед точкой с запятой?! Подробнее о регулярных выражениях см. «Mastering Regular Expressions», Jeffrey E. E Eriedel (O'Reilly, 1997) и «Writing Secure Code», Second Edition, Michael Howard и David LeBlanc (Microsoft Press, 2003)[1]. В первой книге хорошо объясняются регулярные выражения, вторая дает глубокое понимание их использования.

В табл. 3-8 приведены выражения, которые обсуждались в этой главе. Полный список выражений см. в справке по Visual Studio .NET (щелкните Help в диалоговом окне Find и на открывшейся странице справки вы найдете ссылку на все регулярные выражения, поддерживаемые в Visual Studio .NET).

**Табл. 3-8. Некоторые регулярные выражения**

| Выражение | Описание |
|---|---|
| *[ ]* | Поиск приведенного набора символов. Используя символ - (тире), можно задавать диапазон символов. |
| *[^ ]* | Поиск символов, не входящих в приведенный набор символов. |
| *< или >* | Искать в начале или в конце слова. |
| *^ или $* | Искать в начале или в конце строки. |
| . | Любой отдельный символ. |
| \ | Следующий символ строки поиска является литеральным символом, а не регулярным выражением. |
| * | Ни одного, один или несколько символов. |
| *:q* | Строка в кавычках. |
| *:Wh* | Любой символ пустого промежутка. |

## Поиск из командного окна

Часто удобнее выполнять сложные поисковые операции из командного окна. Командное окно предоставляет больше места для ввода строки поиска, что немаловажно при использовании групповых символов шаблона и регулярных выражений.

Чтобы выполнить команду Find из командного окна, введите **Edit.Find** и после этого строку, которую нужно найти в текущем файле, и нажмите Enter. Если требуется установить переключатели, связанные с командой, напечатайте их после строки поиска. Например, если при поиске используются групповые символы, команда может выглядеть так:

```
Edit.Find  Class» /Wild
```

Прочие переключатели команды Find соответствуют описанным параметрам диалогового окна Find. Если вы их очистили, они заново устанавливаются после каждого поиска, как если бы их установили в диалоговом окне Find. Так что нет нужды задавать их каждый раз при поиске из командной строки, Чтобы посмотреть параметры установленные в данный момент, введите **Edit.Find /options**:

[1] *Ховард М., Леблан Д.* «Защищенный код». М.: «Русская Редакция», 2003. — *Прим. перев.*

```
>Edit.Find /options
/wild /doc /names /sub
>
```

Ниже перечислено большинство переключателей, применяемых для команды Find в Visual Studio .NET (табл. 3-9). Такие же переключатели применяются для *Edit.Replace*, *EditFindinFiles* и *EditJKeplaceinFiles*. Чтобы получить полный список переключателей, введите в командном окне **help Edit.Find**.

**Табл. 3-9. Переключатели для команды Find**

| Переключатель | Краткая форма | Описание |
|---|---|---|
| */case* | /c | Чувствительность к регистру |
| */doc* | /d | Поиск в текущем документе |
| */hidden* | /h | Поиск в скрытом коде |
| */markall* | /m | Отмечать каждую находку в области индикаторов |
| */open* | /o | Поиск по всем открытым документам |
| */options* | /t | Показать текущие настройки поиска |
| */proc* | /P | Поиск только в текущей процедуре |
| */sel* | /s | Поиск только в выделенном тексте |
| */up* | /u | Поиск к началу |
| */regex* | Л | Поиск с использованием регулярных выражений |
| */wild* | \| | Поиск с использованием символов шаблона |
| */word* | /w | Искать отдельное слово |

**Примечание** Команда *EditFind* также имеет переключатель */reset*, предназначенный для установки всех параметров поиска в их значения по умолчанию. В текущей реализации этого переключателя имеется ошибка, требующая ввести для переключателя параметр.

## Инкрементный поиск

Инкрементный поиск позволяет сэкономить время. Он обрабатывает по одному символу за раз, проверяя каждое слово из строки поиска от начала файла. Чтобы запустить Инкрементный поиск, нажмите Ctrl+I. После этого указатель мыши изменит форму на стрелку, направленную вниз (рис. 3-23).

После изменения формы стрелки начните печатать искомую строку. По мере ввода букв будут выделяться строковые фрагменты, содержащие набранные буквы. Когда вы завершите ввод образца, который хотите найти, снова нажмите Ctrl+I, чтобы перейти к следующему фрагменту, который соответствует образцу. Вы можете нажимать Ctrl+I, пока не достигнете конца документа, чтобы просмотреть каждый из найденных экземпляров искомого текста. Если вы хотите запустить поиск от конца к началу, нажмите Ctrl+Shift+I. Этот поиск работает аналогично. Чтобы выйти из инкрементного поиска, нажмите Enter или Esc.

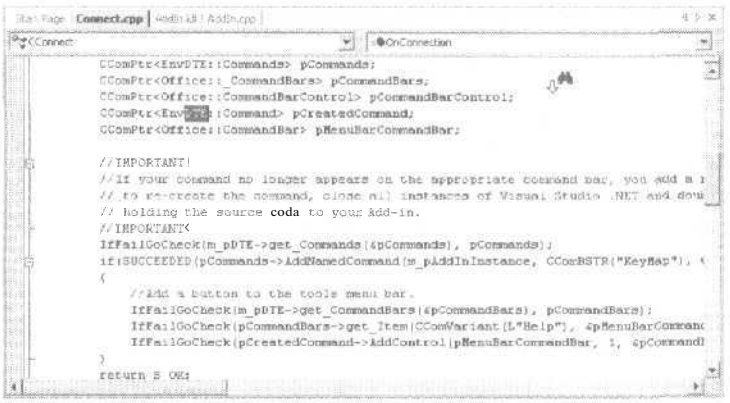
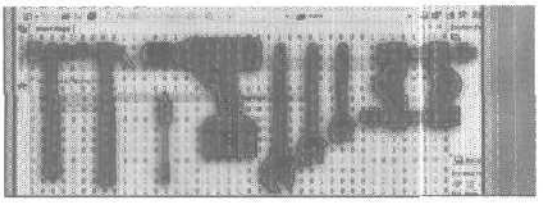



Рис. 3-23. *Запуск инкрементного поиска нажатием Ctrl+I*

# Что дальше

В этой главе мы узнали об именованных командах в редакторе кода IDE, о настройке IDE с применением псевдонимов команд в командном окне, о быстрых клавишах и добавлении элементов меню, связанных с именованными командами и макросами. В главе 4 вы познакомитесь с тем, как расширять IDE с помощью собственных макросов.



ГЛАВА

# 4

# Макросы Visual Studio .NET

Возможности макросов в Microsoft Visual Studio .NET являются, пожалуй, одной из наиболее убедительных причин использовать эту среду разработки. Они предоставляют почти всю функциональность, доступную через объектную модель автоматизации, но в более простой форме с поддержкой сценариев.

В этой главе вы познакомитесь с макросами Visual Studio .NET. Вы узнаете как записать макрос и как отредактировать проект макроса в Macros IDE. Здесь также показано, как расширить функциональность макроса с помощью сборок .NET и как совместно использовать существующий макрос. Кроме того, мы покажем как преобразовать проект макроса в полнофункциональную надстройку (add-in) Visual Studio .NET, используя макрос, поставляемый в составе примеров Visual Studio .NET.

## Макросы: скотч для Visual Studio .NET

Макросы Visual Studio .NET пишутся на языке Visual Basic .NET. Это намного лучше, чем возможности Visual Basic Scripting Edition (VBScript), встроенные в Microsoft Visual C++ 6.0. Visual Basic .NET позволяет задействовать всю функциональность .NET Framework и его собственную объектную модель, предлагая исключительно мощный и привлекательный набор возможностей, которые можно применять для автоматизации выполнения задач в IDE. Фактически вы можете преобразовать любой макрос в надстройку на базе Visual Basic .NET, которую можно компилировать и применять совместно с другими разработчиками.

Как говорилось в главе 1, макросы Visual Studio .NET записываются в файлы с расширением .vsmacros и хранятся в папке VSMacros71, расположенной в папке для проектов Visual Studio .NET. Вы можете задать папку для проектов Visual Studio .NET в диалоговом окне Options на странице Projects and Solutions в папке Environment. Путь по умолчанию — My Documents\Visual Studio Projects.

Обычно макросы Visual Studio .NET создаются одним из двух способов. Вы можете записать макрос в IDE (Ctrl+Shift+R); код, генерируемый во время сеанса записи, будет помещен в метод *MyMacros.RecordingModule.TemporaryMacro*. Альтернативный способ: открыть Macros IDE (Alt+F11) и написать с нуля новый метод. Одна из самых привлекательных черт макросов — то, что они разработаны *идя* автоматизации функциональных возможностей Visual Studio .NET IDE. Это значит, что вы можете просто записать макрос, скопировать сгенерированный код в новый метод и использовать его как основу для своего проекта автоматизации. Эту же технику можно использовать при создании кода надстройки для Visual Studio .NET.

К макросам в IDE обращаются точно так же, как и к любой другой именованной команде. Вы можете ввести имя макроса в окне Command Window (Ctrl+Alt+A), добавить ссылку на макрос на панель инструментов или в меню, назначить макросу комбинацию клавиш, запустить макрос, дважды щелкнув его в окне Macro Explorer, или выполнить макрос прямо из Macros IDE.

**Примечание** Когда вы запускаете макрос двойным щелчком в окне Macro Explorer, фокус возвращается последнему активному окну. Таким образом, вы можете установить активный документ, открыть окно Macro Explorer, дважды щелкнуть макрос, и он окажет действие на последний активный документ.

Мы называем макросы «скотчем для Visual Studio .NET» в лучшем значении этого термина. Скотч делается из прочного материала и помогает выполнять задачи быстро и легко. Мы могли бы описать макросы так: чрезвычайно мощные инструменты IDE, не требующие большого времени на размышления. Вы можете создать макрос для выполнения вашей задачи, а затем удалить его. Макрос можно преобразовать в полнофункциональную надстройку и затем навести глянец в коде в соответствии с вашими пожеланиями.

## Запись макросов

Для этого нажмите комбинацию клавиш Ctrl+Shift+R — она выводит на экран панель инструментов Recorder (рис. 4-1) и создаег модуль макроса с именем *RecordingModule*, если он еще не существует. Заметьте: вы можете приостановить, завершить и даже отменить начатый сеанс записи.



Рис. **4-1.** *Панель инструментов Recorder*

Простейший способ начать знакомство с макросами — записать простейший макрос. Например, вы хотите найти слово *Connect* в тексте вашей программы. Обычно для этой цели применяют команду Find или Find In Files. Однако, используя ее в контексте макроса, вы получите более гибкое решение, которое можно будет задействовать в дальнейшей работе.

Чтобы записать макрос сделайте так.

1. Нажмите Ctrl+Shift+R, чтобы начать запись макроса.
2. Нажмите Ctrl+F, чтобы открыть диалоговое окно Find.
3. Введите слово **Connect** в поле Find What.
4. Щелкните кнопку Find Next.
5. Нажмите Ctrl+Shift+R, чтобы завершить запись макроса.

Теперь у нас есть метод *TemporaryMacro*, записанный в модуле *RecordingModule* (рис. 4-2).

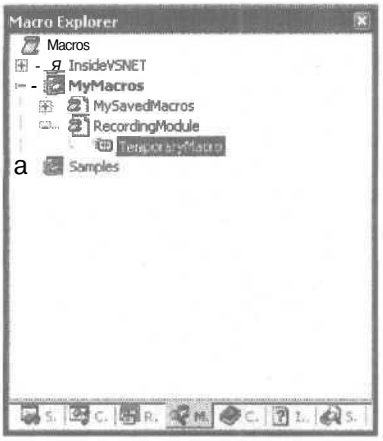



Рис. 4-2. *Окно Macro Explorer*

Ниже представлен листинг, сгенерированный в результате описанных действий. Заметьте, что перемещения мыши и нажатия клавиш (например, клавиши Tab для перемещения по элементам управления диалогового окна Replace) не записываются. Visual Studio .NET ограничивает запись макроса фактическими именами команд, которые были вызваны во время сеанса записи.

```
Imports EnvDTE
Imports System.Diagnostics
Public Module RecordingModule
    Sub TemporaryMacro()
        DTE.ExecuteCommand("Edit.Find")
        DTE.Find.FindWhat = "Connect"
        DTE.Windows.Item("Connect.cpp").Activate()
        DTE.Find.FindWhat = "Connect"
        DTE.Find.Target = vsFindTarget.vsFindTargetCurrentDocument
        DTE.Find.MatchCase = False
        DTE.Find.MatchWholeWord = False
        DTE.Find.Backwards = False
        DTE.Find.MatchInHiddenText = False
        DTE.Find.PatternSyntax = _
            vsFindPatternSyntax.vsFindPatternSyntaxLiteral
        DTE.Find.Action = vsFindAction.vsFindActionFind
        DTE.Find.Execute()
```

```
    End Sub
End Module
```

Чтобы воспроизвести этот макрос, нажмите Ctrl+Shift+P — комбинацию клавиш для команды *Macros.MacrosRecordingModule.TemporaryMacro*Вы увидите открытое диалоговое окно Find и выделенное первое вхождение искомого слова. В данном случае это первое вхождение слова *Connect* в файле Connect.cpp,

Взгляните на строку *DTE.Windows.Item(«Connect.cpp»).Activate()Если* файл Connect.cpp не открыт, эта строка передаст фокус IDE, так что данный макрос не слишком полезен, чтобы сохранить его для применения с различными файлами или проектами. Если закомментировать или удалить эту строку, макрос будет работать с текущим активным документом.

Чтобы сохранить записанный макрос, переименуйте *TemporaryMacro*в окне Macro Explorer или скопируйте записанный код и поместите его в другой модуль или метод макроса.

## Команды макросов

Macro Explorer позволяет управлять макросами из IDE Visual Studio .NET. Получить доступ к командам, связанным с макросом, можно из IDE через подменю Macros меню Tools или через контекстное меню в окне Macro Explorer.

Макросы делятся на проекты, которые содержат модули, хранящие методы. Проекты отображаются в виде иерархического списка в окне Macro Explorer под значком Macro. Щелчок правой кнопкой значка Macro выводит контекстное меню с командами для создания и загрузки проектов макросов. К тем же функциям можно обратиться, введя имя команды в окне Command Window. Вот список команд, относящихся к проектам макросов (табл, 4-1):

**Табл. 4-1. Команды проектов макросов**

| Команда | Описание |
|---|---|
| *Tools.LoadMacroProject* | Вызывает диалоговое окно Add Macro Project, в котором можно выбрать файл проекта макроса. |
| *Tools.NewMacroProject* | Вызывает диалоговое окно New Macro Project, в котором можно сохранить макрос в заданный проект. |
| *Tools.MacrosIDE* | Вызывает Macros IDE. Этой команде соответствует комбинация клавиш Alt+F11. |

Окно Macro Explorer выводится на экран комбинацией Alt+F8. Большинство команд, доступных из контекстных меню в окне Macro Explorer, доступно и из окна Command Window (поскольку элементы в окне Macro Explorer теряют фокус, когда вы переключаетесь в окно Command Window). Можно переименовать проект макроса, щелкнув его правой кнопкой в окне Macro Explorer, а затем выбрав Rename. Это позволяет на месте отредактировать имя проекта макроса. Удаляет проект макроса команда Delete из контекстного меню. Аналогичные пункты контекстного меню доступны для переименования и удаления модулей и методов из окна Macro Explorer,

Вот несколько команд, доступных из отдельного проекта макроса (табл. 4-2):

**Табл. 4-2. Команды проектов макросов**

| Команда | Описание |
|---|---|
| *Tools.Newmodule* | Вызывает диалоговое окно New Module, в котором можно создать новый модуль из окна Macro Explorer. |
| *Tools.Newmacro* | Вызывает Macros IDE с новым методом макроса. |
| *Tools.Edit* | Открывает Macros IDE для отмеченного проекта или модуля. |

Щелчок правой кнопкой макроса в окне Macro Explorer выводит контекстное меню работы с макросами. Команда Run выполняет команду *Tools.Run* для выбранного в данный момент макроса, Rename позволяет отредактировать имя макроса. Изменение имени отражается в именах методов в Macros IDE. Команда Delete удаляет выбранный макрос, а Edit открывает текущий макрос в Macros IDE.

Вы можете выполнять многие задачи в Macro Explorer, не переходя в Macros IDE. Так, вы можете записать макрос, переименовать его для сохранения и даже добавить его к панели инструментов или меню IDE! Вероятно, вы решите, что отказ от IDE ограничивает вас, но он возможен. Чтобы извлечь максимум пользы из макросов Visual Studio .NET, вам потребуется создавать и редактировать их в Macros IDE.

## Редактирование макроса в Macros IDE

Работа в Macros IDE аналогична работе в Visual Studio ,NET. Многие комбинации клавиш работают и в Macros IDE. Редактор Macros IDE поддерживает функцию IntelliSense, а справочная система для макросов интегрирована в эту IDE.

Вы сразу же заметите одно отличие: все загруженные проекты макросов отображаются в окне Project Explorer. Visual Studio .NET поставляется с набором полезных макросов, Чтобы увидеть эти макросы, разверните проект Samples в окне Project Explorer среды Macros IDE (рис. 4-3).

Для каждого проекта макроса используется отдельное пространство памяти, поэтому, чтобы задействовать общие функциональные возможности в нескольких макросах или преимущества общего набора событий среды, объедините создаваемые макросы в один проект. Если вы хотите получить доступ к возможностям другого проекта макроса, просто скопируйте макросы, к которым хотите получить доступ, в свой проект, Например, вы можете скопировать модули из проекта Samples в свой проект, чтобы задействовать функциональность этих макросов.

Чтобы создать новый проект макроса, нужно запустить IDE Visual Studio .NET. Затем вы можете выбрать команду New Macro Project из контекстного меню окна Macro Explorer либо ввести команду **Tools.NewMacroProject** в окне Command Window, чтобы открыть диалоговое окно New Macro Project (рис. 4-4). Введите имя и местоположение файлов вашего проекта и щелкните OK. Нажав комбинацию клавиш Alt+F11, вы откроете окно Macros IDE, где сможете работать с кодом нового проекта.

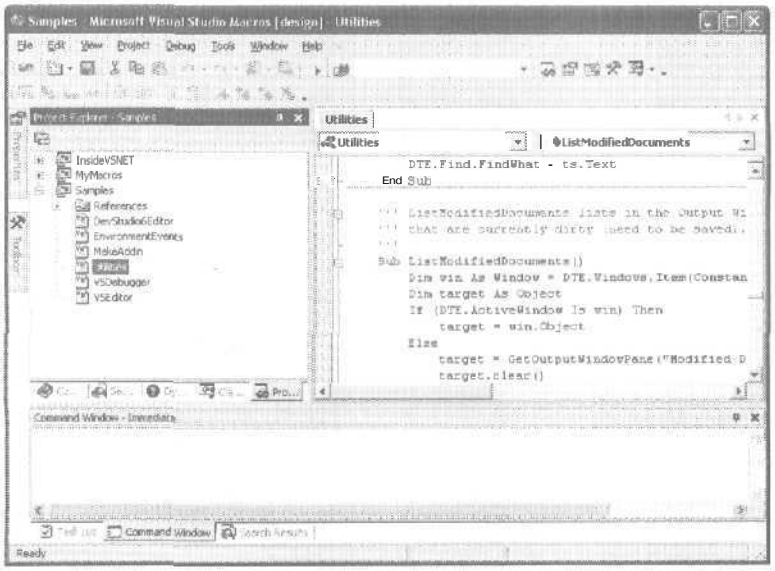



**Рис.** 4-3. *Проект Samples в Macros IDE*

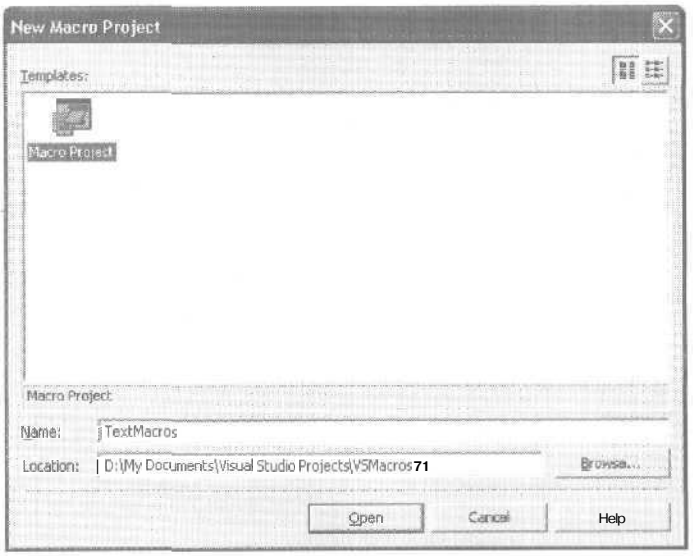



**Рис.** 4-4. *Диалоговое окно New Macro Project*

Взглянув на созданный проект макроса в окне Project Explorer, вы увидите, что по умолчанию в него был добавлен ряд возможностей. Папка References аналогична папке References в IDE Visual Studio .NET. Кроме того, в проект добавляются два новых модуля для выполнения макроса: *EnvironmentEvents* содержит сгенерированный код, который позволяет получить доступ к событиям IDE, а в *Module1* вы будете писать свой код.

**Практикум: переключение между IDE**

Чтобы перейти из Macros IDE в IDE Visual Studio .NET, щелкните кнопку Visual Studio на панели инструментов Macros IDE. В Visual Studio .NET нет панели инструментов, на которой бы аналогичная кнопка выводилась по умолчанию, поэтому вам нужно добавить ее. если вы планируете переключаться в Macros IDE таким способом. Для этого, щелкнув правой кнопкой панель инструментов IDE Visual Studio .NET, выберите команду Customize. На вкладке Commands найдите в категории Tools команду Macros IDE и перетащите ее на панель инструментов, из которой вы собираетесь ее использовать. В Macros IDE есть такая же кнопка с изображением знака бесконечности. Это облегчает переключение между двумя IDE, когда вы работаете над макросами.

Если вы будете разрабатывать много макросов, лучше всего применять машины с двумя мониторами, на один из которых выводится Macros IDE, а на другой — Visual Studio .NET.

Добавление ссылки на проект макроса слегка отличается от добавления ссылки на обычный проект Visual Basic .NET. В диалоговом окне Add Reference, используемом в Macros IDE Project Explorer (рис. 4-5). способа для добавления пользовательских сборок нет.

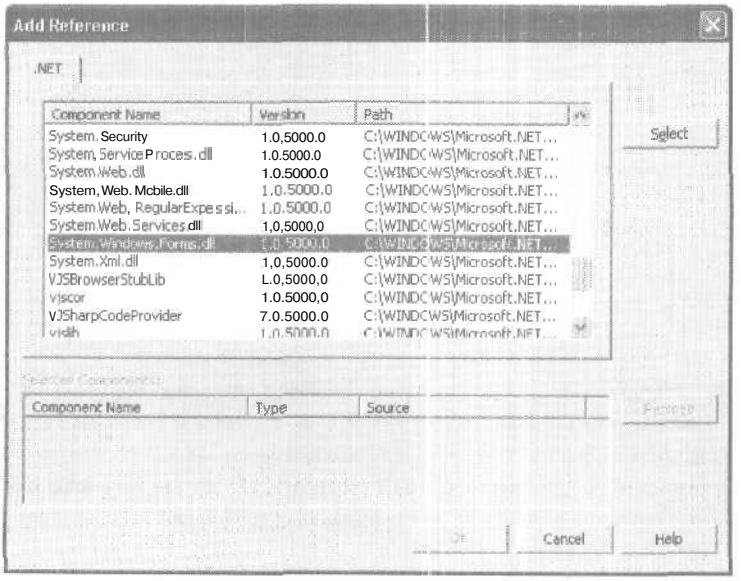



Рис. 4-5. *Диалоговое окно Add Reference*

Чтобы добавить ссылку на вашу собственную сборку, нужно скопировать ее в папку C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\PublicAssemblies. После этого можно добавить ссылку на эту сборку из диалогового окна Add Reference. Используя сборки, вы можете писать функциональную часть макроса на любом языке, обращаясь затем к ней из простого макроса. Вы также можете писать сборки, вызывающие неуправляемый код, и сборки, действующие как

обертки COM для доступа к функциональным возможностям COM из ваших макросов. Рассмотрим несколько примеров, построенных из нового проекта.

## Простой макрос

Ранее мы коснулись поведения команды *File.NewFile* в Visual Studio .NET. Некоторые программисты недовольны тем, что в диалоговом окне, отображаемом этой командой по умолчанию, для создания и открытия нового файла нужен ряд действий мышью или нажатие нескольких комбинаций клавиш. Решение очень простое: создайте макрос, выполняющий требуемые действия, и назначьте ему псевдоним в окне Command Window. Следующий код — это все, что нужно, для создания в TDE нового текстового файла:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module NewFile

    Sub NewTextFile()
        DTE.ItemOperations.NewFile("General\Text File")
    End Sub

End Module
```

Как видите, макрос создан в модуле *NewFile* и содержит единственный метод *NewTextFile*. Единственная строка кода в этом макросе создает новый файл типа Text File в папке General диалогового окна New File. О методе *NewFile*, создающем новый текстовый файл, мы поговорим через минуту. Сейчас важно отметить, что у нас есть макрос, добавляющий нужные возможности к IDE. Чтобы создать этот макрос, мы потратили некоторое время, при этом мы хотели сделать макрос предельно простым.

Есть несколько способов обратиться к макросу, который вы хотите выполнить. Первый: выполнить макрос из окна Macro Explorer в IDE Visual Studio .NET. Этот подход замечательно действует, но не является оптимальным для часто используемых макросов. Второй: создать псевдоним макроса в окне Command Window. Вероятно, это лучший вариант для команд, которые будут вводиться с клавиатуры. Чтобы назначить псевдоним, надо ввести команду alias, псевдоним и имя макроса. IntclliSense автоматически предложит полный вариант, когда вы начнете печатать макрокоманду, поэтому полный вариант строки, назначающей псевдоним, может выглядеть так:

```
>alias nf Macros.InsideVSNET.Chapter04.NewFile.NewTextFile
```

Теперь мы получили новую команду *nf*, которую можно вводить в окне Command Window. Чтобы создать новый текстовый файл, просто нажмите Ctrl+Alt+A и введите *nf*. Конечно, в диалоговом окне Options можно назначить макросу комбинацию клавиш. Если вспомнить, что последовательность комбинаций начинается с «Ctrl+,», хорошим вариантом может быть «Ctrl+,», «Ctrl+N». Кроме того, можно добавить запускающую макрос кнопку на панель инструментов (см. главу 3).

В этом примере важную роль играет оператор *Imports*. API объектной модели Visual Studio .NET содержится в пространстве имен *EnvDTE*. Об объектной модели см. главы 5-12. Здесь мы просто познакомим вас с объектной моделью и предложим несколько наиболее общих функциональных возможностей, которые можно применять в макросах. Большинство тем, описанных в главах второй части книги, применимы как в макросах, так и в надстройках. Фактически вы можете задействовать макросы для быстрой проверки функциональности создаваемых надстроек. Вы сэкономите время, так как обычная отладка надстройки требует ее компиляции и загрузки второго экземпляра IDE. Используя макросы, вы обращаетесь к объектной модели, пишете и тестируете процедуры, а затем добавляете их в свой проект надстройки.

# Работа с макросами

Создаваемые макросы будут использовать объектную модель для доступа и автоматизации различных частей IDE. Ниже мы продемонстрируем применение макросов для автоматизации ряда простых задач и поговорим о применении объектной модели к документам и окнам IDE. Мы также коснемся событий и приведем несколько примеров, которые наставят вас на путь истинный. Многие из приведенных здесь материалов мы подробно обсудим во второй части книги.

## Управление документами и текстом

Некоторые задачи, которые можно выполнять, используя макросы, включают работу с текстом документов. Вам может потребоваться найти текст, изменить выделение или просто вставить текст в документ, Объект *Document* пространства имен *DTE* предоставляет много возможностей, упрощающих управление текстом документов.

Макросы часто работают с текущим документом, который имеет фокус. Получить текущий активный документ в IDE позволяет свойство *DTE.ActiveDocument*, возвращающее объект *Document*. (Вспомните: документ Visual Studio .NET — это окно редактора или дизайнера, открытое в центре IDE.) Если документ является окном редактора, он имеет связанный объект *TextDocument*.

У объекта *TextDocument* есть три свойства, интересные для программистов, желающих управлять текстом внутри объекта: *StartPoint* возвращает объект *TextPoint*, указывающий на начало документа, *EndPoint* возвращает объект, который указывает на конец документа, а *Selection* — объект *TextSelection*, предоставляющий множество свойств и методов для работы с выделенным текстом.

Объект *TextPoint* предоставляет информацию о местоположении для функций редактирования внутри документа. Вы создаете объект *TextPoint* в документе, когда хотите вставить текст в документ или управлять текстом документа либо когда нужно получить сведения об отдельном документе. Объекты *TextPoint* не зависят от выделения текста, и вы можете создавать несколько таких объектов в одном документе.

Взглянем на пару примеров, использующих упомянутые объекты. Хорошо ознакомьтесь с кодом, так как эта информация пригодится для большинства написанного вами кода макросов автоматизации.

Сначала давайте получим *ActiveDocument*, создадим пару объектов *EditPoint* и добавим текст в *ActiveDocument* с помощью следующего фрагмента:

```
Sub CommentWholeDoc()
    Dim td As TextDocument = ActiveDocument.Object
    Dim sp As TextPoint
    Dim ep As TextPoint
    sp = td.StartPoint.CreateEditPoint()
    ep = td.EndPoint.CreateEditPoint()

    sp.Insert("/* ")
    ep.Insert(" */")

End Sub
```

Выполнение этого макроса для документа с кодом Visual C# или Visual C++ закомментирует весь документ, если он ранее не содержал комментариев. Макрос не слишком полезен, но он демонстрирует, как соединить все части вместе. Можно задействовать IntelliSense, чтобы получить справку при создании объектов для экспериментов с другими возможностями.

А вот более практичный макрос: он вставляет текст в документ в зависимости от выделения и создает в документе HTML-комментарий. Этой возможности нет в Visual Studio .NET 2003, и вы можете счесть этот простой макрос настолько полезным, что добавите его к своему набору инструментов. Мы объявим переменную *ts* как объект *TextSelection* и присвоим ей текущий выделенный фрагмент, используя *DTE.ActiveDocument.Selection*:

```
Sub HTMLComment()
    Dim ts As TextSelection = DTE.ActiveDocument.Selection
    ts.Insert("<!- ", vsInsertFlags.vsInsertFlagsInsertAtStart)
    ts.Insert(" ->", vsInsertFlags.vsInsertFlagsInsertAtEnd)
End Sub
```

Данный макрос использует метод *TextSelection Insert* для вставки текста вокруг объекта *Selection*. Метод *Insert* получает два аргумента; строку, которую вы хотите вставить в выделенный фрагмент, и константу *vsInsertFlags*, определяющую место вставки. В первом вызове *Insert* в примере используется значение *vsInsertFlagsAtStart*, а во втором — *vsInsertFlagsAtEnd*. Эти константы перечислены в табл. 4-3.

**Табл. 4-3. Константы *vsInsertFlags***

| Константа | Описание |
|---|---|
| *vsInsertFlagsCollapseToStart* | Свертывает точку вставки от конца выделенного фрагмента до текущего *TextPoint*. |
| *vsInsertFlagsCollapseToEnd* | Свертывает точку вставки от начала выделенного фрагмента до текущего *TextPoint*. |
| *vsInsertFlagsContainNewText* | Заменяет текущий выделенный фрагмент. |
| *vsInsertFlagsInsertAtStart* | Вставляет текст перед началом выделенного фрагмента. |
| *vsInsertFlagsInsertAtEnd* | Вставляет текст сразу за концом выделенного фрагмента. |

Объекты *Selection* и *TextPoint*, а также методы, доступные через *DTE*, предоставляют хорошую основу для различных операций, выполняемых над исходным кодом с помощью макросов.

## Работа с окнами

Окна в Visual Studio .NET управляются через объекты *Window* из *DTE.Windows*. Возможности, предоставляемые объектом *Window*, зависят от типа окна. Если точнее, то от объекта *Window* наследуются *CommandWindow*, *OutputWindow*, *TaskList*, *TextWindow* и *ToolBox*.

Объект *OutputWindow* наиболее подходит для написания макросов. Вы можете применять его для вывода сообщений так же, как применяете *printf* или *Console.Write* в консольных приложениях или *MsgBox* и *MessageBox.Show* в Windows-приложениях.

Чтобы задействовать объект *OutputWindow* для отображения сообщений, нужно создать новый метод, принимающий в качестве аргумента строку. Затем вы можете вызывать этот метод с аргументом так же, как используете для вывода сообщений метод *MsgBox*. В следующем примере приведен метод *MsgWin*, получающий в качестве аргумента строку. Вы можете задействовать этот метод вместо *MsgBox*, когда вам нужно быстро вывести небольшое текстовое сообщение.

```
Sub MsgWin(ByVal msg As String)
    Dim win As Window = DTE.Windows.Item(Constants.vsWindowKindOutput)
    Dim cwin As Window =
        DTE.Windows.Item(Constants.vsWindowKindCommandWindow)
    Dim ow As OutputWindow = win.Object
    Dim owp As OutputWindowPane
    Dim cwp As CommandWindow = cwin.Object
    Dim i As Integer
    Dim exists As Boolean = False
    ' Выяснить, запущен ли метод из окна Command Window.
    ' Если да, направить вывод туда.
    ' Если нет, направить вывод в окно Command,
    If (DTE.ActiveWindow Is cwin) Then
        cwp.OutputString(msg + vbCrLf)
    Else
        ' Определить, существует ли панель с заданным именем.
        ' Если да, следует направить вывод на нее, иначе мы
        ' получим несколько окон с одинаковым именем.
        For i = 1 To ow.OutputWindowPanes.Count
            If ow.OutputWindowPanes().Item(i).Name() = "MsgWin Output" Then
                exists = True
                Exit For
            End If
        Next
        ' Если панель для вывода существует,
        ' используем ее для вывода строки,
        ' в противном случае добавим ее в список.
        If exists Then
            owp = ow.OutputWindowPanes().Item(i)
```

```
        Else
            owp = ow.OutputWindowPanes.Add("MsgWin Output")
        End If
        ' Сделать окно Output видимым, активизировать панель,
        ' и вывести на нее строку.
        win.Visible = True
        owp.Activate()
        owp.OutputString(msg + vbCrLf)
    End If
End Sub
```

*MsgWin* использует одну интересную возможность, найденную в примерах, поставляемых с Visual Studio .NET. Метод определяет, был ли он вызван из окна Command Window. Если да, то вывод направляется пользователю обратно в окно Command Window. Если же метод вызван из макроса, который был запущен через пункт меню, комбинацию клавиш или кнопку, *MsgWin* направляет вывод в окно Output с именем MsgWin Output.

---

**Совет** Проект макроса Samples, поставляемый с Visual Studio .NET, содержит много функционального кода, который вы можете использовать в своих макросах.

---

Чтобы задействовать макрос *MsgWin,* вы должны вызвать его из другого метода. Для примера мы создали метод, который перечислят открытые в данный момент в IDE окна:

```
Sub MsgWinTest()
    Dim wins As Windows = DTE.Windows()
    Dim i As Integer

    For i = 1 To wins.Count
        MsgWin(wins.Item(i).Caption.ToString())
    Next
End Sub
```

На рис. 4-6 показана IDE Visual Studio .NET после вызова макроса *MsgWinText* в IDE.

Базовый вариант макроса *MsgWin* можно улучшить. Довольно просто перегрузить метод *MsgWin,* чтобы разрешить такие действия, как очистка области вывода или добавление заголовка к списку. Для примера создадим перегруженную функцию *MsgWin ,* которая очищает область выводя:

```
Sub MsgWin(ByVal msg As String, ByVal clr As Boolean)
        :
        ' Если clr равно True, очистить область вывода.
        If clr = True Then
            owp.Clear()
        End If
        ' Здесь мы делаем окно Output видимым,
        ' активизируем панель и выводим строку,
        win.Visible = True
```

```
        owp.Activate()
        owp.OutputString(msg + vbCrLf)
    End If
End Sub
```

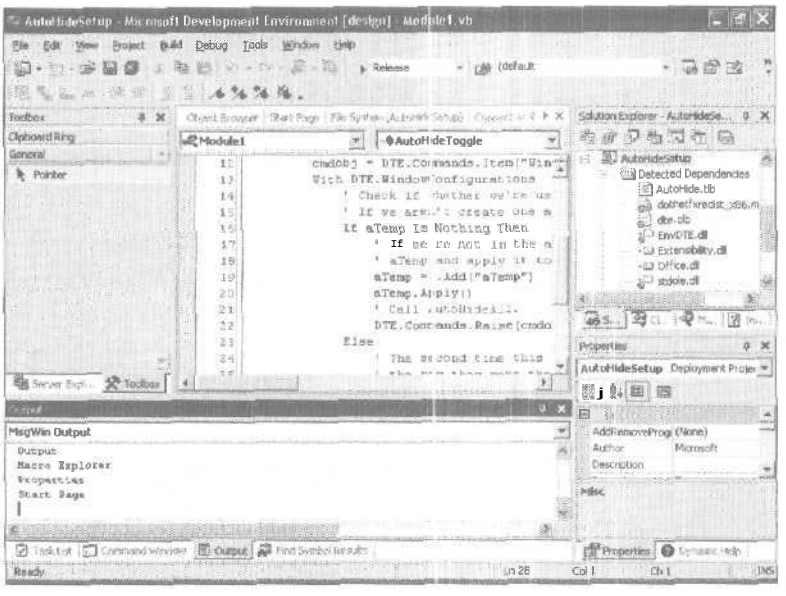


**Рис. 4-6.** *Окно MsgBox Output в IDE*

Конечно, эта перегрузка не делает ничего хорошего, если вы используете макрос способом, показанным в *MsgBoxTest* но вы можете видеть, насколько просто изменить макрос.

Другой способ добавить функциональность в макрос — создать сборку на любом языке программирования, а затем ссылаться на нее из ваших проектов макросов. Мы сделаем это с объектом *CommandWindowPaneEx.*

## Использование сборок в ваших макросах

После того, как вы начнете использовать много макросов, становится очевидной пара вещей. Во-первых, вы можете применять макросы для тестирования сборок .NET. Так, если вы хотите проверить некоторую функциональность, достаточно сделать ссылку на нужную сборку и затем вызывать методы из макроса.

Во-вторых, вы заметите, что вам приходится использовать только Visual Basic, и, если вы не любите его, вы можете потратить много времени на те веши, которые гораздо быстрее сделали бы на другом языке. Как мы уже говорили, можно писать функциональные блоки макросов на языках, отличных от Visual Basic: объедините функции в сборку и сошлитесь на эту сборку из вашего проекта макроса.

Для книги мы написали набор сервисных функций, и вы можете задействовать их в своих макросах и надстройках. В папке Utilities, сопровождающих книгу ресурсов, вы найдете решение Utilities. Оно содержит объект *OutputWindowPaneEx.* Постройте решение и скопируйте файл InsideVSNET.Utilities.dll из папки bin\debug

проекта в папку C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\PublicAssemblies.

После того как файл скопирован в папку Public Assemblies, можно добавить ссылку на сборку из проекта макроса в Macros IDE, щелкнув правой кнопкой папку References в окне Project Explorer и выбрав команду Add Reference или выбрав папку References и введя команду **Project.AddReference** в окне Command Window Macros IDE. На вкладке .NET диалогового окна Add Reference вы увидите новую сборку InsideVSNET.Utilities. Выберите ее в списке, щелкните кнопку Select, а затем OK. Вы увидите новую сборку в списке ссылок в папке References.

Добавив ссылку в проект, полезно ввести в модуль соответствующий оператор *Imports*. В данном случае добавьте в начало модуля строку:

```
Imports InsideVSNET.Utilities
```

Как только вы добавите ссылку, IntelliSense станет предлагать дополнить строку, когда вы будете создавать и применять объект *OutputWindowPaneEx,* Польза этого объекта в том, что он позволяет выбирать, куда направлять вывод: в окно Output IDE Visual Studio .NET или в Macros IDE. В примере мы задаем вывод в Macros IDE, передавая *DTE.MacrosIDE* при создании объекта. Мы также слегка изменили тестовую программу, которая перечисляет открытые окна Macros IDE вместо перечисления окон IDE Visual Studio .NET, как мы делали ранее.

```
Sub OutputWindowPaneExTest()
    Dim owp As New OutputWindowPaneEx(DTE.MacrosIDE)
    Dim wins As Windows = DTE.MacrosIDE.Windows()
    Dim i As Integer

    owp.Activate()

    For i = 1 To wins.Count
        owp.WriteLine(wins.Item(i).Caption.ToString())
    Next
End Sub
```

Кроме выполнения задач, которые можно было проделать с объектом *OutputWindowPane*, *OutputWindowPaneEx* позволяет проделать ряд операций с текстом, посылаемым в окно Output. Как было сказано, вы можете задать IDE в которую будет направлен вывод. Метод *Write* имеет три перегруженных версии, позволяющие задать объект, строку или строку формата и массив параметров. Применяя эти перегруженные версии, можно задать параметры форматирования во многом аналогично использованию методов *System.Console.Write* и *System.Console.WriteLine* в .NET Framework.

## События макросов

Одна из самых мощных возможностей макросов IDE — модель событий — обеспечивает срабатывание макроса при возникновении в IDE заданных событий. Вы можете использовать события для срабатывания макросов, которые создают журналы, сбрасывают тестовые значения или управляют различными частями IDE. Ниже мы покажем, как создать обработчики событий для разных событий IDE.

Проще всего получить обработчик события для проекта макроса через окно Project Explorer в Macros IDE. Раскройте проект и вы увидите модуль *Environment-Events*. Этот файл содержит блок кода, автоматически генерируемый IDE. Ниже приведена важная часть этого кода (атрибуты удалены для экономии места):

```
Public WithEvents DTEEvents As EnvDTE.DTEEvents
Public WithEvents DocumentEvents As EnvDTE.DocumentEvents
Public WithEvents WindowEvents As EnvDTE.WindowEvents
Public WithEvents TaskListEvents As EnvDTE.TaskListEvents
Public WithEvents FindEvents As EnvDTE.FindEvents
Public WithEvents OutputWindowEvents As EnvDTE.OutputWindowEvents
Public WithEvents SelectionEvents As EnvDTE.SelectionEvents
Public WithEvents BuildEvents As EnvDTE.BuildEvents
Public WithEvents SolutionEvents As EnvDTE.SolutionEvents
Public WithEvents SolutionItemsEvents As EnvDTE.ProjectItemsEvents
Public WithEvents MiscFilesEvents As EnvDTE.ProjectItemsEvents
Public WithEvents DebuggerEvents As EnvDTE.DebuggerEvents
```

Как видите, существует множество типов событий, которые можно использовать в IDE. Фактически вы можете задействовать все события DTE, даже если они не включены по умолчанию. Вы можете дополнить список событий, чтобы получить доступ к интересующим вас событиям. Чтобы создать новый обработчик события, выберите тип обрабатываемого события в списке Class Name в верхней части окна кода (рис. 4-7).




**Рис.** 4-7. *Выбор типа обрабатываемого события в списке Class Name*

После того как выбран тип события, в списке Method Name в правом верхнем углу окна кода будут перечислены события, которые можно обрабатывать (рис. 4-8).

Выберите событие, которое вы хотите обрабатывать, и обработчик события будет сгенерирован автоматически. Из сгенерированного обработчика события вы можете вызывать созданные в проекте методы либо просто добавить нужную функциональность прямо в код обработчика события. В приведенном примере мы вызовем функцию *MsgWin* для вывода сообщения об окончании компоновки.

```
Private Sub BuildEvents_OnBuildDone(ByVal Scope As EnvDTE.vsBuildScope, _
    ByVal Action As EnvDTE.vsBuildAction) _
    Handles BuildEvents.OnBuildDone
        MsgWin("Build is done!")
End Sub
```

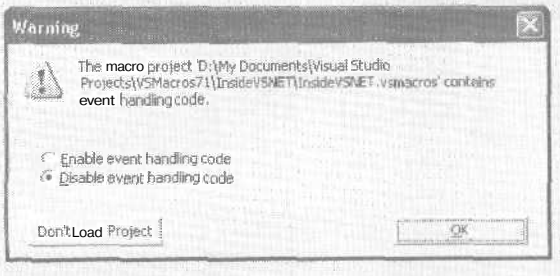



**Рис. 4-8.** *Выбор обрабатываемого события в списке Method Name*

Эти события открывают все возможности для автоматизации и настройки IDE. Работая с событиями, помните, что весь код отдельного проекта макроса совместно использует один и тот же модуль событий. Это значит, что, если вы хотите создать разные обработчики для одного события, надо создать другой обработчик события в другом проекте.

### Безопасность событий

Выполнение кода обработки событий в мощной макросреде, такой как Visual Studio .NET, влияет на безопасность. При первой загрузке проекта макроса, содержащего код обработки событий, вы увидите диалоговое окно:




Вы должны быть уверены в надежности источника загружаемых проектов макросов. Если вы не уверены во включенном в проект коде обработки событий, щелкните кнопку Disable Event Handling Code в диалоговом окне Warning и просмотрите код модуля, прежде чем использовать его.

# Совместное использование макросов

Если вы хотите применять созданные макросы совместно с другими разработчиками, решите, хотите ли вы предоставить в общее пользование исходный код, совместно использовать весь проект или только его часть.

## Экспорт модулей и проектов

Простейший способ совместного использования макроса с другими разработчиками — скопировать его код в сообщение электронной почты или Usenet. Этот подход хорошо работает, если предоставляемые в совместное пользование методы достаточно коротки и не охватывают много модулей. Если метод охватывает несколько модулей, проще будет экспортировать модули или передать весь проект целиком,

Чтобы экспортировать модуль макроса в Visual Studio .NET, откройте Macros IDE и выберите экспортируемый модуль в окне Project Explorer. Нажатие клавиш Ctrl+E вызывает команду *FileSaveSelectedItemsAs*, которая выводит диалоговое окно Export File. В меню File эта команда называется Export.

Диалоговое окно Export File позволяет сохранить модуль как файл .vb, который легко импортировать в другой проект командой *FileAddExisitingItem* (Shift+Alt+A). Не забудьте включить код из модуля *EnvironmentEvents*, если ваш макрос зависит от функций, связанных с событиями.

Если ваш макрос достаточно сложен, вы можете совместно использовать весь проект макроса. Сделать это можно несколькими способами, например, скопировать файл проекта .vsmacros и передать его или сохранить проект макроса в текстовых файлах.

Чтобы проект сохранялся в текстовых файлах, измените в IDE Visual Studio .NET свойство *StorageFormat* для нужного проекта. Выберите его в окне Macro Explorer и измените свойство *StorageFormat* в окне Properties с Binary (.vsmacros) на Text (UNICODE). После этого в папке с проектом макроса будет создано несколько файлов, подобных файлам в папке с обычным проектом Visual Studio .NET. На рис. 4-9 показана папка проекта Samples после выбора текстового формата хранения.




**Рис. 4-9.** *Проект макроса, сохраненный в текстовом формате*

Преимущество рассылки проекта в текстовом формате в том, что программисты могут ознакомиться с исходным текстом вашего проекта до его загрузки в IDE.

Открытие незнакомого проекта макроса в любом приложении всегда представляет определенную угрозу безопасности. Убедитесь по крайней мере, что модуль EnvironmentEvents.vb не содержит неожиданного кода.

Распространение двоичного модуля проекта макроса не защищает исходный код. Если это необходимо, лучше выделить функциональность в отдельную сборку, вызываемую из макроса, как мы уже показывали. А еще лучше преобразовать проект макроса в надстройку.

## Преобразование макроса в надстройку

Visual Studio .NET поставляется с макросом, который входит в проект Samples и позволяет преобразовать проект макроса в надстройку Visual Studio .NET с программой установки. Макрос в модуле *MakeAddin* берет макрос и преобразует его в проект надстройки, который вы можете компилировать и установить в своей среде. Преобразованные в надстройки макросы распространяются в виде двоичных модулей, устанавливаемых на машине пользователя. Заметьте: вам нужно проверить создаваемые таким образом надстройки, чтобы убедиться, что они выполняют свою работу.

Для примера возьмем созданный в главе 1 макрос *AutoHideToggle* и преобразуем его в надстройку с помощью макроса *MakeAddin*. Для начала скопируйте макрос, который вы хотите преобразовать в надстройку, в новый проект макроса. Можете дать проекту имя, совпадающее с именем метода. Убедитесь, что вы построили новый проект, прежде чем использовать его.

Затем создайте в Visual Studio .NET новый проект надстройки, нажав Ctrl+Shift+N чтобы открыть диалоговое окно New Project. Раскройте папку Other Projects, а затем папку Extensibility Projects, чтобы получить доступ к шаблону проекта Visual Studio .NET Add-in (рис. 4-10).

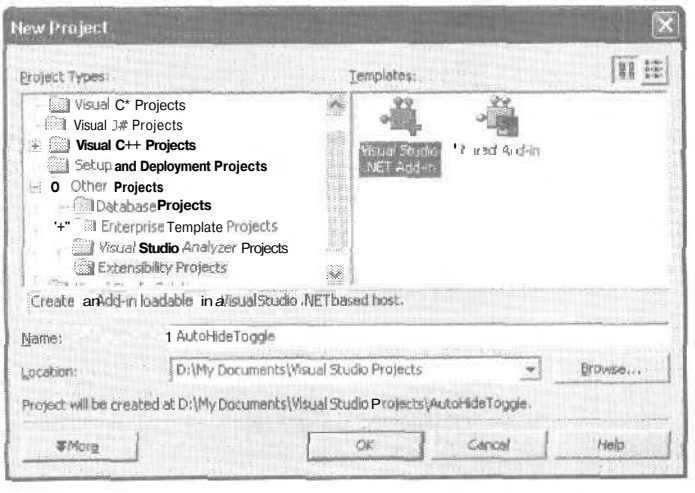



**Рис. 4-Ю.** *Создание нового проекта надстройки*

Дайте новой надстройке соответствующее имя, а затем следуйте указаниям мастера для создания нового проекта надстройки на Visual Basic .NET, устанавливающей команду меню в IDE.

Создав новый проект надстройки, нажмите Ctrl+Alt+A, чтобы вызвать окно Command Window, и введите команду Macros.Samples.MakeAddin.MakeAddinFromMacroProj. В этом окне (рис. 4-11) введите имя проекта макроса, который вы хотите преобразовать в надстройку, и щелкните OK.

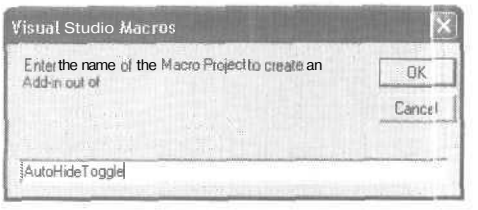
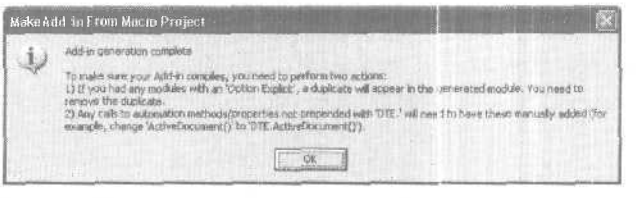



**Рис. 4-11.** *Задание проекта макроса, преобразуемого в надстройку*

Теперь, если все проекты имеют правильный тип, появится просьба подтвердить, что вы хотите выполнить макрос на текущем проекте. Щелкните OK, и проект макроса *MakeAddin* выполнит нужные действия. Когда макрос закончит работу, вы увидите диалоговое окно (рис. 4-12), сообщающее, что макрос *MakeAddin* завершил работу и добавил функциональность макроса из вашего проекта в новую надстройку.




**Ряс. 4-12.** *Диалоговое окно, сообщающее о завершении работы макроса* MakeAddin

Чтобы проверить новую надстройку, нажмите F5 для начала отладки; должен открыться второй экземпляр Visual Studio .NET IDE. В меню Tools второго экземпляра IDE вы увидите меню, содержащее имя вашего проекта и имена макросов, которые были в оригинальном проекте макроса. На рис. 4-13 вы можете видеть, что надстройка AutoHideToggle готова к использованию. Выбор команды меню делает то же. что и запуск макроса в IDE, проделанный в главе 1.

**Примечание** Если вы используете эту надстройку или макрос, описанный в главе 1, помните, что открытая раскладка окон станет раскладкой по умолчанию. Она не будет записана поверх раскладок, поставляемых с Visual Studio .NET, но если вы хотите выбрать одну из них, вернитесь на вкладку My Profile на странице Start Page и выберите альтернативную раскладку в списке Windows Layout. Затем выберите раскладку, которую вы задали при создании профиля, и окна вернутся к их первоначальному размещению.

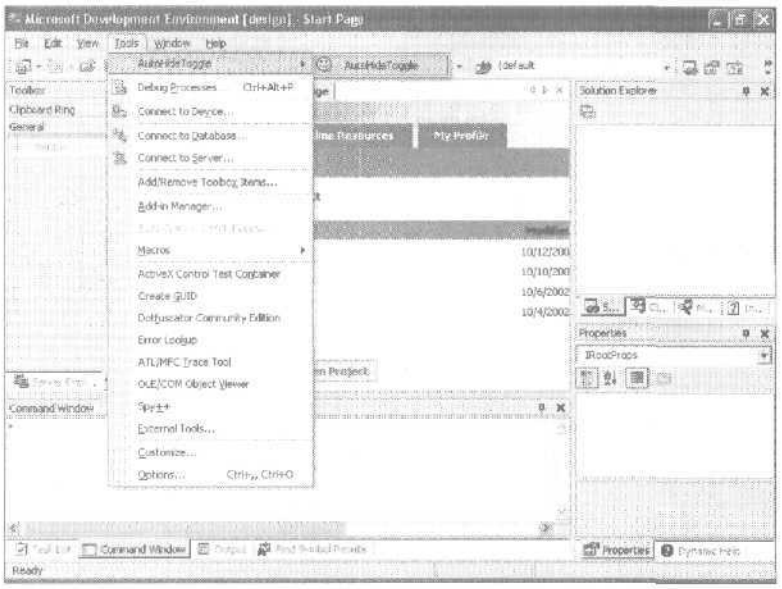



**Рис. 4-13.** *Новая надстройка AutoHideToggle в IDE*

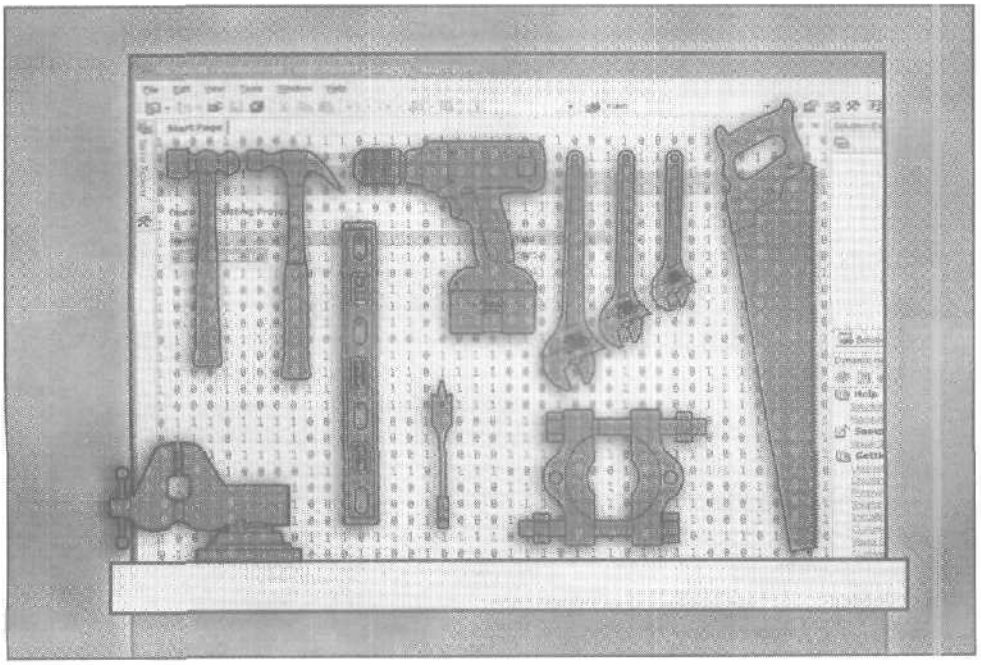
Чтобы использовать новую надстройку в Visual Studio .NET, постройте конечную версию проекта установки, сгенерированного для надстройки, и перейдите в папку, содержащую программу установки. Закройте все экземпляры Visual Studio .NET и запустите программу Setup.exe. Открыв IDE, вы увидите, что новая надстройка установлена и готова к использованию.

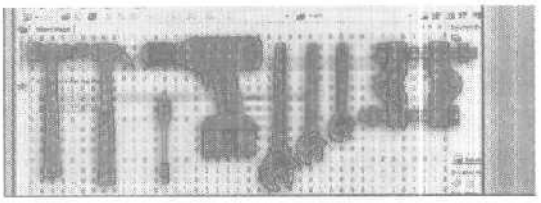
## Что дальше

В этой главе были приведены сведения о возможностях макросов Visual Studio .NET. Вторая часть книги предоставит вам информацию, достаточную для выполнения любых задач автоматизации в Visual Studio .NET.

ЧАСТЬ II

# РАСШИРЕНИЕ VISUAL STUDIO .NET





ГЛАВА

# 5

# Мастер надстроек и объектная модель

С мастером надстроек (Add-in Wizard) Microsoft Visual Studio .NET создавать надстройки не сложней, чем ездить по воскресным улицам: выберите несколько параметров, и мастер создаст готовую надстройку, с которой вы сможете гонять по интегрированной среде разработке (IDE). Конечно, не зная, где у надстройки сцепление, рычаг переключения передач и руль (не говоря уж о педали газа), далеко не уедешь. Так что в первой части этой главы мы познакомим вас с основными агрегатами надстроек. Считайте, что это «инструкция по вождению» надстроек, генерируемых мастером.

Нет смысла рулить надстройками, не зная, куда едешь, поэтому во второй части главы мы выбрали для вас живописный маршрут по программному интерфейсу Visual Studio .NET: объектную модель автоматизации.

## Мастер надстроек

В главе 4 вы узнали об удобном способе автоматизации задач в Visual Studio .NET – макросах, и мы рекомендовали их в качестве первого средства пользовательской настройки IDE. Но иногда, скажем, при разработке коммерческих программ, макросы — не лучший выбор в плане производительности и защиты интеллектуальной собственности. В таких случаях самое подходящее транспортное средство — надстройки, которые являются компилируемыми DLL (обеспечивая лучшую защиту), исполняющимися в IDE (обеспечивая повышенную производительность). И кратчайший путь стать программистом надстроек — использовать мастер Add-in Wizard, который в шесть приемов соберет ваши требования и создаст проект надстройки, подогнанный к вашим потребностям.

## Работа с Add-in Wizard

При выборе пункта меню File | New | Project, Visual Studio .NET предлагает в диалоговом окне New Project набор типов проектов. Развернув узел Other Projects и выбрав Extensibility Projects, вы обнаружите шаблон Visual Studio .NET Add-in (рис. 5-1); двойной щелчок его значка вызовет мастер Add-in Wizard.

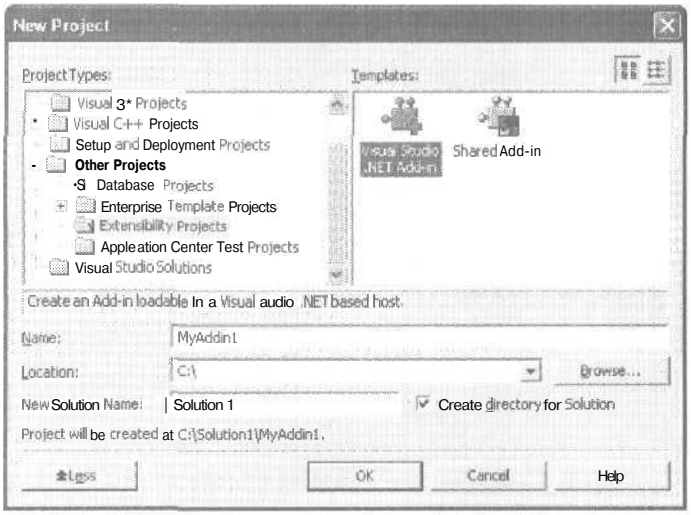



**Рис. 5-1.** *Шаблон Visual Studio .NET Add-in*

Add-in Wizard на шести страницах собирает ваши требования к окончательной форме надстройки. Мастер позволяет вам контролировать следующие параметры.

- **Язык программирования** Add-in Wizard генерирует исходный код надстройки на одном из трех языков: C#, Visual Basic .NET или Visual C++ [с применением Active Template Library (ATL)]. Когда вы пишете надстройки вручную, вы не ограничены этими языками — можно применять любой язык, поддерживающий создание COM-объектов.
- **Хост-приложение** Надстройки могут выполняться в IDE Visual Studio .NET, Macros IDE или в обеих этих средах. За некоторыми исключениями правила, применяемые к надстройкам в IDE Visual Studio .NET, действуют и для надстроек в Macros IDE. (Мы рассмотрим различия между этими хостами в соответствующих разделах.)
- **Имя и описание** Эти параметры позволяют назначить надстройке понятное имя и развернутое описание. Мастер хранит эти значения в реестре, так что они доступны любому клиенту.
- **Команда меню** Add-in Wizard может генерировать код, создающий новый элемент меню для надстройки, что дает пользователю удобный способ загрузки надстройки и выполнения команды.
- **Поддержка компоновки в командной строке** Вы можете создавать надстройки, используемые в компоновках, не требующих вмешательства пользователя. Подобные надстройки не отображают элементы пользовательского

интерфейса, требующие вмешательства пользователя (такие как модальные диалоговые окна).

- **Загрузка при запуске** Для надстройки можно указать, что она загружается автоматически при запуске Visual Studio .NET.
- **Привилегии доступа** Вы можете предоставить доступ к надстройке всем пользователям на машине или только тому пользователю, который установил надстройку,
- **Окно с информацией о программе** Вы можете предоставить информацию о поддержке своей надстройки, которую Visual Studio .NET будет отображать в диалоговом окне About.

Закончив работу, Add-in Wizard создает два проекта: проект надстройки, который строит DLL надстройки, а также проект установки, который создает файл Windows Installer (MSI), который вы можете использовать для распространения своей надстройки.

## Проект надстройки

Надстройки — это DLL, поэтому Add-in Wizard создает проект Class Library для каждой из них. Этот проект содержит исходный файл с именем Connect, определяющий класс надстройки, тоже называемый *Connect*. Класс *Connect* реализует интерфейс *IDTExtensibility2*, который играет роль главного средства взаимодействия между надстройкой и IDE. (*Connect* также реализует *IDTCommandTarget*, если вы выбрали пользовательский интерфейс в мастере.) Вот пять методов интерфейса *IDTExtensibility2* (табл. 5-1):

**Табл. 5-1. Интерфейс *IDTExtensibility2***

| Метод | Описание |
|---|---|
| *OnConnection* | Вызывается при загрузке надстройки. |
| *OnStartupComplete* | Вызывается, когда завершается загрузка Visual Studio .NET. |
| *OnAddInsUpdate* | Вызывается при загрузке и выгрузке надстройки в Visual Studio .NET. |
| *OnBeginShutdown* | Вызывается, когда Visual Studio .NET закрывается. |
| *OnDisconnection* | Вызывается при выгрузке надстройки. |

Файл Connect.cs (листинг 5-1) демонстрирует код, который мастер надстроек генерирует для типичной надстройки на C# с командой меню (за исключением некоторых комментариев).

**Листинг 5-1.** Исходный код надстройки, сгенерированный Add-in Wizard

**Connect.cs**

```
namespace MyAddin1
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.fiuntlae. InteropServices;
```

```
using EnvDTE;

/// <summary>
///   The object for implementing an add-in,
/// </summary>
/// <seealso class='IDTExtensibility2' />
[GuidAttribute("BA857E49-7873-45D2-8335-FCCD4123739E"),
    ProgId("MyAddin1.Connect")]
public class Connect : Object, Extensibility.IDTExtensibility2,
    IDTCommandTarget
{
    /// <summary>
    ///   Implements the constructor for the add-in object.
    ///   Place your initialization code within this method.
    /// </summary>
  ... publicConnect O
    {
    }

    /// <summary>
    ///   Implements the OnConnection method of the
    ///   IDTExtensibility2 interface,
    ///   Receives notification that the add-in is being loaded.
    /// </summary>
    /// <param term='application'>
    ///   Root object of the host application.
    /// </param>
    /// <param term='connectMode'>
    ///   Describes how the add-in is being loaded.
    /// </param>
    /// <param term='addInInst'>
    ///   Object representing this add-in.
    /// </param>
    public void OnConnection(object application,
        Extensibility.ext_ConnectMode connectMode,
        object addInInst, ref System.Array custom)
    {
        applicationObject = (_DTE)application;
        addInInstance = (AddIn)addInInst;

        if (connectMode ==
            Extensibility.ext_ConnectMode.ext_cm_UISetup)
        {
            object []contextGUIDS * new object[] { };
            Commands commands = applicationObject.Commands;
            _CommandBars commandBars =
                applicationObject.CommandBars; '

            try          :
            {
```

*см. след. стр.*

```
            Command command = commands.AddNamedCommand(
                addInInstance, "MyAddin1", "MyAddin1",
                "Executes the command for MyAddin1", true, 59,
                ref contextGUIDS,
                (int)vsCommandStatus.vsCommandStatusSupported +
                (int)vsCommandStatus.vsCommandStatusEnabled);
            CommandBar commandBar =
                (CommandBar)commandBars["Tools"];
            CommandBarControl commandBarControl =
                command.AddControl(commandBar, 1);
        }
        catch(System.Exception /*e*/)
        {
        }
    }
}

/// <summary>
///   Implements the OnDisconnection method of the
///   IDTExtensibility2 interface.
///   Receives notification that the add-in is being unloaded.
/// </summary>
/// <param term='disconnectMode'>
///   Describes how the add-in is being unloaded.
/// </param>
/// <param term='custom'>
///   Array of parameters that are host-application specific.
/// </param>
public void OnDisconnection(
    Extensibility.ext_DisconnectMode disconnectMode,
    ref System.Array custom)
{
}

/// <summary>
///   Implements the OnAddInsUpdate method of the
///   IDTExtensibility2 interface.
///   Receives notification that the collection of add-ins
///   has changed.
/// </summary>
/// <param term='custom'>
///   Array of parameters that arefcost-application specific.
/// </param>
public void OnAddInsUpdate(ref System.Array custom)
{
}

/// <summary>
///   Implements the OnStartupComplete method of the
///   IDTExtensibility2 interface.
```

```
        ///   Receives notification that the host application has
        ///    completed loading,
        /// </summary>
        /// <param term='custom'>
        ///  Array of parameters that are host-application specific.
        /// </param>
        public void OnStartupComplete(ref System.Array custom)
        {
        }

        /// <summary>
        ///   Implements the OnBeginShutdown method of the
        ///   IDTExtensibility2 interface,
        ///   Receives notification that the host application is
        ///   being unloaded.
        /// </summary>
        /// <param term='custom'>
        ///   Array of parameters that are host-application specific.
        /// </param>
        public void OnBeginShutdown(ref System.Array custom)
        {
        }

        /// <summary>
        ///   Implements the QueryStatus method of the
        ///    IDTCommandTarget interface.
        ///     This is called when the command's availability is updated.
        /// </summary>
        /// <param term='commandName'>
        ///  The name of the command to determine state for.
        /// </param>
        /// <param term='neededText'>
        ///   Text that is needed for the command.
        /// </param>
        /// <param term='status'>
        ///   The state of the command in the user interface.
        /// </param>
        /// <param term='commandText'>
        ///   Text requested by the neededText parameter,
        /// </param>
        public void QueryStatus(string commandName,
            EnvDTE.vsCommandStatusTextWanted neededText,
            ref EnvDTE.vsCommandStatus status, ref object commandText)
        {
            if (neededText ==
EnvDTE.vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
            {
                if (commandName == "MyAddin1.Connect.MyAddin1")
                {
                    status =
```

*см. след. стр.*

```
(vsCommandStatus)vsCommandStatus.vsCommandStatusSupported |
vsCommandStatus.vsCommandStatusEnabled;
                }
            }
        }

        /// <summary>
        ///     Implements the Exec method of the IDTCommandTarget Interface.
        ///   This is called when the command is invoked.
        /// </summary>
        /// <param term='commandName'>
        ///   The name of the command to execute.                    :
        /// </param>
        /// <param term='executeOption'>
        ///   Describes how the command should be run.
        /// </param>
        /// <param term='varIn'>
        ///     Parameters passed from the caller to the command handler.
        /// </param>
        /// <param term='varOut'>
        ///     Parameters passed from the command handler to the caller.
        /// </param>
        /// <param term='handled'>
        ///   Informs the caller whether the command was handled.
        /// </param>
        public void Exec(string commandName,
            EnvDTE.vsCommandExecOption executeOption, ref object varIn  ,
            ref object varOut, ref bool handled)
        {
            handled = false;
            if (executeOption ==
                EnvDTE.vsCommandExecOption.vsCommandExecOptionDoDefaul  t)
            {
                if (commandName ==  "MyAddin1.Connect.MyAddin1")
                {
                    handled = true;
                    return;
                }
            }
        }
        private _DTE applicationObject;
        private AddIn addInInstance;
    }
}
```

В начале листинга несколько операторов *using*, сгенерированных мастером для удобства программиста. (Беглый взгляд на Connect.cs показывает, что Add-in Wizard избегает операторов *using* в пользу полностью квалифицированных типов, это очень удобно, когда нужно разобраться, откуда берутся типы надстройки.) Два наиболее важных пространства имен в операторах *using: EnvDTE* и *Extensibility*;

первое определяет типы, применяемые в *IDTExtensibility2*, а второе — типы объектной модели автоматизации. (Третье по важности — пространство имен *Microsoft.Office.Core*, в котором определяются типы для работы с панелями команд в IDE.)

Первый метод в листинге, *OnConnection*, претендует на приз «Самый важный метод надстройки». Visual Studio .NET вызывает его при загрузке надстройки, передавая ей ссылку на корневой объект объектной модели автоматизации в параметре *application*. В коде, сгенерированном Add-in Wizard, параметр *application* приводится к типу *EnvDTE._DTE*, и результат сохраняется в закрытой переменной *applicationObject*. Все дальнейшее взаимодействие надстройки и объектной модели автоматизации осуществляется через переменную *applicationObject*.

Visual Studio .NET также передает надстройке в параметре *addInInst* ссылку на соответствующий ей объект *AddIn*; надстройка хранит эту ссылку в закрытой переменной *addInInstance*.

Оставшаяся часть кода *OnConnection* создает команду в меню Tools. (Этого кода не будет, если вы не указали пользовательский интерфейс в Add-in Wizard.) Код создания меню выполняется в зависимости от оператора *if*:

```
if (connectMode == Extensibility.ext_ConnectMode.ext_cm_UISetup)
```

Параметр *connectMode* содержит значение, описывающее, как была загружена надстройка. Для надстроек, создающих команду в меню, Visual Studio .NET передает значение *Extensibility.ext_ConnectMode.ext_cm_UISetup* при первой загрузке надстройки после ее установки. Это говорит надстройке, что сейчас подходящий момент добавить свои команды в IDE.

Add-in Wizard не генерирует код в теле других четырех методов интерфейса *IDTExtensibility2*: *OnStartupComplete*, *OnAddinsUpdate*, *OnBeginShutdown* и *OnDisconnection*. Два метода интерфейса *IDTCommandTarget*, *QueryStatus* и *Exec*, содержат шаблон для кода, управляющего командой меню надстройки и обрабатывающего щелчки меню. Для обработки вызова команды из меню нужно добавить код в метод *Exec* во втором операторе *if*.

```
if (commandName == "MyAddin1.Connect.MyAddin1")
```

В Connect.cs будет не так уж много кода, даже если вы выберете все параметры в Add-in Wizard, но этот код создает полнофункциональную надстройку,

## Установка и загрузка надстройки

Простейший (и лучший) способ установки новой надстройки — построить проект установки надстройки и установить созданный им MSI-файл. Этот файл является готовым пакетом, который можно развернуть на любой Windows-машине с установленной .NET Framework. (При желании можно распространять .NET Framework вместе с надстройкой; см. об этом главу 13.) Установка такого дистрибутива, состоящего из одного пакета, сводится к двойному щелчку значка MSI-файла в Windows Explorer, Надстройку можно установить и из Visual Studio .NET. Построив проект установки, выберите команду Install в контекстном меню проекта (рис. 5-2). Мастер установки проведет вас через все стадии процесса установки.

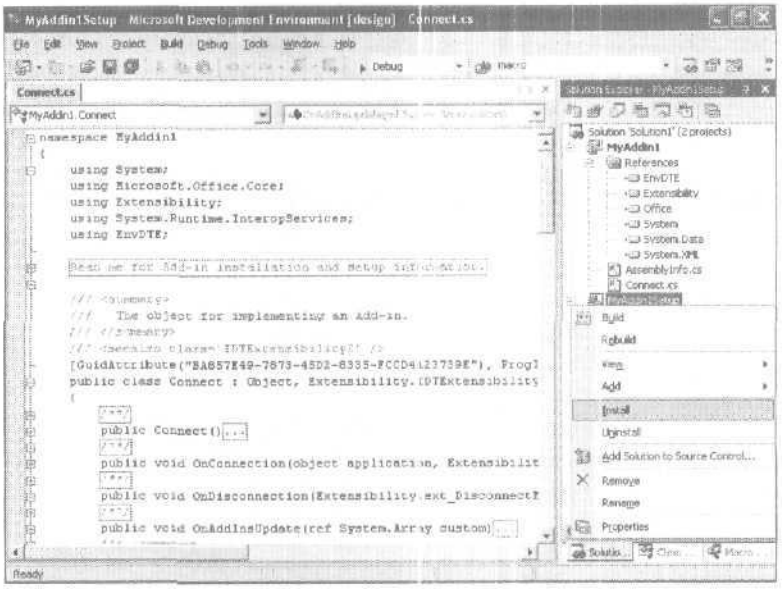


**Рис. 5-2.** *Установка надстройки из Visual Studio .NET*

Способ загрузки вашей надстройки в Visual Studio .NET зависит от некоторых параметров, которые вы выбрали и Add-in Wizard. Если вы выбрали загрузку надстройки при запуске, Visual Studio .NET будет загружать ее автоматически в начале работы. Если вы выбрали пользовательский интерфейс для своей надстройки. при каждом запуске Visual Studio .NET вы сможете загружать надстройку, выбирая соответствующую ей команду в меню Tools (рис. 5-3).

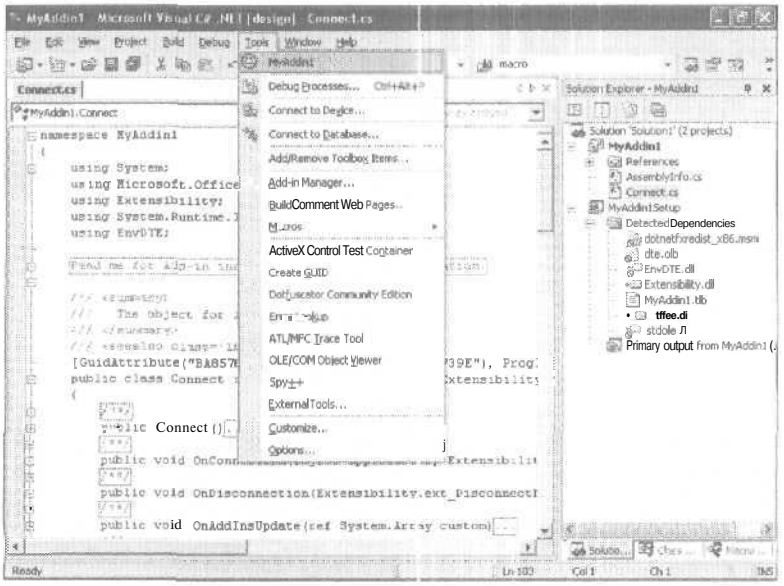


**Рис. 5-3.** *Команда для надстройки в меню*

Если вы не выбрали ни один из этих параметров, надстройку можно загрузить, щелкнув пункт меню Tools|Add-in Manager, который вызывает диспетчер надстроек (Add-in Manager) (рис. 5-4). Add-in Manager позволяет управлять всеми зарегистрированными надстройками: загружать их, выгружать, помечать как загружаемые при запуске и при компоновке из командной строки.

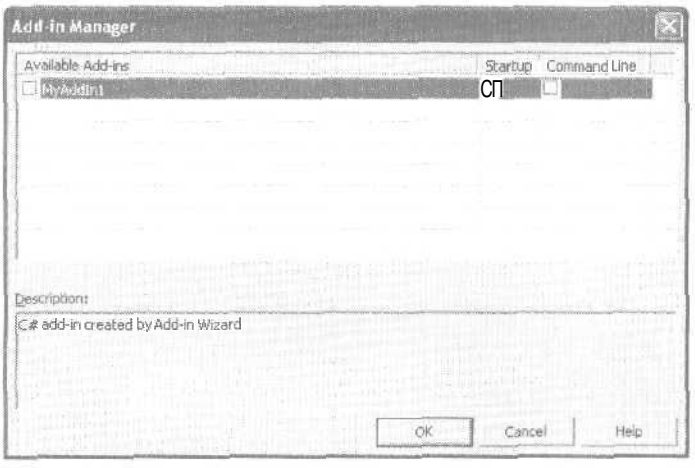



**Рис. 5-4.** *Диалоговое окно Add-in Manager*

## Отладка надстройки

Надстройка — это DLL, так что отладка проекта надстройки не отличается от отладки любого другого проекта Class Library. DLL не может работать сама по себе, ей нужно хост-приложение; для надстройки таким хостом может быть Visual Studio .NET (devenv.exe) или Macros IDE (vsaenv.exe). Add-in Wizard устанавливает свойства отладки проекта надстройки так, что хостом является Visual Studio .NET. Вы можете изменить свойства отладки проекта, щелкнув правой кнопкой проект надстройки в Solution Explorer, выбрав из контекстного меню Properties, а затем -- Configuration Properties | Debugging в диалоговом окне Property Pages (рис. 5-5), Однако обычно вполне подходят значения по умолчанию.

В стандартном сеансе отладки вы открываете проект надстройки в Visual Studio .NET, устанавливаете точки прерываний в исходном коде, а затем запускаете отладчик, выбрав Start в меню Debug (или нажав F5). Отладчик запускает второй экземпляр Visual Studio .NET и подсоединяет себя к этому новому процессу. Вы загружаете отлаживаемую надстройку в этом втором экземпляре Visual Studio .NET, и, когда выполнение надстройки доходит до точки прерывания, выполнение передается отладчику, работающему в первом экземпляре Visual Studio .NET. Теперь можно осуществлять пошаговое выполнение, просматривать содержимое переменных и регистров и т. п.

Отладка надстроек в Macros IDE не сложней их отладки в Visual Studio .NET. Единственная загвоздка в том, что БЫ не можете открыть Macros IDE из экземпляра Visual Studio .NET и затем присоединить отладчик из этого экземпляра к Macros IDE. Дело в том, что эти два процесса будут взаимоблокироваться, если Visual Studio

.NET вызовет событие макроса, когда выполнение остановлено в отладчике. Рекомендуемый способ отладки надстроек в Macros IDE аналогичен отладке надстроек в Visual Studio .NET: откройте проект надстройки в Visual Studio .NET, запустите второй экземпляр Visual Studio .NET, откройте Macros IDE во втором экземпляре Visual Studio .NET, присоедините отладчик из первого экземпляра Visual Studio .NET к процессу Macros IDE, загрузите надстройку в Macros IDE и приступайте к обычной отладке. (Облегченно вздохнув!)

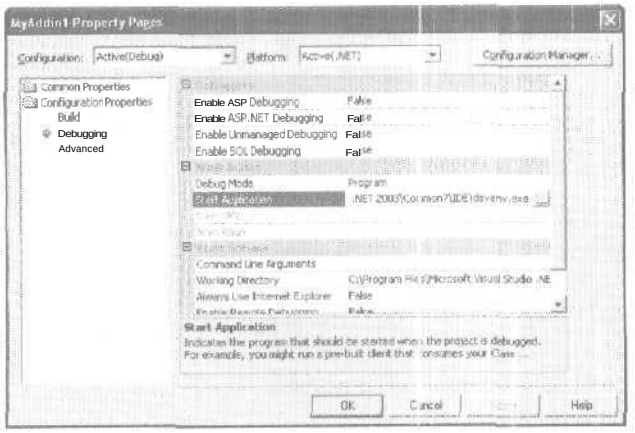



Рис. 5-5. *Свойства отладки проекта надстройки*

И последнее: никто не заставляет вас работать с отладчиком Visual Studio .NET, если у вас есть другой любимый отладчик. Любой отладчик, способный работать с DLL, загружаемыми с задержкой. — к вашим услугам (например, Microsoft CLR Debugger, поставляемый с .NET Framework SDK, — рис. 5- 6).

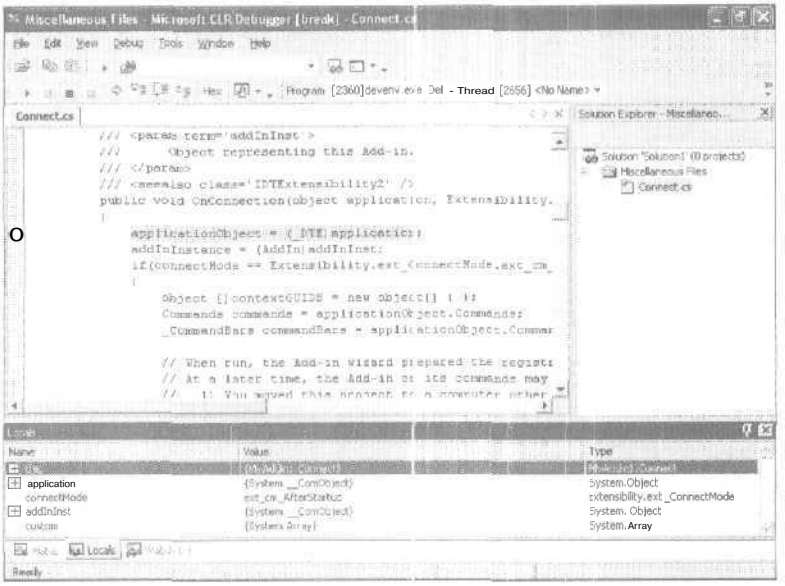



Рис. 5-6. *Отладка надстройки* с *помощью CLR Debugger*

# Объектная модель автоматизации

Одного взгляда на схему объектной модели автоматизации достаточно, чтобы согласиться с тем, что разработчики Visual Studio .NET позаботились о настройке среды программирования. Более НО объектов модели автоматизации дают полный контроль над IDE: от решений до отдельных параметров любой функции.

## Объекты автоматизации

Как следует из ее названия, объектная модель автоматизации предоставляет свою функциональность через объекты — такая парадигма знакома большинству современных программистов. Каждый объект инкапсулирует небольшую часть Visual Studio .NET и предоставляет к ней программный доступ через совокупность методов. свойств и событий. Объекты организованы в иерархию, в которой методы и свойства одного объекта имеют доступ к дочерним объектам внизу иерархии и родительским вверху.

На вершине этой объектной модели — объект *DTE.* Надстройки получают ссылку на *DTE* через параметр *application* события *OnConnection*; макросы используют ссылку на *DTE* в предопределенной глобальной переменной с именем *DTE.* Имея ссылку на *DTE,* вы можете добраться до любого объекта в иерархии. Следующий короткий список поможет вам составить общее представление о доступных объектах,

- Объекты *Solution, Project* и *ProjectItem* дают вам программный доступ ко всем аспектам управления проектами.
- Объекты *Document* и *TextDocument* позволяют работать с документами в IDE.
- Объекты *TextSelection* и *TextPoint* позволяют редактировать текст в окне редактора; объект *EditPoint* ~ вносить изменения непосредственно в текстовый буфер.
- Объекты *FileCodeModel, CodeNamespace, CodeInterface,CodeVariable, CodeFunction* и аналогичные им позволяют манипулировать конструкциями кода, независимо от языка программирования.
- Объекты *CommandWindow, TaskList, OutputWindow, ToolBox* и другие подобные объекты предоставляют контроль над отдельными окнами инструментов в fDE.
- Объекты *Command* предоставляют доступ к командам среды.
- Объекты *Debugger, Process, Program, Thread, StackFrame* и другие связанные с ними позволяют управлять отладчиком Visual Studio .NET.

В оставшейся части книги эти и другие объекты описываются подробнее

## Принципы объектной модели

Везде, где это можно, объектная модель автоматизации следует нескольким простым принципам, перечисленным ниже. (Не забывайте об этих правилах, и вы не будете тратить время, агонизируя от вопроса, почему *AddIns(0)* выдает *COMException* или почему *Activate* не выводит ваше сообщение @#&%*!.)

- Все объекты используют только типы автоматизации.
- Все объекты имеют свойство *DTE,* возвращающее объект *DTE.*
- Имена наборов имеют форму множественного числа от имени составляющих их объектов. Например, *Documents* — набор объектов *Document*

- Наборы доступны через свойства, имеющие имена самих наборов.
- У наборов есть метод (не свойство) *Item* для доступа к их элементам.
- В управляемых языках метод *Item* набора принимает объект, а в неуправляемых языках — *VARIANT*.
- Метод *Item* набора принимает числовой индекс или имя элемента набора.
- Числовые индексы наборов начинаются с 1.
- У наборов есть метод *Count,* который в управляемых языках возвращает *SystemInt32*, а в неуправляемых — *long*.
- У элементов наборов есть свойство *Collection,* возвращающее набор-владелец; все остальные объекты имеют свойство *Parent,* возвращающее родительский объект.
- Элемент добавляется в набор методом *Add* набора; элемент удаляется из набора методами *Remove* или *Delete* элемента. *(Delete* применяется к элементам, связанным с пользовательским интерфейсом, таким как объекты *ToolBoxTab*; *Remove* применяется к элементам не связанным с пользовательским интерфейсом, таким как объекты *AddIn*.)
- Объект *DTEEvents* предоставляет доступ ко всем интерфейсам событий.
- Общий объект может вернуть один или несколько конкретных объектов через свойство *Object,* которое принимает независимое от регистра имя конкретного объекта.
- Создание окна не приводит автоматически к его отображению; чтобы сделать окно видимым, нужно установить его свойство *Visible* в *True*.
- Свойства, возвращающие полный путь к дисковому файлу, называются *FullName*.
- Метод *Activate* устанавливает фокус на видимом элементе пользовательского интерфейса; если элемент невидим, *Activate* не всегда делает видимым его контейнер.

## События автоматизации

Объекты автоматизации говорят *о* том, *что* можно делать с Visual Studio .NET, а события автоматизации — *когда* можно выполнять те или иные действия. В каждой функциональной группе объектной модели автоматизации определены события, позволяющие отслеживать активность Visual Studio .NET и выполнять соответствующие действия. События автоматизации вместе с объектами автоматизации дают контроль над всеми аспектами Visual Studio .NET.

### Подключение к событиям автоматизации

Подключиться к событиям автоматизации довольно просто, но синтаксис подключения во всех языках, поддерживаемых Visual Studio .NET, различен. Мы начнем с C#, так как его синтаксис демонстрирует все возможности механизма событий.

*DTEEvents* ~ первый шаг на пути изучения событий автоматизации. Это свойство возвращает интерфейс *EnvDTEEvents*, который определяет неизменяемые свойства (табл. 5-2). Каждое из свойств *Events* возвращает интерфейс события с тем же именем, что и у свойства, а каждый интерфейс события определяет набор событий, на которые можно подписаться.

**Табл. 5-2. Свойства *EnvDTE.Events***

| Свойство | События |
|---|---|
| *BuildEvents* | *OnBuildBegin*<br>*OnBuildDone*<br>*OnBuildProjConfigBegin*<br>*OnBuildProjConfigDone* |
| *CommandEvents** | *AfterExecute*<br>*BeforeExecute* |
| *DebuggerEvents* | *OnContextChanged*<br>*OnEnterBreakMode*<br>*OnEnterDesignMode*<br>*OnEnterRunMode*<br>*OnExceptionNotHandled*<br>*OnExceptionThrown* |
| *DocumentEvents** | *DocumentClosing*<br>*DocumentOpened*<br>*DocumentOpening*<br>*DocumentSaved* |
| *DTEEvents* | *ModeChanged*<br>*OnBeginShutdown*<br>*OnMacrosRuntimeReset*<br>*OnStartupComplete* |
| *FindEvents* | *FindDone* |
| *MiscFilesEvents (возвращает ProjectItemsEvents)* | *ItemAdded*<br>*ItemRemoved*<br>*ItemRenamed* |
| *OutputWindowEvents** | *PaneAdded*<br>*PaneClearing*<br>*PaneUpdated* |
| *SelectionEvents* | *OnChange* |
| *SolutionEvents* | *AfterClosing*<br>*BeforeClosing*<br>*Opened*<br>*ProjectAdded*<br>*ProjectRemoved*<br>*ProjectRenamed*<br>*QueryCloseSolution*<br>*Renamed* |
| *SolutionItemsEvents* (возвращает *ProjectItemsEvents*) | *ItemAdded*<br>*ItemRemoved*<br>*ItemRenamed* |
| *TaskListEvents** | *TaskAdded*<br>*TaskModified*<br>*TaskNavigated*<br>*TaskRemoved* |
| *TextEditorEvents** | *LineChanged* |
| *WindowEvents** | *WindowActivated*<br>*WindowClosing*<br>*WindowCreated*<br>*WindowMoved* |

* *C#* не позволяет ссылаться на эти свойства, используя стандартный синтаксис свойств, поскольку их аксессор *get* принимает параметры. Нужно явно обращаться к аксессорам *get*.

Прежде чем подписаться на любое из событий из табл. 5-2, надо определить функцию, которая будет обрабатывать событие, а для этого нужна сигнатура события. В документации Visual Studio .NET описаны сигнатуры всех событий автоматизации; имея эту информацию вы можете определить функцию обработки события, прототип которой будет соответствовать нужному событию.

Затем вам потребуется делегат, определенный для события. В модели событий .NET Framework делегат — это класс-оболочка функции обратного вызова (например. обработчика события), обеспечивающая безопасность типов. Делегаты событий автоматизации имеют имена. соответствующие шаблону *_disp<интерфейс события>_<имя события>EventHandler*. Чтобы добавить свой обработчик события к списку подписчиков события, вы создаете новый экземпляр делегата события, передавая конструктору делегата обработчик события, а затем назначаете делегат событию, используя синтаксис +=. Сделать это проще, чем может поначалу показаться. Рассмотрим пример.

Допустим, компания, в которой вы работаете, протоколирует свои ежедневные компоновки, и вам нужно добавить в эти протоколы специальную информацию, скажем, название отдела и фамилию сотрудника, производившего компоновку. Один из способов — написать надстройку, которая реагирует на событие *BuildEvents.OnBuildBegin* (оно возникает перед компоновкой) и записывать специальную информацию в журнал из обработчика этого события. В документации представлена такая сигнатура этого события:

```
void OnBuildBegin(vsBuildScope scope, vsBuildAction action);
```

При возникновении этого события Visual Studio .NET передает информацию о компоновке через параметры *scope* и *action*: *scope* говорит о том, что будет компоноваться (решение, пакет или проект), а *action* — о виде компоновки (компоновка, полная перекомпоновка, очистка или развертывание). Создаваемый вами обработчик события должен иметь такую же сигнатуру- Назовите его, как хотите. но неплохо упомянуть название события:

```
public class Connect : Object, Extensibility.IDTExtensibility2
{
    ⋮
    private void OnBuildBegin(vsBuildScope scope. vsBuildAction action)
    {
        // Занести в журнал информацию о компоновке
        ⋮
    }
}
```

Создав обработчик события, нужно связать его с событием. Из табл. 5-2 вы знаете, что события компоновки относятся к интерфейсу *BuildEvents*, так что делегат для события *OnBeginBuild* называется *_dispBuildEvents_OnBeginBuildEventHandler*. Следующий код привязывает обработчик события:

```
public class Connect : Object, Extensibility.IDTExtensibility2
I
    ⋮
    public void OnConnection(object application,
```

```
        Extensibility.ext_ConnectMode connectMode,
        object addInInst,
        ref System.Array custom)
    {
        applicationObject = (_DTE)application;
        addInInstance = (AddIn)addInInst;

        buildEvents = applicationObject.Events.BuildEvents;
        buildEvents.OnBuildBegin +=
            new _dispBuildEvents_OnBuildBeginEventHandler(
                this.OnBuildBegin);
        :
    }
    :
    public void OnDisconnection(
        Extensibility.ext_DisconnectMode disconnectMode,
        ref System.Array custom)
    {
        buildEvents.OnBuildBegin -=
            new _dispBuildEvents_OnBuildBeginEventHandler(
                this.OnBuildBegin);
        :
    }
    :
    private void OnBuildBegin(vsBuildScope scope, vsBuildAction action )
    {
        // Заносим информацию о компоновке в журнал.
        :
    }
    :
    private _DTE applicationObject;
    private AddIn addInInstance;
    private EnvDTE.BuildEvents buildEvents;
    :
}
```

Принцип прост: *OnConnection* привязывает обработчик события *OnBuildBegin* к событию при загрузке надстройки, а *OnDisconnection* отключает обработчик от события при выгрузке надстройки.

**Важно!** Если вы не отмените подписку на все события до выгрузки надстройки, то получите старую добрую утечку памяти. В нормальных ситуация надстройки становятся жертвами сборщика мусора, когда Visual Studio .NET их выгружает. Но когда надстройка не отменила подписку на событие, она не освобождается сборщиком мусора, так как событие по-прежнему удерживает ссылку на нее. И, подобно привидениям из фильма «Шестое чувство», которые «даже не знали, что мертвы», надстройка ведет себя как ни в чем не бывало и продолжает обрабатывать события.

### Обработчики событий макросов

Подключаться к событиям в надстройках просто, а в макросах — еще проще. Каждый новый проект макроса начинает свою жизнь с модуля *EnvironmentEvents,* в котором определены переменные событий для всех интерфейсов событий, перечисленных в табл. 5-2, а инициализируются эти переменные автоматически средой Macros IDE. Когда модуль *EnvironmentEvents* открыт в окне редактора, вы можете добавить новый обработчик события, выбрав переменную события в выпадающем списке Class Name, а затем выбрать событие, к которому хотите подключиться, в выпадающем списке Method Name (рис. 5-7). Macros IDE генерирует пустой обработчик события — вам остается лишь добавить код.

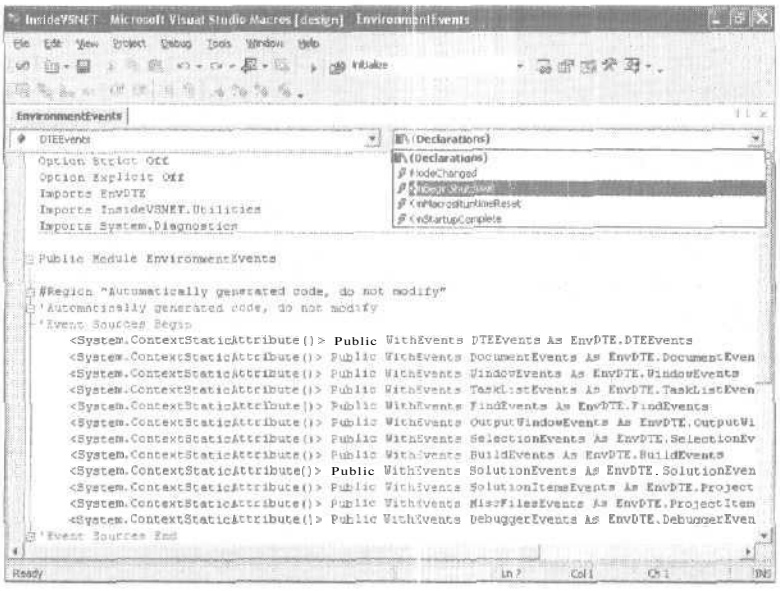



**Рис.** 5-7. *Добавление обработчика событий макроса*

Чтобы отменить подписку на событие, вы просто удаляете обработчик события из модуля *EnvironmentEvents.* Не изменяйте и не удаляйте переменные событий из области, помеченной «Automatically generated code, do not modify* (автоматически сгенерированный код; не изменять), так как после этого проект макроса может не скомпоноваться.

#### Определение собственных переменных событий

Хотя модуль *EnvironmentEvents* — удобное место для создания и управления обработчиками событий макросов, у вас могут быть причины определить некоторые обработчики событий в других модулях (скажем, чтобы не позволить слишком разрастись *EnvironmentEvents).* Чтобы создать обработчик событий в другом модуле, прежде всего в этом модуле: нужно объявить переменную события. Следующее объявление создает переменную *MyWindowEvents* для обработки событий окна:

```
<System.ContextStaticAttribute()> Public WithEvents _
    MyWindowEvents As EnvDTE.WindowEvents
```

Открыв модуль, содержащий это объявление, в окне редактора, вы сможете сгенерировать пустой обработчик события так же, как в модуле *EnvironmentEvents*, выбрав переменную *MyWindowEvents* в выпадающем списке Class Name и нужное событие — в выпадающем списке Method Name. Однако в отличие от модуля *EnvironmentEvents* новый обработчик события не будет получать события автоматически. Дело в том, что Macros IDE инициализирует переменные событий только в *EnvironmentEvents*, а когда вы создали собственную переменную события, вам придется инициализировать ее самостоятельно.

Для инициализации переменной события ей назначается свойство *DTE.Events*:

```
MyWindowEvents = DTE.Events.WindowEvents
```

Вы можете поместить это назначение в некоторый стандартный макрос и запускать его вручную. В некоторых ситуациях это приемлемо, но ваш обработчик пропустит все события, возникшие до запуска этого макроса. Если обработчик должен обрабатывать все события с момента запуска среды, нужно как-то автоматизировать назначение. Интерфейс *DTEEvents* определяет два события, которые вам для этого пригодятся: *OnStartupComplete*, которое возникает, когда завершается загрузка Visual Studio .NET, и *OnMacrosRuntimeReset*, которое возникает, когда проект макроса повторно загружается в память. Создав обработчики для обоих этих событий в *EnvironmentEvents* и выполнив назначение в каждом обработчике, вы получите автоматическую инициализацию своих переменных событий:

```
Public Module EnvironmentEvents
    :
    Public Sub DTEEvents_OnMacrosRuntimeReset() _
            Handles DTEEvents.OnMacrosRuntimeReset
        MyModule.MyWindowEvents = DTE.Events.WindowEvents
    End Sub

    Public Sub DTEEvents_OnStartupComplete() _
            Handles DTEEvents.OnStartupComplete
        MyModule.MyWindowEvents = DTE.Events.WindowEvents
    End Sub
    :
End Module
```

### Фильтрованные события

Вообще-то надо внимательно подходить к обработке событий, так как большое число обработчиков снижает производительность Visual Studio .NET. Однако фильтрованные события позволяют избежать этого ограничения. Фильтрованные события позволяют обрабатывать события, поступающие от одних объектов, и игнорировать — от других. Так, вместо того чтобы получать события *WindowClosing* для каждого окна, вы можете выбрать получение этих событий только от окна Task List. Свойства *DTEEvents* (табл. 5-3) позволяют создавать фильтрованные события.

**Табл. 5-3. Свойства фильтрованных событий**

| Свойство *DTE.Events* | Фильтрация по |
|---|---|
| *CommandEvents* | имени команды |
| *DocumentEvents* | объекту *Document* |
| *OutputWindowEvents* | имени области окна вывода |
| *TaskListEvents* | категории в списке задач |
| *TextEditorEvents* | объекту *TextDocument* |
| *WindowEvents* | объекту *Window* |

Допустим, вы хотите обрабатывать события от конкретного окна в IDE. Из табл. 5-3 видно, что свойство *WindowEvents* принимает в качестве фильтра объект *Window*. Вы получаете нужный объект *Window* из набора *EnvDTE.Windows*, передавая его методу *Item* соответствующую константу *EnvDTE.Constants.vsWindowKindxxx*. Передайте объект *Window* свойству *WindowEvents*, и вы получите интерфейс события для этого конкретного окна. Следующий код демонстрирует, как ограничить события окон лишь окном Macro Explorer:

```
' Объявление переменной события.
<System.ContextStaticAttribute()> Public WithEvents _
    MacroExplorerWindowEvents As EnvDTE.WindowEvents

Sub InitializeMacroExplorerFilter()
    Dim macroExplorerWindow As EnvDTE.Window

    macroExplorerWindow = _
        DTE.Windows.Item(EnvDTE.Constants.vsWindowKindMacroExplorer)
    MacroExplorerWindowEvents = _
        DTE.Events.WindowEvents(macroExplorerWindow)
End Sub

Sub MacroExplorerWindowEvents_WindowActivated( _
        ByVal GotFocus As EnvDTE.Window, _
        ByVal LostFocus As EnvDTE.Window) _
        Handles MacroExplorerWindowEvents.WindowActivated
    ' Описание: Выполняется при каждой активизации
    '           окна Macro Explorer.
    :
End Sub
```

### Практикум: нефильтрованные и фильтрованные события

Хотите увидеть разницу между нефильтрованными и фильтрованными событиями? Тогда откройте окно Output и проведите эксперимент, в котором используется макрос в модуле *FilteredEvents*.

1. Запустите макрос *InitializeUnfilteredEvents*, инициализирующий переменные событий *DocumentEvents* и *TextEditorEvents* без фильтров. *FilteredEvents* определяет обработчики событий, которые обрабатывают события *DocumentSaved* и *LineChanged* для этих двух переменных событий.

2. Создайте два новых текстовых файла,
3. В обоих созданных файлах введите строку текста и нажмите Enter. В окне Output для каждой новой строки будет выведено сообщение «LineChanged fired» (Произошло событие LineChanged).
4. Сохраните оба файла, В окне Output появится сообщение «DocumentSaved fired* (Произошло событие DocumentSaved ) для каждого сохраненного файла. -

Поскольку переменные событий *DocumentEvents* и *TextEditorEvents* были инициализированы без фильтров, соответствующие обработчики событий получают события для всех документов. Теперь оставьте открытыми оби созданных файла и попробуйте аналогичный эксперимент с применением фильтрованных событий,

1. Запустите макрос *InitializeFilteredEvents*.Он создает файл New File и инициализирует переменные событий *DocumentSaved* и *LineChanged* так, что они относятся только к событиям этого файла,
2. В файле New File введите строку текста и нажмите Enter. В окне Output появится сообщение «LineChanged fired*.
3. Переключайтесь на каждый из созданных вами текстовых файлов и повторяйте предыдущий шаг. Заметьте: в окне Output **ни** для одного из этих файлов сообщений не появляется.
4. Сохраните оба текстовых файла, И опять никаких сообщений в окне Output.
5. Сохраните New File. В этот раз в окне Output появится сообщение «DocumentSaved fired».

### События позднего связывания

Языки программирования, интегрированные в Visual Studio .NET, предлагают события позднего связывания, которые отслеживают состояние собственных типов проектов; их можно рассматривать как фильтрованные события для проектов. Эти привязанные к проектам события перечислены в табл. 5-4.

**Табл. 5-4. События позднего связывания**

| Язык | Интерфейс события |
|---|---|
| Visual C# | *CSharpBuildManagerEvents*<br>*CSharpProjectsEvents*<br>*CSharpProjectItemsEvents*<br>*CSharpReferencesEvents* |
| Visual Basic | *VBBuildManagerEvents*<br>*VBImportsEvents*<br>*VBProjectsEvents*<br>*VBProjectItemsEvents*<br>*VBReferencesEvents* |
| Visual C++ | *CodeModelEvents*<br>*VCProjectEngineEventsObject* |

Имена интерфейсов событий совпадают с именами интерфейсов, от которых они порождены, но с префиксом, специфичным для языка: *VBProjectsEvents*, *CSharpProjectItemsEvents* и т. д. (Интерфейсы событий Visual C++ — исключение, так как порождаются от другой базы.) Вы получаете нужный интерфейс события, передавая его имя методу *DTE.Events.GetObject*. Скажем, в следующем коде инициализируется переменная события с интерфейсом, который обрабатывает все события C#-проекта:

```
<System.ContextStaticAttribute()> Public WithEvents _
    CSharpProjectsEvents As EnvDTE.ProjectsEvents

Sub InitializeCSharpProjectsEvents()
    CSharpProjectsEvents = DTE.Events.GetObject("CSharpProjectsEvents")
End Sub
```

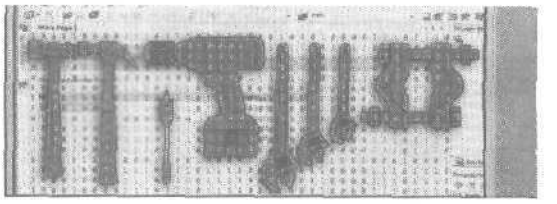
Когда код выполнится, все обработчики событий, созданные для переменной *CsharpProjectsEvents*, будут получать события только C#-проекта.

## Что дальше

Наша воскресная прогулка закончена. Мы обследовали сельскую местность Add-in Wizard и проехались по берегу объектной модели автоматизации — пора и за работу. Ставьте свою надстройку в гараж, поднимайте капот и откручивайте гайки. В следующей главе мы разберем надстройку по винтику, а затем соберем снова.

ГЛАВА

# 6



# Архитектура надстроек

В главе 5 показано, что самый простой способ создать надстройку — это запустить мастер Add-in Wizard из состава Microsoft Visual Studio .NET. Давно известно, что простые пути далеко не всегда являются лучшими — это справедливо и тогда, когда предстоит освоить незнакомую технологию. Давайте исходить из того, что хорошие *мастера* — это не *костыли*, без которых не обойтись, а лишь удобный инструментарий, который следует использовать, зная при этом, что и сам можешь написать эквивалентный код. Мы, конечно, тоже постараемся научить вас всему, что надо для написания надстроек, подобных создаваемых мастером. К концу главы, если постараетесь, то создадите и собственный мастер вроде Add-in Wizard.

Не верите? Войдите в IDE, откройте Command Window и... начнем. В следующем разделе вы изучите основы конструирования надстроек и начнете писать их код в старомодной манере — своими руками и с нуля.

## Создание надстройки с нуля

В листинге 6-1 показан исходный код нашей первой надстройки — Basic. Можно рассматривать ее как сгенерированную мастером надстройку, с которой сдраны все покровы. Размеры такой минималистской надстройки невелики. Связано это с тем, что к ней предъявляется единственное требование: она должна представлять открытый класс, наследующий и реализующий интерфейс *ExtensibilityIDTExtensibility2*.Файл Basic.cs, написанный на языке C#, удовлетворяет этому требованию, определяя класс *Basic,* наследующий интерфейс *IDTExtensibility2*и реализующий 5 методов этого интерфейса: *OnConnection*, *OnStartupComplete*,*OnAddInsUpdate*,*OnBeginShutdown* и *OnDisconnection*. Вы не увидите метода *Main,* поскольку Basic, как и все надстройки, реализуется в форме DLL Точкой входа служит метод *OnConnection*. Надстройка Basic реализует этот метод, отображая собственное имя в информационном окне.

**Листинг 6-1.** Исходный код надстройки Basic

Basic.cs

```
using System;
using System.Windows.Forms;
using Extensibility;

public class Basic : IDTExtensibility2
{
    public void OnConnection(object application,
        ext_ConnectMode connectMode,
        object addInInst,
        ref Array custom)
    {
        MessageBox.Show("Basic Add-in");
    }

    public void OnStartupComplete(ref Array custom)
    {
    }

    public void OnAddInsUpdate(ref Array custom)
    {
    }

    public void OnBeginShutdown(ref Array custom)
    {
    }

    public void OnDisconnection(ext_DisconnectMode removeMode,
        ref Array custom)
    {
    }
}
```

## Компиляция надстройки Basic

Если в текстовый файл вставить исходный код из листинга 6-1 *и* назвать этот файл Basic.cs, то затем можно из командной строки скомпилировать надстройку Basic командой:

```
csc /t:library /r:"c:\program files\microsoft visual studio .net 2003\
common7\ide\publicassemblies\extensibility.dll" basic.cs
```

Флаг */t:library* говорит компилятору C#, что из исходного файла нужно создать DLL (Basic.dll), а *tyii2.r/r;«c:\programjiles\microsofl visual studio net 2003\common7\ide\publicassemblies\extensibility.dll»*указывает компилятору сборку, содержащую пространство имен *Extensibility* (Extensibility.dll). Пространство имен *Extensibility* задает три типа: интерфейс *IDTExtensibility2* а также перечисления *ext_ConnectMode и ext_DisconnectMode*, задающие значения, передаваемые методам *OnConnection* и *OnDisconnection* соответственно.

**Совет** Печатать длинные ссылки в командной строке скучно, не говоря о том, что игра на клавишах способствует развитию профессиональной болезни кистей рук. Альтернативой является добавление ссылки к списку ссылок по умолчанию, находящемуся в глобальном файле CSC.rsp, путь к которому <WinDir>\Microsoft.NET\Framework\<Version>\CSC.rsp. Скажем, добавив в этот файл */r:«c:\program files\microsoft visual studio .net 2003\common7\ide\publicassemblies\extensibility.dll»*, можно скомпилировать надстройку Basic короткой командой;

```
csc /t:library basic.cs
```

## Регистрация надстройки Basic как COM-объекта

К этому моменту надстройка уже существует в виде файла Basic.dll. Точнее, существует управляемая надстройка, определенная в сборке, хранимой в файле Basic.dll. А теперь пора сказать нечто смешное и грустное по поводу надстроек Visual Studio .NET: Visual Studio .NET не может быть хостом управляемой надстройки, даже созданной собственным мастером, без помощи извне. Дело в том, что сама Visual Studio по большей части является неуправляемым приложением. Visual Studio .NET во многом унаследовала свою функциональность от Visual Studio *6,* включая способность быть хостом надстроек. Но, поскольку Visual Studio 6 имеет дело только с COM-надстройками, Visual Studio .NET не может управлять другими их видами.

К счастью, на помощь пришло COM-взаимодействие. Разработчики .NET Framework с самого начала исходили из того, что мир не выбросит на ветер огромные вложения в COM и не начнет все переделывать заново под управляемый код, так что они сделали все, чтобы COM-классы и .NET-классы могли бесшовно взаимодействовать. Все, что нужно сделать .NET-классу, чтобы он мог восприниматься как COM, — это зарегистрировать себя как COM-класс. В частности, управляемая надстройка требует следующих элементов реестра.

- **HKEY_CLASSES_ROOT\<ProgID>** Раздел, чье имя является ProgID класса, реализующего *IDTExtensibility2.*
- **HKEY_CLASSES_ROOT\<ProgID>\CLSID** Раздел, чье значение по умолчанию является CLSID класса, реализующего *IDTExtensibility2.*
- **HKEY_CLASSES_ROOT\CLSID\<CLSID>** Раздел, чье имя является CLSID класса, реализующего *IDTExtensibility2.*
- **HKEY_CLASSES_ROOT\CLSID\InprocServer32** Раздел, чьи именованные значения идентифицируют класс, реализующий *IDTExtensibility2,* и сборку, в которой этот класс определен.

На рис. 6-1 показаны элементы реестра для надстройки Basic.

**Примечание** Visual Studio .NET не единственный участник COM-взаимодействия. С той минуты, как управляемая надстройка корректно зарегистрирована, любая среда, такая как Visual Basic 6, способная быть хостом COM-надстроек, может быть хостом и управляемой надстройки (конечно, если .NET Framework установлена на этой машине).




Рис. 6-1. *COM-взаимодействие: элементы реестра для надстройки Basic*

К счастью, нет нужды добавлять вручную элементы реестра: они создаются автоматически при регистрации надстроек утилитой .NET Framework Assembly Registry (RegAsm). Следующая команда зарегистрирует класс *Basic* как COM-класс:

```
regasm /codebase basic.dll
```

---

Примечание Не обращайте внимания на сообщение, выдаваемое RegAsm в конце работы, о том, что флаг */codebase* должен использоваться только для сборок со строгими именами, Флаг */codebase* генерирует именованное значение *CLSID\InprocServer32\CodeBase*— без него CLR не сможет найти сборку.

---

Программа RegAsm генерирует по умолчанию ProgID и CLSID для каждого класса, входящего в сборку. (См. врезку «GUID и FUID» о задаваемых по умолчанию CLSID.) Значение по умолчанию для ProgID является тем же, что и полностью квалифицированное имя класса. Например, код:

```
namespace Outer
{
    namespace Inner
    {
        public class HyClass ()
        {
        }
    }
}
```

создаст ProgID по умолчанию — *Outer.Inner.MyClass.* Поскольку класс *Basic* не был вложен в пространство имен, его ProgID есть просто *Basic,*

**GUID и FUID**

- Традиционно CLSID представляет собой глобально уникальный идентификатор (GUID) — 16-байтовое число, гарантирующее уникальность во времени и пространстве. Однако при ближайшем рассмотрении значение по умолчанию CLSID, генерируемое программой RegAsm, напоминает GUID только внешне. Фактически же RegAsm всегда генерирует тот же CLSID, когда задается одно и то же полностью квалифицированное имя класса независимо от того, когда это число генерируется. Мы назовем это число FUID — *for-the-most-part* unique *identifier* (частично уникальный идентификатор) — в иерархии уникальности оно следует за GUID.

  Это значит, что, если два программиста в разные моменты времени н:д двух разных машинах на двух разных континентах регистрируют класс, названный *Basic*, их классы будут иметь один и тот же FUID: 992B0D1F-A395-34F5-BFC9-EGA3E9385293.

  Ну, хорошо, вы не можете запретить применение имени класса, так тщательно выбранное вами, но совершенно не обязательно использовать FUID, присвоенный классу программой RegAsm. И здесь приходит на помощь *System.Runtime.InteropServices.GuidAttribute*. Применительно к классу *GuidAttribute* переопределяет данное по умолчанию значение FUID и позволяет присвоить GUID по вашему выбору. (Для выбора GUID много возможностей: запустите утилиту Create GUID [GuidGen], создающую «честный» GUID, позволяющий вызывать ваш собственный класс.)

## Регистрация надстройки Basic в Visual Studio .NET

Basic.dll представляет собой надстройку, полностью готовую к функционированию. но Visual Studio .NET все же не знает о ее существовании. Надстройки уведомляют о своей доступности интегрированную среду разработке (IDE) Visual Studio .NET через один из элементов реестра:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\AddIns\<ProgID>
- HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\7.1\AddIns\<ProgID>

А Macros IDE они становятся известны через один из соответствующих элементов реестра *VSA*:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VSA\7.1\AddIns\<ProgID>
- HKEY_CURRENT_USER\Software\Microsoft\VSA\7.1\AddIns\<ProgID>

**Примечание** Отсюда и далее будем ссылаться только на надстройки в IDE Visual Studio .NET. Однако информация в оставшейся части главы применима в равной степени и к надстройкам в Macros IDE.

Add-in Manager заполняет свой список надстроек, основываясь на этих двух разделах *AddIns*. Элемент HKEY_LOCAL_MACHINE делает надстройку доступной всем пользователям данной машины и, поскольку модификация куста реестра HKEY_LOCAL_MACHINE требует наличия прав администратора, надстройка, зарегист] >иро-

ванная таким способом, известна как *надстройка администратора*. Элемент HKEY_CURRENT_USER делает ее доступной пользователю, создавшему этот элемент, что отражается в параметрах пользователя HKEY_USERS; не удивительно, что зарегистрированная таким образом надстройка известна как *надстройка пользователя*. В обоих случаях именем раздела является ProgID класса, реализующего *ID TExtensibility2*.

Для регистрации надстройки Basic с целью обращения к ней всех пользователей машины запустите редактор реестра (RegEdit) и создайте следующий раздел (предполагается, что у вас права администратора):

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\
AddIns\Basic
```

Затем запустите Visual Studio .NET и выберите Tools j Add-in Manager, чтобы открыть диалоговое окно Add-in Manager (рис. 6-2). Заметьте: в списке Add-in Manager появилась наша надстройка Basic По умолчанию Add-in Manager отображает ProglD, который он находит в разделе *AddIns*. (В разделе «Именованные значения реестра для надстроек» мы рассмотрим именованные значения для раздела реестра ProglD, позволяющие управлять именем и другими свойствами надстроек.)

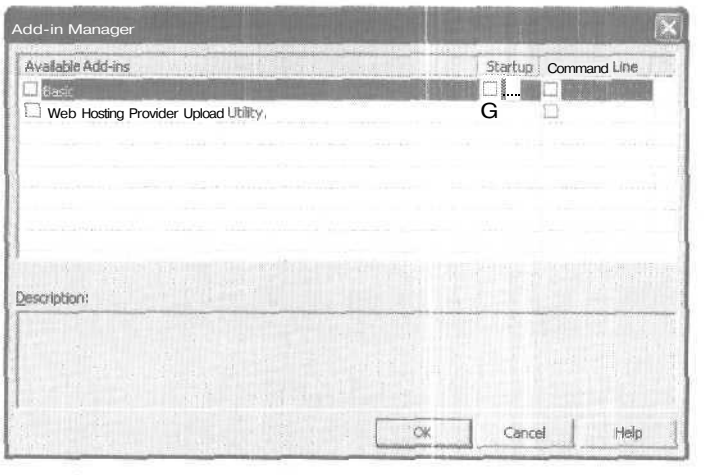



**Рис. 6-2.** *Диалоговое окно Add-in Manager, показывающее надстройку Basic*

Если установить флажок слева от Basic и щелкнуть OK, Visual Studio .NET загрузит надстройку. Если все этапы работы выполнялись должным образом, Visual Studio .NET вызовет метод *OnConnection* надстройки Basic, и вы увидите такое диалоговое окно (рис. 6-3):




**Рис. 6-3.** *Диалоговое окно, отображаемое надстройкой Basic*

Вот вы и создали надстройку с нуля. Далее вы узнаете, что происходит с надстройкой в среде Visual Studio .NET, когда надстройка загружается, когда выгружается и что происходит между этими двумя событиями.

# События надстроек

Надстройками управляют события. Практически все, что делает надстройка, она делает в ответ на внешние воздействия, а подталкивает ее на эти действия Visual Studio .NET через интерфейс *IDTExtensibility2*. Начнем наше исследование событий надстроек с последовательности, в которой Visual Studio .NET вызывает методы *IDTExtensibility2*.

## Последовательность событий надстроек

Вызовы методов *IDTExtensibility2*, которые мы будем также называть событиями, возникают в предсказуемые моменты жизненного цикла надстройки. Рис. *6-4* показывает последовательность событий с момента загрузки надстройки и до момента ее выгрузки.

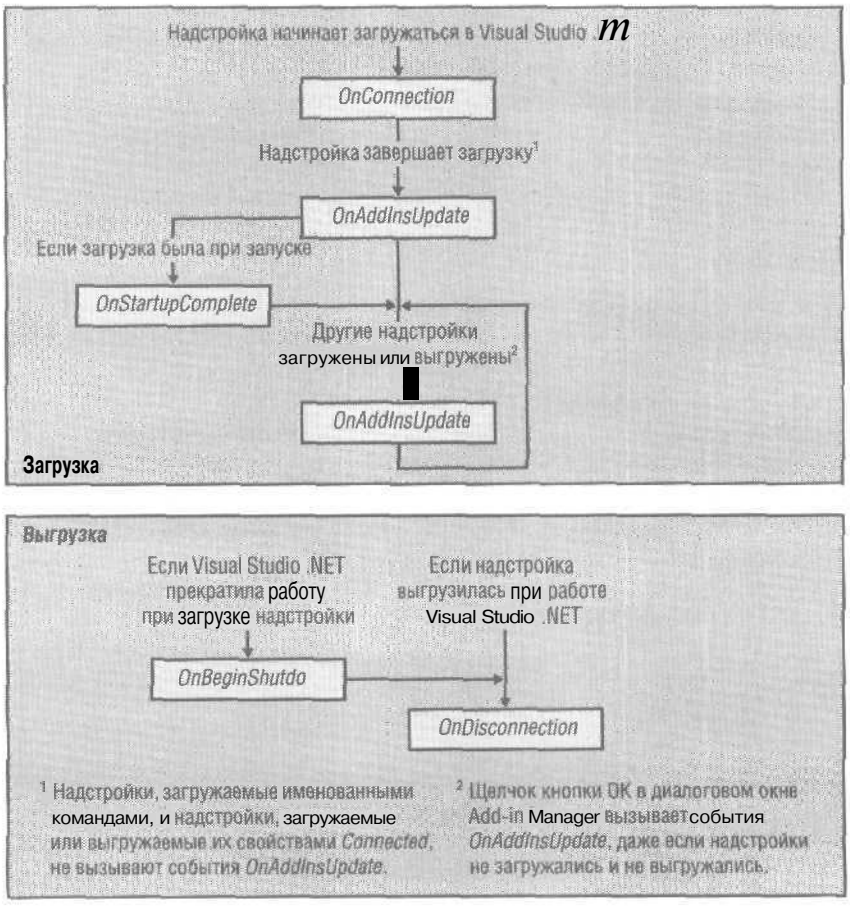



**Рис. 6-4.** *Последовательность событий надстроек*

О действиях, инициируемых событиями, можно догадаться по имени события. Порядок возникновения событий тоже во многом предсказуем: событие *OnConnection* возникает, когда надстройка загружается, *OnDisconnection* — при выгрузке и т. д. Но такая прямолинейность последовательности событий является лишь кажущейся: того, кто верит всему, что написано в документации, ожидает несколько сюрпризов. И первый из них связал с начальным событием *OnAddInsUpdate*. Документация гласит, что надстройка получает уведомление о событии *OnAddInsUpdate*. только когда другие надстройки загружаются и выгружаются, — на самом же деле надстройка сама инициирует первое такое событие. Это просто ошибка документации, от Visual Studio .NET мы вправе ожидать более обдуманных действий в уведомлении надстроек. Ведь надстройка, получив событие *OnConnection*, уже знает, что она загружена: нет необходимости в повторном уведомлении об этом через событие *OnAddInsUpdate*.

Второй сюрприз связан с диспетчером Add-in Manager, который инициирует события *OnAddInsUpdate* всякий pan при нажатии кнопки OK независимо от того, была ли надстройка загружена/выгружена. Причина в том, что нажатие OK автоматически сбрасывает список надстроек, тем самым инициируя последовательность событий *OnAddInsUpdate*. Хорошо ли, нет ли, но это стандартное поведение, и не следует полагать, что любое событие *OnAddInsUpdate*,происшедшее после первого, говорит на самом деле о чем-то важном.

Третья неожиданность: надстройки, загружаемые командами, не инициируют события *OnAddInsUpdate*,а также этого не делают надстройки, загружаемые/выгружаемые их свойствами *Connected*. (О командах см. главу 7.) Разумного объяснения такому поведению нет. так что назовем вещи своими именами — это ошибка.

## Жизненный цикл надстройки

Увидеть всю последовательность событий можно, запустив надстройку-пример LifeCycle. Она обрабатывает каждое событие *IDTExtensibility2*,отображая имя события в окне Output (листинг 6-2). После построения и регистрации LifeCycle загрузите ее в Visual Studio .NET, используя Add-in Manager. Затем попытайтесь загрузить и выгрузить другие надстройки, такие как Basic, чтобы инициировать различные события *IDTExtensibility2*.Чтобы вызвать событие *OnStartupComplete*, нужно прежде всего в Add-in Manager установить флаг Startup для надстройки LifeCycle, а затем перезапустить Visual Studio .NET. Для вызова события *OnBeginShutdown* закройте Visual Studio .NET когда LifeCycle загружена.

**Листинг 6-2.** Исходный код надстройки LifeCycle

**LifeCycle.cs**

```
namespace InsideVSNET
{
    namespace AddIns
    {
        using EnvDTE;
        using Extensibility;
        using InsideVSNET.Utilities;
```

```
using System;
using System.Runtime.InteropServices;

[GuidAttribute("D0C34F40-6A93-4601-B456-5A972E9C8D24")]
public class LifeCycle : IDTExtensibility2
{
    private string title = "LifeCycle";
    private OutputWindowPaneEx output;
    public void OnConnection(object application,
        ext_ConnectMode connectMode,
        object addInInst,
        ref Array custom)
    {
        this.output = new OutputWindowPaneEx((DTE)application,
            this.title);
        this.output.WriteLine("OnConnection event fired");
    }
    public void OnStartupComplete(ref Array custom)
    {
        this.output.WriteLine("OnStartupComplete event  fired");
    }

    public void OnAddInsUpdate(ref Array custom)
    {
        this.output.WriteLine("OnAddInsUpdate event fi  red");
    }

    public void OnBeginShutdown(ref Array custom)
    {
        this.output.WriteLine("OnBeginShutdown event f  ired");
    }

    public void OnDisconnection(ext_DisconnectMode rem  oveMode,
        ref Array custom)
    {
        this.output.WriteLine("OnDisconnection event f  ired");
    }
}
}
}
```

Вы уже знаете, как создать надстройку с нуля в командной строке, теперь пора научиться этому, работая в Visual Studio .NET. (В конце концов эта книга вовсе не о том, как командная строка может упростить жизнь программисте)

Для построения проекта LifeCycle сначала создайте новое решение, выбран File | New | Blank Solution. Добавьте в решение проект LifeCycle, выбрав File | Add Project | New Project и Visual C# Projects j Empty Project в диалоговом окне Add New Project. Наконец, добавьте к проекту CS-файл LifeCycle.cs, выбрав File j Add New Item и Code File в диалоговом окне Add New Item.

Далее нужно изменить два свойства проекта: тип результата (output type) и COM-взаимодействие, По умолчанию пустой проект построит консольное приложение; для изменения типа проекта на тот, что создает DLL, щелкните правой кнопкой имя проекта и в контекстном меню выберите пункт Properties. В диалоговом окне Property Pages выберите Common Properties | General в левом окне, а затем в правом — Class Library из выпадающего списка Output Type. В этом же окне можно разрешить COM-взаимодействие, что заставит Visual Studio .NET добавить в реестр элементы, позволяющие надстройке функционировать как COM-компоненту. Для включения COM-взаимодействия выберите Configuration Properties | Build в левом окошке диалогового окна LifeCycle Property Pages и выберите в правом окошке True в выпадающем списке Register For COM Interop,

**Примечание** При изменении типа результата следует щелкнуть Apply до того, как переходить к выбору значения в выпадающем списке Register For (ЮМ Interop. Если же вы переключитесь на Configuration Properties | Build, прежде чем примените новый тип результата, вас постигнет неудача: вы должны будете закрыть и повторно открыть диалоговое окно, чтобы выбрать новое значение параметра Register For COM Interop.

После настройки проекта введите код из листинга 6-2 в CS-файл. Если в этот момент попытаться построить проект, компилятор сообщит об ошибке, указав, что он не может найти пространства имен *EnvDTE, Extensibility, Utilities* и типы, которые они определяют. Добавьте ссылки на эти пространства имен, щелкнув правой кнопкой References в Solution Explorer и выбрав Add Reference в контекстном меню. На вкладке .NET диалогового окна Add References выберите из списка компонент envdte, extensibility и InsideVSNET.Utilities. Теперь компоновка LifeCycle.dll пройдет без проблем.

# Интерфейс *IDTExtensibility2*

Как вы знаете, в основе каждой надстройки лежит реализация интерфейса *IDTExtensibility2*. Visual Studio .NET вызывает методы этого интерфейса всякий раз, когда нужно уведомить надстройку о важных событиях, таких как загрузка/выгрузка другой надстройки или завершение работы Visual Studio .NET. Взаимодействие надстройки и Visual Studio .NET не является улицей с односторонним движением — напротив, благодаря интерфейсу *IDTExtensibility2* надстройка имеет доступ и может управлять всей объектной моделью автоматизации Visual Studio .NET.

## Пространство имен *EnvDTE*

Прежде чем заняться исследованием отдельных методов интерфейса *IDTExtensibility2*, разумно взглянуть с общих позиций на истинную цель надстроек — управление объектами из пространства имен *EnvDTE*. Имя *EnvDTE* — это аббревиатура от Environment Development Tools Extensibility (средства расширения инструментов среды разработки), что довольно точно описывает его назначение: здесь определена объектная модель автоматизации Visual Studio .NET. Документация по Visual Studio .NET включает схему, описывающую в этой модели иерархию из более чем

140 объектов, определенных в пространстве имен *EnvDTE*. Надстройки, описанные в этой книге, будут использовать большинство из этих объектов, но некоторые объекты представляют особый интерес:

- *DTE* корневой объект модели;
- ***DTE.AddIn*** объект, представляющий надстройку;
- ***DTE.AddIns*** набор объектов *AddIn*, включающий все надстройки, зарегистрированные в IDE Visual Studio .NET;
- ***DTE.Solution.AddIns*** набор объектов *AddIn*, связанных с решением.

Несколько последующих примеров будет посвящено работе с объектами *DTE*, *DTEAddIn* и ***DTEAddIns,*** которые в совокупности позволяют управлять как вишими собственными, так и другими надстройками. Об объекте *DTESolutionAddIns* см. главу 8.

**Примечание** Обеспечение реализации интерфейса *IDTExtensibility2*— главная цель класса надстройки, но не единственная. Класс надстройки, как и всякий другой, может определять любое число методов, свойств и событий, не связанных с интерфейсом *IDTExtensibility2*. Модель автоматизации обеспечивает доступ к надстройке, используя свойство *AddIn.Object*, возвращающее интерфейс *IDispatch*. Вот как вызвать открытый метод *DisplayMessage* из надстройки *MyAddIn.Connect*:

```
Dim dispObj As Object = DTE.AddIns.Item("MyAddIn.Connect").Object
dispObj.DisplayMessage("IDispatch a message to you.")
```

## OnConnection

Пожалуй, самый важный из методов *IDTExtensibility2*— *OnConnection*. Он обеспечивает надстройку ссылкой на главный объект, что необходимо для прямого взаимодействия с IDE. Прототип метода *OnConnection* таков:

```
public void OnConnection(object application,
    ext_ConnectMode connectMode,
    object addInInst,
    ref Array custom);
```

Параметр *application* содержит ссылку на экземпляр *EnvDTE.DTE*, представляющий корневой объект в модели автоматизации. Формально *application* задает ссылку на экземпляр *EnvDTE.DTEClass*, реализующий интерфейс *EnvDTE.DTE*, который в свою очередь является наследником интерфейса *EnvDTE._DTE*. Этот последний интерфейс и содержит нужные типы. Чтобы получить типы интерфейса *_DTE*, нужно привести тип *application к DTE* или *_DTE* в зависимости от ваших предпочтений, (См. врезку «Озвучим очевидное», чтобы понять, откуда пришли все эти подчеркивания в *EnvDTE.)* Почти каждая надстройка, делающая хоть что-то полезное, нуждается в объекте *DTE,* так что обычно в первых же операторах метода *OnConnection* объект *DTE* присваивается глобальной переменной.

### Озвучим очевидное

Глядя на подчеркивания, разбросанные, как мусор, по пространству имен *EnvDTE*, вы, возможно, подумаете что программисты, проектируя Visual Studio .NET, слегка обкурились. Большинство имен типов в *EnvDTE* не слишком похоже на другие имена типов .NET Framework; само имя *EnvDTE* нарушает принятые в -КЕТ Framework Pascal-подобные правила именования. Однако всем этим странным именам в *EnvDTE* есть законная причина — COM. (Это позволяет полагать, что программисты курили вполне легитимное сырье.)

Изначальная объектная модель расширяемости — расширяемость периода проектирования (Design Time Extensibility, DTE) — начала жить как COM-компонент в предыдущей версии Visual Studio. Вместо того чтобы переписать этот компонент в виде управляемого кода, разработчики Visual Studio .NET предпочли предоставить его функциональность через COM-взаимодействие. Сборка *EnvDTE,* отображаемая в диалоговом окне Add Reference, была сгенерирована утилитой Type Library Importer (TlbImp) из библиотеки типов компонента расширяемости (dte.olb). Когда эта утилита обрабатывает библиотеку типов, она сохраняет все имена в неприкосновенности. В случае *EnvDTE* результатом явилось то, что пространство имен несет наследие COM с точностью до каждого подчеркивания.

Конечно, разработчики Visual Studio решили правильно — не переписывать компонент расширяемости предыдущей версии Visual Studio .NET, так что теперь мы должны мириться с этими веселенькими именами и надеяться, что наши пальцы не отвалятся от непрестанного нажатия Shift+<подчеркиваниеX

Параметр *connectMode* сообщает надстройке обстоятельства, при которых последняя была загружена. Значения параметра задаются перечислением *Extensibility.ext_ConnectMode* (табл. 6-1).

**Табл. 6-1. Перечисление *Extensibility.ext* ConnectMode**

| Константа | Значение (Int32) | Описание |
|---|---|---|
| *ext_cm_AfterStartup* | *0x00000000* | Загружена после запуска Visual Studio .NET. |
| *ext_cm_Startup* | *0x00000001* | Загружена в момент запуска Visual Studio .NET. |
| *ext_cm_External* | *0x00000002* | Загружена внешним клиентом. (Больше не используется Visual Studio .NET.) |
| *ext_cm_CommandLine* | *0x00000003* | Загружена из командной строки. |
| *ext_cm_Solution* | *0x00000004* | Загружена с решением. |
| *ext_cm_UISetup* | *0x00000005* | Загружена для настройки пользовательского интерфейса. |

Настройка может проверять значения *connectMode* и после этого вести себя нужным образом. Например, когда надстройка опознает *ext_cm_UISetup* она понимает, что выполняется первый раз и может добавить свои команды в меню и инструментальные панели IDE. (Подобный код для случая *ext_cm_UISetup* генерирует Add-in Wizard.)

Параметр *addInInst* передает надстройке ссылку на собственный экземпляр *AddIn*, которую она может хранить для дальнейшего использования. (Экземпляр *AddIn* неоценим для открытия набора-предка надстройки.) Наконец, каждый из методов *IDTExtensibility2* включает параметр *custom*, позволяющий передать надстройке массив данных, специфических для хоста. Visual Studio .NET всегда передает в *custom* пустой массив.

## OnStartupComplete

Событие *OnStartupComplete* генерируется только в надстройках, загружающихся при запуске Visual Studio .NET. Вот прототип *OnStartupComplete:*

```
public void OnStartupComplete(ref Array custom);
```

Такая надстройка не может быть уверена, что ее инициализация успешно пройдет при обработке *OnConnection*. — она может быть загружена слишком рано, так что обращение к еще не загруженным компонентам Visual Studio .NET приведет к ошибке. В таких случаях надстройке лучше задействовать *OnStartupComplete,* так как при обработке этого события гарантировано, что Visual Studio .NET уже работает.

## OnAddInsUpdate

Событие *OnAddInsUpdate* генерируется, когда надстройка появляется или покидает среду Visual Studio .NET. Надстройка может использовать это событие для установления связей с другими надстройками. Прототип *OnAddInsUpdate* таков:

```
public void OnAddInsUpdate(ref Array custom);
```

Отсутствие полезных параметров свидетельствует о пассивно-агрессивной природе события *OnAddInsUpdate.* Работа надстройки может быть прервана сообщением об изменении состояния другой надстройки. Но в сообщении не говорится о том, какая это была надстройка и почему она сгенерировала событие, Если вам нужно знать, какая надстройка отвечает за появление события, то идентифицировать ее придется самому. К счастью, нужное исследование позволяет провести набор *DTE.AddIns*,. Этот набор содержит список объектов *AddIn* (по одному объекту на каждую зарегистрированную надстройку) и каждый объект *AddIn* имеет свойство *Connected,* показывающее статус соединения. Получить ту или иную надстройку из набора *AddIns* можно, передавая методу *AddIns.Item* индекс в виде ProgID или числа (начиная с 1). Если запрашиваемого индекса в наборе нет, генерируется исключение *COMException*,— «неправильный индекс», а при удаче возвращается ссылка на *AddIn.* Вот пример одного способа проверки статуса соединения для InsideVSNET.AddIns.LifeCycle:

```
public void OnAddInsUpdate(ref Array custom)
{
    try
    {
        AddIn addIn =
            this.dte.AddIns.Item("InsideVSNET.AddIns.LifeCycle ");
```

```
        if (addIn.Connected == true)
        {
            // InsideVSNET.AddIns.LifeCycle подключена.
        }
        else
        {
            // InsideVSNET.AddIns.LifeCycle не подключена.
        }
    }
    catch (COMException)
    {
        // InsideVSNET.AddIns.LifeCycle не является
        // зарегистрированной надстройкой.
    }
}
```

Конечно, все еще остается загадкой, была ли именно надстройка InsideVSNET.AddIns.LifeCycle причиной возникновения события.

Надстройка LoadUnload (листинг 6-3) делает то, с чем не смог справиться предыдущий пример: она определяет, какая надстройка вызвала событие *OnAddInsUpdate*.

**Листинг 6-3.** Исходный код LoadUnload

**LoadUnload.cs**

```
namespace InsideVSNET
{
    namespace AddIns
    {
        using EnvDTE;
        using Extensibility;
        using InsideVSNET.Utilities;
        using Microsoft.Office.Core;
        using System;
        using System.Collections;
        using System.Runtime.InteropServices;

        [GuidAttribute("B2FCDEBF-1536-4EA2-9F1A-81878A9C028D"),
            ProgId("LoadUnload.Connect")]
        public class Connect : Object, IDTExtensibility2, IDTCommandTarget
        {
            private DTE dte;
            private AddIn addInInstance;
            private SortedList addInsList = new SortedList();
            private AddIns addInsCollection;
            private OutputWindowPaneEx output;
            private string title = "LoadUnload";

            public Connect()
            {
```

```
}

public void OnConnection(object application,
    ext_ConnectMode connectMode,
     object addInInst,
    ref Array custom)
{
    this.dte = (DTE)application;
    this.addInInstance = (AddIn)addInInst;
    this.addInsCollection = this.dte.AddIns;

    foreach (AddIn addIn In this.addInsCollection)
    {
        this.addInsList[addIn.ProgID] = addIn.Connected;
    }

    this.output = new OutputWindowPaneEx(this.dte, this.title);
    :
}

public void OnDisconnection(ext_DisconnectMode disconnectMode,
    ref Array custom)
{
}

public void OnAddInsUpdate(ref Array custom)
{
    this.addInsCollection.Update();

    foreach (AddIn addIn in this.addInsCollection)
    {
        if (this.addInsList.Contains(addIn.ProgID) )
        {
            if (addIn.Connected !=
                (bool)this.addInsList[addIn.ProgID ])
            {
                string action = addIn.Connected ?
                    "loaded" : "unloaded";
                this.output.WriteLine(addIn.ProgID +
                    " was " + action, this.title);
            }
        }
        else
        {
            string action = addIn.Connected ?
                " and loaded" : String.Empty;
            this.output.WriteLine(addIn.ProgID +
                " was added" + action, this.title);
        }
```

*см. след. стр.*

```
                this.addInsList[addIn.ProgID] * addIn.Connected;
            }
        }

        public void OnStartupComplete(ref Array custom)
        {
        }

        public void OnBeginShutdown(ref Array custom)
        {
        }
        ⋮
    }
}
}
```

LoadUnload хранит список выполняемых надстроек и статусов их соединений в переменной *addInsList* типа *SortedList.* Всякий раз, когда возникает *OnAddInsUpdate,* LoadUnload сравнивает статусы соединения всех надстроек из своего внутреннего списка и статусы надстроек, полученные из набора *DTE.AddIns:* если найдено расхождение, то тем самым определен и инициатор события. Взгляните на первую часть главного цикла из листинга 6-3:

```
this.addInsCollection.Update();

foreach (AddIn addIn in this.addInsCollection)
;
    if (this.addInsList.Contains(addIn.ProgID))
    {
        if (addIn.Connected !=
            (bool)this.addInsList[addIn.ProgID])
        {
            string action = addIn.Connected ?
                "loaded" : "unloaded";
            this.output.WriteLine(addIn.ProgID +
                " was " + action, this.title);
        }
    }
    ⋮
```

Переменная *addInsCollection* содержит ссылку на набор *DTEAddIns,* и вызов *Update* синхронизирует набор с реестром, так что в него включаются любые вновь созданные надстройки. (Add-in Manager при каждом запуске выполняет работу, эквивалентную *Update.*)После вызова *Update,* главный цикл проходит по всем текущим надстройкам в *addInsCollection* и проверяет, имеется ли каждая надстройка в его внутреннем списке. Если да, свойство *Connected* сравнивается со значением, хранимым во внутреннем списке, если значения разнятся, то сравнение свойств *Connected* позволяет определить, была ли надстройка только что загружена (true) или выгружена (false). Если же текущей надстройки нет во внутреннем списке

*addInsList*, значит, надстройка была зарегистрирована между предыдущим и текущим событиями *OnAddInsUpdate*.

Вот вторая часть главного цикла, обрабатывающая появление новых надстроек:

```
    :
    else
    {
        string action = addIn.Connected ?
            " and loaded" : String.Empty;
        this.output.WriteLine(addIn.ProgID +
            " was added" + action, this.title);
    }

    this.addInsList[addIn.ProgID] = addIn.Connected;
}
```

Последний оператор либо пишет текущее значение *Connected* в существующий элемент, либо создает новый элемент для вновь зарегистрированной надстройки.

LoadUnload, конечно, далека от совершенства (например, надстройки, загружаемые по команде, появляются и исчезают незамеченными), но для демонстрационных целей ее вполне хватает,

## OnBeginShutdown

Вот прототип *OnBeginShutdown*:

```
public void OnBeginShutdown(ref Array custom);
```

Это событие возникает, когда IDE прекращает работу при работающей надстройке. Хотя останов JDE может быть отменен, *OnBeginShutdown* не поддерживает механизм отмены, так что надстройка должна считать, что прекращение работы IDE неизбежно, и освободить ресурсы. Надстройка, управляющая состоянием IDE, может использовать это событие для восстановления начальных параметров IDE.

## OnDisconnection

Это событие напоминает *OnBeginShutdown* тем, что сигнализирует о завершении жизни надстройки; но в отличие от *OnBeginShutdown* это не обязательно связано с прекращением работы IDE. Событие *OnDisconnection* предоставляет надстройке больше сведений, чем *OnBeginShutdown*. Прототип *OnDisconnection* таков:

```
public void OnDisconnection(ext_DisconnectMode removeMode,
    ref Array custom);
```

Заданные перечислением *IDTExtensibility2.ext_DisconnectMode* значения параметра *removeMode* говорят надстройке, почему она была выгружена. Значения перечисления *ext_DisconnectMode* приведены в табл. 6-2.

Перечисление *ext_DisconnectMode* служит целям, подобным *ext_ConnectMode*: оно позволяет надстройке изменить свое поведение, чтобы соответствовать изменившимся обстоятельствам. Так, надстройка, получившая *ext_dm_UISetupComplete*, вероятно, не станет заниматься освобождением ресурсов, поскольку была загружена только для инициализации.

**Табл. 6-2. Перечисление *Extensibility.ext_DisconnectMode***

| Константа | Значение (Int32) | Описание |
|---|---|---|
| *ext_dm_HostShutdown* | *0x00000000* | **Выгружена** при останове Visual Studio .NET. |
| *ext_dm_UserClosed* | *0x00000001* | Выгружена, тогда как Visual Studio продолжает выполняться. |
| *ext_dm_UISetupComplete* | *0x00000002* | Выгружена после настройки пользовательского интерфейса. |
| *ext_dm_SolutionClosed* | *0x00000003* | Выгружена, когда было закрыто решение. |

# Реестр и именованные значения надстройки

Как вы помните, надстройка становится известной Visual Studio .NET, создавая подраздел реестра с именем, соответствующим ProgID, в разделе *AddIns*. Этот подраздел может содержать несколько именованных значений, позволяющих настроить поведение надстройки. Ниже мы рассмотрим возможные именованные значения и производимый ими эффект.

## Разделы *CommandPreload* и *PreloadAddinState*

Функциональность большинства надстроек доступна через команды меню и кнопки инструментальных панелей IDE: когда выбираешь команду или нажимаешь кнопку, соответствующие элементы интерфейса загружают надстройку и передают ей соответствующую команду для обработки. Конечно, такие элементы пользовательского интерфейса не появляются чудесным образом — их создает сама надстройка и добавляет их в интерфейс IDE при своей первоначальной загрузке. Но если не будет действий пользователя, нажимающего на кнопки, что заставит надстройку загрузиться первый раз, чтобы создать эти кнопки и другие элементы интерфейса? Решение проблемы «яйцо или курица» начинается с именованного значения *CommandPreload* и кончается *PreloadAddinState*. Вновь установленная надстройка задает значение *CommandPreload* равным *0x1*, сообщая тем самым Visual Studio .NET, что хочет при следующем запуске IDE быть загруженной с целью добавления элементов пользовательского интерфейса в IDE (процесс, известный как *предварительнаязагрузка*). Рис. 6-5 иллюстрирует, как Visual Studio .NET выполняет предварительную загрузку надстроек.

**Примечание** Вовсе не обязательно, чтобы предварительно загруженные надстройки создавали элементы интерфейса; они вольны выполнять любой вид инициализации «первой загрузки», например, создавать файлы, добавлять элементы в реестр и т. д.

При запуске Visual Studio .NET выполняется предварительная загрузка всех надстроек, значение *CommandPreload* которых равно *0x1*, но при условии, что они еще не были загружены. Способ, которым Visual Studio .NET определяет, была ли надстройка предварительно загружена, зависит от того, какая это надстройка — пользователя или администратора. Для первых значение *CommandPreload* позволяет определить, была ли она предварительно загружена, поскольку Visual Studio .NET после предварительной загрузки надстройки изменяет это значение на *0x2*

(рис. 6-5). Для вторых значение *CommandPreload* никогда не меняется, однако Visual Studio .NET располагает другим механизмом, позволяющим и в этом случае проследить, проводилась ли предварительная загрузка надстройки. Таковым является раздел *PreloadAddinState:* он содержит список надстроек администратора и статус их предварительной загрузки. Прежде чем Visual Studio .NET предварительно загрузит надстройку администратора, проверяется раздел *PreloadAddinState:* если раздела нет, надстройки нет в списке или она есть в списке и значение для нее равно *0x1*, Visual Studio .NET знает, что надстройка еще не была загружена.

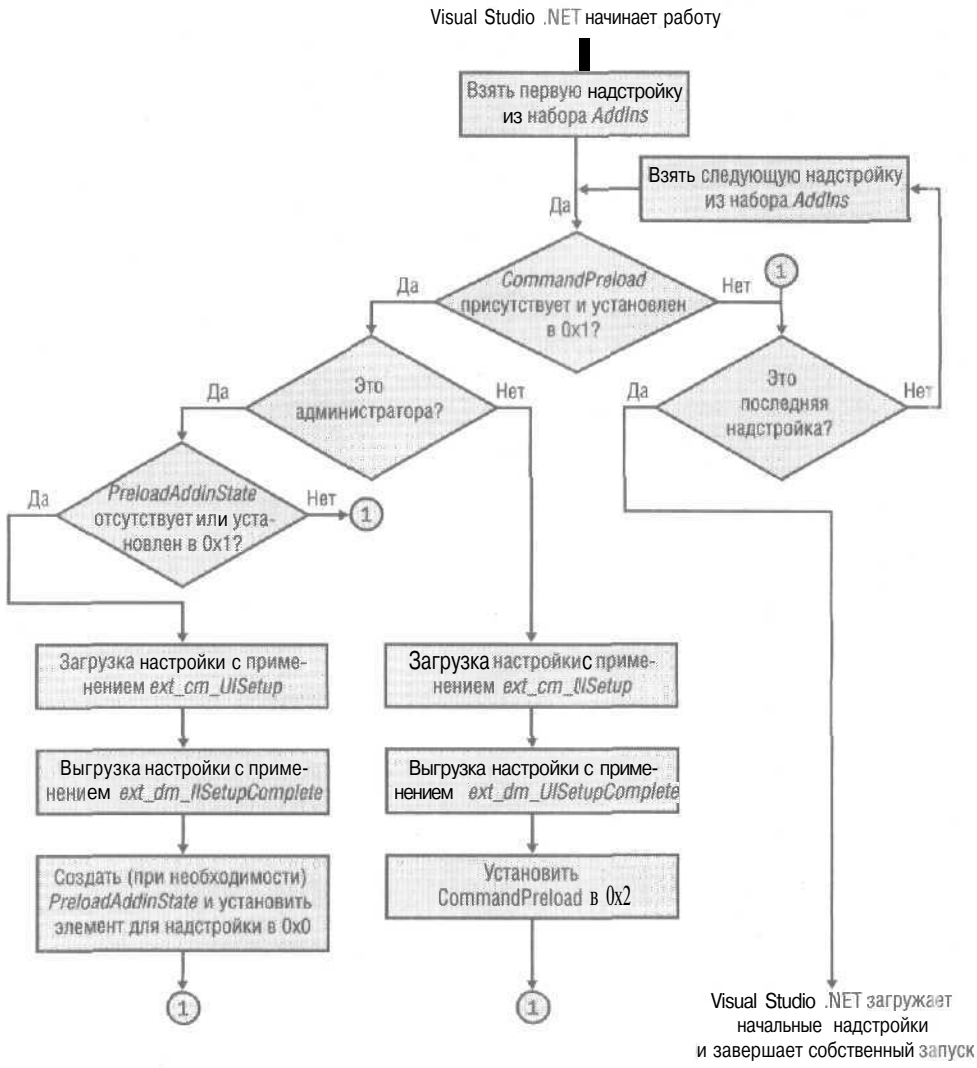



**Рис. 6-5.** *Предварительная загрузка надстроек*

Для надстроек обоих типов Visual Studio .NET предварительно загружает надстройки «молча», передавая им значение *ext_cm_UISetup* в событии *OnConnection*,

а затем выгружает их прямо после завершения *OnConnection*, передав им значение *ext_dm_UISetupComplete* в событии *OnDisconnection*. После предварительной загрузки надстройки администратора Visual Studio .NET создаст раздел *PreloadAddinState* (если его нет) и задаст для надстройки значение *0x0*; после предварительной загрузки надстройки пользователя Visual Studio .NET установит для него значение *CommandPreload* в *0x2*.

**Предупреждение** Когда мы говорим, что Visual Studio .NET «молча» выполняет предварительную загрузку, мы и имеем в виду именно *молча*. При нормальных условиях Visual Studio .NET выдаст сообщение, если при загрузке надстройки возникнет ошибка, и даже предложит удалить эту надстройку из раздела *AddIns* реестра. При предварительной загрузке Visual Studio .NET не даст ни малейшей подсказки о проблемах с надстройкой.

Раздел *PreloadAddinState* размещается в кусте реестра HKEY_CURRENT_USER в разделе Software\Microsoft\VisualStudio\7.1\AddIns\PreloadAddinState. Этот раздел создается не при установке Visual Studio .NET, а по требованию, когда Visual Studio .NET в первый раз производит предварительную загрузку надстройки администратора для пользователя. Так что надстройка администратора, установившая *CommandPreload* в *0x1*, становится автоматически загружаемой для всех пользователей, включая и тех, кто добавляется уже после установки надстройки.

**Совет** Если нужно восстановить элементы, добавляемые надстройкой в интерфейс пользователя, задайте *PreloadAddinState* значение *0x1*, если это надстройка администратора, или установите *CommandPreload* в *0x1* – если пользователя; при следующем запуске Visual Studio .NET для надстройки будет осуществлена предварительная загрузка. Если же вы предпочитаете принудительные меры, можно восстановить элементы пользовательского интерфейса для каждой надстройки командой:

```
devenv /setup
```

Эта команда восстановит IDE в своем начальном состоянии (при этом удалив элементы интерфейса всех надстроек, а заодно и все пользовательские изменения в IDE), изменит для надстроек пользователя значения *CommandPreload* с *0x2* на *0x1* и удалит раздел *PreloadAddinState*. В результате Visual Studio .NET предварительно загрузит все надстройки при следующем запуске. Грубовато, но эффективно.

## LoadBehavior и CommandLineSafe

Именованное значение *LoadBehavior* управляет тем, как надстройка загружается, а также отображает текущее состояние загрузки. В табл. 6-3 приведены возможные значения *LoadBehavior*. Эти значения являются битовыми флагами, так что теоретически их можно комбинировать, задавая тем самым собственные параметры загрузки. На практике же есть одно исключение: комбинация флагов ведет себя неотличимо от отдельных флагов.

**Табл. 6-3. Значения *LoadBehavior***

| Флаг | Значение | Описание |
|---|---|---|
| *ID_UNLOADED* | *0x0* | Надстройка в настоящее время не загружена. |
| *ID_STARTUP* | ***0x1*** | Надстройка должна загружаться при запуске среды. |
| *IDJLOADED* | *0x2* | Надстройка в настоящее время загружена. |
| *ID_COMMAND_LINE* | *0x4* | Надстройка должна загружаться при компоновке из командной строки. |

Значения *ID_UNLOADED* и *IDJLOADED* более не служат полезным целям, Для определения состояния загрузки надстройки служит свойство *AddIn.Connected*.

Флаг *ID_STARTUP* уведомляет Visual Studio .NET о необходимости загрузить надстройку в момент запуска IDE. В частности, этот флаг следует включить для надстроек, отслеживающих события IDE, — им с самого начала нужно быть активными, иначе они могут пропустить некоторые действия. Те же надстройки, которых не очень заботят события среды, могут опустить этот флаг и ждать, пока они не будут загружены по требованию.

Флаг *ID_COMMAND_LINE* сигнализирует, что надстройка загружается при компоновке из командной строки. Учтите: в Visual Studio .NET 2002 этот флаг не работает так, как следует из описания. В версии 2002 единственный эффект этого флага в том, что он запрещает Add-in Manager загружать данную надстройку (да и этого эффекта нет, если одновременно присутствует флаг *ID_STARTUP*).

Необязательное именованное значение *CommandLineSafe* предполагает совместную работу с флагами *ID_STARTUP* и *ID_COMMAND_LINE*, чтобы гарантировать успех компоновки, не требующей вмешательства оператора. Значение *CommandLineSafe,* равное *0x1*, указывает, что надстройка не имеет интерфейса, требующего взаимодействия с пользователем, во всяком случае когда компоновка запускается из командной строки. Отсутствие элемента *CommandLineSafe* или значение *CommandLineSafe,* равное *0x0,* указывает на то, что надстройка не нужна при компоновке из командной строки. В настоящее время значение *CommandLineSafe* не влияет на загрузку надстройки и применяется только в информационных целях.

Безопасно ли **все это?**

Очевидно, проектировщики Visual Studio .NET имели в виду что-то особенное, когда создавали значения в реестре, связанные ,с командной строкой. Продолжительные компоновки, в том числе запускаемые на ночь, характерны для индустрии ПО, поэтому любое качество, которое увеличивает шансы на успешную компоновку, не требующую вмешательства, ценятся системными администраторами на вес золота. Что до надстроек, то в Visual Studio .NET 2002 включены все элементы, необходимые для полноценной компоновки из командной строки, но здесь эти компоненты не связаны между собой. А вот как выглядит компоновка из командной строки в Visual Studio .NET 2003.

1. Разработчик инициирует компоновку из командной строки,
2. Visual Studio .NET загружает все надстройки, отмеченные как *ID_COMMAND_LINE* и передает значение *ext_cm_CommandLine* событию *OnConnection* каждой надстройки.

*см. след, стр.*

3. Сборка использует значение *ext_cm_CommandLine* как сигнал для выключения своего пользовательского интерфейса.

Но в Visual Studio .NET 2002 компоновка из командной строки так не работает. Здесь *ID_COMMAND_LINE* не влияет на загрузку надстройки; надстройки, отмеченные как *ID_STARTUP*, ошибочно загружаются и Visual Studio .NET 2002 не передает им значение *ext_cm_CommandLine* ни при каких обстоятельствах.

Не следует беспокоиться: все эти проблемы уже решены в Visual **Studio** .NET 2003. Что же до Visual Studio .NET 2002, то и здесь можно обойти эти проблемы, отметив надстройку как *ID_STARTUP* и используя код, подобный следующему, в котором просматриваются свойства *System.Environment.CommandLine* в поисках признака компоновки из командной строки:

```
if (Regex.IsMatch(System.Environment.CommandLine, "/build"))
{
    // Компоновка из командной строки.
    System.Console.WriteLine("Command-line build- phooey on GUI!");
}
else
{
    // Компоновка не из командной строки.
    MessageBox.Show("GUI for you!");
}
```

Если поместить этот код в обработчике события *OnConnection*, надстройка в момент загрузки может решить, отображать ли ей пользовательский интерфейс.

## *SatelliteDLLPath* и *SatelliteDLLName*

Если предполагается применять надстройку в разных странах, особое внимание следует уделить локализации. Используемая во всем мире надстройка не должна требовать от пользователей знания определенного языка — она должна общаться с каждым пользователем на его родном языке. Надстройка, поддерживающая локализацию, хранит локализуемые ресурсы в сателлитной DLL В период выполнения она ищет в этой DLL ресурсы, соответствующие местному языку, и применяет их для настройки интерфейса пользователя. Таким способом одна надстройка может поддерживать любое число языков, предоставляя для каждого местного языка соответствующие ресурсы в сателлитной DLL.

Сборка уведомляет о размещении своих сателлитных DLL в именованных значениях *SatelliteDLLPath* и *SatelliteDLLName*. *SatelliteDLLName* хранит имя сателлитной DLL (из чего следует, что все локализованные сателлитные DLL имеют одно и то же имя файла). *SatelliteDLLPath* хранит корневую папку для сателлитных DLL Каждая локализованная сателлитная DLL размещается в своей папке, вложенной в корневую, и имя папки является идентификатором языка, поддерживаемым этой DLL. Например, идентификатор языка U.S. English — 1033, так что сателлитную DLL для U.S. English можно найти в <SatelliteDLLPath>\1033\<SatelliteDLLName>. В пе-

риод выполнения можно найти сателлитную DLL для текущего местного диалекта через свойство *AddIn.SatelliteDllPath* либо метод *DTE.SatelliteDllPath*.

**Практикум: проверка сателлитных ОН**

; Документация утверждает, что Visual Studio .NET ищет сателлитные DLL в таком порядке:

```
<SatelliteDLLPath>\<Default system locale>\<SatelliteDLLName>
<SatelliteDLLPath>\<Default user locale>\<SatelliteDLLName>
<SatelliteDLLPath>\<OS setup locale>\<SatelliteDLLName>
<SatelliteDLLPath>\1033\<SatelliteDLLName>
<SatelliteDLLPath>\<SatelliteDLLName>
```

Увы, документация неправа, по крайней мере частично. Так как же Visual Studio .NET находит сателлтные DLL? Посмотрите сами. Макрос *SatelliteDLL-Probing* создает каталог satellites, содержащий подкаталоги для 133 местных диалектов — от Afrikaans до Zulu, — и копирует сателлитную DLL в каждый каталог. В основном цикле макроса многократно вызывается метод *DTE.SatelliteDllPath*: после каждого вызова макрос отображает в окне Output путь к сателлитной DLL и соответствующий местный диалект, затем текущая сателлитная DLL удаляется, так что на следующем шаге цикла метод *SatelliteDllPath* найдет очередное местоположение.

Если несколько раз запустить макрос *SatelliteDLLProbing* изменяя местный диалект пользователи между вызовами, то можно заметить, что местный диалект пользователя не имеет большого значения. Системный местный диалект всегда переопределяет местный диалект пользователя — для текущего пользователя это означает, что он может и не увидеть диалект. Вы также можете заметить, что *SatelliteDllPath* никогда не найдет сателлитную DLL в <SatelliteDLLPath>, что противоречит документации. Наиболее интересно, однако, то упорство, с которым метод *SatelliteDllPath* отыскивает отдельную DLL. Вместо того чтобы остановиться после проверки подкаталога для системного местного диалекта, не найдя там ничего нужного, *SatelliteDllPath* начинает рыскать по другим подкаталогам, как собака по мусорной свалке. Полагаем, здесь такая логика: уж лучше какой-нибудь язык, чем совсем без языка. (Не можем найти английский? Попробуем на каталонском!) На самом деле *SatelliteDllPath* проходит в своих поисках и подкаталоги, не связанные с местными диалектами, и, если находит сателлитную DLL, ее и использует.

Следует понимать, что локализация — сложный вопрос и алгоритм поиска сателлитной DLL в Visual Studio .NET это подтверждает. Если местный диалект установлен корректно, алгоритм работает прекрасно, но если допустить ошибку, то теряется вся его эффективность. Хорошая новость в том, что Visual Studio .NET использует тот же механизм для собственной локализации, поэтому ваша надстройка всегда будет найдена, если использовать для нее тот же местный диалект, что и при установке Visual Studio .NET.

## *FriendlyName* и *Description*

Именованные значения *FriendlyName* и *Description* позволяют указывать понятное имя и давать краткое описание надстройки. Пример приложения, использующего эти значения, — Add-in Manager, который заполняет список надстроек значениями *FriendlyName* и отображает в окне Description значение *Description* выбранной надстройки.

Каждое из значений *FriendlyName* и *Description* хранят либо читабельную строку, либо ID строкового ресурса из сателлитной DLL данной надстройки (в форме «#<resource ID>»). Выделение строкового ресурса из сателлитной DLL выполняется довольно просто; соответствующий код можно найти в примере Add-in Manager Plus.

## *AboutBoxDetails* и *AboutBoxIcon*

Именованные значения *AboutBoxDetails* и *AboutBoxIcon* отображаются в диалоговом окне About Visual Studio .NET (рис. 6-6). Это окно отображает значения *FriendlyName* в списке Installed Products; когда пользователь выбирает надстройку из этого списка, диалоговое окно выводит для нее значок *AboutBoxIcon* под надписью Product Details, а в текстовом поле рядом со значком — информацию *AboutBoxDetails*.

**Примечание** Как *AboutBoxDetails,* так и *AboutBoxIcon* должны быть заданы в реестре, прежде чем в диалоговом окне About будет отображаться информация о надстройке; однако одно или оба значения могут быть фиктивными.

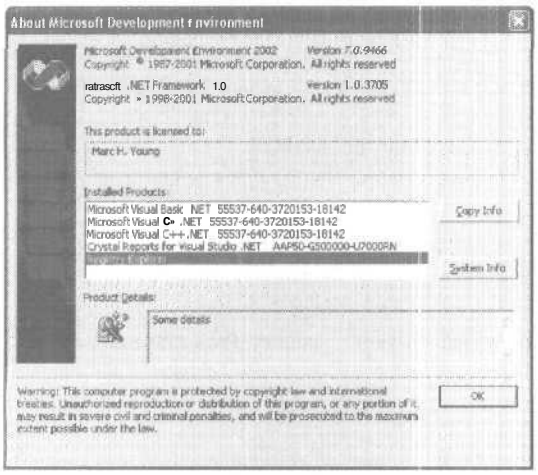



**Рис** 6-6. *Информация о надстройке в окне About Visual Studio .NET*

Формат значения *AboutBoxDetails* такой же, как и для значений *FriendlyName* и *Description*: строка с кратким описанием либо ID, задающий строковый ресурс из сателлитной DLL (в форме «#<resource ID>»).

Значение *AboutBoxIcon* имеет немного большую гибкость, чем его напарник, Надстройка может хранить информацию о значке в одном из четырех форматов:

- строка, представляющая ID ресурса значка в сателлитной DLL (в форме «#<resource ID>»);
- строка, представляющая путь к файлу значка;
- строка, представляющая путь к исполняемому файлу и ID ресурса значка внутри этого файла (в форме «<path>,<resource ID>»); например, «C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\devenv.exe, 1200» ссылается на значок приложения Visual Studio .NET;
- двоичные данные значка.

Последний формат дает возможность надстройке хранить всю свою «личную» информацию (включая *FriendlyName*, *Description* и *AboutBoxDetails*) внутри реестра, а не разбросанную по отдельным файлам.

Хранение данных значка в реестре столь же просто, как и следующий макрос, который принимает в качестве параметров раздел надстройки в реестре и путь к файлу значка:

```
Sub SetAboutBoxIcon(ByVal addInKey As RegistryKey, _
    ByVal iconPath As String)

    Dim iconFile As FileStream = File.OpenRead(iconPath)
    Dim iconData(iconFile.Length) As flyte
    Dim i As Integer

    For i = 0 To iconFile.Length - 1
        iconData(i) = iconFile.ReadByte
    Next

    addInKey.SetValue("AboutBoxIcon", iconData)
End Sub
```

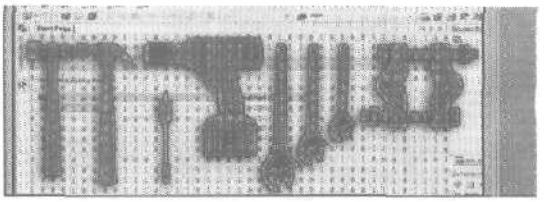
Примечание Честь и хвала Кенни Керру за его классы управления ресурсами значков! Нас одолевали тягостные раздумья, как решить неблагодарную задачу написания управляемого кода, вытаскивающего значки из исполняемых файлов Win32, но Кенни сделал это и избавил нас от этих проблем, Так что мы рады передать ему нашу огромную благодарность, которую только может вместить это короткое примечание. (Мы использовали его классы для извлечения значков в надстройке Add-in Manager Plus и не могли нарадоваться результатам работы.) Вы можете найти его утилиту Icon Browser, для которой первоначально и были написаны классы управления ресурсами значков, и еще многие полезные «новинки» от Кенни на его Web-узле: http://www.kennyandkarin.com/kenny.

## Что дальше

Следующая глава удовлетворит потребность, хоть немного присущую каждому из нас: приятно скомандовать: «ПРЫГАЙ!» и в ответ услышать: «ВЫСОКО?» Как вскоре будет показано, макросы, надстройки и сама Visual Studio .NET представляют собой наиболее лояльных и покорных солдат, всегда готовых и стремящихся выполнить любые ваши приказания: все, что нужно делать, — это отдавать правильные команды. Если вы перевернете эту страницу, мы покажем вам как... «ДАВАЙ ПРЫГАЙ!»

ГЛАВА

# 7



# Команды

Команды — наиболее фундаментальный механизм взаимодействия между пользователем и интегрированной средой разработки (IDE). Эта глава освещает использование существующих и создание новых команд из надстроек и макросов.

## Что такое команда?

Если вам приходилось разрабатывать для Microsoft Windows программы с пользовательским интерфейсом, вы, наверное, знакомы с управляемой событиями моделью программирования. Когда пользователь щелкает кнопку на форме, выбирает пункт меню или нажимает клавишу на клавиатуре, программа получает уведомление об этом действии. Если вы программируете на уровне Windows SDK, скажем, на языке Visual C++, при выполнении пользователем какого-то действия, ваша программа получает сообщение, детализирующее произошедшее. Если вы применяете такие языки, как Microsoft Visual Basic .NET или Microsoft Visual C# .NET, подобное уведомление произойдет в форме вызова обработчика события. Такие уведомления — это команды, выдаваемые пользователем, и программа обрабатывает эти их, что-то делая в ответ,

В Visual Studio .NET для информирования кода о фактах взаимодействия пользователя с IDE применяется способ уведомления, похожий на передачу сообщений в Win32. Однако из-за сложности и большого числа команд в IDE Visual Studio NET *маршрутизация* команды, т. е. передача уведомления надлежащему обработчику этого уведомления, не сводится к простому получению сообщения. Допустим, пользователь выбрал File | New File. Поскольку существует целый ряд присоединяемых программ (add-on) [не путать с надстройками (add-in)], таких как Visual C++ .NET, Visual Basic .NET и Visual C# .NET, то Visual Studio .NET нужно определить, какая из этих программ должна обработать выбранный пункт меню. Когда

Win32-программа обрабатывает сообщение, существует только один цикл его обработки, но поскольку в Visual Studio .NET несколько возможных обработчиков команды, команду надо направить на обработку подходящему коду. Каждая из присоединяемых программ резервирует глобально уникальный идентификатор (GUID) для однозначной идентификации себя, и каждая доступная команда связывает себя с GUID конкретной присоединяемой программы. Когда пользователь выполняет команду, определяется связанный с ней GUID, ищется присоединяемая программа, обрабатывающая этот GUID, и команда посылается данной программе.

Команде требуется и другая идентифицирующая информация. Каждая команда связана с идентифицирующим ее GUID, и все команды, относящиеся к одной присоединяемой программе, имеют один и тот же GUID, следовательно, присоединяемая программа не сможет отличить, скажем, команду New File от команды New Project. Для устранения неоднозначности между командами, имеющими одинаковый GUID, каждой команде в этой группе присваивается некоторое число, или ГО. Каждая присоединяемая программа откликается только на собственные команды, поэтому ID можно присваивать, не конфликтуя с командами других присоединяемых программ, ведь GUID всех программ различны. Комбинация GUID и ID однозначно определяет каждую отдельную команду.

---

Примечание Команда в Visual Studio .NET существует независимо от любых элементов пользовательского интерфейса (скажем, пунктов меню) этой команды. Команды могут создаваться и уничтожаться, и для какой-то команды элемент пользовательского интерфейса может быть не создан никогда. А вот обратное невозможно — элемент пользовательского интерфейса всегда имеет связанную с ним команду.

---

## Поиск команд

В Visual Studio .NET любая команда, которую может выполнить пользователь, представлена в объектной модели объектом *Command*, а список этих объектов содержится в наборе *Commands*. Как и для других объектов, входящих в наборы, для перебора объектов *Command* можно применять стандартные конструкции, такие как ключевые слова *foreach* (Visual C# ) или *ForEach* (Visual Basic .NET), используя которые, можно создать макрос, просматривающий список объектов *Command*.

```
Sub WalkCommands()
    Dim cmd As EnvDTE.Command
    For Each cmd In DTE.Commands
      'Здесь используется объект EnvDTE.Command.
    Next
End Sub
```

Метод *Item* набора *Commands* работает немного иначе, чем методы *Item* других наборов объектов. *Commands.Item* получает как параметр знакомый вам числовой индекс, а еще и дополнительный необязательный аргумент. Если вы применяете способ числовой индексации, вам нужно задать второму аргументу -1. Рассматриваемый метод имеет дополнительный аргумент, поскольку, как уже говорилось, для однозначной идентификации команды служит пара GUID и ID. И

можно использовать GUID и ID для индексации набора *Commands*, при этом GUID в виде строки передается как первый аргумент, а ID команды — как второй. Следующий макрос показывает поиск команды открытия файла с использованием пары GUID и ID:

```
Sub FindFileOpenCommand()
    Dim cmd As EnvDTE.Command
    cmd = DTE.Commands.Item("{5EFC7975-14BC-11CF-9B2B-  00AA00573819}", 222)
End Sub
```

Чтобы писать код, подобный приведенному выше, требуется найти и выучить GUID и ID каждой из команд (а их тысячи) да еще научиться правильно их вводить во избежание программных ошибок. У метода *CommandsItem* существует более легкий для запоминания формат индексации набора — индексация по имени команды.

## Имена команд

Запоминание GUID и ID каждой команды требует жутких усилий, и Visual Studio .NET определяет простое для запоминания текстовое представление большинства команд. Эти имена построены по общему образцу: текст меню верхнего уровня, в котором находится элемент пользовательского интерфейса данной команды, затем точка, тексты каждого из вложенных подменю, разделяемые точками, и, наконец, текст элемента меню, связанного с командой. Любые символы, отличные от букв и чисел (кроме точек-разделителей и подчеркиваний), затем удаляются из этой строки. Возвращаясь к примеру с поиском объекта *Command* для команды открытия файла и используя новый способ поиска по имени команды, получим макрос:

```
Sub FindFileOpenCommandByName()
    Dim command As EnvDTE.Command
    command = DTE.Commands.Item("File.OpenFile")
End Sub
```

Чтобы определить GUID и ID этой команды, можно задействовать свойства *GUID* и *ID* объекта *Command*. Сделаем это для поиска пары GUID и ID, использованной в макросе *FindFileOpenCommand*, показанном ранее. Вот макрос, который можно для этого применить:

```
Sub FindGuidIDPair()
    Dim guid As String
    Dim id As Integer
    Dim command As EnvDTE.Command
    command = DTE.Commands.Item("File.OpenFile")
    guid = command.Guid
    id = command.ID
    MsgBox(guid + ", " + id.ToString())
End Sub
```

Диалоговое окно Options позволяет найти имена всех доступных команд (рис. 7-1). На левой панели можно выбрать Environment | Keyboard, чтобы отобразить список имен всех команд.

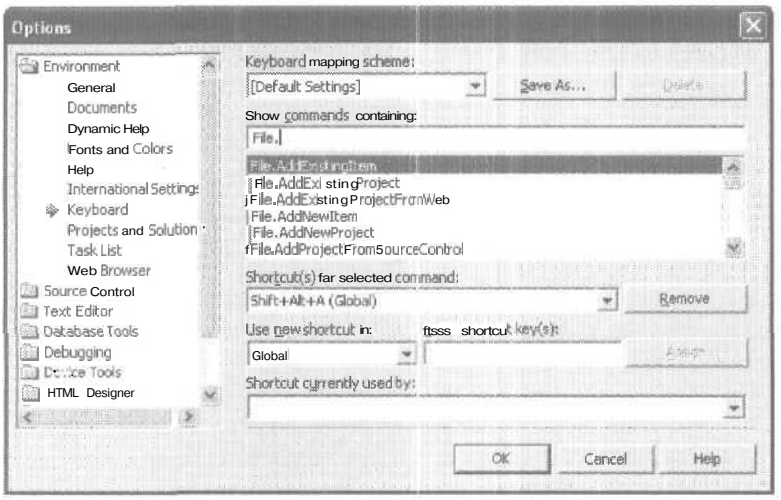



Рис. 7-1. *Диалоговое окно Options*

Для нахождения имен всех доступных команд можно применить и объектную модель. Сделаем это с помощью набора *EnvDTE.Commands* в таком макросе:

```
Sub CreateCommandList()
    Dim command As EnvDTE.Command
    Dim output As New OutputWindowPaneEx(DTE, "Create Command List")

    For Each command In DTE.Commands
        If (command.Name <> Nothing) Then
            output.WriteLine(command.Name)
        End If
    Next
End Sub
```

После запуска макрос помещает в окно Output имена всех доступных команд. Этот макрос также проверяет, что имя команды не установлено в *Nothing*. Проверка производится потому, что, если команда не имеет имени, для нее вместо имени возвращается *Nothing* в Visual Basic .NET или *null* в C#. .NET Framework при попытке использования этого пустого значения создает объект *System.String* со значением, равным пустой строке (""). Однако в этом макросе мы отфильтровываем все безымянные команды, сравнивая получаемое имя со значением *Nothing*. Безымянные команды обычно служат для закрытого внутреннего взаимодействия в Visual Studio .NET, и пользователь, как правило, не может обращаться к ним. Мы не советуем вам применять эти команды, так как это может привести к непредсказуемым последствиям.

## Выполнение команд

Команды должны обеспечить пользователю возможность обратиться к Visual Studio .NET для выполнения некоторых действий. Инициировать выполнение команды можно разными способами, и самый распространенный — выбор пункта меню или нажатие кнопки на панели инструментов. Но есть и другие. Так, если вы на-

писали макрос, отвечающий стандартам записи макросов (он определен как открытый, не возвращает значения и не принимает аргументов, если только эти аргументы не являются необязательными строковыми параметрами), средства поддержки макросов распознают этот макрос и создадут для него команду. Двойной щелчок этого макроса в окне Macro Explorer выполнит связанную с ним команду, которая будет обрабатываться редактором макросов. Третий способ выполнения команды — использование метода *DTE.ExecuteCommand*, который запускает команду, заданную ее именем, как если бы пользователь выбрал пункт меню, связанный с этой командой.

Чтобы запустить нашу команду *File.OpenFile* с использованием метода *ExecuteCommand*, надо написать примерно такой код:

```
Sub RunFileOpenCommand()
    DTE.ExecuteCommand("File.OpenFile")
End Sub
```

После вызова метода *ExecuteCommand* выполнение макроса или надстройки приостанавливается до тех пор, пока вызванная команда не закончит работу.

И, наконец, способ для продвинутых пользователей — напечатать имя команды в командном окне, Как отмечалось в главе 1, командным называется окно, работающее в текстовом режиме, в котором вы печатаете имена команд; когда пользователь нажимает клавишу Enter, команда запускается. Имя команды, которое вы печатаете в командном окне, — это то же самое имя, которое возвращает свойство *CommandName* и которое можно передавать как аргумент методу *ExecuteCommand*.

### Создание команд запуска макросов

Как вы помните, макросы, отвечающие определенному формату, автоматически включаются в число команд, и этим командам присваивается имя. Имя команды состоит из строки *Macros*, имени проекта макроса, имени модуля или класса, в котором реализован макрос, и имени макроса; части разделены точками. В этом формате макрос *TurnOnLineNumbers* из проекта макроса Samples, устанавливаемого с Visual Studio .NET, получит имя *Macros.Samples.Utilities.TurnOnLineNumbers*. Можно вводить это имя в командном окне или обращаться к этому имени из другого макроса, например, так:

```
Sub RunCommand()
    DTE.ExecuteCommand("Macros.Samples.Utilities.TurnOnLineNumbers")
End Sub
```

## Создание команд для надстроек

Пора создать собственную команду. Как уже говорилось, когда активизируется команда, встроенная в Visual Studio .NET, по GUID, связанному с этой командой, определяется соответствующая присоединяемая программа, и команда передается на обработку данной программе. Точно так же и для создаваемых вами команд нужно указать, кто будет заниматься ее обработкой. Команду можно динамически создавать и удалять, но ее создание требует наличия связанной с ней надстройки,

которую Visual Studio .NET могла найти и использовать в качестве обрабатывающей программы. Метод *AddNamedCommand,* предназначенный для создания команд, содержится в объекте-наборе *DTE.Commands,* вот его сигнатура:

```
public EnvDTE.Command AddNamedCommand(EnvDTE.AddIn AddInInstance,
    string Name, string ButtonText, string Tooltip, bool MSOButton,
    int Bitmap = 0, ref object[] ContextUIGUIDs,
    int vsCommandDisabledFlagsValue = 16)
```

Аргументы у этого метода таковы.

- ***AddInInstance*** Объект *AddIn,* обрабатывающий команду.
- ***Name*** Имя команды. Может содержать только буквы, цифры и подчеркивания. Любой другой использованный символ заменяется подчеркиванием,
- ***ToolTip*** Текст подсказки для данной команды.
- ***MSOButton*** Равен *True,* если значок, отображаемый для данной команды на элементах пользовательского интерфейса, будет использовать одно из изображений, предопределенных для кнопок панели команд; если равен *False,* изображение для значка извлекается из сателлитной DLL, указанной в регистрационной информации данной надстройки.
- ***Bitmap*** Если аргумент *MSOButton* равен *True,* то это индекс предопределенного изображения кнопки панели команд. Список доступных предопределенных изображений приведен на HTML-странице в каталоге CommandUIBmps, находящемся среди файлов примеров, поставляемых с этой книгой. Если *MSOButton* равен *False,* то это идентификатор ресурса растрового изображения в сателлитной DLL.
- ***ContextUIGUIDs*** Этот параметр зарезервирован для будущих версий. Здесь следует передавать пустой массив элементов типа *System.Object.*
- ***vsCommandDisabledFlagsValue*** Это состояние доступности кнопки, принимаемое по умолчанию. Если надстройка, обрабатывающая вызов данной команды, еще не загружена, вместо немедленной ее загрузки, чтобы определить, как отображать данную команду, используется значение этого аргумента. Значение этого аргумента применяется вместо значения, возвращаемого аргументом *StatusOption* метода *IDTCommandTarget.QueryStatus,* который мы еще обсудим.

Метод *AddNamedCommand* добавляет элемент к внутреннему списку команд, поддерживаемому Visual Studio .NET. Полное имя команды, которое можно использовать в командном окне или в качестве значения аргумента метода *ExecuteCommand,* складывается из ProgID соответствующей надстройки, точки и значения параметра *Name.* Так, если имя, переданное методу *AddNamedCommand,* — *MyCommand,* а ProgID надстройки — *MyAddin.Connect,* то имя созданной команды будет *MyAddin.Connect.MyCommand.*

Все команды, добавляемые этим методом, также имеют назначаемую им пару GUID и ID. GUID, используемый для всех команд, которые создает метод *AddNamedCommand,* определяется константой *EnvDTE.Constants.vsAddInCmdGroup;* значение ID при первом вызове метода *AddNamedCommand* равно 1, при каждом следующем вызове увеличивается на 1. Исходя из типа данных, используемого для ID, всего

может быть создано 4 294 967 296 команд, прежде чем начнутся конфликты между двумя командами для надстроек.

## Обработка команд

Для вновь созданной команды нашему коду нужен способ, позволяющий Visual Studio .NET осуществить обратный вызов надстройки, чтобы известить ее о вызове команды. Обычно, когда макрос или надстройка хотят, чтобы их проинформировали о выполнении пользователем некоторого действия, устанавливается связь с событием. Но обработчики команд работают иначе; вместо привязки к источнику события надстройка должна реализовать специальный интерфейс. Причина отказа от применения событий проста, Когда активизируется команда, реализуемая надстройкой, а сама надстройка, обрабатывающая команду, еще не загружена, код этой надстройки загружается в память и запускается путем вызова метода *OnConnection* и других методов интерфейса *IDTExtensibility2*, как если бы открыли диалоговое окно диспетчера надстроек (Add-in Manager) и установили флажок для данной надстройки. Поскольку надстройка является загружаемой по вызову (загружается в момент запуска команды), код надстройки не может предварительно подключиться к обработчику события.

Интерфейс для обработки вызова команд — *IDTCommandTarget* — создан по модели интерфейса *IOleCommandTarget* из Win32 SDK, но с изменениями, направленными на совместимость с *IDispatch* и на удобство использования:

```
public interface IDTCommandTarget
{
    public void Exec(string CmdName,
        EnvDTE.vsCommandExecOption ExecuteOption, ref object VariantIn,
        ref object VariantOut, ref bool Handled);

    public void QueryStatus(string CmdName,
        EnvDTE.vsCommandStatusTextWanted NeededText,
        ref EnvDTE.vsCommandStatus StatusOption,
        ref object CommandText);
};
```

При активизации все команды, созданные вашей надстройкой, управляются через данный интерфейс его методом *Exec*. Метод *Exec* имеет следующие аргументы.

- *CmdName* Полное имя команды. Чтобы определить, какую команду выполнить, ваша надстройка должна осуществить сравнение с этой строкой (с учетом регистра), так как этому методу на обработку передаются запросы на выполнение всех команд, созданных данной надстройкой.
- *vsCommandExecOption* Обычно передает значение из перечисления *vsCommandExecOptionDoDefault*, информирующее надстройку о том, какую именно работу, связанную с этой командой, выполнить.
- *VariantIn* Как вы увидите ниже, команде в качестве аргумента могут передаваться данные, и делается это именно с помощью этого аргумента. Если команда активизируется через пользовательский интерфейс (меню или панель инструментов), значение этого параметра — *null* или *Nothing* (в зависимости от языка, на котором разработана надстройка).

- ***VariantOut*** *Служит* для передачи данных от надстройки к вызывающей стороне. Однако Visual Studio .NET будет игнорировать любое значение, возвращаемое вашей командой.
- ***Handled*** Позволяет надстройке передать обратно в Visual Studio .NET значение, указывающее, обработала ли надстройка данную команду. Если возвращается *true,* последующей обработки команды не требуется, если *false* — Visual Studio .NET продолжит поиск обработчика для данной команды. Поиск завершится неудачей, так как нет других обработчиков команд, принимающих команду с такой же парой GUID и ID, как у команды, созданной данной надстройкой.

## Состояние команды

Команда и ее пользовательский интерфейс не обязательно должны быть постоянно доступны пользователю. К примеру, команда надстройки может быть доступна, только когда активным является окно текстового редактора. Можно управлять состоянием команды: доступна, недоступна или зафиксирована (latched) (когда в пункте меню за кнопкой нарисована галочка, а в случае панели инструментов вокруг кнопки нарисован прямоугольник). Состояние изменяется при помощи метода *QueryStatus* интерфейса *IDTCommandTarget.* Если надстройка еще не загружена, состояние по умолчанию определяется значением, переданным в качестве последнего аргумента метода *AddNamedCommand.* Однако когда надстройка загружена (при выполнении команды или вручную в диалоговом окне Add-in Manager), для определения состояния команды вызывается метод *QueryStatus.* Этот метод имеет следующие аргументы.

- ***CmdName*** Имеет то же значение, что и аргумент *CmdName* метода *Exec* интерфейса *IDTCommandTarget.*
- ***NeededText*** Всегда установлен в *vsCommandStatusTextWantedNone.* Надстройка всегда должна проверять наличие именно этого значения, так как прочие значения зарезервированы для будущих версий Visual Studio .NET.
- ***StatusOption*** Надстройка должна задать значение этого параметра, позволяющего Visual Studio .NET узнать, поддерживается данная команда надстройки (*vsCommandStatusSupported*) или нет (*vsCommandStatusUnsupported*) доступна ли команда для вызова (*vsCommandStatusEnabled*) невидим ли пользовательский интерфейс команды (*vsCommandStatusInvisible*) отображать ли команду как зафиксированную (*vsCommandStatusLatched*). При задании текущего состояния команды, перечисленные значения можно соединять логическим «или» и передавать обратно через данный аргумент.
- ***CommandText*** В настоящее время этот параметр не используется в Visual Studio .NET, и его нельзя изменять.

Периодически, скажем, когда фокус ввода переходит от одного окна к другому или когда отображается меню, содержащее команду надстройки, Visual Studio .NET вызывает для этой команды метод *QueryStatus,* чтобы убедиться, что пользовательский интерфейс синхронизирован с состоянием команды. Очень важно создавать код, реализующий метод *QueryStatus,* как можно более эффективным, иначе пользовательский интерфейс может стать заторможенным. Предположим,

вы создаете команду, копирующую активный в данный момент файл в некоторый каталог в локальной сети. Может появиться искушение поместить в метод *QueryStatus* этой команды проверку наличия сетевого подключения и поиск в сети целевого каталога. И если и сеть, и каталог назначения присутствуют, сделать команду доступной, а в противном случае — недоступной. Но такие проверки могут занять до минуты и более. Пользователю будет гораздо лучше, если данная команда будет доступна постоянно, а если при попытке выполнить команду сеть или каталог назначения недоступны, он просто получит сообщение об ошибке.

### Определение состояния команды программно

Иногда нужно определять доступность/недоступность команды программно, скажем, при вызове команды через *DTE.ExecuteCommand*. Все команды, и созданные на основе макросов, и созданные надстройкой, и встроенные в Visual Studio .NET, поддерживают метод *QueryStatus,* Когда вы выполняете метод *DTE.ExecuteCommand* и требуемая команда недоступна (метод *QueryStatus* возвращает значение, свидетельствующее о том, что она недоступна в данный момент), возникнет исключение (если используется язык, поддерживаемый .NET Framework).

Проверить доступность команды и избежать возникновения ошибки позволяет свойство *Command.IsAvailable*. Например, чтобы убедиться перед вызовом, что команду *Build.BuildSolution* можно вызывать, используйте такой код:

```
Sub CheckAvailabilityC)
    If (DTE.Commands.Item("Build.BuildSolution").IsAvailable = True) Then
        DTE.ExecuteCommand("Build. BuildSolution")
    End If
End Sub
```

## Как найти обработчик команды надстройки

Когда пользователь вызывает вашу команду, Visual Studio .NET надо определить, какая надстройка обрабатывает данную команду, и тогда она сможет вызывать соответствующие методы интерфейса *IDTCommandTarget*.Сначала анализируется имя команды; как уже отмечалось, первая часть полного имени команды — ProgID надстройки, а оставшаяся часть — значение, переданное методу *AddNamedCommand* в параметре *Name*. Для поиска надстройки Visual Studio .NET выделяет из имени команды ProglD и проверяет, загружена ли надстройка, соответствующая этому ProglD, а если нет, загружает ее. Затем Visual Studio .NET ищет интерфейс *IDTCommandTarget* (который должен быть реализован для объекта, реализуемого *IDTExtensibility2*) экземпляра объекта надстройки и вызывает метод *Exec*, передавая ему первым параметром имя команды.

Если во время этого процесса не удается найти нужную надстройку, пользователю выдается диалоговое окно (рис. 7-2).

Если пользователь щелкнет кнопку Yes, команда будет удалена методом *Command.Delete*, а вместе с ней все ее элементы пользовательского интерфейса. Если надстройку удалось загрузить, но у объекта надстройки не был найден интерфейс *IDTCommandTarget,* считается, что метод *QueryStatus* для этой команды возвратил флаг *vsCommandStatusUnsupported*.

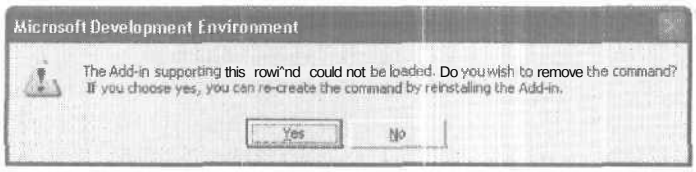



**Рис.** 7-2. *Диалоговое окно, отображаемое Visual Studio .NET, если не удается загрузить надстройку для вызванной команды*

# Пользовательский интерфейс команды

Visual Studio позаимствовала свои панели инструментов и систему меню у комплекта приложений Microsoft Office. Панели команд обеспечивают настраиваемый пользовательский интерфейс, единый для всех приложений Office, то же самое касается и Visual Studio .NET. А поскольку панели команд также поддерживают объектную модель, эти приложения поддерживают и единую модель программирования для доступа к структуре панели команд.

Отправная точка для доступа к объектам панели команд — свойство *DTE.CommandBars*, которое возвращает объект *Microsoft.Office.Core.CommandBars*, определяемый в сборке Office.dll. Следующий код макроса демонстрирует получение этого объекта:

```
Sub GetCommandBars()
    Dim commandBars As Microsoft.Office.Core.CommandBars
    commandBars = DTE.CommandBars
End Sub
```

## Объектная модель панели команд

Объектная модель панели команд имеет древовидную иерархическую структуру, как и объектная модель Visual Studio .NET. В корне дерева располагается набор объектов *Microsoft.Office.Core.CommandBars*, включающий все панели команд и контекстные меню, а также строку главного меню. Каждая панель команд включает набор элементов управления, имеющих тип *Microsoft.Office.Core.CommandBarControl*. После извлечения какого-либо *CommandBarControl* его можно преобразовать к нескольким типам. Первый тип — *CommandBarButton* — это любой элемент на панели управления, который пользователь может щелкнуть для выполнения некоторого действия; это аналогично выполнению команды Visual Studio .NET. Для получения объекта типа *CommandBarButton*, нужно выполнить приведение типа для объекта *CommandBarControl*:

```
Sub GetCommandBarButton()
    Dim commandBarBtn As Microsoft.Office.Core.CommandBarButton
    Dim commandBarCtl As Microsoft.Office.Core.CommandBarControl
    Dim commandBarCtls As Microsoft.Office.Core.CommandBarControls

    'Ищем панель команд View.
    commandBarCtls = DTE.CommandBars.Item("View").Controls
    'Извлекаем из меню первый элемент управления.
```

```
    commandBarCtl = commandBarCtls.Item(1)
    'Преобразуем CommandBarControl в объект CommandBarButton.
    commandBarBtn = CType(commandBarCtl, _
        Microsoft.Office.Core.CommandBarButton)
    MsgBox(commandBarBtn.Caption)
End Sub
```

Объект, возвращаемый из набора *Controls*, может быть преобразован к типу *CommandBarPopup,* если этот элемент — корневой узел подменю. Пример этого — пункт New меню File; когда пользователь помещает указатель над этим меню, появляется подменю. Так же можно получить *CommandBarPopup,* если элемент находится в выпадающем меню кнопки со стрелкой, например, New Project | New Blank Solution на панели команд Standard:

```
Sub GetCommandBarPopup()
    Dim commandBar As Microsoft.Office.Core.CommandBar
    Dim cmdBarControl As Microsoft.Office.Core.CommandBarControl
    Dim cmdBarPopup As Microsoft.Office.Core.CommandBarPopup

    'Ищем панель команд "Standard"
    commandBar = DTE.CommandBars.Item("Standard")
    'Находим первый элемент управления на этой панели
    cmdBarControl = commandBar.Controls.Item(1)
    'Преобразуем CommandBarControl в объект CommandBarPopup
    cmdBarPopup = CType(cmdBarControl, _
        Microsoft.Office.Core.CommandBarPopup)
    MsgBox(cmdBarPopup.Controls.Item(1).Caption)
End Sub
```

Всплывающее меню само является панелью команд. Всплывающее меню нельзя напрямую привести к типу *CommandBar,* но этот объект имеет свойство *CommandBar,* возвращающее объект *CommandBar,* который имеет набор элементов управления (как можно увидеть из предпоследней строки кода предшествующего макроса).

## Главная панель команд

Свойство *DTE.CommandBars* возвращает набор всех объектов типа *CommandBar,* имеющихся в Visual Studio .NET, но наиболее часто используемая панель команд — главное меню, в котором находятся пункты File, Edit, View и ряд дополнительных; все они являются объектами типа *CommandBar* из набора *DTE.CommandBars.* Но, поскольку в наборе может находиться несколько элементов с одинаковыми именами, индексация набора с использованием имен не всегда работает. Так, существует несколько объектов *CommandBar* с именем View, и при индексации набора с помощью строки *View* можно получить не тот, что требуется. Следующий макрос может возвратить панель команд View для редактора SQL, всплывающее меню развертывания проекта или меню View:

```
Sub GetView()
    Dim cmdbars As Microsoft.Office.Core.CommandBars
    Dim commandBar As Microsoft.Office.Core.CommandBar
```

```
    cmdbars = DTE.CommandBars
    commandBar = cmdbars.Item("View")
End Sub
```

Обходя эту ситуацию, вы можете найти объект *CommandBar* для строки меню, который называется *MenuBar,* и затем найти панель команд подменю View:

```
Sub GetMenuCommandBar()
    Dim commandBar As Microsoft.Office.Core.CommandBar
    Dim cmdBarControl As Microsoft.Office.Core.CommandBarControl
    Dim cmdBarPopupView As Microsoft.Office.Core.CommandBarPopup
    Dim cmdBarView As Microsoft.Office.Core.CommandBar

    'Извлечение панели команд МепиВаг.
    commandBar = DTE.CommandBars.Item("MenuBar")
    'Поиск меню View.
    cmdBarControl = commandBar.Controls.Item("View")
    'Преобразование к типу CommandBarPopup
    cmdBarPopupView = CType(cmdBarControl, _
        Microsoft.Office.Core.CommandBarPopup)
    'Получить объект CommandBar для меню View
    cmdBarView = cmdBarPopupView.CommandBar
    MsgBox(cmdBarView.Name)
End Sub
```

По умолчанию, если мастер надстроек генерирует надстройку, которую нужно поместить в меню Tools, генерируется код для размещения пункта меню в меню Tools строки меню. Если нужно переместить пользовательский интерфейс команды в другое меню, можно просто изменить строку «*Tools*» названием другого меню; но в выборе правильного меню надо проявлять осторожность. Легко допустить ошибку, выбрав неправильную панель команд, что приведет к потере из виду кнопки команды, так как она будет размещена там, где вы не ожидаете ее найти.

## Добавление новой панели команд

Имея объект *Command* (найденный индексированием набора *Commands* или полученный при добавлении новой команды) и после использования описанных ранее методов для поиска подходящей панели команд, можно добавить к найденной панели новую кнопку, при щелчке которой будет активизироваться данная команда. Это делает метод *CommandAddControl.* Добавленная этим методом команда сохраняется на диске и при следующем запуске Visual Studio .NET автоматически создастся заново. Точно так же, когда элемент управления добавляется на панель команд методом *AddControl,* этот элемент и его местоположение сохраняются на диске, и при следующем запуске Visual Studio .NET изменения вступают в силу. Первый аргумент метода *AddControl* — объект типа *CommandBar,* куда надо добавить кнопку. Второй аргумент определяет порядковый номер этого элемента управления по отношению к остальным элементам на панели команд (если это значение 1, элемент управления будет первым на панели инструментов, если 2 — вторым и т. д.).

Вы можете потратить время на задание индекса, определяющего место элемента управления, но он так и не появится там, где вы ожидали. Причина в том, что панель

команд может иметь один и более разделителей (линий, нарисованных между двумя соседними элементами), которые разбивают элементы управления на логические группы. Эти разделители сами являются элементами управления на панели инструментов, и их следует учитывать при вычислении требуемой позиции. Но в расчет должны приниматься не только разделители, но и элементы управления, которые в данный момент невидимы, поскольку метод *QueryStatus* возвратил для них значение *vsCommandStatusInvisible*. Если добавляемый элемент управления нужно поместить в конец панели команд, для определения последней позиции можно задействовать свойство *Controls.Count*:

```
Sub AddControl()
    Dim command As EnvDTE.Command
    Dim commandBar As Microsoft.Office.Core.CommandBar

    'Найти команду File.OpenFile.
    command = DTE.Commands.Item("File.OpenFile")
    'Найти панель команд Tools.
    commandBar = DTE.CommandBars.Item("Tools")
    'Добавить в меню Tools элемент управления, при щелчке
    'которого выполняется команда File.OpenFile,
    command.AddControl(commandBar, commandBar.Controls.Count + 1)
End Sub
```

Заметьте: используемый индекс не фиксирует элемент управления в абсолютной позиции. Если вы добавили элемент управления в позицию 1, а затем добавили еще один элемент управления в ту же позицию 1, первый элемент переместится на вторую позицию.

Иногда, если стандартный набор панелей команд не соответствует вашим потребностям, имеет смысл создать новую панель инструментов для размещения ваших кнопок. Объектная модель панели команд позволяет создавать новые панели инструментов, но это может не дать требуемого результата. Создаваемые так панели команд являются временными, т. е., если выйти из Visual Studio .NET и заново запустить ее, они будут уничтожены. А раз уж пользовательский интерфейс кнопок для команд продолжает сохраняться от запуска к запуску, то хочется, чтобы и панели команд также сохранялись от запуска к запуску. Объектная модель Visual Studio .NET позволяет сделать это методом *Commands.AddCommandBar*:

```
public object AddCommandBar(string Name, EnvDTE.vsCommandBarType Type, _
  Microsoft.Office.Core.CommandBar CommandBarParent = null, int Position = 1)
```

Метод имеет следующие аргументы.

- ***Name*** Заголовок, отображаемый на панели команд.
- ***Type*** Значение из перечисления *vsCommandBarType*. Если оно равно *vsCommandBarTypeToolbar*, создаваемая панель команд может пристыковываться к верхнему, левому, нижнему или правому краю окна Visual Studio .NET. Если оно равно *vsCommandBarTypeMenu*, панель команд добавляется в качестве подменю к другой панели команд. Если это значение равно *vsCommandBarTypePopup*, создается контекстное меню.

- ***CommandBarParent*** Если значение, передаваемое параметром *Type,* равно *vsCommandBarTypeToolbar* или *vsCommandBarTypePopup,* данное значение должно быть равно *null* или *Nothing* (в зависимости от используемого языка). Если значение, передаваемое параметром *Type,* равно *vsCommandBarTypeMenu,* вновь создаваемое меню должно иметь корнем объект панели команд.
- ***Position*** Это значение используется, только если параметр *Type* установлен в *vsCommandBarTypeMenu.* Оно определяет положение новой команды меню на родительской панели команд. Данное значение имеет тот же смысл, что и параметр *Position* метода *AddControl.*

Способ отображения вновь созданной панели команд зависит от ее типа. Если это новое меню, соответствующий ей пункт меню будет скрыт от пользователя, пока панель команд для этого пункта меню не будет заполнена кнопками. Если это новая панель инструментов, свойству *Visible* возвращаемого объекта *CommandBar* должно быть задано *True.* Если это всплывающее меню, можно показать его пользователям методом *CommandBar.ShowPopup,* принимающего два аргумента: *x* и *y,* задающие координаты левого верхнего угла всплывающего меню.

## Нестандартные изображения

Ряд стандартных изображений в Visual Studio .NET можно помещать в пункты меню и на кнопки панелей команд, но они не покрывают всех потребностей. Чтобы использовать собственную картинку, нужно зарегистрировать для вашей надстройки сателлитную DLL, содержащую в своих ресурсах нужную картинку (о создании сателлитной DLL см. главу 6), а еще надо изменить вызов метода *Commands.AddNamedCommand,* чтобы Visual Studio .NET могла найти изображение. Во-первых параметру *MSOButton* метода *AddNamedCommand* надо задать *false,* т. е. указать, что нужно использовать не встроенное изображение, а изображение из сателлитной DLL. И во-вторых, параметру *Bitmap* надо задать идентификатор ресурса (ID) данного изображения в сателлитной DLL.

Используемые в Visual Studio .NET изображения должны иметь определенный формат: 16x16 пикселов и 16 цветов. Visual Studio .NET позволяет нарисовать изображение с прозрачными участками. Чтобы сделать участок прозрачным, ему нужно задать цвет 0,254,0 RGB (red, green, blue). (Этот цвет не будет светло-зеленым цветом, отображаемым на цветовой палитре редактора изображений Visual Studio .NET или программы Paint в Windows.)

Создать изображение, имеющее правильный размер и количество цветов, может оказаться непросто. Пример CustomBitmap показывает, как это делается. Установочный проект был изменен, чтобы можно было установить сателлитную DLL, содержащую нестандартное изображение, в нужное место, а также чтобы правильно зарегистрировать данную сателлитную DLL. Установив этот пример с помощью установочного проекта, вы увидите, что изображение, появляющееся на кнопке команды в меню Tools, — пустое. Причина в том, что файл изображения (CustomBitmap\ CustomBitmapUI\untitled.bmp) — пустой шаблон, окрашенный в прозрачный цвет; вы можете использовать его как заготовку при создании собственного изображения. Этот пример также показывает, как изменить вызов метода *AddNamedCommand,* чтобы сослаться на нестандартное изображение. Как будет видно из

следующего кода, мы изменили аргумент *MSOButton* с *true* на *false,* а аргумент *Bitmap* — с 59 на 1 (1 — это идентификатор ресурса нашего изображения в сателлитной сборке):

```
Command command = commands.AddNamedCommand(addInInstance, "CustomBitmap",
    "CustomBitmap", "Executes the command for CustomBitmap", false, 1,
    ref contextGUIDS, (int)vsCommandStatus.vsCommandStatusSupported +
    (int)vsCommandStatus.vsCommandStatusEnabled);
```

# Восстановление потерянной команды и ее пользовательского интерфейса

При разработке надстроек (особенно когда вы только закончили создание надстройки) пользовательский интерфейс команды и сама команда время от времени куда-то пропадают. Это происходит из-за способа сохранения информации, связанной с командой. При закрытии Visual Studio .NET вся информация о местоположении меню и панелей инструментов, включая как встроенные команды Visual Studio .NET, так и команды, созданные вами, сохраняется в файле на диске. Но если происходит что-то помешавшее Visual Studio .NET сохранить этот файл, каша команда и ее пользовательский интерфейс могут потеряться.

Допустим, с помощью мастера вы создали надстройку и начали отлаживать итоговый проект. Если вы закроете отлаживаемый экземпляр Visual Studio .NET, выбрав пункт File Exit (или иным законным способом), информация о панелях инструментов сохранится в файле на диске. Когда вы закроете тот экземпляр Visual Studio .NET, который вы использовали для разработки этой надстройки, его сведения о панелях инструментов также сохранятся, но при этом информация, о местоположении вашей панели инструментов, сгенерированная отлаживаемым экземпляром, будет затерта. Таким образом, вы потеряете всю информацию о команде, созданной вашей надстройкой.

Данная информация может быть потеряна и по другой причине. Предположим, вы запустили мастер надстроек и сгенерировали надстройку, а затем начали ее отладку, нажав клавишу F5. Но, вместо того чтобы потом закрыть отлаживаемый экземпляр Visual Studio .NET (выбрав File | Exit или другим законным способом), вы решили прекратить отладку в отладчике. И этот экземпляр Visual Studio .NET не получил возможности сохранить информацию о командах на диске, т. е. эта информация потерялась.

## devenv /setup

Вы можете воссоздать свою команду несколькими способами. Во-первых, закрыть все экземпляры Visual Studio .NET и в окне командной строки (сеанс MS DOS) ввести **devenv /setup.** Это заставит Visual Studio .NET запуститься и ревизовать всю информацию о панелях команд, удаляя любые команды или объекты *CommandBar*, созданные всеми надстройками. Запуск *devenv /setup* также приведет к очистке соответствующих флагов, чтобы при следующем запуске Visual Studio .NET заставить надстройки заново создать информацию о своих командах.

**Внимание!** *devenv /setup* вызывает нежелательные побочные эффекты: при обновлении информации о командах: все команды, созданные надстройками, а также вся настройка команд (например, перемещение кнопки с одной панели команд на другую) будут потеряны.

Воссоздать команду позволяет также надстройка ResetCmdBarInfo, включенная в файлы примеров этой книги. ResetCmdBarInfo создает команду в меню Tools. Выбор данного пункта меню приводит к созданию экземпляра Visual Studio .NET, для которого задан параметр командной строки */setup*. Когда Visual Studio .NET заново создаст информацию о панелях инструментов, она закончит работу. Это имеет такой же эффект, как ввод команды **devenv /setup** в командной строке.

## Нестандартнаярегистрация

Другой способ повторного создания команды — очистить флаг *CommandPreload* (см. главу 6), приводящий к регистрации вашей надстройки во время компоновки. Если компонуемая надстройка написана на Visual C++, можно изменить код регистрации надстройки, чтобы очистить этот флаг в файле .rgs, но . NET-сборка регистрируется не так, как COM-объекты Visual C++. Выполнить нестандартную системную регистрацию при регистрации сборки как COM-объекта позволяют атрибуты .NET. Библиотеки . NET Framework содержат два атрибута, относящихся к пространству имен *System.Runtime.InteropServices*: *ComRegisterFunctionAttribute* и *ComUnregisterFunctionAttribute*. Эти атрибуты при размещении их перед статическими (C#) или общими (Visual Basic .NET) открытыми методами, принимающими аргумент типа *System.Type*, вызываются во время RegAsm-фазы регистрации .NET-компонента как COM-объекта (это происходит при каждой компоновке проекта надстройки). Таким образом, вы включаете в объявление класса надстройки подобный код:

```
[ComRegisterFunctionAttribute]
public static void RegisterFunction(Type t)
{
    string progID = String.Empty;

    foreach (System.Attribute attrib in t.GetCustomAttributes(false))
    {
        if (attrib.GetType().FullName ==
            "System.Runtime.InteropServices.ProgIdAttribute")
        {
            ProgIdAttribute progIdAttrib = (ProgIdAttribute)attrib;
            progID = progIdAttrib.Value;
        }
    }

    if (progID != String.Empty)
    {
        RegistryKey key = Registry.CurrentUser.OpenSubKey(
            @"Software\Microsoft\VisualStudio\7.1\AddIns\" + progID,
            true);
```

```
        if (key != null)
        {
            if (((int)key.GetValue("CommandPreload", -1)) == 2)
            {
                key.SetValue("CommandPreload", 1);
            }
        }

        key = Registry.CurrentUser.OpenSubKey(
            @"Software\Microsoft\VSA\7.1\AddIns\" + progID, true);

        if (key != null)
        {
            if (((int)key.GetValue("CommandPreload", -1)) == 2)
                key.SetValue("CommandPreload",1 ) ;
        }

        key = Registry.CurrentUser.OpenSubKey(
            @"Software\Microsoft\VisualStudio\7.1\PreloadAddinState",
            true);

        if (key != null)
        {
            key.DeleteValue(progID, false);
        }

        key = Registry.CurrentUser.OpenSubKey(
            @"Software\Microsoft\VSA\7.1\PreloadAddinState", true);

        if (key != null)
        {
            key.DeleteValue(progID, false);
        }
    }
}

[ComUnregisterFunctionAttribute]
public static void UnregisterFunction(Type t)
{
    string progID = String.Empty;

    foreach (System.Attribute attrib in t.GetCustomAttributes(false))
    {
        if (attrib.GetType().FullName ==
            "System.Runtime.InteropServices.ProgIdAttribute")
        {
            ProgIdAttribute progIdAttrib = (ProgIdAttribute)attrib;
            progID = progIdAttrib.Value;
        }
    }
```

```
    if (progID != String.Empty)
    {
        RegistryKey key = Registry.CurrentUser.OpenSubKey(
            @"Software\Microsoft\VisualStudio\7.1\PreloadAddinState",
            true);

        if (key != null)
        {
            key.DeleteValue(progID, false);
        }

        key = Registry.CurrentUser.OpenSubKey(
            @"Software\Microsoft\VSA\7.1\PreloadAddinState", true);

        if (key != null)
        {
            key.DeleteValue(progID, false);
        }
    }
}
```

Что же делает этот код? Первые несколько строк каждой из функций, используя отражение, ищут класс, внутри которого реализована сама эта функция. Затем идет код, перебирающий список атрибутов этого класса в поиске атрибута, устанавливающего ProgID надстройки при ее регистрации в качестве COM-объекта — атрибута *ProgIdAttribute*. Когда этот атрибут найден, из него извлекается ProgID надстройки. Оставшаяся часть кода просто проверяет определенные значения реестра; если они присутствуют и установлены в 2. они сбрасываются обратно в 1. Сбрасываются и значения для редактора макросов Visual Studio .NET и самой Visual Studio .NET (если они есть). Если одно из этих приложений не связано с вашим приложением, вы должны удалить связанную с ним проверку реестра. Точно так же. если появятся иные приложения. построенные по технологии Visual Studio .NET и вы зададите, чтобы ваша надстройка загружалась для этих приложений, вам придется добавить соответствующую проверку значений в реестре для этих приложений. Надстройка ReRegisterCS из файлов примеров книги демонстрирует сброс данных флагов.

**Замечание** Этот метод регистрации будет восстанавливать только потерянные команды, но не приведет к полному созданию всех необходимых параметров регистрации надстройки. Этот метод позволяет добавить к регистрации сборки собственный код, но приведенный здесь пример этого не делает.

Эти методы полезны и в другой ситуации, кроме восстановления потерянных команд: для перемещения пользовательского интерфейса панели команд. Допустим, вы получили стандартный код настройки с помощью мастера надстроек, размещающего кнопку в меню Tools. Однако затем вы решили, что пользовательский интерфейс для вашей команды должен находиться на другой панели команд, скажем, в меню View. После изменения кода для нахождения требуемого меню вы

должны очистить флаг *CommandPreload*, чтобы запустить код для размещения элементов пользовательского интерфейса.

### Производительность надстройки

Чтобы убедиться, что ваши команды созданы корректно, и проработать проблемы, которые мы обсудили, вы можете сделать так, чтобы ваша надстройка загружалась при запуске Visual Studio .NET, вызывать метод *AddNamedCommand* для создания своих команд после загрузки надстройки, а затем вызывать метод *Delete*, чтобы удалить эти команды при выгрузке надстройки. Не делайте так — производительность упадет. Группа автоматизации Visual Studio .NET разработала команды так, чтобы пользователь надстройки не страдал от наличия у него установленных надстроек.

Допустим, пользователь имеет 10 установленных надстроек Если каждая будет загружаться при запуске, придется создать, инициализировать и выполнить код 10 разных компонентов и только для того, чтобы сделать доступным пользовательский интерфейс этих надстроек, даже если пользователь не собирается с ними работать. Было бы не так страшно загружать надстройки при запуске, но пользователь будет вынужден ждать запуска Visual Studio .NET заметно дольше. Проблема команд, не сохраняющихся от одного экземпляра Visual Studio .NET до другого, — это проблема разработчика, и лишь в исключительно редких случаях она видна пользователю.

Однажды пользователь обратился в Microsoft по поводу ошибки: Visual Studio .NET запускалась больше минуты. Microsoft привела все стандартные причины замедления производительности: сильная фрагментация диска, фоновые процессы, отнимающие слишком много процессорного времени, слишком малый объем оперативной памяти. А проблема заключалась в надстройке, загружаемой при запуске и заново создающей свои команды при каждой загрузке.

## Удаление команды

Можно удалить только команды, созданные надстройками; команды, определенные Visual Studio .NET, и команды макросов удалить нельзя, при попытке сделать это будет сгенерирована ошибка. Команды удаляются методом *Command.Delete*, он удаляет не только имя команды, но и любые элементы пользовательского интерфейса, связанные с этой командой, например, кнопки на панели команд.

```
Sub DeleteMyCommand()
    DTE.Commands.Item("MyAddin.Connect.MyCommand").Delete()
End Sub
```

При удалении надстройки должны быть удалены любые команды, созданные ею. но по умолчанию код, генерируемый мастером надстроек, этого не делает Для удаления команд нужно добавить к компоненту-надстройке *нестандартное действие*, т. е. код, выполняемый проектом установки (который создан мастером надстроек) при установке и/или удалении. При установке и удалении инсталлятор загружает компонент-надстройку и ищет открытый класс, производный от *System.Con-*

*ftgurationJnstallJnstaller* и использующий атрибут *RunInstaller(true).* Если таковой находится, вызываются его методы *Install* и *Uninstall,* что позволяет выполнить любой нестандартный код. Создать компонент подобного типа несложно. В диалоговом окне Add New Item вы можете вставить элемент проекта Installer Class (рис. 7-3).

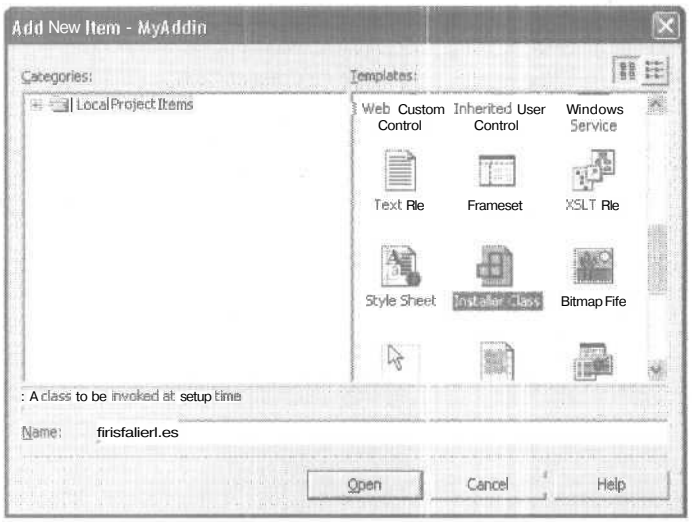



**Рис. 7-3.** *Диалоговое окно Add New Item с выбранным элементом Installer Class*

Следующий ниже код реализует методы *Uninstall*и *Install* класса *Installer.* Эти методы сначала проверяют отсутствие работающих экземпляров процесса devenv.exe (выполняемый файл для Visual Studio .NET). Если какие-либо экземпляры запущены, пользователю предлагается закрыть их, и после закрытия нестандартный обработчик запускает новый экземпляр Visual Studio .NET с помощью модели автоматизации. Метод *Uninstatt,* продолжающий выполнение, вызывает метод *Delete* для удаления команды. Вы можете изменить этот код для поиска и удаления команд, созданных вашей надстройкой.

Этот код также помогает установить надстройку методом *Install,* Когда в реестре установлен флаг *CommandPreload,* при следующем запуске Visual Studio .NET разрешает надстройкам создавать команды. Вы уже видели, с какими проблемами можно столкнуться, если Visual Studio .NET будет закрыта неожиданно или если будет создано несколько ее экземпляров, прежде чем информация о команде может быть записана на диск. Используя нестандартное действие, можно создать экземпляр Visual Studio .NET посредством модели автоматизации и затем сразу же закрыть этот процесс. Это позволит Visual Studio .NET создать любые команды и сохранить эти данные, что предохранит вашего пользователя от попадания в ловушку перезаписи информации о команде.

```
bool IsVSRunning()
{
    System.Diagnostics.Process []processes =
        System.Diagnostics.Process.GetProcesses();
    foreach(System.Diagnostics.Process process in processes)
    {
```

```
        //Помещаем это в try/catch. Если процесс
        //не принадлежит пользователю, при попытке получить
        //ProcessModule будет сгенерировано исключение,
        try
        {
            if((process != null) && (process.MainModule != null))
            {
                //Получить имя файла процесса
                //и сравнить его с 'devenv.exe'.
                string fileName = process.MainModule.FileName;
                fileName = System.IO.Path.GetFileName(fileName);
                if(System.String.Compare(fileName, "devenv.exe") == 0)
                    return true;
            }
        }
        catch(System.Exception ex)
        {
        }
    }
    return false;
}

public override void Uninstall(System.Collections.IDictionary savedState)
{
    base.Uninstall(savedState);
    System.Windows.Forms.MessageBox.Show("Uninstall");
    while(IsVSRunning())
    {
        System.Windows.Forms.MessageBox.Show("A running instance of Visual " +
            Studio .NET was found.\n\nPlease close all copies of Visual " +
            "Studio .NET, then press OK.", "Uninstall...",
            System.Windows.Forms.MessageBoxButtons.OK);
    }
    EnvDTE.DTE dte = new EnvDTE.DTEClass();
    //Измените имя следующей команды
    //на имя созданной вами команды.
    //Это можно делать несколько раз
    //по числу созданных команд.
    dte.Commands.Item("MyAddin1.Connect.MyAddin1",-1).Delete();
}

public override void Install(System.Collections.IDictionary stateSaver)
{
    base.Install(stateSaver);
    System.Windows.Forms.MessageBox.Show("Install");
    while(IsVSRunning())
    {
        System.Windows.Forms.MessageBox.Show("A running instance of Visual " +
            "Studio .NET was found.\n\nPlease close all copies of Visual " +
            "Studio .NET, then press OK.", "Install...",
```

```
            System.Windows.Forms.MessageBoxButtons.OK);
    }
    EnvDTE.DTE dte = new EnvDTE.DTEClass();
}
```

После добавления этого компонента к вашему проекту надстройки останется добавить его к списку нестандартных действий, доступных для установщика. Чтобы сделать это, нужно открыть редактор нестандартных действий (Custom Actions editor) проекта установщика: щелкните правой кнопкой установочный проект и выберите View | Custom Actions. Затем добавьте основной выходной поток (primary output) проекта надстройки к узлам Uninstall и Install этого редактора, щелкая правой кнопкой каждый из узлов в окне Custom Actions, выбирая Add Custom Action, дважды щелкая элемент Application Folder в диалоговом окне Select Item In Project и выбирая Primary Output From Addin Project Name.

Нет нужды добавлять нестандартное действие к проекту установщика, чтобы удалить свои команды при деинсталляции. Удаление ненужных команд удобно для пользователя, но вам не надо вручную удалять созданные вами команды. При деинсталляции надстройки остаются все ее команды вместе с их пользовательским интерфейсом. Если пользователь попытается выполнить одну из команд, для которых не может быть найдена обрабатывающая их надстройка, появится диалоговое окно (рис. 7-2), Если пользователь щелкнет Yes, эта команда и все элементы ее пользовательского интерфейса будут удалены.

## Параметры команд

Команды, создаваемые вами для надстроек, могут принимать параметры. Можно передать параметры обработчику команды, если команда вызывается из командного окна или методом *DTE.ExecuteCommand*.Допустим, ваша надстройка создала команду по имени *MyAddin.Connect.Command*. Если пользователь напечатает в командном окне:

```
MyAddin.Connect.Command
MyAddin.Connect.Command My Parameters
```

то метод *IDTCommandTarget.Exec* вашей надстройки будет вызван дважды. Первый раз параметр *VarIn* получит пустую строку. Во второй раз *VarIn* будет содержать строку *«My Parameters»*. Все, что находится после имени команды (кроме начальных и конечных символов пустых промежутков), дословно копируется в параметр *VarIn*. Заботы по выделению из этого списка поддерживаемых параметров и их обработке возлагаются на надстройку. Параметры также можно передать при активизации команды методом *ExecuteCommand*; до сих пор мы видели передачу методу *ExecuteCommand* только одного аргумента — полного имени команды. Однако ему можно передать и второй, необязательный аргумент — параметр команды, который передается обработчику команды точно так же, как и при передаче данных команде, вызываемой в командном окне.

Команды, имеющие параметры, часто требуют ввода этих параметров согласно определенному синтаксису. Для ознакомления с ним пользователь может заглянуть в документацию, но ему проще обратиться за справкой к самой команде. Общий формат обращения за помощью к утилитам командной строки Windows —

напечатать имя требуемой программы и аргумент /? вслед за ним. Командное окно Visual Studio .NET также поддерживает подобные справки. Если пользователь в командном окне напечатает имя команды и параметр /? то будет вызван наш метод *IDTCommandTarget.Exec* с параметром *ExecuteOption*, к значению которого операцией логического сложения будет добавлено значение *vsCommandExecOption.vsCommandExecOptionShowHelp*. Получив это значение, ваша надстройка должна выбрать наилучший способ отображения справки, например, отобразить Web-страницу с документацией. Надстройка CommandHelp из файлов примеров книги демонстрирует отображение справки при задании для команды параметра /?.

Макросы также могут принимать параметры. Для определения макроса с параметрами нужно сделать все аргументы необязательными, строковыми, с заданными по умолчанию значениями:

```
Sub OptionalArguments(Optional ByVal optArg As String = "I am optional")
    MsgBox(optArg)
End Sub
```

Этот макрос можно вызвать из командного окна или методом *ExecuteCommand*, используя либо только имя макроса, либо имя и аргументы. Если при вызове макроса аргумент не задан, пользователю выдается окно с сообщением «I am optional», Если аргумент задан, в окне сообщения используется текст аргумента. Макросы не имеют специальных возможностей, связанных с активизацией справки, но их можно смоделировать, проверяя передаваемое значение аргумента: если это строка «/?», пользователь запрашивает справку. Для подключения работы со справкой приведенный выше макрос можно переписать:

```
Sub OptionalArguments2(Optional ByVal optArg As String = "I am optional")
    If (optArg = "/?") Then
      Dim helpString As String
      helpString = "Usage: OptionalArguments [optArg]"
      helpString = helpString + vbLf
      helpString = helpString + "This macro will display in " + _
          "a message box the passed argument."
      MsgBox(helpString)
    Else
      MsgBox(optArg)
    End If
End Sub
```

## Привязка к клавишам

Мы рассмотрели массу способов вызова команды, но каждый из них требует либо перехода на мышь, либо набора длинной строки команды. Для упрощения доступа многим командам назначены комбинации клавиш. Так, вместо того чтобы мышью открывать меню File, затем выбирать подменю Open и щелкать File, можно просто использовать комбинацию Ctrl+O, и окно Open File откроется,

Каждой команде можно назначить одну или более комбинаций клавиш. Свойство *Command.Bindings* позволяет читать или устанавливать комбинации для данной команды. При обращении к этому свойству для извлечения набора установ-

ленных комбинаций возвращается массив объектов, содержащий строки с описанием каждой комбинации, который можно перебирать в поиске каждой комбинации. Данное свойство позволяет назначить комбинации клавиш; допускается задание как отдельной строки для установки одной комбинации, так и массива строк для одновременной установки нескольких комбинаций. Вот как получить комбинацию клавиш:

```
Sub GetFileOpenBindings()
    Dim bindings As Object()
    Dim binding As Object
    bindings = DTE.Commands.Item("File.OpenFile").Bindings
    For Each binding In bindings
        MsgBox(CStr(binding))
    Next
End Sub
```

Каждая установленная комбинация имеет определенный формат, и вы должны строго придерживаться этого формата во избежание возникновения ошибок. Вот этот формат:

```
Область действия::МодифицирующаяКлавиша+Клавиша,
    МодифицирующаяКлавиша2 +Клавиша2
```

А вот описание его частей.

- Область действия — контекст, в котором можно применять данную комбинацию клавиш. Так, нажатие Ctrl+B в HTML-редакторе сделает выделенный текст полужирным; если же активен редактор изображений (загружено изображение и окно активно), эта комбинация выбирает в качестве инструмента кисть. Если не активен ни HTML-редактор, ни редактор изображений, Ctrl+B отображает диалоговое окно с точками прерывания отладчика. Существует ряд контекстов, но самый распространенный — Global. При этом контексте команда доступна везде, если только не активен более узкий контекст. Сочетание Ctrl+B отображает точки прерывания отладчика, если нажимается в контексте Global, тогда как в контексте HTML Editor Design View оно приводит к полужирному начертанию текста, и, когда активен HTML-редактор, последний вариант использования данной комбинации имеет преимущество.
- Модифицирующие клавиши — системные клавиши, удерживаемые в нажатом состоянии, когда нажимается другая клавиша. Это Alt, Shift и Ctrl. Можно задать одну или более модифицирующих клавиш; они могут появляться в любом порядке, но должны быть разделены символом «+».
- Клавиша — любая клавиша стандартной 102-клавишной клавиатуры (что исключает экзотические клавиши новой Windows-клавиатуры, такие как клавиша Windows, клавиши мультимедиа, а также Fn на клавиатурах некоторых ноутбуков). Если клавиша является системной, скажем, F1, F2 и т. д. или Esc, то модифицирующая клавиша не требуются. Если клавиша — алфавитно-цифровая, модифицирующая клавиша обязательна. Это не может быть одна клавиша Shift — необходимо включить еще или Alt, или Ctrl, или обе, так как некоторые сочетания клавиш зарезервированы и не могут быть перезаписаны (например, A и Shift+A зарезервированы для текстового редактора). Клавиши,

которые можно использовать, включают алфавитно-цифровые клавиши, символы, не требующие нажатия Shift (вроде запятой или точки), и значения Bkspce, Del, Ins. пробел, функциональные от F1 до F24, Enter. Break, стрелка-вверх, стрелка-вниз, стрелка-влево, стрелка-вправо, Tab. Home, End, PgUp, PgDn, Esc, NUM *, NUM -, NUM + и NUM /, Клавиши Caps Lock, Print Screen, Num Lock и Pause зарезервированы операционной системой, и их нельзя использовать.

- МодифицирующаяКлавиша2 и Клавиша2 образуют другой набор сочетаний клавиш. Они необязательны и могут служить для уточнения команд, входящих в группу. Так, может быть надстройка, использующая сочетание Ctrl+Shift+D, O для открытия файла в некотором окне инструментов и сочетание Ctrl+Shift+D, S для сохранения этого файла. Для Клавиши2 нет ограничений по ее группированию с модифицирующими клавишами, а значит, модифицирующая Клавиша2 необязательна.

Используя этот формат, можно назначить комбинацию клавиш Ctrl+Shift+D,O в контексте Global, применяя код, подобный следующему:

```
Sub SetKeyBinding()
    Dim command As Command
    command = DTE.Commands.Item("File.OpenFile")
    command.Bindings = "Global::Ctrl+Shift+D,O"
End Sub
```

Это обращение к свойству *Bindings* устанавливает только одну комбинацию клавиш для команды, но за один раз можно назначить и несколько сочетаний, передавая массив элементов типа *System.Object,* присвоив каждому из элементов значение типа *System.String:*

```
Sub SetKeyBindings()
    Dim command As Command
    Dim bindings(1) As Object
    bindings(0) = "Global::Ctrl+Shift+D,O"
    bindings(1) = "Global::Ctrl+Shift+D,F"
    command = DTE.Commands.Item("File.OpenFile")
    command.Bindings = bindings
End Sub
```

Когда делаются изменения в привязках комбинаций клавиш к командам, все комбинации, связанные с данной командой удаляются, и, если какие-то из назначаемых комбинаций уже используются другой командой, они также удаляются. После запуска подобных макросов пользователи, привыкшие применять Ctri+O для открытия файла, могут удивиться, почему знакомые им комбинации исчезли, однако эти параметры можно сохранить так:

```
Sub SetKeyBindingsPreserve()
    Dim command As Command
    Dim bindings() As Object
    Dim preserveLength As Integer

    command = DTE.Commands.Item("File.OpenFile")
    'Извлечь текущие комбинации для команды.
    bindings = command.Bindings
```

```
    'Определить их количество
    preserveLength = bindings.Length
    'Добавить к массиву 2 элемента (помните, preserveLength
    'будет на 1 больше, чем количество элементов, и,
    'добавляя к нему 1, мы увеличим массив на 2 элемента).
    ReDim Preserve bindings(preserveLength + 1)
    'Назначим и наши новые комбинации.
    bindings(preserveLength) = "Global::Ctrl+Shift+D,O"
    bindings(preserveLength + 1) = "Global::Ctrl+Shift+D,F"
    command.Bindings = bindings
End Sub
```

При попытке изменить связанные с командой комбинации клавиш вы можете получить исключение с сообщением: «Setting key bindings valid only when using a non-built-in key binding scheme» (установка комбинаций клавиш допустима, только когда используется схема комбинаций клавиш, не являющаяся встроенной). Это исключение вызвано тем, что схемы комбинаций клавиш, зашитые в Visual Studio .NET, доступны только для чтения. Чтобы изменить схему, вы или ваш пользователь должны сначала переключиться на схему клавиш, определяемую пользователем. Это можно сделать через пользовательский интерфейс — в диалоговом окне Options на вкладке Environment | Keyboard, щелкнув кнопку Save As, а также присвоив свойству Scheme из набора свойств Environment | Keyboard значение пути. Если в момент установки этого значения файл .vsk (схемы клавиатуры сохраняются в файле с расширением .vsk) не существовал, он будет создан.

Следующий макрос создает файл схемы клавиатуры для Visual Studio .NET в каталоге данных приложения пользователя:

```
Sub ChangeKeyboardScheme()
    Dim props As EnvDTE.Properties
    Dim prop As EnvDTE.Property
    Dim path As String
    Dim folder As System.Environment.SpecialFolder

    'Найти свойство Scheme из набора свойств Environment, Keyboard,
    props = DTE.Properties("Environment", "Keyboard")
    prop = props.Item("Scheme")
    'Задать путь к месту, где хотим сохранить информацию.
    folder = System.Environment.SpecialFolder.ApplicationData
    path = System.Environment.GetFolderPath(folder)
    path = path + "\Microsoft\VisualStudio\7.1\scheme.vsk"
    'Установить путь.
    prop.Value = path
End Sub
```

### Практикум: создание команды и краткого справочника комбинаций клавиш

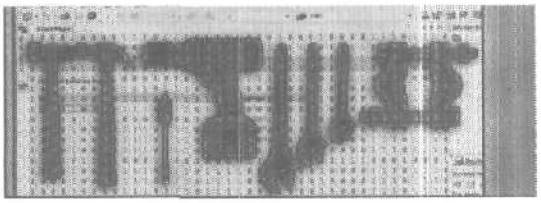
Мне нравится иметь на руках распечатку всех команд Visual Studio .NET и назначенных им сочетаний клавиш. Когда я ловлю себя на том, что использую какую-либо команду неоднократно, то заглядываю в эту распечатку и пы-

таюсь запомнить сочетание клавиш, активизирующее эту команду. Поиск нужной команды в справочнике требует времени, но это окупается ростом производительности. Вот макрос, с помощью которого я создаю свой кратки й справочник:

```
Sub GenerateCommandCheatSheet()
        Dim cmd As Command
        Dim selection As TextSelection
        Dim binding As String
        Dim newDocument As Document
        'Открыть новый документ, где будет храниться информация.
        newDocument = DTE.ItemOperations.NewFile( _
            "General\Text File").Document
        selection = newDocument.Selection
        'Перебор всех команд:
        For Each cmd In DTE.Commands
            If (cmd.Name <> Nothing) Then
                Dim str As String
                'Получить имя команды и отформатировать
                'для получения красивого результата:
                str = cmd.Name
                str = str.PadRight(70)
                'Вывести имя команды и ее комбинации клавиш.
                If (cmd.Bindings.Length > 0) Then
                    For Each binding In cmd.Bindings
                        str = str + binding
                        selection.Insert(str + vbLf)
                        str = ""
                        str = str.PadRight(70)
                    Next
                Else
                    selection.Insert(str + vbLf)
                End If
            End If
        Next
    End Sub
```

# Что дальше

В следующей главе мы сосредоточимся на создании и изменении решений и проектов Visual Studio .NET с помощью объектной модели. Вы также узнаете, как работать с измененными таким образом решениями и как решения и проекты компилируются в работающие программы.



ГЛАВА

# 8

# Программное управление решениями и проектами

Microsoft Visual Studio .NET богата инструментами. Один из таких инструментов — система управления проектами. В проектах можно создавать файлы и управлять ими, а также компилировать для получения конечной программы. Их этой главы вы узнаете, как управлять проектами, применяя объектную модель автоматизации.

## Работа с решениями

В Visual Studio .NET решение (solution) является основной единицей управления проектами, Решение — это контейнер для любого количества проектов, работающих совместно и формирующих единую программу. Каждый проект в решении может содержать файлы с кодами, компилирующиеся при создании программы, папки для облегчения управления файлами и ссылки на другие программные компоненты, которые могут использоваться в проекте. Решение управляется через окно Solution Explorer, в котором можно добавить, удалить и изменить проекты и содержащиеся в них файлы. При открытии файла решения для него создается узел в Solution Explorer, а каждый проект, добавленный в решение, отображается как вложенный узел.

В объектной модели Visual Studio .NET решение представляется объектом *EnvDTESolution*, который можно получить через свойство *Solution* объекта *DTE*:

```
Sub GetSolution()
    Dim solution As EnvDTE.Solution
    solution = DTE.Solution
End Sub
```

## Создание, загрузка и выгрузка решений

Чтобы задействовать объект *Solution* с его методами и свойствами, вам не потребуется создавать или открывать файл решения. Можно использовать объект *Solution* даже если узел решения в Solution Explorer не отображается. В Visual Studio .NET всегда есть открытое решение, даже если оно существует только в памяти, а не на диске. Если решение в памяти не изменялось со времени последнего сохранения, то при открытии файла решения с диска оно выгружается и заменяется решением с диска. Если решение в памяти изменялось (например, в него добавлен новый или существующий проект), то при его закрытии вам будет задан вопрос о сохранении решения на диске.

Чтобы сохранить решение из программы, можно вызвать метод *Solution.SaveAs*, передав ему полный путь (включая имя файла и расширение .sln), указывающий, где на диске сохранить решение. Однако это не всегда допустимо и может вызывать исключение, поскольку до того, как метод *Solution.SaveAs* можно будет использовать, файл решения должен быть сохранен на диске или загружен из существующего файла. Чтобы разрешить сохранение файла решения можно задействовать метод *Create*. Он применяется для указания такой информации, как место сохранения файла и имя решения. Комбинируя методы *Create* и *SaveAs*, можно не только создавать решение, но и сохранять его:

```
Sub CreateAndSaveSolution()
    DTE.Solution.Create("C:\", "Solution")
    DTE.Solution.SaveAs("C:\Solution.sln")
End Sub
```

Файл решения, сохраненный на диске через пользовательский интерфейс или объектную модель, можно открыть методом *Solution.Open*. Используя путь к файлу из макроса CreateAndSaveSolution, можно открыть наше решение:

```
DTE.Solution.Open("C:\Solution.sln")
```

При вызове этого метода текущее решение выгружается, и открывается указанный файл решения. Когда при загрузке файла решения закрывается текущее решение, вы не получите уведомления об этом, даже если текущее решение было изменено. Это значит, что вы не сможете сохранить изменения. Макрос или надстройка может задействовать свойство *ItemOperations.PromptToSave*, чтобы предложить сохранение решения. Объект *ItemOperations*, доступный из свойства *DTE.ItemOperations*, содержит методы и свойства для операций с файлами. Одно из свойств — *PromptToSave* — отображает диалоговое окно, позволяющее выбрать файлы для сохранения, и возвращает значение, указывающее на нажатую кнопку. Кроме того, оно сохраняет выбранные файлы. Если файлы сохранять не надо, это свойство не будет выводить диалоговое окно, а сразу вернет значение, указывающее на нажатие кнопки OK. Корректно сохранить измененные файлы и открыть решение позволяет комбинация свойства *PromptToSave* и метода *Open*:

```
Sub OpenSolution()
    Dim promptResult As vsPromptResult
    'Предлагаем сохранить все открытые и измененные файлы:
    promptResult = DTE.ItemOperations.PromptToSave
```

```
    'Если пользователь не нажал кнопку Cancel,
    'открываем файл решения с диска:
    If promptResult <> vsPromptResult.vsPromptResultCancelled Then
        DTE.Solution.Open("C:\Solution.sln")
    End If
End Sub
```

Теперь вы знаете как создавать, сохранять и открывать решение. Единственный упущенный этап жизненного цикла решения — его закрытие. Объект *Solution* поддерживает метод *Close,* который можно применять для закрытия файла решения. Этот метод принимает необязательный параметр типа *Boolean,* позволяющий предписать Visual Studio .NET сохранить файл при закрытии. Если в нем передать *true,* то перед закрытием файл решения будет сохранен; если *false* — любые изменения будут отклонены.

## Перебор проектов

Объект *Solution* представляет собой набор объектов *Project* и, будучи набором, содержит метод *Hem,* через который можно найти проект в решении. Как и методы *Item* других объектов, он поддерживает числовую индексацию, а еще и передачу строки для поиска проекта. Однако версия метода *Solution.Item* со строкой отличается от подобных версий других методов *Item. Solution.Item* принимает не имя проекта, а уникальное имя. Уникальное имя, как следует из названия, однозначно определяет проект в решении. Уникальные имена служат для индексации набора проектов потому, что Visual Studio .NET допускает загрузку двух проектов с одинаковым именем, расположенных в разных папках на диске. (Visual Studio .NET 2003 требует, чтобы все проекты в решении имели разные имена.) Так как в Visual Studio .NET версии 7.1 и более поздних допускается загрузка нескольких проектов с одинаковым именем, то для того чтобы отличить один проект от другого при вызове метода *Item,* одного имени недостаточно. Получить уникальное имя проекта можно из свойства *Project.UniqueName.* Следующий макрос получает это значение для всех проектов, загруженных в решение:

```
Sub EnumProjects()
    Dim project As EnvDTE.Project
    For Each project In DTE.Solution
        MsgBox(project.UniqueName)
    Next
End Sub
```

Объект *Solution* — не единственный набор всех загруженных проектов. Объект *Solution* содержит свойство *Projects,* которое тоже возвращает набор доступных проектов, и при переборе и индексации проектов работает так же, как объект *Solution.* Реализация одинаковой функциональности в двух разных местах может показаться избыточной, но в своих исследованиях команда, работавшая над объектной моделью Visual Studio .NET, обнаружила, что разработчики не распознают объект *Solution* как набор. Поэтому, чтобы упростить разработчикам поиск набора проектов, они добавили этот набор *Projects.* Макрос *EnumProjects* можно переписать для использования набора *Projects.*

```
Sub EnumProjects2()
    Dim project As EnvDTE.Project
    For Each project In DTE.Solution.Projects
        MsgBox(project.UniqueName)
    Next
End Sub
```

В наборах *Solution* и *Projects* можно найти список проектов, но иногда вам потребуется искать проекты, выбранные в древовидном списке окна Solution Explorer. Это позволяет сделать свойство *DTE.ActiveSolutionProjects*. При вызове этого свойства Visual Studio .NET определяет выбранные в Solution Explorer элементы. Если выбран узел проекта, к возвращаемому списку объектов добавляется объект *Project* для выбранного проекта. Если выбран элемент проекта, то к возвращаемому списку также добавляется проект, содержащий этот элемент. Затем из списка удаляются все дубликаты, и он возвращается. Вот макрос, демонстрирующий использование этого свойства:

```
Sub FindSelectedProjects()
    Dim selectedProjects As Object()
    Dim project As EnvDTE.Project
    selectedProjects = DTE.ActiveSolutionProjects
    For Each project In selectedProjects
        MsgBox(project.UniqueName)
    Next
End Sub
```

## Захват событий решения

При работе с решениями Visual Studio .NET инициирует события, позволяющие надстройкам и макросам получать уведомления о выполняемых действиях. Эти события инициируются через объект *SolutionEvents*, доступ к которому можно получить через свойство *Events.SolutionEvents*. Захват событий решения выполняется, как обычно, — открытием модуля *EnvironmentEvents* любого проекта макроса, выбором объекта *SolutionEvents* в левом раскрывающемся списке из верхней части окна редактора кода, а также выбором имени события в правом раскрывающемся списке этого окна.

Вот сигнатуры и описания событий решения.

- ***void Opened()*** Инициируется сразу после открытия файла решения.
- ***void Renamed(string OldName)*** Этот обработчик события вызывается сразу после переименования файла решения на диске. Единственный аргумент, передаваемый обработчику, указывает полный путь к файлу решения до его переименования.
- ***void ProjectAdded(EnvDTE.Project Project)*** Это событие инициируется при добавлении проекта в решение. Аргумент, передаваемый обработчику, — объект *EnvDTEProject* для добавленного проекта.
- ***void ProjectRenamed(EnvDTE.Project Project, string OldName)*** Это событие инициируется после переименования проекта в решении. Обработчику передаются два аргумента. Первый аргумент типа *EnvDTEProject* представляет

объект для переименованного проекта. Второй — это строка, содержащая полный путь к файлу проекта до его переименования.

- ***void ProjectRemoved(EnvDTE.Project Project)*** Это событие инициируется прямо перед удалением проекта из решения. Обработчик принимает в виде аргумента объект *EnvDTE Project* для удаляемого проекта. Как и в случае с событием *BeforeClosing*, не следует в обработчике изменять удаляемый проект, так как он уже был сохранен на диске (если вы указали, что файл следует сохранять) перед удалением и любые изменения проекта будут сброшены.
- ***void QueryCloseSolution(reboo fCancel)*** Это событие инициируется непосредственно перед тем, как Visual Studio .NET начинает закрывать файл решения. Обработчику передается один аргумент — ссылка на переменную типа *Boolean*. Надстройка или макрос могут блокировать закрытие решения, установив этот параметр в *true*, или разрешить закрытие, установив его в *false*. Не следует злоупотреблять запретом на закрытие решения — пользователи могут быть неприятно удивлены, если макрос или надстройка будут препятствовать закрытию решения.
- ***void BeforeClosing()*** Это событие инициируется перед тем, как файл решения будет закрыт, но уже после того, как он был сохранен (если вы указали, что файл следует сохранять). Поскольку во время инициации события шанс сохранить файл уже упущен, из обработчика не следует вносить изменения в решение, так как они будут сброшены.
- ***void AfterClosing()*** Это событие инициируется сразу по завершении процедуры закрытия решения.

Пример SolutionEvents (листинг 8-1) иллюстрирует подключение к каждому из этих событий. После его загрузки при инициации каждого события надстройка будет выводить окно с краткой справкой о событии. Обработчик события *QueryCloseSolution* позволяет отменить закрытие решения.

**Листинг 8-1.** Исходный код надстройки для событий решения

SolutionEvents.cs

```
namespace SolutionEvents
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;

    [GuidAttribute("1FF0C203-8036-4A54-A71A-B0F82BA60B0C"),
    ProgId("SolutionEvents.Connect")]

    public class Connect : Object, Extensibility.IDTExtensibility2

    {
```

```
public Connect()
{
}

public void OnConnection(object application,
    Extensibility.ext_ConnectMode connectMode,
    object addInInst, ref System.Array custom)
{
    applicationObject = (_DTE)application;
    addInInstance = (AddIn)addInInst;

    //Устанавливаем переменную делегата solutionEvents,
    //используя свойство DTE.Events.SolutionEvents:
    solutionEvents = (EnvDTE.SolutionEvents)
        applicationObject.Events.SolutionEvents;

    //Устанавливаем все доступные обработчики, создавая
    //новые экземпляры соответствующих делегатов:
    solutionEvents.AfterClosing += new
        _dispSolutionEvents_AfterClosingEventHandler(AfterClosing);
    solutionEvents.BeforeClosing += new
        _dispSolutionEvents_BeforeClosingEventHandler(BeforeClosing);
    solutionEvents.Opened += new
        _dispSolutionEvents_OpenedEventHandler(Opened);
    solutionEvents.ProjectAdded += new
        _dispSolutionEvents_ProjectAddedEventHandler(ProjectAdded);
    solutionEvents.ProjectRemoved += new
        _dispSolutionEvents_ProjectRemovedEventHandler
        (ProjectRemoved);
    solutionEvents.ProjectRenamed += new
        _dispSolutionEvents_ProjectRenamedEventHandler
        (ProjectRenamed);
    solutionEvents.QueryCloseSolution += new
        _dispSolutionEvents_QueryCloseSolutionEventHandler
         (QueryCloseSolution);
    solutionEvents.Renamed += new
        _dispSolutionEvents_RenamedEventHandler(Renamed);
}

public void OnDisconnection(
    Extensibility.ext_DisconnectMode disconnectMode,
    ref System.Array custom)
{
    //Надстройка закрывается. Отсоединяем обработчики:
    solutionEvents.AfterClosing -= new
        _dispSolutionEvents_AfterClosingEventHandler(AfterClosing);
    solutionEvents.BeforeClosing -= new
        _dispSolutionEvents_BeforeClosingEventHandler(BeforeClosing);
    solutionEvents.Opened -= new
        _dispSolutionEvents_OpenedEventHandler(Opened);
```

*см. след. стр.*

```
        solutionEvents.ProjectAdded -= new
            _dispSolutionEvents_ProjectAddedEventHandler(ProjectAdded);
        solutionEvents.ProjectRemoved -= new
            _dispSolutionEvents_ProjectRemovedEventHandler
            (ProjectRemoved);
        solutionEvents.ProjectRenamed -= new
            _dispSolutionEvents_ProjectRenamedEventHandler
            (ProjectRenamed);
        solutionEvents.QueryCloseSolution -= new
            _dispSolutionEvents_QueryCloseSolutionEventHandler(
            QueryCloseSolution);
        solutionEvents.Renamed -= new
            _dispSolutionEvents_RenamedEventHandler(Renamed);
    }

    public void OnAddInsUpdate(ref System.Array custom)
    {
    }

    public void OnStartupComplete(ref System.Array custom)
    {
    }

    public void OnBeginShutdown(ref System.Array custom)
    {
    }

    //Обработчик-делегат SolutionEvents.AfterClosing:
    public void AfterClosing()
    {
        System.Windows.Forms.MessageBox.Show(
            "SolutionEvents.AfterClosing",
            "Solution Events");
    }

    //Обработчик-делегат SolutionEvents.BeforeClosing:
    public void BeforeClosing()
    {
        System.Windows.Forms.MessageBox.Show(
            "SolutionEvents.BeforeClosing",
            "Solution Events");
    }

    //Обработчик-делегат SolutionEvents.Opened:
    public void Opened()
    {
        System.Windows.Forms.MessageBox.Show("SolutionEvents.Opened",
            "Solution Events");
    }
```

```
//Обработчик-делегат SolutionEvents.ProjectAdded.
//Отображает UniqueName добавляемого проекта.
public void ProjectAdded(EnvDTE.Project project)
{
    System.Windows.Forms.MessageBox.Show(
        "SolutionEvents.ProjectAdded\nProject: " + project.UniqueName,
        "Solution Events");
}

//Обработчик-делегат SolutionEvents.ProjectRemoved.
//Отображает UniqueName удаляемого проекта.
public void ProjectRemoved(EnvDTE.Project project)
{
    System.Windows.Forms.MessageBox.Show(
        "SolutionEvents.ProjectRemoved\nProject: " +
        project.UniqueName, "Solution Events");
I

//Обработчик-делегат SolutionEvents.ProjectRemoved.
//Отображает UniqueName переименовываемого проекта
//и полный путь файла до его переименования.
public void ProjectRenamed(EnvDTE.Project project, string oldName)
{
    System.Windows.Forms.MessageBox.Show(
        "SolutionEvents.ProjectRenamed\nProject: " +
        project.UniqueName
        + "\nOld project name: " * oldName, "Solution Events") ;
}

//Обработчик-делегат SolutionEvents.QueryCloseSolution.
//Дает возможность отменить закрытие решения.
public void QueryCloseSolution(ref bool cancel)
{
    if(System.Windows.Forms.MessageBox.Show(
        "SolutionEvents.QueryCloseSolution\nContinue with close?",
        "Solution Events",
        System.Windows.Forms.MessageBoxButtons.YesNo)==
        System.Windows.Forms.DialogResult.Yes)

        cancel = false;
    else
        cancel = true;
}

//Обработчик- делегат SolutionEvents.QueryCloseSolution.
//Отображает полный путь к решению
//до и после переименования.
public void Renamed(string oldName)
{
    System.Windows.Forms.MessageBox.Show(
```

*см. след стр.*

```
                "SolutionEvents.Renamed\nNew solution name: " +
                applicationObject.Solution.FullName + "\nOld solution name: "
                + oldName, "Solution Events");
        }

        private _DTE applicationObject;
        private AddIn addInInstance;

        //Переменная делегатов-обработчиков:
        private EnvDTE.SolutionEvents solutionEvents;
    }
}
```

**Практикум: ошибка с отключением событий?**

Меня часто спрашивают, нет ли ошибки с событиями, поскольку события могли неожиданно теряться и боже не инициироваться, хотя код для отключения события ни разу не запускался. Эта проблема возникает вследствие распространенной ошибки в программировании, связанной с принципом действия сборщика мусора в .NET Framework. Взгляните на следующий код, подключающийся к событию переименования решения:

```
public void ConnectSolutionEvents()
{
    EnvDTE.SolutionEvents solutionEvents;
    solutionEvents = (EnvDTE.SolutionEvents)
    applicationObject.Events.SolutionEvents;
    solutionEvents.Renamed += new
        _dispSolutionEvents_RenamedEventHandler(Renamed);
}
```

При вызове этого метода для подключения к событию *Renamed* переменной *solutionEvents* присваивается экземпляр объекта *SolutionEvents*. Но *solutionEvents* является локальной переменной метода *ConnectSolutionEvents*, *и* в результате при возврате из *ConnectSolutionEvents*она отмечается как подлежащая сбору мусора. Обычно событие инициируется один-два раза, но сборщик мусора видит, что эта переменная может быть удалена из памяти, и удаляет ее, отключая таким образом обработчик события. Чтобы код вашего обработчика событий работал корректно, переместите переменную *solutionEvents* за пределы метода в область видимости класса. Это не даст отключить обработчик события до выгрузки класса. Помните, что описанный случай применим ко всем обработчикам событий, подключенных с помощью .NET Framework, а не только к событию *Solution Renamed*.

## Надстройки для решений

Из главы 6 вы знаете, что Visual Studio .NET позволяет писать служебный код и создавать надстройки. Загруженная надстройка работает, пока ее не выгрузят или не закроют Visual Studio .NET. Решения могут загружать и исполнять надстройки

так же, как Visual Studio .NET. При загрузке решения Visual Studio .NET ищет в файле решения ссылки на надстройки. Если такие ссылки есть, она загружает эти надстройки и вызывает те же методы интерфейса *IDTExtensibility2* (*OnConnection*, *OnDisconnection* и т. д.), что и для надстроек, загруженных в обычном режиме.

В сравнении с обычными надстройками создание надстроек для решений предлагает такие преимущества, как использование надстройки только при открытии и запуске определенного решения. Например, вы хотите сосчитать, сколько раз проводилась компоновка решения. Для этого можно создать надстройку, загружаемую в обычном режиме, но если надстройка должна отслеживать компоновки одного определенного решения, то, будучи постоянно загруженной, она лишь зря тратит системные ресурсы.

Мастер Add-in Wizard не позволяет создавать надстройки для решений, но нужный основной код можно получить, создав обычную надстройку и внеся в нее изменения. Единственное отличие надстроек для решений от обычных в том, как Visual Studio .NET сообщается о необходимости загрузки надстройки. Обычная надстройка хранит информацию в реестре, а надстройки для решений — прямо в файле решения (.sln). Зная это, вы легко можете переделать обычную надстройку в надстройку для решения. Надо запустить Add-in Wizard и выбрать нужный язык, не изменяя прочих параметров, установленных по умолчанию. Когда мастер закончит работу и создаст код, удалите проект установки для проекта этой надстройки, Проект установки служит в основном для внесения параметров в системный реестр, а поскольку надстройки для решений их не используют, проект установки не нужен.

**Примечание** Если вы создадите надстройку для решений, следуя этим указаниям, ее можно будет загружать как обычную, так как при генерации кода мастер создает параметры реестра для обычной надстройки. Наличие этих дополнительных параметров в реестре может быть не очень удобно для разработчика надстройки, но не создаст нежелательных последствий для пользователей этой надстройки. Если вы не хотите связываться с диалоговым окном Add-in Manager или желаете устранить возможность случайной загрузки надстройки для решений в обычном режиме, эти разделы реестра можно удалить.

Затем необходимо зарегистрировать надстройку в файле решения. Свойство *SolutionAddIns* возвращает набор *AddIns* — такой же, что возвращается свойством *DTE.AddIns*, но содержащий только надстройки, зарегистрированные в файле решения. Для регистрации надстройки в решении служит метод *AddIns.Add*. Вот его сигнатура и параметры:

```
public EnvDTE.AddIn Add(string ProgID, string Description, string Name ,
    bool Connected)
```

Ниже приведено описание передаваемых методу аргументов.

- ***ProgID*** COM-совместимый ProgID надстройки, сопоставляемой решению.
- ***Description*** Описание надстройки. Используется только для заметок и игнорируется Visual Studio .NET.

- ***Name*** Короткое имя надстройки. Как и описание, этот параметр игнорируется Visual Studio .NET.
- ***Connected*** Если этому аргументу задано true, надстройка будет загружена после сопоставления решению и будет загружаться при каждом следующем открытии решения; если же false, надстройка будет сопоставлена решению, но не загружена и не будет загружаться при повторных открытиях решения.

Поскольку вы не сможете сопоставить надстройку файлу решений через реестр (для этого потребуется код), вам придется выбрать способ запуска кода для вставки надстройки в решение. Если он исполняется в общей среде, где исходный код регистрируется системой контроля (такой как Microsoft Visual SourceSafe), администратор проекта может написать и запустить макрос с нужным кодом, а затем зарегистрировать файл решения; тогда надстройка будет запускаться у всех, кто в следующий раз получит от системы управления новую версию решения. Другой вариант — создать мастер, вызывающий *AddInsAdd* после создания кода для проекта. Однако самое простое — использовать для управления надстройками решений что-либо вроде диалогового окна Add-in Manager.

Visual Studio .NET не содержит диспетчер надстроек для решений, но его можно создать, задействовав надстройку (обычную, а не для решений). Надстройка SolutionAddinManager добавляет команду в контекстное меню узла решения в Solution Explorer. При выборе этой команды появляется диалоговое окно, через которое можно загружать, выгружать, добавлять и удалять надстройки для решений.

# Работа с элементами проектов

Решение содержит несколько проектов, каждый из которых содержит файлы, составляющие программу Файлы в проекте можно перебирать и программировать.

## Перебор элементов проекта

Файлы в проекте упорядочиваются иерархически. Проект может содержать любое количество файлов и папок, каждая из которых может содержать собственные файлы и папки. Для соответствия иерархии проекта объектная модель проекта также имеет иерархическую структуру. Набор *ProjectItems* представляет узлы, содержащие элементы, а объекты *ProjectItem* — каждый элемент в этом наборе. Для перебора иерархии служат объекты *ProjectItems* и *ProjectItem*. Следующий макрос перебирает первый уровень иерархии объектов *ProjectItems* и *ProjectItem*, получая объект *ProjectItems* верхнего уровня через свойство *Project.ProjectItems*:

```
Sub EnumTopLevelProjectItems()
    Dim projItem As EnvDTE.ProjectItem
    Dim projectProjectItems As EnvDTE.ProjectItems
    Dim project As EnvDTE.Project

    'Ищем первый проект в решении:
    project = DTE.Solution.Projects.Item(1)
    'Получаем набор элементов проекта:
    ProjectProjectItems = project.ProjectItems
    'Просматриваем список элементов в наборе:
    For Each projItem In projectProjectItems
```

```
        MsgBox(projItem.Name)
    Next
End Sub
```

Некоторые элементы проекта, такие как папки, являются одновременно и элементом иерархии проекта и контейнером для других файлов и папок. Поскольку эти папки представляют собой и элементы, и наборы элементов, то в иерархической модели проекта папка представляется и объектом *ProjectItem*, и *ProjectItems*. Чтобы определить, является ли узел *ProjectItem* контейнером для других узлов *ProjectItem*, можно вызвать свойство *ProjectItem.ProjectItems*, которое возвращает набор *ProjectItems*, если узел может содержать вложенные элементы. Перебрать все объекты *ProjectItem* и *ProjectItems* в проекте позволяет рекурсивная макрофункция вроде этой:

```
Sub EnumProjectItems(ByVal projItems As EnvDTE.ProjectItems)
    Dim projItem As EnvDTE.ProjectItem
    'Ищем все объекты ProjectItem в данном наборе;
    For Each projItem In projItems
        MsgBox(projItem.Name)
        'И просматриваем все вложенные элементы текущего элемента:
        EnumProjectItems(projItem.ProjectItems)
    Next
End Sub

Sub EnumProject()
    Dim project As EnvDTE.Project
    'Ищем первый проект в решении:
    project = DTE.Solution.Projects.Item(1)
    EnumProjectItems(project.ProjectItems)
End Sub
```

Вначале макрос *EnumProject* ищет набор *ProjectItems* данного проекта, а затем вызывает подпрограмму *EnumProjectItems*, которая найдет все объекты *ProjectItem*, содержащиеся в наборе. Если объект *ProjectItem* сам является набором, макрос будет рекурсивно вызывать подпрограмму *EnumProjectItems* для отображения содержащихся в нем элементов.

Наборы объектов *ProjectItem* могут содержаться не только в папках. Некоторые файлы, такие как файлы Windows Forms и Web Forms, также являются наборами файлов. Каждый из этих типов файлов имеет соответствующие файлы ресурсов (.resx-файлы), а файлы Web Forms, кроме того, имеют соответствующий файл фонового кода. В обычном состоянии Solution Explorer не указывает, являются ли эти файлы контейнерами для других файлов, но его можно настроить для отображения файлов, содержащихся в таких типах файлов. Чтобы все файлы форм отображались в древовидном списке Solution Explorer как раскрывающиеся узлы, выберите из меню Project команду Show All Files. Когда выполняется макрос *EnumProject* (см. выше), свойство *ProjectItem.ProjectItems* возвращает набор, содержащий объекты *ProjectItem* для этих вложенных элементов. Код наподобие макроса *EnumProject* будет возвращать одинаковые значения независимо от того, выбрана ли в меню команда Show All Files. Эта команда влияет только на пользовательский интерфейс Solution Explorer.

Для поиска определенного элемента в проекте можно комбинировать методы перебора файлов и файлов в папках. Допустим, вы создали решение для приложения Windows Forms и изменили проект так (рис. 8-1):




**Рис. 8-1.** *Приложение Windows Forms с вложенными ресурсами*

Используя объект *ProjectItem* и набор *ProjectItems,* можно написать такой макрос, как этот, выполняющий поиск файла Bitmapl.bmp:

```
Sub FindBitmap()
    Dim project As EnvDTE.Project
    Dim projectProjectItems As EnvDTE.ProjectItems
    Dim resourcesProjectItem As EnvDTE.ProjectItem
    Dim resourcesProjectItems As EnvDTE.ProjectItems
    Dim bitmapsProjectItem As EnvDTE.ProjectItem
    Dim bitmapsProjectItems As EnvDTE.ProjectItems
    Dim bitmapProjectItem As EnvDTE.ProjectItem

    'Получаем проект:
    project = DTE.Solution.Item(1)
    'Получаем список элементов проекта:
    projectProjectItems = project.ProjectItems
    'Получаем элемент для папки Resources:
    resourcesProjectItem = projectProjectItems.Item("Resources")
    'Получаем набор элементов в папке Resources:
    resourcesProjectItems = resourcesProjectItem.ProjectItems
    'Получаем элемент для папки Bitmaps:
    bitmapsProjectItem = resourcesProjectItems.Item("Bitmaps")
    'Получаем набор элементов в папке Bitmaps:
    bitmapsProjectItems = bitmapsProjectItem.ProjectItems
    'Получаем элемент для файла Bitmap1.bmp:
    bitmapProjectItem = bitmapsProjectItems.Item("Bitmap1.bmp")
    MsgBox(bitmapProjectItem.Name)
End Sub
```

Для поиска определенного файла можно просматривать иерархическое дерево *ProjectItem* и *ProjectItems,* но иногда может потребоваться более быстрый и легкий способ поиска объекта *ProjectItem* для файла с определенным именем, содержащегося в проекте. Чтобы найти элемент, передав часть пути к файлу на диске, можно задействовать метод *FindProjectItem* объекта *Solution.* Допустим, в папке Addins

корневого каталога диска C созданы два проекта надстроек (при помощи Add-in Wizard). Обе надстройки — MyAddin1 и MyAddin2 — содержат файл с именем Connect.cs. Найти файл Connect.cs в каждом проекте позволяет макрос:

```
Sub FindItem()
    Dim projectItem As EnvDTE.ProjectItem
    projectItem = DTE.Solution.FindProjectItem("Connect.cs")
End Sub
```

Однако, поскольку *FindProjectItem* возвращает любой файл с указанным именем, вы не сможете определить, какой объект *ProjectItem* будет возвращен: *ProjectItem* для Connect.cs из MyAddin1 или *ProjectItem* для Connect.cs из MyAddin2. Чтобы усовершенствовать поиск, в качестве имени искомого файла можно указать чуть более длинный путь, как это сделано в следующем макросе, в котором добавлен < имя папки на диске, содержащей версию Connect.cs для MyAddinl:

```
Sub FindItemWithFolder()
    Dim projectItem As EnvDTE.ProjectItem
    projectItem = DTE.Solution.FindProjectItem("MyAddin1\Connect.cs")
End Sub
```

Разумеется, для поиска *ProjectItem* можно указать не только часть пути, но и путь целиком, в точности определяющий искомый элемент:

```
Sub FindItemWithFullPath()
    Dim projectItem As EnvDTE.ProjectItem
    projectItem = _
        DTE.Solution.FindProjectItem("C:\Addins\MyAddin1\Connect.cs")
End Sub
```

## Добавление и удаление элементов проекта

Добавить новые файлы в проект можно двумя способами. Первый — использовать методы *AddFromDirectory*, *AddFromFile*, *AddFromFileCopy* и *AddFromTemplate* интерфейса *ProjectItems* (о котором см. главу 9). Второй — использовать объект *ItemOperations*, который предлагает массу методов для работы с файлами, облегчающих файловые операции. Различие между применением методов объекта *ProjectItems* и методов *ItemOperations* в том, что объект *ProjectItems* дает надстройке или макросу более точный контроль над тем, где в проекте будет расположен создаваемый файл. Объект *ItemOperations* более ориентирован на работу с пользовательским интерфейсом; он добавляет новый файл в проект или папку, выбранную в Solution Explorer, или, если выбран файл, он добавляет элемент в проект или папку, содержащую этот файл. Эти особенности позволяют записывать макросы. Если вы запустите запись и добавите файл через Solution Explorer, в макрос запишется вызов одного из методов объекта *ItemOperations*. Выбранный элемент указывает место, куда добавляются файлы при вызове соответствующего метода.

Один метод объекта *ItemOperations* — *AddExistingItem* — в качестве единственного аргумента принимает путь к файлу на диске и добавляет этот файл в выбранный проект или папку в проекте. В зависимости от типа проекта файл может быть скопирован в папку проекта до добавления, или в проект может быть добавлена ссылка без копирования файла. Проекты Visual Basic .NET и C# пост-

роены на основе каталогов. А значит, отображаемая в Solution Explorer иерархия повторяется на диске, и любые файлы из проекта должны находиться в папке проекта или одной из вложенных папок. Проекты Visual C++ немного отличаются: файл, включенный в проект, может быть расположен в любом месте диска, а не только в папке проекта или вложенной в нее папке. Например, допустим, что файл с именем file.txt расположен в корневом каталоге C:\. Если запустить макрос:

```
Sub AddExistingItem()
    DTE.ItemOperations.AddExistingItem("C:\file.txt")
End Sub
```

когда в Solution Explorer выбран проект C# или Visual Basic .NET или один из вложенных в них элементов, то file.txt будет скопирован в каталог или подкаталог, содержащий файл проекта, а затем добавлен к проекту. Но если выбран проект Visual C++, файл останется на месте, а к проекту будет добавлена ссылка на него.

В то время как *AddExistingItem* добавляет в проект файл с диска, *AddNewItem* создает новый файл и добавляет его в проект. Этот метод принимает два аргумента и имеет следующую сигнатуру:

```
public EnvDTE.ProjectItem AddNewItem(string Item = "General\Text File",
    string Name = "")
```

Новый элемент можно добавить через пользовательский интерфейс, щелкнув проект правой кнопкой и выбрав из контекстного меню команду Add | Add New Item, которая откроет диалоговое окно Add New Item (рис. 8-2).

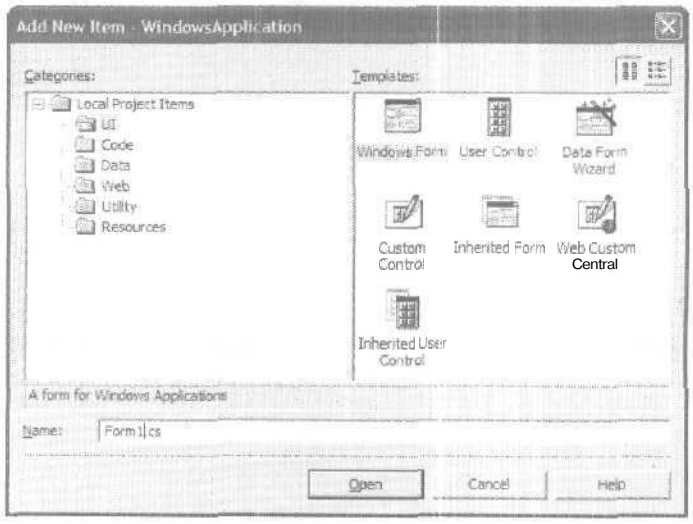



Рис. 8-2. *Диалоговое окно Add New Item для проекта C#*

Это диалоговое окно связано с методом *AddNewItem* тем, что первый параметр *AddNewItem* определяет тип добавляемого файла и этот тип файла можно найти через диалоговое окно. Тип файла определяется по пути к выделенному элементу из древовидного списка, каждая часть которого отделяется обратной косой чер-

той и по названию элемента в списке из правой части диалогового окна. Так, при добавлении в проект файла Windows Forms узел верхнего уровня древовидного списка (Local Project Items) конкатенируется с символом обратной косой черты. Затем к этой строке добавляется строка «*UI*», так как это узел дерева, содержащий добавляемый элемент Windows Forms, а за ней следует еще одна обратная косая черта. Наконец, добавляется имя элемента, показанное в правой части диалогового окна, — строка «*Windows Form*», и в итоге получается строка, которая может быть передана *AddNewItem*: «*Local ProjectItems\UI\WindowForm*». Второй аргумент этого метода — просто имя создаваемого файла с расширением. Если в этом параметре передается пустая строка, то создается и используется имя файла по умолчанию.

Иногда может потребоваться удалить элемент, добавленный в проект. Объект *ProjectItem* предоставляет два метода для удаления элементов из проекта: *Remove* и *Delete,* Оба удаляют элемент из проекта, но *Delete* более деструктивен, поскольку он также стирает файл с жесткого диска, перемещая его в Корзину.

# Работа с объектами проектов, специфичными для отдельных языков

Объектная модель проектов Visual Studio .NET предоставляет функциональность, общую для всех типов проектов. Однако некоторые проекты поддерживают дополнительную, уникальную функциональность. Так, проект C# включает узел ссылок (references node), а проект Setup — нет. Возможность программно добавлять и удалять эти ссылки дала бы большую гибкость в написании надстроек, макросов и мастеров. Программирование таких специфических функций обеспечивается расширяемостью объектной модели Visual Studio .NET, которая позволяет каждому типу проекта предоставлять дополнительные методы и свойства, кроме определенных в объекте *Project.*

Доступ к специальному типу объекта предоставляет свойство *Object* объекта *Project.* Это свойство возвращает объект типа *System.Object*, который можно преобразовать в тип объектной модели, поддерживаемой используемым языком. Наиболее широко применяемое проектное расширение — объект *VSProject*,доступный в типах проектов Visual Basic .NET и C#.

## Проекты *VSProject*

Интерфейс *VSLangProj.VSProject*определяет расширения объекта *EnvDTE.Project*для проектов Visual Basic .NET или C#. Получив интерфейс *EnvDTEProject* для одного из этих типов проектов, можно перейти к интерфейсу *VSLangProj.VSProject* через вызов свойства *Project.Object*. Следующий макрос, предполагающий, что первый проект в наборе *Projects* — это проект Visual Basic или C#, получает объект *VSProject* для этого проекта:

```
Sub GetVSProject()
    Dim project As EnvDTE.Project
    Dim vsproject As VSLangProj.VSProject
    project = DTE.Solution.Projects.Item(1)
    vsproject = CType(project.Object, VSLangProj.VSProject)
End Sub
```

### Ссылки

Ссылки — это указатели на программные компоненты, которые могут применяться в проекте для сокращения количества кода, создаваемого программистом. Проект использует информацию о типах, содержащуюся в ссылке для отображения информации в форме завершения операторов IntelliSense. Кроме того, ссылка предоставляет сведения компилятору для интерпретации идентификаторов программного кода. Ссылка может быть сборкой или другим проектом, загруженным в решение. Можно формировать ссылки на COM-компоненты, создавая оболочки для библиотеки информации о типах COM-объектов посредством сборок для взаимодействия. Ссылки можно добавлять через пользовательский интерфейс. Для этого надо щелкнуть правой кнопкой узел References в проекте Visual Basic или C#, выбрать из контекстного меню команду Add Reference и в появившемся диалоговом окне выбрать компонент.

Используя объект *VSLangProjReferences* можно перебирать, добавлять и удалять ссылки. Получить объект *References* позволяет свойство *VSProjectReferences*. Например, следующий код получает объект *References* и перебирает ссылки, добавленные в проект:

```
Sub EnumReferences()
    Dim proj As EnvDTE.Project
    Dim vsproj As VSLangProj.VSProject
    Dim references As VSLangProj.References
    Dim reference As VSLangProj.Reference
    proj = DTE.Solution.Projects.Item(1)
    vsproj = proj.Object
    references = vsproj.References
    For Each reference In references
        MsgBox(reference.Name)
    Next
End Sub
```

Чтобы добавить ссылку в сборку, вызовите метод *ReferencesAdd* и передайте ему путь к сборке. Если копия сборки такой же версии и с тем же открытым ключом не хранится в глобальном кэше сборок (global assembly cache, GAC), метод *Add* копирует сборку в каталог выходных файлов проекта. Это нужно для того, чтобы при запуске/загрузке выходных файлов проекта другой сборкой по ссылке загружалась правильная копия сборки. Следующий макрос добавляет ссылку в сборку:

```
Sub AddReferenceToAssembly()
    Dim vsproj As VSLangProj.VSProject
    Dim proj As EnvDTE.Project
    proj = DTE.Solution.Projects.Item(1)
    vsproj = CType(proj.Object, VSLangProj.VSProject)
    vsproj.References.Add("C:\Program Files\Microsoft Visual Studio .NET " _
        & "2003\Common7\IDE\PublicAssemblies\extensibility.dll")
End Sub
```

Этот код ищет объект *VSProject* для проекта и добавляет ссылку на сборку метаданных Extensibility.dll (предполагая, что Visual Studio .NET установлена в каталоге по умолчанию) — ту же сборку, что содержит определение интерфейса *IDTEx-*

*tensibility2*, используемого для создания надстроек. Б виде ссылок в проект нельзя добавлять сборки, хранящиеся в GAC, так как системы проектов Visual Basic и C# различают файлы для компоновки и файлы, применяемые в период выполнения компонента.

Во время разработки компонент из проекта в составе решения может потребоваться компоненту из другого проекта. Создать ссылку из одного проекта на другой позволяет метод *ReferencesAddProject*.Он принимает объект *Project ч* добавляет ссылку к этому проекту:

```
Sub AddProjectReference()
    Dim vsproj As VSLangProj.VSProject
    Dim proj As EnvDTE.Project
    'Ищем проект, в который будет добавлена ссылка:
    proj = DTE.Solution.Projects.Item(1)
    vsproj = CType(proj.Object, VSLangProj.VSProject)
    'Ищем проект, на который будем ссылаться:
    proj = DTE.Solution.Projects.Item(2)
    'Создаем ссылку из проекта в проект:
    vsproj.References.AddProject(proj)
End Sub
```

Добавление ссылок на COM-объекты требует нескольких специфичных для COM параметров, которые могут вызвать затруднения у программистов, не сталкивавшихся с COM: GUID библиотеки типов или идентификатор библиотеки типов (LIBID), описывающей COM-компонент, а также основной и дополнительный номер версии этой библиотеки типов. Используя эти параметры, можно добавить ссылку на библиотеку типов COM-компонента, а Visual Studio .NET автоматически создаст сборку для взаимодействия с этой библиотекой типов. Следующий макрос добавляет ссылку на библиотеку типов для Windows Media Player:

```
Sub AddCOMReference()
    Dim vsproj As VSLangProj.VSProject
    Dim proj As EnvDTE.Project
    proj = DTE.Solution.Projects.Item(1)
    vsproj = CType(proj.Object, VSLangProj.VSProject)
    vsproj.References.AddActiveX( _
        "{22D6F304-B0F6-11D0-94AB-0080C74C7E95}", 1, 0)
End Sub
```

### Web-ссылки

.NET Framework не только облегчает разработку традиционного ПО, но и рождает новые методики создания программ. Одна из этих новых методик задействует Web-сервисы XML. Web-сервисы позволяют разрабатывать ПО через Интернет, размещая код программ на сервере, к которому могут подключаться программы с компьютеров пользователей. С помощью Visual Studio .NET подключение персонального приложения к Web-сервисам XML сводится к добавлению Web-ссылки. При создании ссылки на Web-сервис XML с сервера скачивается специальный файл, написанный на языке Web Services Description Language (WSDL) и из WSDL-файла создается прокси-класс (класс, содержащий логику для перевода вызова метода

или свойства через Интернет с клиентского компьютера на сервер). В дальнейшем этот прокси-класс можно задействовать для вызовов Web-сервиса XML.

Следующий макрос добавляет в проект Web-ссылку. Он получает объект *VSProject* для проекта и вызывает метод *AddWebReference*, передавая ему URL Web-сервиса. Этот пример использует Web-сервис TerraServer, предоставляемый Microsoft, который предлагает подробную географическую информацию и спутниковые изображения территории США. Этот Web-сервис расположен по адресу *http://terraserver.homeadvisor.msn.com/TerraService.asmx*

```
Sub AddTerraServerWebRef()
    Dim vsProj As VSLangProj.VSProject
    Dim serviceURL As String
    'Устанавливаем URL Web-сервиса TerraServer.
    serviceURL = "http://terraserver.microsoft.net/TerraService.asmx"
    'Ищем VSProject для проекта.
    vsProj = DTE.Solution.Projects.Item(1).Object
    'Добавляем Web-ссылку.
    vsProj.AddWebReference(serviceURL)
End Sub
```

При создании этой Web-ссылки с сервера скачивается WSDL-файл, описывающий Web-сервис XML, затем генерируется прокси-класс для сервиса и автоматически добавляется в проект. Этот класс помещается в пространство имен, определяемое адресом сервера, представленным в обратном порядке. Так, если Web-сервис расположен по адресу *www.microsoft.com*, пространство имен для него будет носить имя *com.microsqft.www*. В этом примере TerraServer расположен по адресу *terraserver.homeadvisor.msn.com,* поэтому применяется пространство имен *com.msn.homeadvisor.terraserver*. После добавления в проект ссылки на Web-сервис им можно пользоваться, просто вызывая методы созданного прокси-класса.

### Практикум: использование Web-сервиса XML

Работая над этой книгой, я купил электронный самонаводящийся телескоп, который может автоматически искать звезды. Но, прежде чем его использовать, я должен был ввести широту и долготу моего местоположения — Редмонд, шт. Вашингтон. Но как это узнать? Решением было написать небольшую программу, которую можно запустить на любом компьютере, подключенном к Интернету, и определить свое положение при помощи Web-сервиса TerraServer, — своего рода дешевый Web-ориентированный GPS-навигатор.

Создав новое консольное приложение C# и добавив ссылку на Web-сервис TerraServer, я получил все, что нужно для определения положения города в пределах США. Для определения положения, переменная типа *Place,* хранящая имя города, штата и страны, передается методу *TerraService.GetPlaceFacts*. Этот метод возвращает объект типа *PlaceFacts*, который содержит широту и долготу указанного города. Ниже приведен код, который я использовал для определения географического положения; чтобы определить положение другого города, надо лишь изменить строки с именами города, штата и страны.

```
static void Main(string[] args)
{
    //Объявляем переменные:
    com.msn.homeadvisor.terraserver.PlaceFacts placeFacts;
    com.msn.homeadvisor.terraserver.TerraService terraService;
    com.msn.homeadvisor.terraserver.Place place;

    //Создаем необходимые объекты:
    terraService = new com.msn.homeadvisor.terraserver.TerraService();
    place = new com.msn.homeadvisor.terraserver.Place();

    //Ищем широту и долготу для Редмонда, шт. Вашингтон:
    place.City="Redmond";
    place.State = "Washington";
    place.Country = "USA";

    //Вызываем Web-сервис и получаем нужную информацию:
    placeFacts = terraService.GetPlaceFacts(place);

    //Отображаем информацию в консоли:
    System.Console.Write(place.City + " Latitude and Longitude: ");
    System.Console.Write(placeFacts.Center.Lat.ToString());
    System.Console.Write(" ");
    System.Console.WriteLine(placeFacts.Center.Lon.ToString());
}
```

### *Imports*

Чтобы упростить свою жизнь, программисты могут включать в исходный код Visual Basic .NET и C# операторы *Imports* и *using,* сокращая идентификаторы, применяемые для доступа к пространствам имен, определяемым библиотеками кода. Так, для вывода информационного окна можно указать имя класса длинным, конкретным идентификатором:

```
System.Windows.Forms.MessageBox.Show("Hello World")
```

Но если этот код встречается много раз, вам придется снова и снова набирать этот идентификатор, что может привести к ошибке. Чтобы сократить объем печатаемого текста, в Visual Basic можно вызвать оператор *Imports*:

```
Imports System.Windows.Forms
```

Далее в программе можно применять сокращенную запись:

```
MessageBox.Show("Hello World")
```

Visual Basic также позволяет вводить операторы импорта через диалоговое окно Property Pages проекта (рис. 8-3) и не набирать оператор *Imports* в исходном коде. Действуя через диалоговое окно вместо набора оператора в коде, можно применять импортирование в любых файлах проекта, а не только в тех, что используют оператор *Imports*.

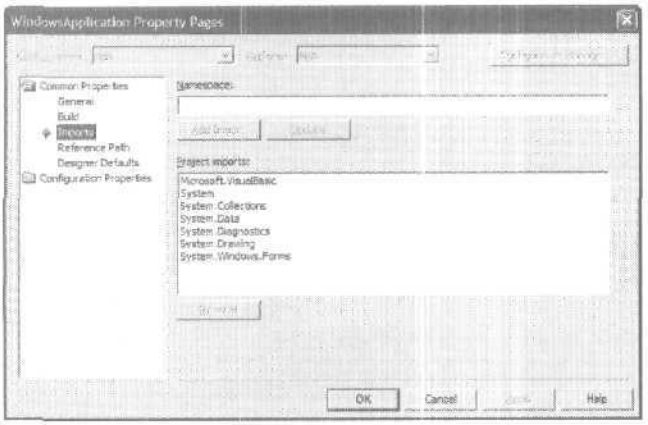



**Рис. 8-3.** *Ввод операторов импортирования через диалоговое окно Property Pages проекта*

Используя набор *VSProject.Imports* можно перебирать, удалять и добавлять импортирование во всем проекте. Следующий макрос добавляет в проект Visual Basic .NET пространство имен *System.XML*:

```
Sub AddSystemXMLImport()
    Dim vsProj As VSLangProj.VSProject
    Dim vsImports As VSLangProj.Imports
    vsProj = DTE.Solution.Projects.Item(1).Object
    vsProj.Imports.Add("System.Xml")
End Sub
```

**Примечание** Объект *Imports* допустим только для типов проектов Visual Basic .NET. При попытке получить доступ к этому объекту для проектов другого типа C# вернет *null*, а Visual Basic .NET — *Nothing*.

### Project.Properties

Каждому проекту сопоставлен набор параметров, позволяющих управлять взаимодействием с этим проектом. Их можно устанавливать через узел Common Properties диалогового окна Property Pages проекта. Параметры включают имя компонента, собираемого компилятором, тип создаваемого проекта (.exe или .dll) и параметры раскладки (layoutoptions) для HTML-дизайнера. Эти параметры можно устанавливать и программно, применяя свойство *Properties* объекта *Project*. Оно возвращает тот же объект *Properties*, что используется Visual Studio .NET для установки параметров. Следующий макрос просматривает список свойств проекта, а также значения и типы каждого свойства:

```
Sub WalkVSProjectProperties()
    Dim project As EnvDTE.Project
    Dim properties As EnvDTE.Properties
    Dim [property] As EnvDTE.Property
    Dim owp As InsideVSNET.Utilities.OutputWindowPaneEx
    owp = New InsideVSNET.Utilities.OutputWindowPaneEx(DTE, _
```

```
        "Project properties")
    project = DTE.Solution.Projects.Item(1)
    properties = project.Properties
    For Each [property] In properties
        owp.WriteLine("Name: " + [property].Name)
        owp.WriteLine("Value: " + [property].Value.ToString())
        owp.WriteLine("Type: " + [property].Value.GetType().FullName)
        owp.WriteLine()
    Next
End Sub
```

Этот объект *Property* можно задействовать не только для чтения значений свойств, но и для установки свойств проекта. Это иллюстрирует следующий макрос. Он устанавливает значок для использования в проекте после компиляции, Этот код предполагает, что значок с именем Icon.ico расположен в одном каталоге с файлом проекта.

```
Sub SetProjectIcon()
    Dim project As EnvDTE.Project
    Dim [property] As EnvDTE.Property
    Dim projectPath As String
    project = DTE.Solution.Projects.Item(1)
    'Получаем объект Property для значка:
    [property] = project.Properties.Item("ApplicationIcon")
    'Создаем путь к значку на основе пути к проекту:
    projectPath = project.FullName
    projectPath = System.IO.Path.GetDirectoryName(projectPath)
    projectPath = projectPath + "\Icon.ico"
    'Устанавливаем проекту этот значок:
    [property].Value = projectPath
End Sub
```

# Работа со вспомогательными типами проектов Visual Studio .NET

Для облегчения работы с файлами в решении Visual Studio .NET включает вспомогательные проекты. Они позволяют отслеживать файлы, не входящие ни в один другой проект, загруженный в решение. Поскольку в этих проектах могут храниться файлы любого типа, такие как файлы с исходными кодами программ и документы Microsoft Word, они не подлежат компиляции в программу. И, поскольку вспомогательные проекты являются частью Visual Studio .NET и не сопоставлены определенному языку, они доступны всем пользователям Visual Studio .NET и не требуют установленных версий Visual Basic .NET, C# или Visual C++.

## Проект Miscellaneous Files

Вам может потребоваться открыть файл, не входящий в состав существующих проектов. При открытии такой файл автоматически добавляется в проект с именем Miscellaneous Files, В отличие от проектов других типов для него на диске не создается файл проекта, но вы получаете удобный способ доступа к открытым фай-

лам, которые не являются частью других проектов, открытых в решении. Воспринимайте проект Miscellaneous Files как список недавно открытых документов — при открытии файла в проект добавляется элемент для этого файла, а при закрытии — удаляется. По умолчанию проект Miscellaneous Files и содержащиеся в нем файлы не отображаются в иерархическом дереве Solution Explorer, но их можно легко включить, открыв диалоговое окно Tools Options, выбрав узел Environment | Documents и установив флажок Show Miscellaneous Files In Solution Explorer.

Проект Miscellaneous Files имеет сопоставленное ему уникальное имя, которое в отличие от других проектов не меняется со временем. Это имя — «<MiscFiles>» — определяется константой *vsMiscFilesProjectUniqueName*. Следующий макрос получает объект *Project* для проекта Miscellaneous Files:

```
Sub FindMiscFilesProject()
    Dim project As EnvDTE.Project
    Dim projects As EnvDTE.Projects
    projects = DTE.Solution.Projects
    project = projects.Item(EnvDTE.Constants.vsMiscFilesProjectUniqueName)
End Sub
```

При открытии первого файла в проекте Miscellaneous Files добавляется элемент в набор *SolutionProjects*, реализующий интерфейс *Project* Он работает в точности, как интерфейс *Project*, реализуемый такими проектами, как проекты Visual Basic .NET или C#, за исключением того, что несколько свойств при вызове вернут *null* или *Nothing* или инициируют исключение *System.NotImplementedException*. Вот методы и свойства объекта *Project,* которые возвращают определенные значения для проекта Miscellaneous Files, а также объектов *ProjectItem* и *ProjectItems*, содержащихся в этом проекте (табл. 8-1):

**Табл. 8-1. Методы и свойства объектов *Project, ProjectItems* и *ProjectItem***

| *Project* | *ProjectItems* | *ProjectItem* |
|---|---|---|
| *DTE* | *DTE* | *DTE* |
| *ProjectItems* | *Parent* | *Collection* |
| *Name* (неизменяемый) | *Item* | *Name* (неизменяемый) |
| *UniqueName* | *GetEnumerator* / *_NewEnum* | *FileCount* |
| *Kind* | *Kind* | *Kind* |
| *FullName* | *Count* | *FileNames* |
| | *ContainingProject* | *SaveAs* |
| | | *Save* |
| | | *IsOpen* |
| | | *Open* |
| | | *Delete* |
| | | *Remove* |
| | | *ExpandView* |
| | | *ContainingProject* |
| | | *IsDirty* |

Добавлять новые файлы в проект Miscellaneous Files можно методом *ItemOperations.NewFile*:

```
public EnvDTE.Window NewFile(string Item = "General\Text File",
    string Name = "", string ViewKind =
    "{00000000-0000-0000-0000-000000000000}")
```

Применяя использованный ранее способ для вычисления первого параметра *ItemOperations.AddNewItem*, можно найти значение, которое следует передать методу *NewFile*. Второй параметр имеет такой же смысл, что и второй параметр метода *ItemOperations.AddNewItem*, — имя добавляемого файла (с расширением) — и, если передается пустая строка, вычисляется имя по умолчанию. Последний аргумент указывает, в каком виде открыть файл после добавления. Эти значения можно найти в классе *EnvDTE.Constants*; они начинаются с имени *vsViewKind*.

## Проект Solution Items

Он почти идентичен проекту Miscellaneous Files с некоторыми отличиями. Файлы, открытые в проекте Miscellaneous Files, удаляются из этого проекта при закрытии решения; элементы, добавленные в проект Solution Items, остаются в проекте даже после закрытия решения. Поскольку эти файлы остаются в проекте, о нем можно думать, как о способе рационального ведения хозяйства, поскольку он позволяет хранить под рукой такие элементы как документация, заметки и другие файлы, которые могут понадобиться в работе над проектом. Как и Miscellaneous Files, проект Solution Items не может участвовать в компоновке приложения, но поддерживает объект *EnvDTE.Project* (правда, ограниченной функциональности). К проекту Solution Items подходят все методы, приведенные в табл. 8-1. Подобно Miscellaneous Files, для проекта Solution Items зарезервировано уникальное имя для индексации набора *Solution.Projects*, так что в списке проектов решения можно найти и Solution Items. Можно использовать константу *vsSolutionItemsProjectUniqueName*:

```
Sub FindSolutionItemsProject()
    Dim project As EnvDTE.Project
    Dim projects As EnvDTE.Projects
    projects = DTE.Solution.Projects
    project = projects.Item( _
        EnvDTE.Constants.vsSolutionItemsProjectUniqueName)
End Sub
```

Добавить элементы в Solution Items позволяют методы *AddNewItem* и *AddExistingItem* объекта *ItemOperations*. Для этого нужно, чтобы сначала проект Solution Items был выделен в окне Solution Explorer. А поскольку Solution Items создается, только когда пользователь вручную добавляет файл в проект, прежде чем добавить в него файл, убедитесь, что такой проект существует.

## Внемодельные проекты

Все типы проектов, рассмотренные до этого момента, реализовывали объект *Project*, который можно применять в макросах или надстройках. Однако несколько типов проектов, таких как проект базы данных или проект, выгруженный командой

Project | Unload Project, сами не реализуют объект *Project,* Чтобы обеспечить некоторое программное управление проектами таких типов, Visual Studio -NET поддерживает внемодельный тип проекта (unmodeled project type). Этот тип предоставляет реализацию объекта *Project,* которая поддерживает только общие для всех типов проектов свойства: *DTE, Kind и Name.* Все другие свойства и методы в этой реализации объекта *Project* при вызове возвращают значения, не имеющие смысла, или инициируют исключение, и их не следует использовать в макросах или надстройках. Отличить внемодельный проект от других позволяет свойство *Project-Kind.* которое возвращает константу *EnvDTE.Constants.vsProjectKindUnmodeled*если проект является внемодельным. Следующий макрос перебирает все проекты, загруженные в решение и определяет внемодельные:

```
Sub FindUnmodeledProjects()
    Dim i As Integer
    For i = 1 To DTE.Solution.Projects.Count
        Dim project As EnvDTE.Project
        project = DTE.Solution.Projects.Item(i)
        If (project.Kind = EnvDTE.Constants.vsProjectKindUnmodeled) Then
            MsgBox(project.Name + " is unmodeled")
        End If
    Next
End Sub
```

**Примечание** Вы, наверное, заметили, что при выборе проектов из решения макрос *FindUnmodeledProjects*использует метод цифровой индексации вместо перечислителя. Это следствие ошибки в Visual Studio .NET, которая не вернет корректный объект *Project* при использовании перечислителя. Эта ошибка проявляется только для внемодельных проектов; для других типов проектов перечислитель подходит прекрасно.

## События проектов и элементов проекта

Подобно решению, инициирующему события, чтобы позволить надстройкам и макросам реагировать на действия пользователей, разные типы проектов также инициируют события, информирующие надстройки и макросы о действиях пользователей. Подключаться к событиям, инициируемым разными типами проектов, надо по-разному, но каждый тип проекта поддерживает одинаковые интерфейсы для работы с событиями. Каждый тип проекта инициирует два класса событий: действия, выполняемые с проектом, и действия, выполняемые с элементами проекта, Вот события, инициируемые проектом:

```
void ItemAdded(ByVal Project As EnvDTE.Project)
void ItemRemoved(ByVal Project As EnvDTE.Project)
void ItemRenamed(ByVal Project As EnvDTE.Project,
    ByVal OldName As String)
```

А это сигнатуры событий элементов проекта:

```
void ItemAdded(ByVal ProjectItem As EnvDTE.ProjectItem)
void ItemRemoved(ByVal ProjectItem As EnvDTE.ProjectItem)
void ItemRenamed(ByVal ProjectItem As EnvDTE.ProjectItem, _
    ByVal OldName As String)
```

Проекты и элементы проектов инициируют три отдельных события: *ItemAdded*, *ItemRemoved*, *ItemRenamed*. Они указывают на то, что проект или элемент проекта был добавлен, удален или переименован соответственно.

Подключение к событиям проектов и элементов проекта из макросов требует большего, чем подключение к другим типам событий Visual Studio .NET. Так, чтобы подключиться к событиям решения из проекта макроса, вначале открывается модуль *EnvironmentEvents*. Затем из раскрывающегося списка Class Name в левой части текстового окна в этом модуле выбирается объект события *SolutionEvents*, а из раскрывающегося списка Method Name — метод обработчика событий решения. Однако подключиться к событию проекта или элемента проекта из макроса не так просто: придется написать небольшой соединительный код для подключения событий. Вначале объявляется переменная для события добавлением следующего кода в модуль *EnvironmentEvents* (этот пример использует события проекта C#):

```
<System.ContextStaticAttribute()> _
Public WithEvents csharpProjectItemsEvents As EnvDTE, ProjectItemsEvents
```

После ввода этого кода в левом раскрывающемся списке над кодом появляется элемент для модуля макроса *EnvironmentEvents*. Выберите этот элемент, чтобы заполнить правый раскрывающийся список событиями для этого объекта, и выберите каждое событие, чтобы создать нужный код для захвата события. На этом этапе обработчик событий для проектов C# вызываться еще не будет, так как еще надо установить переменную событий *csharpProjectItemsEvents* на экземпляр объекта *ProjectItemsEvents*. Чтобы установить ее на экземпляр нужного объекта событий, создайте обработчик для *DTEEvents.OnStartupComplete* и разместите в нем код для подключения к событиям почти так же, как вы делали бы в надстройке:

```
Private Sub DTEEvents_OnStartupComplete() _
    Handles DTEEvents.OnStartupComplete

    csharpProjectItemsEvents = _
        DTE.Events.GetObject("CSharpProjectItemsEvents")
End Sub
```

После создания этого обработчика при перезапуске Visual Studio .NET инициируется обработчик *OnStartupComplete*, что приведет к подключению переменной событий. Разумеется, можно поместить этот же код в макрос и запустить его; Visual Studio .NET необязательно перезапускать для установки переменной событий. Вот пример такого макроса:

```
Sub ConnectCSharpProjectItemsEvents()
    csharpProjectItemsEvents = _
        DTE.Events.GetObject("CSharpProjectItemsEvents")
End Sub
```

Можно подключаться не только к событиям проектов Miscellaneous Files и Solution Items, но и проектов другого типа. Для этого надо изменить строку, передающуюся в метод *Events.GetObject*. Так, чтобы подключиться к событиям проекта Visual Basic .NET и событиям его элементов, используйте строки *VBProjectsEvents* и *VBProjectItemsEvents*. Строки *VJSharpProjectsEvents* и *VJSharpProjectItemsEvents* позволят подключиться к событиям проекта Microsoft Visual J#, а *eCSharpProjectsEvents* и *eCSharpProjectItemsEvents* — к событиям приложений C# Smart Device Application. Строки *eVBProjectsEvents* и *eVBProjectItemsEvents* помогут подключиться к событиям приложений Visual Basic .NET smart device application. Пример ProjectEvents демонстрирует подключение ко всем этим событиям проектов и элементов проектов. Он подключается к событиям, предоставляемым каждым типом проектов, и при инициации любого события выводит окно с описанием этого события.

# Управление конфигурациями компоновки

Редактирование и управление проектом — важная часть процесса разработки, но основное время тратится на компоновку и компиляцию проекта, а не на перемещение файлов в нем. Visual Studio .NET предоставляет объектную модель для компоновки решения и управления компиляцией проектов в этом решении. Корневой объект для управления компоновкой решения называется *SolutionBuild*; доступ к нему осуществляется через свойство *Solution.SolutionBuild*. Объект *ConfigurationManager* позволяет управлять компоновкой каждого проекта в решении. Доступ к нему осуществляется через свойство *Project.ConfigurationManager*.

## Управление параметрами решения

Для управления компоновкой решения Visual Studio .NET использует конфигурации решения (solution configurations). Конфигурации решения — это группа конфигураций проектов, описывающих процесс компоновки проектов в решении. Конфигурация проекта попросту говорит различным компиляторам, как создавать код для проекта. Каждый проект может содержать несколько конфигураций, между которыми можно переключаться в конфигурации решения, управляя процессом компоновки кода компиляторами. Самыми распространенными конфигурациями решений и проектов являются отладочная (debug) и финальная (release), которые предписывают выполнять компоновку проекта с отладочной информацией и оптимизацией кода соответственно. При создании проекта, такого как проект Windows Forms, Visual Studio .NET создает отладочную конфигурацию решения, содержащую отладочную конфигурацию проекта Windows Forms, и финальную конфигурацию решения, содержащую финальную конфигурацию проекта Windows Forms. Вы вправе создавать новые конфигурации решения, содержащие любые из имеющихся конфигураций проектов, или новые конфигурации проектов, которые можно загрузить в любую конфигурацию решения.

### Объекты *SolutionConfiguration* и *SolutionContext*

В объектной модели конфигурации решений представляются набором *SolutionConfigurations*, который содержит объекты *SolutionConfiguration*. Поскольку объект *SolutionConfigurations* является набором, для него можно применять стандартные методы перебора и использовать метод *Item* для поиска определенного объекта

*SolutionConfiguration* по имени. Для создания новой конфигурации решения служит метод *SolutionConfigurationsAdd*, который копирует существующую конфигурацию решения и переименовывает ее указанным именем:

```
public EnvDTE.SolutionConfiguration Add(string NewName,
    string ExistingName, bool Propagate)
```

А вот аргументы, передаваемые этому методу.

- ***NewName*** Имя новой конфигурации решения, Оно не может совпадать с именем любой из существующих конфигураций решения и должно следовать правилам именования файлов. (Оно не может содержать символы \, /, :, *, ?, «, < или >.)
- ***ExistingName*** Имя существующей конфигурации решения, копируемой для создания новой конфигурации решения, или строка «*<Default>*». Если указана строка «*<Default>*», то в качестве источника для копирования используется текущая активная конфигурация решения.
- ***Propagate*** Если этому параметру задано *true,* при создании новой конфигурации решения делаются копии всех конфигураций проектов, входящих в эту конфигурацию решения, им присваивается то же имя, что и новой конфигурации решения, и все эти копии конфигураций проектов загружаются в новую конфигурацию решения. Если же задано *false,* создается новая конфигурация решения, и ей сопоставляются те же конфигурации проектов, которые входили в исходную конфигурацию решения.

Объект *SolutionConfiguration* имеет один метод и одно свойство, которые следует упомянуть отдельно. Имя одного из этих методов — *Activate*; при выполнении компоновки как через пользовательский интерфейс, так и через объектную модель с применением метода *SolutionBuild.Build*, текущий активный объект *SolutionConfiguration* представляет используемую в компоновке конфигурацию. Таким образом, активизация какой-либо конфигурации решения приводит к использованию в любых действиях по компоновке активной конфигурации. Второй важный элемент — свойство *SolutionContexts*. Как сказано выше, *SolutionConfiguration* — это контейнер проектов в решении и конфигураций проектов, сопоставленных данной конфигурации решения. Свойство *SolutionConfiguration.SolutionContexts* возвращает список этих проектов и конфигураций этих проектов, которые надо задействовать при компоновке.

Чтобы выбрать конфигурацию проекта, используемую при компоновке решения, укажите в свойстве *ConfigurationName* объекта *SolutionContext* имя любой конфигурации проекта, поддерживаемой этим проектом. Следующий макрос изменяет отладочную конфигурацию решения для применения финальной конфигурации проекта, загруженного в решение:

```
Sub ChangeProjectConfiguration()
    Dim solutionBuild As EnvDTE.SolutionBuild
    Dim solutionCfgs As EnvDTE.SolutionConfigurations
    Dim solutionCfg As EnvDTE.SolutionConfiguration
    Dim solutionContext As EnvDTE.SolutionContext
    'Ищем отладочную конфигурацию решения:
    solutionBuild = DTE.Solution.SolutionBuild
```

```
    solutionCfgs = solutionBuild.SolutionConfigurations
    solutionCfg = solutionCfgs.Item("Debug")
    'Получаем контекст решения для первого проекта:
    solutionContext = solutionCfg.SolutionContexts.Item(1)
    'Изменяем отладочный контекст для сборки финальной
    'конфигурации проекта:
    solutionContext.ConfigurationName = "Release"
    'Переустанавливаем флаг сборки для этого контекста:
    solutionContext.ShouldBuild = True
End Sub
```

Модификацией *SolutionContext* можно не только выбрать конфигурацию проекта, которую следует использовать в конфигурации решения, но и устанавливать другие параметры, например, указывающие, следует ли собирать конфигурацию проекта. Такое действие выполняется в предпоследней строке предыдущего макроса, в которой свойству *ShouldBuild* задается *true.* В этом макросе такое действие необходимо, поскольку ожидается, что при первом создании отладочной конфигурации решения она не содержит финальных конфигураций проектов. Поэтому финальная конфигурация не собирается в этой конфигурации решения, и при включении финальной конфигурации проекта в отладочную конфигурацию решения это состояние «не собирать» переносится вместе с ним.

### Стартовые проекты

Когда вы запускаете решение (как правило, клавишей F5), компоновщик проекта вначале проверяет что все проекты, подлежащие компоновке, представлены последними версиями, затем просматривает список проектов, объявленных как стартовые, и запускает их по очереди. Создать список стартовых проектов можно через пользовательский интерфейс, щелкнув правой кнопкой мыши узел решения в Solution Explorer и выбрав из контекстного меню команду Set StartUp Projects. Вы увидите диалоговое окно Solution Property Pages (рис. 8-4), где можно установить стартовые проекты для решения, содержащего четыре приложения Windows Forms.

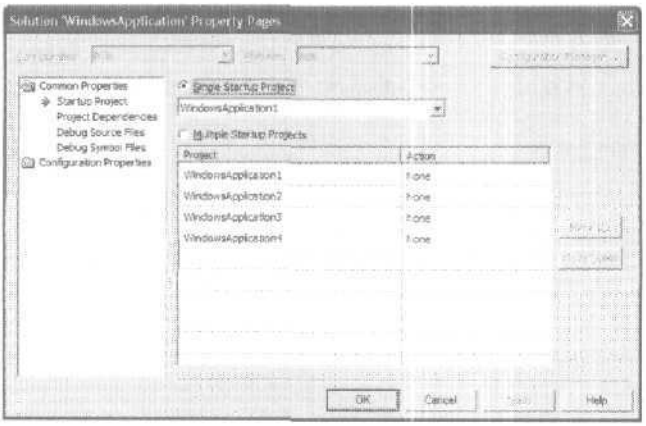



**Рис. 8-4.** *Установка проектов, выполняющихся при запуске решения*

Стартовые проекты также можно установить через объектную модель в свойстве *SolutionBuild.StartupProjects.* Этому свойству присваивается значение типа *System.Qbject,* содержащее проекты, которые должны выполняться при запуске решения. Значение, передаваемое *StartupProjects,* может быть представлено в двух видах: строка, указывающая уникальное имя проекта (таким образом, для запуска будет установлен единственный проект), или массив из *System.Qbject* (содержащий одно или несколько уникальных имен проектов, что приведет к запуску нескольких проектов).

Допустим, открытое решение содержит два проекта, каждый из которых нужно назначить стартовым проектом. Для этого можно использовать код вроде этого:

```
Sub SetStartupProjects()
    Dim startupProjects(1) As Object
    startupProjects(0) = DTE.Solution.Projects.Item(1).UniqueName
    startupProjects(1) = DTE.Solution.Projects.Item(2).UniqueName
    DTE.Solution.SolutionBuild.StartupProjects = StartupProjects
End Sub
```

Если стартовым нужно назначить только один проект, код будет выглядеть так:

```
Sub SetStartupProject()
    Dim startupProject As String
    startupProject = DTE.Solution.Projects.Item(1).UniqueName
    DTE.Solution.SolutionBuild.StartupProjects = startupProject
End Sub
```

Устанавливая стартовые проекты, следите за тем, чтобы назначать только проекты, подлежащие компоновке. Если среди проектов, переданных *SolutionBuild.StartupProjects,* встретится, например, уникальное имя проекта Miscellaneous Files или Solution Items, возникнет ошибка.

**Примечание** Visual Studio .NET содержит ошибку, влияющую на применение свойства *SolutionBuild.StartupProjects* для нескольких проектов. При изменении этого свойства с запуска одного проекта на запуск нескольких изменится список стартовых проектов. Однако переключатель Multiple Startup Projects не будет установлен; его потребуется установить вручную. Установка свойства *SolutionBuildStartupProjects* не повлияет на состояние этого переключателя.

### Зависимости проектов

В работе с решением, содержащим несколько проектов, может получиться, что компоненты одного проекта будут зависеть от результата работы другого проекта. Пример — проект элемента управления UserControl, размещаемого на форме приложения Windows Forms с именем WinForm. Поскольку изменения в проекте UserControl могут повлиять на использование этого элемента управления проектом Windows Forms, проект UserControl должен быть откомпилирован до компиляции проекта WinForm. Чтобы обозначить отношения между двумя проектами, надо создать зависимости проектов (project dependency). Зависимости между двумя или более проектами можно изобразить с помощью графа зависимостей; на рис 8-5

показан граф зависимостей проектов WinForm и UserControl. Стрелка указывает на проект, от которого зависит другой проект.




**Рис. 8-5.** *Граф зависимостей, показывающий зависимость проекта WinForm от проекта UserControl*

Предположим, мы добавили в решение новый проект: библиотеку классов ClassLib, реализующую функциональность, используемую как проектом WinForm. так и UserControl. Граф зависимостей для этого решения показан на рис, 8-6.

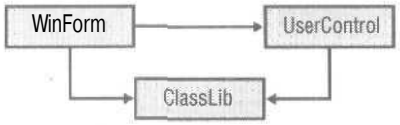



**Рис. 8-6.** *Граф зависимостей для трех проектов*

Из этого графа видно, что проект WinForm не может быть собран до того, как будут собраны проекты UserControl и ClassLib. До компоновки проекта UserControl следует собрать только проект ClassLib. Если при компоновке решения система компоновки решит начать с проекта UserControl, это приведет к предварительной компоновке проекта ClassLib. Если система компоновки выберет первым проект ClassLib, он будет собран тут же, так как не зависит от других проектов. При компоновке проекта UserControl проект ClassLib не собирается повторно, поскольку предыдущая версия не устарела. Независимо от того, какой проект система компоновки выберет первым, последним будет собран WinForm. так как он зависит от двух других проектов.

С графом зависимостей может возникнуть проблема, если создать циклическую зависимость, в которой один или более проектов окажутся взаимозависимыми. Допустим, проект WinForm зависит от проекта UserControl, проект UserControl – от проекта ClassLib, а ClassLib — от WinForm. Образуется циклическая зависимость, показанная на рис. 8-7.

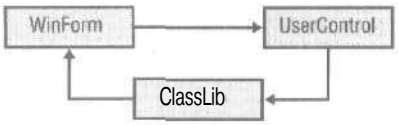



**Рис. 8-7.** *Граф циклической зависимости трех проектов*

При компоновке проекта WinForm вследствие зависимости будет инициирована компоновка UserControl. Это в свою очередь инициирует компоновку проекта ClassLib, который зависит от WinForm. Если система компоновки Visual Studio .NET окажется неспособна обнаружить зацикливание, попытки найти первый проект для компоновки будут продолжаться бесконечно. Но Visual Studio .NET достаточно умна для того, чтобы обнаруживать циклические зависимости и запрещать их.

Зависимости между проектами можно создавать через пользовательский интерфейс командой меню Project | Project Dependencies. Откроется диалоговое окно

Project Dependencies (рис. 8-8). В нем показаны все проекты, которые могут быть отмечены как зависимости для проекта UserControl. Флажок WinForm затенен, так как установлена зависимость WinForm от UserControl, и Visual Studio .NET не допустит создания цикла между проектами WinForm и UserControl.




Рис. 8-8. *Установка зависимостей проектов*

Зависимости сборки можно также установить через объектную модель. Свойство *SolutionBuild.BuildDependencies* возвращает объект *BuildDependencies*, представляющий собой набор объектов *BuildDependency*. Этот набор индексируется методом *Item* — ему можно передать числовой индекс, объект *EnvDTE.Project* или уникальное имя проекта. Каждый проект решения имеет свой объект *EnvDTE.BuildDependency*, через свойство *RequiredProjects* которого можно добавлять, удалять или получать зависимости для проекта. Следующий макрос отображает в окне Output доступные проекты открытого решения и все проекты, от которых он зависит.

```
Sub Depends()
    Dim projectDep As EnvDTE.BuildDependency
    Dim project As EnvDTE.Project
    Dim owp As New InsideVSNET.Utilities.OutputWindowPaneEx(DTE,
        "Build dependencies")

    For Each projectDep In DTE.Solution.SolutionBuild.BuildDependencies
        Dim reqProjects As Object()

        owp.Write("The project ")
        owp.Write(projectDep.Project.Name)
        owp.WriteLine(" relies on:")
        reqProjects = projectDep.RequiredProjects
        If (reqProjects.Length = 0) Then
            owp.WriteLine(vbTab + "<None>")
        Else
            For Each project In reqProjects
                owp.WriteLine(vbTab + project.Name)
```

```
            Next
        End If
        owp.WriteLine()
    Next
End Sub
```

Объект *BuildDependency* позволяет создавать макросы и надстройки, устанавливающие зависимости между двумя и более проектами. Допустим, в нашем текущем примере загружено решение, содержащее проекты WinForm, UserControl и ClassLib, но зависимости не установлены. Объект *BuildDependency* поддерживает три метода для модификации списка проектов, от которых зависит данный проект: *AddProject*, *RemoveProject* и *RemoveAllProjects*. *AddProject* и *RemoveProject* принимают уникальное имя проекта, который надо добавить или удалить из списка зависимостей данного проекта. *RemoveAllProjects* не принимает аргументов и удаляет все зависимости проекта. Следующий макрос — *SetDependencies* — создает зависимости для решения из трех проектов, соответствующие графу зависимостей на рис. 8-6:

```
Sub SetDependencies()
    Dim buildDependencies As EnvDTE.BuildDependencies
    Dim buildDependency As EnvDTE.BuildDependency
    Dim project As EnvDTE.Project

    Dim winFormUniqueName As String
    Dim userControlUniqueName As String
    Dim classLibUniqueName As String

    'Собираем уникальные имена каждого проекта.
    For Each project In DTE.Solution.Projects
        If (project.Name = "WinForm") Then
            winFormUniqueName = project.UniqueName
        ElseIf (project.Name = "UserControl") Then
            userControlUniqueName = project.UniqueName
        ElseIf (project.Name = "ClassLib") Then
            classLibUniqueName = project.UniqueName
        End If
    Next

    buildDependencies = DTE.Solution.SolutionBuild.BuildDependencies
    For Each buildDependency In buildDependencies
        If (buildDependency.Project.Name = "WinForm") Then
            buildDependency.RemoveAllProjects()
            'Добавляем все проекты, кроне WinForm.
            'в список зависимостей:
            buildDependency.AddProject(userControlUniqueName)
            buildDependency.AddProject(classLibUniqueName)
        ElseIf (buildDependency.Project.Name = "UserControl") Then
            buildDependency.RemoveAllProjects()
            'Добавляем зависимость для проекта ClassLib:
            buildDependency.AddProject(classLibUniqueName)
```

```
        End If
    Next
End Sub
```

## Управление параметрами проекта

Конфигурации решений объединяют конфигурации проектов. Каждый проект содержит несколько конфигураций, управляющих тем, как компилятор создает программный код для этого проекта. Поскольку проекту может быть сопоставлено несколько конфигураций, можно создавать разные версии программы.

### Объект *ConfigurationManager*

Конфигурации проекта управляются через объект *ConfigurationManager,* который содержит набор объектов *Configuration* и позволяет создавать новые конфигурации. Конфигурации проекта представлены в форме координатной сетки, где по одной оси меняется тип конфигурации (например, отладочная или финальная), а по другой — платформа, для которой собирается конфигурация. В настоящее время Visual Studio .NET поддерживает платформу Win32 для 32-битных версий Windows на процессоре x8б, .NET, если проект компилируется для платформы Microsoft .NET, и Pocket PC и Windows CE, если проект собирается для платформ Windows CE. Поскольку собирать проекты одновременно для нескольких платформ нельзя, на второй оси всегда указывается только одна координата.

Найти определенную конфигурацию проекта можно несколькими способами. Первый — применить знакомый метод *Item,* доступный во всех объектах-наборах. Но в отличие от методов *Item* большинства других наборов метод *ConfigurationManager.Item* требует наличия двух параметров. Первый может быть числовым индексом и адресует всю сетку платформ и конфигураций. Кроме того, используя *Item* можно указать *Configuration* напрямую, передав методу имя конфигурации в первом параметре и имя платформы во втором. Допустим, в Solution Explorer открыт проект Visual C++. Найти объект *Configuration* для отладочной сборки под Win32 можно так:

```
Sub RetrieveDebugWin32Configuration()
    Dim conflg As Configuration
    Dim project As EnvDTE.Project
    project = DTE.Solution.Projects.Item(1)
    config = project.ConfigurationManager.Item("Debug", "Win32")
End Sub
```

Другой способ получить определенную конфигурацию — использовать методы *ConfigurationManager.ConfigurationRow* и *ConfigurationManager.Platform* которые принимают тип сборки и имя платформы соответственно. Эти методы возвращают набор объектов *Configuration,* просматривая который, можно найти интересующий элемент. Метод *ConfigurationRow* возвращает список всех конфигураций с указанным именем; метод *Platform* возвращает список всех конфигураций для заданной платформы. Эти методы особенно полезны, когда надо изменить параметры конфигураций, которые тесно взаимосвязаны, скажем, просмотреть все конфигурации Win32 проекта Visual C++ и включить управляемые расширения,

что позволит вашей программе задействовать .NET Framework в коде C++. Следующий пример кода делает именно это. После поиска конфигураций Win32, имеющихся в проекте, он получает объект *Properties* для этих конфигураций и задает их свойствам *ManagedExtension*значение *true,* позволяя компилятору создавать код, способный работать с .NET Framework.

```
Sub SetManagedExtensionsProperty()
    Dim configManager As ConfigurationManager
    Dim configs As Configurations
    Dim config As Configuration
    Dim project As EnvDTE.Project
    project = DTE.Solution.Projects.Item(1)
    configManager = project.ConfigurationManager
    configs = configManager.Platform("Win32")
    For Each config In configs
        Dim prop As EnvDTE.Property
        prop = config.Properties.Item("ManagedExtensions")
        prop.Value = True
    Next
End Sub
```

На основе существующих конфигураций проекта можно создавать новые точно так же, как можно создавать новые конфигурации решения, копируя имеющиеся конфигурации решения. Новые конфигурации проекта создает метод *ConfigurationManager.AddConfigurationRow*Он принимает в качестве параметров имя новой конфигурации и имя существующей конфигурации, которая играет роль шаблона для создания новой конфигурации, *AddConfigurationRow*также принимает как аргумент значение типа *Boolean.* Этот параметр с именем *Propagate*действует так же, как параметр *Propagate* метода *SolutionConfigurations.Add*но с противоположным эффектом. При вызове метода *SolutionConfigurations.Add*с параметром *Propagate,* установленным в *true,* создается копия конфигурации решения и всех конфигураций проекта, которые она включает. А при вызове метода *AddConfigurationRow* с параметром *Propagate,* равным *true,* копируется текущая активная конфигурация решения, ее имя устанавливается как имя новой конфигурации проекта, и в новую конфигурацию решения включается созданная конфигурация проекта.

---

**Примечание** Объект *ConfigurationManager* содержит метод *AddPlatform,* который работает почти так же, как *AddConfigurationRow,* но добавляет в сетку конфигураций новый ряд платформ. При вызове этого метода в любой из существующих на данный момент версий языков Microsoft генерируется исключение, так как для этих типов проектов нельзя добавлять новые платформы. Это не значит, что данный метод не будет работать с языками сторонних производителей или будущими версиями языков программирования от Microsoft.

---

Большинство типов проектов поддерживает только один тип платформы, а некоторые проекты, такие как проекты установки, не поддерживают никаких платформ — создаваемый результат не привязывается к определенной платфор-

ме. Проекту установки не важно, для какой платформы предназначено его содержимое — Win32 или .NET; его задача — хранить файлы, которые должны быть установлены на компьютер пользователя, поэтому при компоновке проекта установки платформа не имеет значения. Поскольку сетка типов конфигурации компоновки не может быть одномерной, для таких типов проектов создается псевдоплатформа с именем «*<N/A>*».

### Свойства конфигураций проектов

Конфигурации проектов отличаются устанавливаемыми значениями свойств. Например, одно из отличий между отладочной и финальной конфигурацией в том, что в отладочной конфигурации не оптимизируется код, что облегчает отладку, а в финальной конфигурации оптимизация включается, ускоряя выполнение. Такие свойства устанавливаются через объект, возвращаемый свойством *ConfigurationProperties*. Как было показано в макросе *SetManagedExtensionsProperty* это свойство возвращает объект *EnvDTE.Properties* — тот же объект, что служит в Visual Studio .NET для установки значений свойств различных объектов. Следующий макрос получает отладочную и финальную конфигурации проекта, читает свойство конфигурации *Optimize* (типа *Boolean*), инвертирует его и записывает обратно в конфигурацию. Таким образом, свойство *Optimize* каждой конфигурации меняет свое значение на противоположное.

```
Sub SwapOptimizationSettings()
    Dim project As EnvDTE.Project
    Dim configManager As EnvDTE.ConfigurationManager
    Dim configs As EnvDTE.Configurations
    Dim config As EnvDTE.Configuration
    Dim props As EnvDTE.Properties

    'Ищем объект ConfigurationManager для проекта:
    project = DTE.Solution.Projects.Item(1)
    configManager = project.ConfigurationManager

    'Получаем отладочные конфигурации,
    configs = configManager.ConfigurationRow("Debug")
    'Просматриваем каждую конфигурацию в ряду отладочных конфигураций.
    For Each config In configs
        Dim optimize As Boolean
        'Получаем свойство Optimize конфигурации.
        props = config.Properties
        optimize = props.Item("Optimize").Value
        'Инвертируем значение.
        props.Item("Optimize").Value = Not optimize
    Next

    'Повторяем для финальных конфигураций.
    configs = configManager.ConfigurationRow("Release")
    For Each config In configs
        Dim optimize As Boolean
        'Получаем свойство Optimize конфигурации.
```

```
        props = config.Properties
        optimize = props.Item("Optimize").Value
        'Инвертируем значение.
        props.Item("Optimize").Value = Not optimize
    Next

End Sub
```

## События компоновки

На каждой стадии компоновки Visual Studio .NET инициирует события, которые могут быть перехвачены макросом или надстройкой, что позволяет использовать служебный код. Таких событий четыре. Вот их сигнатуры:

```
void OnBuildBegin(EnvDTE.vsBuildScope Scope, EnvDTE.vsBuildAction Action);
void OnBuildProjConfigBegin(string Project, string ProjectConfig,
    string Platform, string SolutionConfig);
void OnBuildProjConfigDone(string Project, string ProjectConfig,
    string Platform, string SolutionConfig, bool Success);
void OnBuildDone(EnvDTE.vsBuildScope Scope, EnvDTE.vsBuildAction Action);
```

А вот описания обработчиков событий.

- ***OnBuildBegin*** Событие инициируется перед началом компоновки. Обработчику этого события передаются два аргумента. Первый — перечисление типа *EnvDTE.vsBuildScope.* Он может принимать значения *vsBuildScopeBatch* (если вы запустили пакетную компоновк> одного или нескольких проектов), *vsBuildScopeProject* (если вы запустили компоновку одного проекта, щелкнув его правой кнопкой мыши и выбрав команду Build) или *vsBuildScopeSolution* (если вы выбрали компоновку активной конфигурации решения). Второй аргумент имеет тип *EnvDTE.vsBuildAction* и может принимать значения *vsBuildActionBuild* (если для конфигурации проекта или решения выполняется компиляция), *vsBuildActionClean* (если результат компоновки конфигурации проекта или решения удаляется с диска), *vsBuildActionDeploy* (если конфигурация проекта или решения развертывается в месте назначения) или *vsBuildActionRebuildAll* (если для конфигурации проекта или решения выполняется повторная компоновка, даже когда зависимости проектов не гарантируют повторной компоновки).
- ***OnBuildProjConfigBegin*** Событие инициируется в начале компоновки конфигурации проекта. Обработчику передаются четыре аргумента типа *string:* первый содержит уникальное имя собираемого проекта, второй — имя собираемой конфигурации, третий — имя платформы, а четвертый — имя собираемой конфигурации решения.
- ***OnBuildProjConfigDone*** Событие инициируется после компоновки конфигурации проекта. Обработчику передаются те же аргументы, что и обработчику *OnBuildProjConfigBegin,* плюс значение типа *Boolean,* указывающее, как прошла компоновка конфигурации: успешно *(true)* или нет *(false).*
- ***OnBuildDone*** Событие инициируется по завершении всех этапов компоновки независимо от результата.

Среди примеров к книге есть BuildEvents, иллюстрирующий подключение к каждому из событий компоновки. При вызове каждого обработчика события переданная ему информация отображается в окне вывода, которое содержит сведения об аргументах, переданных каждому обработчику. Так, если создать решение, содержащее два проекта (ClassLibrary1 и ClassLibrary2), загрузить этот пример и провести компоновку решения, выбрав команду меню Build | Build Solution, отобразится следующая информация:

```
OnBuildBegin
    Scope: vsBuildScopeSolution
    Action: vsBuildActionBuild

OnBuildProjConfigBegin
    Project: ClassLibrary1.csproj
    Platform: .NET
    Solution Configuration: Debug

OnBuildProjConfigDone
    Project: ClassLibrary1.csproj
    Platform: .NET
    Solution Configuration: Debug
    Success: True

OnBuildProjConfigBegin
    Project: ..\ClassLibrary2\ClassLibrary2.csproj
    Platform: .NET
    Solution Configuration: Debug

OnBuildProjConfigDone
    Project: ..\ClassLibrary2\ClassLibrary2.csproj
    Platform: .NET
    Solution Configuration: Debug
    Success: True

OnBuildDone
    Scope: vsBuildScopeSolution
    Action: vsBuildActionBuild
```

Эта информация отражает этапы компоновки данного решения. Компоновка начинается с вызова обработчика события *OnBuildBegin,* за которым следует компоновка каждой конфигурации проекта из состава конфигурации решения по очереди с вызовом в конце обработчика события *OnBuildDone,* сигнализирующего об окончании процесса компоновки. В Visual Studio .NET 2003 события *OnBuildProjConfigBegin* и *OnBuildProjConfigEnd* инициируются друг за другом без других событий компоновки между ними. Однако макрос или надстройка не должны полагаться на такой порядок событий, если вы планируете задействовать этот код в будущих версиях Visual Studio .NET, так как эти версии могут использовать преимущества многопроцессорных компьютеров, собирая одну конфигурацию проекта на одном процессоре, а другую — на другом. Если макрос или надстройка привязаны к такому порядку инициации событий, код может работать неверно.

# Сохранение информации проектов и решений в сеансах IDE

Иногда надстройке или макросу может потребоваться сохранить информацию в файле проекта или решения. Объектная модель позволяет это сделать с помощью объекта *EnvDTE.Globals*. Получить его можно через свойство *Globals* проекта или решения:

```
Sub SolutionGlobals()
    Dim globals As EnvDTE.Globals
    globals = DTE.Solution.Globals
End Sub

Sub ProjectGlobals()
    Dim globals As EnvDTE.Globals
    globals = DTE.Solution.Projects.Item(1).Globals
End Sub
```

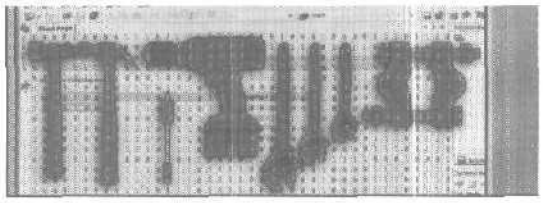
Объект *Globals* для объектов *Solution* и *Project* действует почти так же, как объект *Globals* для объекта DTE. Но, во-первых, если макрос или надстройка сохраняют информацию в файле решения или проекта, то даже если для этой переменной установлен флаг *VariablePersists* данные могут не записаться в файл. Дело в том, что внесение изменений в переменную переводит файл проекта или решения в измененное состояние (modified state). Если вы закроете файл решения или проекта без сохранения измененных файлов, данные не запишутся. Во-вторых, в отличие от объекта *EnvDTE.Globals* для объекта DTE, который может хранить информацию во многих форматах, данные, хранящиеся в файле проекта или решения, могут быть только в строковом формате. Поскольку файлы проектов и решения являются текстовыми, вся хранимая в них информация также должна представляться в текстовом виде. Это не значит, что в объекте *Globals* проекта или решения нельзя хранить информацию иного типа. Просто при записи этих данных в файл проекта или решения будет предпринята попытка преобразовать данные в строку. Если она не удастся, данные не запишутся. Кроме того, поскольку при хранении в файлах проектов и решений информация преобразуется в строку, то после восстановления объекта *Globals* по содержимому этих файлов все данные также будут представлены в виде строки. Макрос или надстройка должны сами определить нужный формат этих данных.

Выше в этой главе вы видели сценарий надстройки для решения, которая при каждой компоновке увеличивала значение счетчика, учитывая число компоновок. Объект *Solution.Globals* предлагает хорошее место для хранения этого значения, что показано в примере BuildCounter. Этот пример при загрузке в решение сначала подключается к событию *OnBuildDone*. При каждой инициации события *OnBuildDone* он проверяет наличие переменной *BuildCounter* в объекте *Globals* для решения, Если она существует, ее значение увеличивается и сохраняется в объекте *Globals*. В противном случае сохраняется значение *1*. Код обработчика события *OnBuildDone* таков:

```
void OnBuildDone(EnvDTE.vsBuildScope Scope, EnvDTE.vsBuildAction Action)
{
    //Считаем число компоновок, сохраняя значение счетчика
    //в файле решения через объект Globals:
    Globals globals;
    globals = applicationObject.Solution.Globals;
    if(globals.get_VariableExists("BuildCounter"))
    {
        //Счетчик существует. Увеличиваем его:
        System.Int32 int32;
        int32 = System.Int32.Parse((string)globals["BuildCounter"]);
        int32++;
        globals["BuildCounter"] = int32.ToString();
    }
    else
    {
        //Переменная никогда не создавалась. Начинаем счет:
        globals["BuildCounter"] = 1.ToString();
        globals.set_VariablePersists("BuildCounter", true);
    }
}
```

## Что дальше

В этой главе мы рассмотрели совместное использование фрагментов объектной модели для программного управления многими типами проектов, которые могут быть загружены в решение. Мы также познакомились с различными объектами, позволяющими модифицировать процесс компоновки проектов. В следующей главе мы используем эти концепции управления проектами в создании мастеров для Visual Studio .NET.



ГЛАВА

# 9

# Мастера Visual Studio .NET

В этой главе мы обсудим создание мастеров для Microsoft Visual Studio .NET.

## Общий обзор

Мастера позволяют упростить выполнение сложных или часто повторяющихся задач. В Windows масса разных мастеров, например, мастера, помогающие подключить принтер или сеть или разобраться в причинах возникновения нештатной ситуации. А в Visual Studio .NET мастера применяются для генерации кода.

### Типы мастеров

В Visual Studio .NET можно создавать и запускать мастера трех типов. Возможно, самый известный — это мастер нового проекта (New Project wizard), который генерирует код для нового проекта. Мастера этого типа активизируются при выборе пользователем элемента на правой панели диалогового окна New Project, отображеное командой File | New | Project.

Второй тип мастера — мастер добавления нового элемента (Add New Item wizard). После создания нового проекта пользователю часто бывает нужно добавить к нему новые файлы, скажем, классы, изображения или Web-страницы. Обычный способ вызова этих мастеров — щелкнуть правой кнопкой данный проект в окне Solution Explorer и выбрать Add | Add New Item. Из появившегося диалогового окна Add New Item можно запускать мастеров.

Реже всего используют нестандартный мастер (Custom wizard). Его нельзя вызвать напрямую из Visual Studio .NET — только явно задав в макросе, в надстройке или в другом мастере. Нестандартный мастер не может быть классифицирован как мастер нового проекта или добавления нового элемента, однако он может вести пользователя по этапам выполнения и этих задач. Такие мастера помогают упростить выполнение любых работ в Visual Studio .NET.

Мастеря любого типа создаются по одной схеме: создание COM-объекта, реализующего мастер, создание .vsz-файла (позволяющего Visual Studio .NET узнать о существовании мастера) и, наконец, создание шаблонов исходного кода.

## Создание объекта мастера

Мастер любого типа — это просто COM-объект, реализующий интерфейс *EnvDTE.IDTWizard*. Метод *Execute* — единственный метод этого интерфейса, и вызывается он, когда Visual Studio .NET загружает данный мастер. Вот его сигнатура:

```
public interface IDTWizard
{
    public void Execute(object Application, int hwndOwner,
        ref object[] ContextParams,
        ref object[] CustomParams,
        ref EnvDTE.wizardResult retval)
}
```

Методу *Execute* передается ряд аргументов.

- ***Application*** Объект *DTE* того экземпляра Visual Studio .NET, в котором мастер будет работать.
- ***hwndOwner*** Описатель окна, которое мастер может задействовать как родительское для любых элементов пользовательского интерфейса, которые он создает.
- ***ContextParams*** Массив элементов типа *object*, описывающий состояние Visual Studio .NET в момент старта мастера.
- ***CustomParams*** Массив элементов типа *object,* содержащий данные, определенные автором мастера и передаваемые объекту мастера.

Именно в методе *Execute* выполняется вся работа мастера. Visual Studio .NET не накладывает ограничений на реализацию мастера, кроме необходимости реализации COM-объектом интерфейса *IDTWizard*. Мастер может отображать пользовательский интерфейс, чтобы задать пользователю вопросы или он может задействовать информацию, передаваемую посредством различных аргументов методу *Execute*. Метод *Execute* — это что-то вроде функции *Main* в консольных приложениях Visual Basic или Visual C#: вызванный, он может делать все, что угодно.

Аргумент *ContextParams,* передаваемый методу *Execute*. — это массив элементов, заполненный значениями, вводимыми пользователем в диалоговом окне Add New Item или New Project. Этот массив также содержит ряд других значений, передаваемых от Visual Studio .NET и помогающих мастеру определить, как генерировать код. Эти значения массива меняются в зависимости от того, какой мастер запускается: нового проекта, добавления нового элемента или нестандартный. Вот аргументы, передаваемые для разных типов проектов и порядок, в котором эти типы появляются, (табл. 9-1 и 9-2).

**Табл. 9-1. Значения массива *ContextParams*, передаваемые мастеру нового проекта**

| Значение | Описание |
|---|---|
| *Тип мастера* | Значение *EnvDTE.Constants.vsWizardNewProject*. |
| *Имя проекта* | Имя создаваемого проекта. В него не входит расширение имени файла. |
| *Локальный каталог* | Каталог, в котором нужно создать проект или решение. |
| *Установочный каталог* | Место на диске, куда установлена Visual Studio .NET. |
| *Взаимоисключение* | Если равно *true*, мастер должен закрыть текущее решение и создать новое, если *false* — текущее решение закрывать не надо и данный проект нужно добавить в открытый в данный момент файл решения. |
| *Имя решения* | Если задано, то это имя решения, которое необходимо создать. Имя решения доступно, если в диалоговом окне New Project установлен флажок Create Directory For Solution. |
| *Скрытый* | Булевский флаг, указывающий, должен ли мастер запускаться без отображения элементов пользовательского интерфейса. Если он равен *true*, при генерации кода применяются значения по умолчанию. |

**Табл. 9-2. Значения массива *ContextParams*, передаваемые мастеру добавления нового элемента**

| Значение | Описание |
|---|---|
| *Тип мастера* | Значение *EnvDTE.Constants.vsWizardAddItem*. |
| *Имя проекта* | Имя проекта, к которому нужно добавить новый элемент. |
| *Элементы проекта* | Набор *EnvDTE.ProjectItems*, к которому нужно добавить данный элемент. |
| *Новое место элемента* | Каталог диска, в котором нужно создать данный элемент. |
| *Новое имя элемента* | Имя, введенное пользователем в поле Name диалогового окна Add New Item. |
| *Установочный каталог продукта* | Каталог, в котором установлен соответствующий язык программирования. |
| *Скрытый* | Булевский флаг, указывающий, должен ли мастер запускаться без отображения элементов пользовательского интерфейса, Если он равен *true*, при генерации кода применяются значения по умолчанию. |

Мы не приводим таблицу значений параметров для нестандартного мастера, так как они не регламентируются Visual Studio .NET, — они определяются соответствующей надстройкой, макросом и даже другим мастером. Тип нестандартного мастера и передаваемые ему параметры *CustomParams* мы обсудим позже.

При запуске мастер должен убедиться, что первый элемент массива контекстных параметров — константа *EnvDTE.Constants.vsWizardNewProject* (если это мастер нового проекта) или константа *EnvDTE.Constants.vsWizardAddItem* (если это мастер добавления нового элемента). Если GUID не соответствует типу мастера, реализуемому данным объектом, объект должен вернуть код ошибки *wizardResult.wizardResultFailure* в аргументе *retval* метода *IDTWizard.Execute*.

Примечание При проверке GUID, передаваемого в качестве типа мастера, нужно выполнить сравнение без учета регистра, так как константы *vsWizardNewProject* и *vsWizardAddItem* могут иметь иной регистр, чем значение, передаваемое от Visual Studio .NET в первом элементе массива *ContextParams*.

Вот пример реализации мастера и код, извлекающий элементы из массива *ContextParams* (листинг 9-1).

**Листинг 9-1.** Исходный код надстройки, реализующей мастер

**Wizard.cs**

```
using System;
using System.Runtime.InteropServices;

namespace BasicWizard
{
    [GuidAttribute("E5D0A8B2-A449-4d3b-B47B-99494D23A58B"),
    ProgIdAttribute("MyWizard.Wizard")]
    public class Wizard : EnvDTE.IDTWizard
    {
        public void Execute(object Application, int hwndOwner,
                            ref object[] ContextParams,
                            ref object[] CustomParams,
                            ref EnvDTE.wizardResult retval)
        {
            EnvDTE.DTE application = (EnvDTE.DTE)Application;
            string wizardType = (string)ContextParams[0];

            if(System.String.Compare(wizardType,
                EnvDTE.Constants.vsWizardNewProject, true) == 0)
             {
                string newProjectName = (string)ContextParams[1];
                string newProjectLocation = (string)ContextParams[2];
                string visualStudioInstallDirectory *
                    (string)ContextParams[3];
                bool exclusiveProject = (bool)ContextParams[4];
                string newSolutionName = (string)ContextParams[5];
                bool runSilent = (bool)ContextParams[6];
            }
            else if(System.String.Compare(wizardType,
                EnvDTE.Constants.vsWizardAddItem, true) -- 0)
            {
                string projectName = (string)ContextParams[1];
                EnvDTE.ProjectItems projectItems =
                    (EnvDTE.ProjectItems)ContextParams[2];
                string newItemLocation = (string)ContextParams[3];
                string newItemName = (string)ContextParams[4];
                string productInstallDirectory = (string)ContextParams[5];
```

*см. след. стр.*

```
                bool runSilent = (bool)ContextParams[6];
            }
            else
            {
                //ОШИБКА! Неизвестный тип мастера.
            }
        }
    }
}
```

# Создание .vsz-файла

Как вы помните из главы *6,* чтобы создать надстройку, нужно предоставить Visual Studio .NET данные, позволяющие ей узнать, что надстройка доступна для загрузки. Эти данные, размещаемые в системном реестре, включают ProgID и некоторые сведения, уточняющие способ загрузки надстройки. Мастеру тоже требуется способ заявить о своей доступности, но в отличие от надстроек для этого нужно поработать с файловой системой. А именно — создать иерархию каталогов на диске и поместить в ней .vsz-файлы.

Файл .vsz имеет простой текстовый формат. В начале файла находится строка *«VSWIZARD7.0»,* сообщающая Visual Studio .NET, что данный файл содержит определение мастера и этот мастер может работать в Visual Studio .NET версии 7 и дальнейших. В следующей строке текста находится оператор, начинающийся с *«Wizard=»,* за которым следует ProgID или идентификатор класса (ClassID) COM-объекта, реализуемого мастером. Если использовать ProgID кода Wizard.cs из листинга 9-1, то данная строка .vsz-файла будет выглядеть так:

```
Wizard=MyWizard.Wizard
```

Можно задействовать и формат с ClassID:

```
Wizard={E5D0A8B2-A449-4d3b-B47B-99494D23A58B}
```

После строки с ProgID или ClassID можно поместить список данных, определяемых пользователем. Там могут находиться любые строковые данные, которые нужно передать мастеру и которые могут быть статическими (помещаемыми вручную в .vsz-файл во время разработки) или генерируемыми при установке мастера программой установки. Каждая из строк этих данных начинается с *«Param=»,* и мастер может запрашивать любое количество таких записей (в том числе 0). Вот пример этих данных:

```
Param=Hello World
Param=Second line of data
```

Каждое выражение *Param* передается как элемент массива *CustomParams* при вызове метода *Execute* мастера и может быть извлечено в мастере кодом, подобным следующему коду C#:

```
for(int i = 0 ; i < CustomParams.Length ; i++)
{
    string data = (string)CustomParams[i];
```

```
    System.Windows.Forms.MessageBox.Show(data);
}
```

Примечание Даже в случае Visual Studio .NET 2003 (версия 7.1) первая строка .vsz-файла должна начинаться с *'VSWIZARD7.0'*, а не с *'VSWIZARD7.1'*.

## Где сохранить .vsz-файл

Чтобы пользователь смог запустить мастер, нужно поместить .vsz-файл в определенное место на диске, где его могут найти и сделать доступным для запуска диалоговые окна New Project или Add New Item. При выводе этих окон происходит чтение некоторого каталога или ряда каталогов на диске с целью открыть подкаталоги и прочитать содержащиеся там файлы. Имена этих каталогов вставляются в дерево в левой части диалогового окна, а любые подкаталоги вставляются как подчиненные элементы данного узла дерева. Когда пользователь выбирает узлы дерева, файлы, находящиеся в соответствующем этому узлу каталоге, отображаются в правой части диалогового окна.

К примеру, на рис. 9-1 показано диалоговое окно New Project с иерархией каталогов, к которой в каталог Extensibility Projects (тот каталог диска, в котором хранятся .vsz-файлы мастера надстроек) добавлен каталог A Sub Folder. Когда содержимое каталога Extensibility Projects копируется в каталог A Sub Folder, оно появляется в правой части диалогового окна, если выбрать этот новый каталог. Каталог A Sub Folder был создан в каталоге C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\Extensibility Projects (используется размещение, выбираемое при установке по умолчанию).

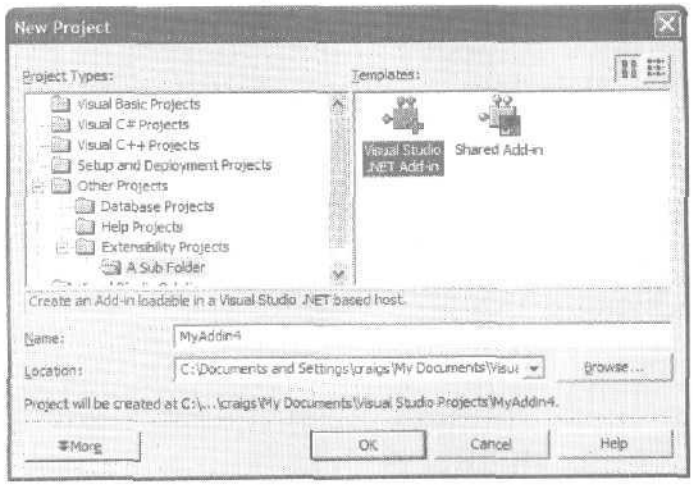



Рис. 9-1. *В диалоговом окне New Project показан новый подкаталог*

Место для размещения .vsz-файлов нового проекта можно найти программно, используя свойство *TemplatePath* объекта *Solution*. Следующий макрос отображает окно с сообщением, показывающее каталог, в котором можно сохранить .vsz-

файлы, чтобы они появились в узлах проектов Visual Basic и проектов Visual C# на правой стороне диалогового окна New Project:

```
Sub VSZLocation()
    ' Отобразить .vsz-путь для проектов Visual Basic.
    MsgBox(DTE.Solution.TemplatePath( _
        VSLangProj.PrjKind.prjKindVBProject))

    ' Отобразить .vsz-путь для проектов Св.
    MsgBox(DTE.Solution.TemplatePath( _
        VSLangProj.PrjKind.prjKindCSharpProject))
End Sub
```

**Примечание** Свойство *TemplatePath* названо неудачно — ему больше подошло бы имя *VSZFilePath*. Не путайте слово *Template* (шаблон) в имени данного свойства с шаблонами файлов (которые мы обсудим ниже).

Константы *prjKindVBProject* и *prjKindCSharpProject*, определенные в сборке метаданных VSLangProj.dll, — это GUID, представленные в виде строки. Как вам известно, существуют типы проектов, отличные от проектов Visual Basic и C#, но констант, которые можно передать свойству *TemplatePath* для этих типов проектов, не найти ни в одной из сборок. GUID для этих типов проектов можно найти вручную в системном реестре. В разделе HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\Projects находится список GUID, каждый из которых определяет тип проекта, поддерживаемый в Visual Studio .NET. Заменив аргумент в *Solution.TemplatePath* на один из этих GUID, получим путь к месту, где нужно сохранять .vsz-файлы, чтобы для данного типа проектов в диалоговом окне New Project появилась запись, соответствующая требуемому мастеру. Если в указанной области реестра мы будем искать vcproj (расширение файлов проектов Visual C++), то обнаружим, что GUID для проектов Visual C++ - {8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}. Этот GUID можно задействовать для определения пути к месту, где можно хранить .vsz-файлы, чтобы они появлялись в диалоговом окне New Project в узле Visual C++ Projects:

```
Sub VSZLocation2()
    ' Отобразить .vsz-путь для проектов Visual C++:
    MsgBox(DTE.Solution.TemplatePath( _
        "{07CD18B1-3BA1-11d2-890A-0060083196C6}"))
End Sub
```

Для каждого найденного .vsz-файла отображается свой значок (рис. 9-1). Можно связать изображение с .vsz-файлом, поместив файл со значком (.ico) в каталог с тем же именем, что и .vsz-файл, но с расширением .ico. Для мастера надстроек .vsz-файл называется Visual Studio .NET Add-in.vsz. Когда пользователь выбирает соответствующий каталог, ищется файл Visual Studio .NET Add-in.ico, и применяется хранящееся в нем изображение; если же значок для .vsz-файла не найден, используется значок, определенный в Windows как значок по умолчанию для файлов.

Диалоговое окно Add New Item использует структуру каталогов так же, как окно New Project. Вы можете добавлять новые каталоги, и любые выбранные пользова-

телем .vsz-файлы будут задействованы для запуска мастера. Единственное различие между Add New Item и New Project в том. что соответствующие им каталоги, содержащие .vsz-файлы (каталоги шаблонов), размещаются в разных местах. Вот структура каталогов после того, как она была изменена для диалогового окна Add New Item и в каталоге был размещен шаблон текстового файла (рис. 9-2):

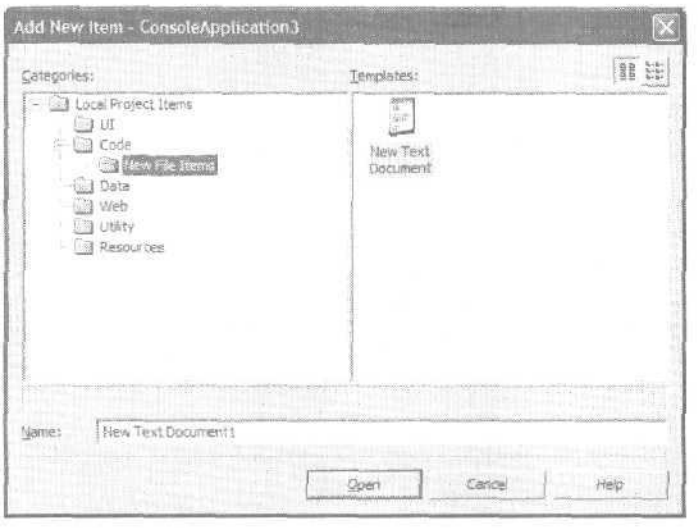



**Рис. 9-2.** *Пользовательский каталог в диалоговом окне Add New Item*

Определение места размещения .vsz-файлов для данного диалогового окна очень похоже на определение пути для диалогового окна New Project, но используется другой метод. Вместо метода *Solution.TemplatePath*, GUID типа проекта передается методу *Solution.ProjectItemsTemplatePath*. Определить путь к .vsz-файлам, соответствующим проектам C# для диалогового окна Add New Item, позволяет макрос:

```
Sub ProjectItemVSZLocation()
    ' Отобразить .vsz-путь проектов C#:
    MsgBox(DTE.Solution.ProjectItemsTemplatePath( _
        VSLangProj.PrjKind.prjKindCSharpProject))
End Sub
```

# Создание шаблона мастера

Мастера создают новый проект или добавляют файл с кодом к существующему проекту. Но откуда берется код для проектов или элементов проекта? Ответ: из файлов шаблонов. Шаблоны — это файлы с исходным кодом, которые мастер добавляет к решению или к существующему проекту, Эти файлы размещаются на диске, и. когда мастер хочет добавить проект или файл, шаблон проекта и файлы, на которые проект ссылается, или файл (для мастера добавления нового элемента) копируются в каталог, указанный пользователем, и добавляются к решению или проекту.

Обычно шаблоны создаются при помощи одного из мастеров для генерации проекта или элемента проекта, а затем файлы нового проекта изменяются в со-

ответствии с конкретными требованиями того проекта или элемента проекта, который нужно создать. Так же генерируется и код, создаваемый и добавляемый к решению, когда вы запускаете мастер надстроек. Сначала мы с помощью мастера библиотеки классов Visual Basic или C# генерируем базовый проект, а затем изменяем его, чтобы реализовать надстройку. Мастер надстроек находит этот проект и добавляет его к решению, а потом файлы созданного проекта изменяются согласно параметрам, заданным пользователем при запуске мастера,

## Использование файлов шаблонов

Созданные файлы шаблонов нужно добавить к решению или проекту, В Visual Studio .NET это можно сделать несколькими способами. Б главе 8 мы исследовали модель проекта, но оставили вне обсуждения два метода объекта *Solution: AddFromFile* и *AddFromTemplate*. Оба служат для добавления шаблонов к решению. *AddFromFile* добавляет в файл решения ссылку на проект, оставляя файл проекта в том же месте на диске, где он и находился. Вызов этого метода приводит к тому же результату, что и щелчок правой кнопкой в окне Solution Explorer узла решения, выбор Add | Add Existing Project и перемещение к файлу проекта. Мастера, однако, обычно добавляют к решению лишь копию шаблона проекта, иначе после того, как пользователь, для которого сгенерирован проект, внесет изменения в шаблон проекта, следующие запуски того же мастера будут добавлять ссылки на этот измененный проект. Поэтому мастера должны обычно вызывать метод *AddFromTemplate* копирующий шаблон проекта и связанные с ним файлы в каталог назначения и затем добавляющий к решению ссылку на эту копию. Вот сигнатура метода *AddFromTemplate*:

```
public EnvDTE.Project
AddFromTemplate(string FileName, string Destination,
    string ProjectName, bool Exclusive = false)
```

Метод имеет следующие аргументы.

- ***FileName*** Полный путь к шаблону проекта.
- ***Destination*** Местоположение на диске, куда копируется проект и файлы, на которые он ссылается. Мастер должен создать этот путь назначения до того, как будет вызван метод *AddFromTemplate*.
- ***ProjectName*** Имя, присваиваемое файлу проекта, и имя в Solution Explorer, где этот файл был скопирован. Не добавляйте к этому аргументу расширение типа проекта.
- ***Exclusive*** Если этот параметр установлен в *true,* перед добавлением шаблона проекта закрывается текущее решение и создается новое. *Если false*, решение не закрывается, и вновь создаваемый проект добавляется к открытому в это время решению.

**Примечание** Если параметр *Exclusive* равен *true*, при вызове *AddFromFile* или *AddFromTemplate* существующий проект закрывается без предоставления пользователю возможности сохранения измененных им файлов. Чтобы позволить ему это сделать, перед обращением к этим методам нужно вызвать свойство *ItemOperations.PromptToSave*.

Методы *AddFromTemplate* и *AddFromFile* добавляют шаблон проекта независимо от его местоположения на диске, т. е, нет нужды размещать файлы в каком-то определенном месте — достаточно поместить их там, где удобно. Обычно файлы шаблонов хранят в каталоге Templates, который находится в том же каталоге, что и COM-объект, реализующий мастер. Если мастер создан на языке, поддерживаемом .NET Framework, определить путь для шаблонов позволяет отражение, например:

```
string templatePath =
    System.Reflection.Assembly.GetExecutingAssembly().Location;
templatePath = System.IO.Path.GetDirectoryName(templatePath) +
    "\\Templates\\";
```

Метод *AddFromTemplate* добавляет к решению шаблон проекта, но набор *ProjectItems* содержит ряд методов для добавления файлов к существующему проекту: *AddFromDirectory*, *AddFromFileCopy*, *AddFromFile* и *AddFromTemplate*. *AddFromDirectory* получает в качестве параметра путь к каталогу на диске; этот каталог обследуется рекурсивно, и к проекту добавляются все содержащиеся в нем файлы и подкаталоги. Методы *AddFromFileCopy* и *AddFromFile* выполняют одну и ту же базовую операцию, добавляя к проекту ссылку на заданный файл. Однако *AddFromFileCopy* перед добавлением ссылки копирует файл в структуру каталогов этого проекта. Метод *AddFromFileCopy* отличается от метода *AddFromTemplate* из набора *ProjectItems* (не путать с методом *AddFromTemplate* объекта *Solution*) тем, что *AddFromTemplate* копирует файл для проекта в каталог диска, а после добавления файла этот проект может внести в этот файл некоторые изменения.

Вот сигнатуры и параметры этих методов:

```
public EnvDTE.ProjectItem AddFromDirectory(string Directory)
public EnvDTE.ProjectItem AddFromFileCopy(string FilePath)
public EnvDTE.ProjectItem AddFromFile(string FileName)
public EnvDTE.ProjectItem AddFromTemplate(string FileName, string Name)
```

- ***Directory*** Исходный каталог на диске.
- ***FilePath / FileName*** Местоположение файла, который нужно скопировать или на который надо добавить ссылку.
- ***Name*** Итоговое имя файла. Это имя должно иметь расширение типа файла.

Каждый из этих методов возвращает *ProjectItem*, объект, который можно задействовать для выполнения операций над добавляемым файлом (скажем, открытие файла или доступ к его содержимому).

## Имена файлов решения и мастер нового проекта

Когда запускается мастер нового проекта, имя файла решения может быть задано, а может быть и не задано в массиве *ContextParams* в зависимости от того, установил ли пользователь флажок Create Directory For Solution, который становится виден после того, как пользователь щелкнет More в диалоговом окне New Project. Если этот флажок установлен, становится доступным поле New Solution Name, позволяя пользователю задать новое имя каталога для решения. Если пользователь не установил флажок, при создании проекта посредством *Solution.AddFromTemplate*

нужно применить имя, заданное для проекта в массиве *ContextParams* как имя проекта, имя файла решения (если аргумент *взаимоисключение* в массиве *ContextParams* установлен в *true* и в этот момент решение не открыто) и имя каталога на диске, содержащего эти файлы. В этом каталоге должны также размещаться файлы решения и проекта. Если пользователь установил флажок, имеет силу аргумент *Имя решения* из списка контекстных параметров, и нужно применить его значение для именования корневого каталога решения и файла решения.

Создать файл решения и присвоить ему имя можно методом *Solution.Create* (см. главу 8), передавая ему в качестве аргументов путь к месту хранения файла решения и имя решения. В каталоге для файла решения нужно создать новый каталог для размещения файла проекта, и вы должны назвать и этот каталог, и проект именем проекта, передаваемым в массив *ContextParams*.

## Замены

Когда вы используете шаблон для создания нового проекта или нового файла. генерируемый код скорее всего не удовлетворяет всем требованиям мастера, который вы хотите создать. Так, если запущен мастер библиотеки классов C# (C# Class Library Wizard), генерируемый класс получает имя *Class1,* Пользователь может вручную переименовать этот класс, но лучше будет динамически присвоить классу имя, отражающее вид классов, генерируемых мастером (скажем, имя *Wizard,* если данный класс реализует мастер). Это можно сделать, заменив определенные текстовые выражения в файлах шаблона после того, как они были добавлены к решению или проекту. Чтобы сделать замену, вы используете объектную модель редактора для поиска этих выражений и затем изменяете их текст. Таким выражением может быть любой текст, помещенный в файл, но обычно они имеют определенный формат, отличающийся от остального текста в файле. Стандартное выражение, применяемое для резервирования места для имени класса, выглядит так *%CLASSNAME%*а шаблон класса с включенными в него выражениями — так:

```
public class %CLASSNAME%
{
    public %CLASSNAME%()
    {
        //
        // TODO: Add constructor logic here
        //
    }
}
```

Следующий макрос — *MakeReplacements*— заменяет выражения в файле. Некоторые концепции, использованные в макросе, например объекты *EnvDTE.TextPoint,* объясняются в главе 11.

```
Sub MakeReplacements(ByVal projectItem As EnvDTE.ProjectItem, _
                     ByVal token As String, _
                     ByVal replaceWith As String)
    Dim window As EnvDTE.Window
    Dim textDocument As EnvDTE.TextDocument
    Dim textRanges As EnvDTE.TextRanges
```

```
    Dim findOptions As Integer
    findOptions = EnvDTE.vsFindOptions.vsFindOptionsFromStart + _
        EnvDTE.vsFindOptions.vsFindOptionsMatchCase + _
        EnvDTE.vsFindOptions.vsFindOptionsMatchWholeWord

    ' Открыть указанный элемент проекта.
    ' Файл открывается, но его не видно:
    window = projectItem.Open(EnvDTE.Constants.vsViewKindTextView)

    ' Найти объект TextDocument для данного элемента проекта:
    textDocument = window.Document.Object("TextDocument")

    ' Заменить текст выражения текстом из replaceWith:
    textDocument.ReplacePattern(token, replaceWith, _
                                findOptions, textRanges)
End Sub
```

Если у вас открыт файл с определением класса и он сделан активным документом (с помощью объекта *ProjectItem*, возвращаемого методами *Add* из набора *ProjectItems*), для замены выражения *%CLASSNAME%* на имя класса *MyClass* можно вызвать такой макрос:

```
Sub MakeReplacements()
    MakeReplacements(DTE.ActiveWindow.ProjectItem, _
        "%CLASSNAME%", "MyClass")
End Sub
```

Мастер надстроек позволяет также удалить текст, находящийся между двумя выражениями. Так, можно удалить код, создающий кнопку на панели команд, когда на его странице Choose Add-in Options был сброшен флажок Yes, Create A Tools' Menu Item (создать пункт меню Tools). Следующий макрос удаляет текст между двумя выражениями:

```
Sub DeleteBetweenTokens(ByVal projectItem As EnvDTE.ProjectItem, _
                        ByVal token1 As String, _
                        ByVal token2 As String)
    Dim window As EnvDTE.Window
    Dim textDocument As EnvDTE.TextDocument
    Dim tokenEndPoint As EditPoint
    Dim tokenStartPoint As EditPoint
    Dim findOptions As Integer
    findOptions = EnvDTE.vsFindOptions.vsFindOptionsMatchCase + _
                  EnvDTE.vsFindOptions.vsFindOptionsMatchWholeWord

    ' Открыть указанный элемент проекта.
    ' Файл открывается, но его не видно:
    window = projectItem.Open(EnvDTE.Constants.vsViewKindTextView)

    ' Найти объект TextDocument для данного элемента проекта:
    textDocument = window.Document.Object("TextDocument")

    ' Создать точки редактирования для поиска:
```

```
    tokenEndPoint = textDocument.StartPoint.CreateEditPoint()
    tokenStartPoint = textDocument.StartPoint.CreateEditPoint()

    ' Цикл, пока не будут найдены все начальные/конечные точки:
    While (tokenStartPoint.FindPattern(token1, findOptions))
        If (tokenEndPoint.FindPattern(token2, findOptions, tokenEndPoint)) _
            Then

            ' Распространить выделение между
            '  начальным/конечным выражениями:
            textDocument.Selection.MoveToPoint(tokenStartPoint, False)
            textDocument.Selection.MoveToPoint (tokenEndPoint,True)

            ' Удалить выделенное:
            textDocument.Selection.Delete()
        Else
            Exit While
        End If
    End While
End Sub
```

Если код шаблона был изменен, чтобы выглядеть примерно так:

```
public class %CLASSNAME%
(
    public %CLASSNAME%()
    !
        //
        // TODO: Add constructor logic here
        //
    }

%BEGINOPTIONALCODE%
    void SomeOptionalCode()
    {
    }
%ENDOPTIONALCODE%
}
```

то после запуска данного макроса:

```
Sub MakeReplacements2()
    MakeReplacements(DTE.ActiveWindow.ProjectItem, _
        "%CLASSNAME%", "MyClass")
    DeleteBetweenTokens(DTE.ActiveWindow.ProjectItem, _
                        "%BEGINOPTIONALCODE%", "%ENDOPTIONALCODE%")
End Sub
```

результатом будет следующий код:

```
public class MyClass
{
    public MyClass()
```

```
    {
        //
        // TODO: Add constructor logic here
        //
    }
}
```

## Необрабатываемые шаблоны для мастеров добавления нового элемента

Мастер добавления нового элемента обычно используется для добавления файла и последующего его изменения с помощью замен. Однако иногда файл шаблона не требует модификации после его присоединения к проекту. Пример тому - присоединение текстового файла. Когда пользователь выбирает присоединение к проекту текстового файла, добавляется пустой файл. Создание объекта мастера только для того, чтобы присоединить пустой текстовый файл, — неоправданная трата дискового пространства (для хранения DLL мастера) и времени. Чтобы обойти это, Visual Studio .NET позволяет применять *необрабатываемые шаблоны*. При отображении диалогового окна Add New Item, Visual Studio .NET не только ищет и отображает .vsz-файлы на правой панели этого диалогового окна, она также показывает любые файлы, находящиеся в том же каталоге, где размещаются ,vsz-файлы. Если пользователь выберет один из этих файлов необрабатываемых шаблонов в диалоговом окне Add New Item, для этого файла выполняется аналог метода *AddFromFileCopy*: файл копируется в структуру каталогов того проекта, к которому добавляется данный файл, а затем включается в проект. Чтобы создать необрабатываемый шаблон, нужно просто создать файл и поместить его в место, указываемое путем, возвращаемым при вызове метода *ProjectItemsTemplatePath*, а затем указать требуемый тип проекта.

# Нестандартные мастера

В Visual Studio .NET встроена поддержка создания только двух типов мастеров: нового проекта и добавления нового элемента. Однако иногда требуется создать мастер иного типа. Visual Studio .NET поддерживает расширяемую архитектуру мастера, что позволяет создать собственный, или нестандартный, мастер, например, позволяющий вставлять в существующий файл кода стандартные конструкции кода, такие как реализация по умолчанию классов, методов или свойств,

Чтобы создать нестандартный мастер, нужно создать COM-объект, реализующий интерфейс *IDTWizard*, и .vsz-файл для этого мастера — точно так же, как это делается для мастеров нового проекта или добавления нового элемента. Но при этом вы сами выбираете список контекстных параметров, которые будет принимать этот мастер, а кроме того, нестандартный мастер активизируется посредством вашего кода, а не через диалоговое окно. Список контекстных аргументов, передаваемых мастеру при запуске, поставляет сведения о том, как должен работать данный мастер. Если мастер должен запускаться как мастера нового проекта или создания нового элемента, Visual Studio .NET сама вычисляет нужный массив контекстных параметров и передает этот массив методу *IDTWizard.Execute*. Нестан-

дартный мастер запускается программно (или из надстройки, или из другого мастера), и вызывающее приложение заполняет массив контекстных параметров.

**Примечание** Не существует ограничений на контекстные параметры, которые можно передать нестандартному мастеру, но мы рекомендуем делать первым аргументом GUID, известный вызывающей стороне и мастеру. Мастер затем должен проверить, что переданный ему GUID совпадает с ожидаемым. Это помогает предохранить мастер от некорректного использования контекстных параметров и возникновения исключительной ситуации или краха.

### Откуда взялись нестандартные мастера?

Программисты, работавшие над автоматизацией Visual Studio .NET, не предусмотрели возможность создания нестандартных мастеров для обеспечения дополнительного способа расширить программу вашими наработками. Нестандартные мастера были рождены необходимостью. Разработчики Visual C++ нуждались в способе добавления к классам новых функций и переменных из окна просмотра классов (Class View), и они хотели делать это способом, напоминающим работу мастеров. Для этого они добавили к списку типов мастеров нестандартные мастера. Они создали мастера для добавления функций и переменных, и путем вызова метода *DTE.LaunchWizard* с необходимыми параметрами для программного запуска этих мастеров они предоставили пользователям возможность изменять класс знакомым им способом. Таким образом, щелкая правой кнопкой класс C++ в окне Class View и выбирая Add \ Add Function или Add [ Add Variable, вы на самом деле запускаете нестандартный мастер.

## Программный запуск нестандартного мастера

Ваша программа может вручную загрузить COM-объект мастера, найти его интерфейс *IDTWizard* и передать нужные значения методу *Execute,* но проще позволить выполнить большую часть этой работы Visual Studio .NET. Для программного запуска мастера можно применить модель автоматизации, вызвав метод *DTE.LaunchWizard*, который принимает в качестве аргумента путь к .vsz-файлу и массив контекстных параметров. Поскольку этому методу передается путь к .vsz-файлу, этот файл может располагаться в любом месте диска — его не обязательно располагать в каком-то конкретном месте, как для мастеров других типов. После вызова метод *LaunchWizard* порождает COM-объект мастера, определяемый в .vsz-файле, и создает массив нестандартных параметров, содержащихся в .vsz-файле. Метод *LaunchWizard* затем передает методу *Execute* мастера все нужные параметры.

В примере CustomWizard можно увидеть применение метода *LaunchWizard*. Мастер из этого примера сначала проверяет, что первый аргумент массива ContextParams совпадает с GUID ожидаемого типа мастера, а затем просто перебирает список переданных контекстных и нестандартных параметров, отображая для каждого найденного в этих массивах элемента окно сообщения. Вот макрос, запускающий этот мастер:

```
Sub CallCustomWizard()
    Dim contextParams(1) As Object
    contextParams(0) = "{9A4B2CFF-7A69-4671-BFA5-AE0D0C44AEFB}"
    contextParams(1) = "Hello world!"
    DTE.LaunchWizard("C:\samples\CustomWizard.vsz", contextParams)
End Sub
```

Если пример мастера размещен в каталоге C:\samples, этот макрос помещает тип мастера и строку *«Hello world!»* в массив и затем вызывает *LaunchWizard*. Для мастера определен GUID {9A4B2CFF-7A69-4671-BFA5-AE0D0C44AEFB}, и он ожидает найти эту строку в первом элементе массива *ContextParams*, если мастер не найдет там эту строку, он откажется работать и сразу вернет управление и ошибку.

## Связывание нестандартных мастеров

Можно использовать метод *LaunchWizard* для связывания мастеров, что означает вызов одного мастера из другого для упрощения создания проекта. Допустим, вам требуется решение, содержащее Web-сервис XML, а также приложение Windows Forms для получения данных от Web-сервиса XML и отображения их пользователю. Создать проект Windows Forms легко: вы запускаете мастер Windows Form для создания шаблона проекта Windows Form, а затем создаете объект мастера, который добавит этот шаблон проекта к решению. Однако создать Web-сервис XML для данного решения не так просто, как создать шаблон и добавить его к решению. Проект Web-сервиса XML лежит на Web-сервере. Ни одного файла из проекта Web-сервиса XML, кроме файла проекта, нет на локальном компьютере — файл проекта просто указывает на сервер и то место на сервере, где хранятся файлы. Мастер может общаться с Web-сервером по протоколу, такому как Front Page server extensions, выдавая ему нужные команды для сохранения файлов Web-сервиса XML, но проще вызвать мастер Web-сервисов (Web Service Wizard) для создания и сохранения этих файлов.

Пример проекта ChainWizard, включенный в файлы примеров этой книги, демонстрирует, как это делается. Вот часть метода *Execute* данного мастера:

```
const string serviceName = "ChainWizardWebService";
object []contextParamsChain = new object[7];
EnvDTE.wizardResult wizardResultChain;
EnvDTE.DTE dte = (EnvDTE.DTE)Application;

// Добавить наш Web-сервис путем заполнения контекстных параметров
// и подключиться к мастеру Web-сервисов.
contextParamsChain[0] = EnvDTE.Constants.vsWizardNewProject;
contextParamsChain[1] = serviceName;
contextParamsChain[2] = "http://localhost/" + serviceName;
contextParamsChain[3] = System.IO.Path.GetDirectoryName(dte.FullName);
contextParamsChain[4] = (bool)ContextParams[4];
contextParamsChain[5] = "";
contextParamsChain[6] = false;
string webSvcTemplatePath = dte.Solution.get_TemplatePath(
    VSLangProj.PrjKind.prjKindCSharpProject);
webSvcTemplatePath += "CSharpWebService.vsz";
```

```
wizardResultChain = dte.LaunchWizard(webSvcTemplatePath,
    ref contextParamsChain);
```

Этот код создает и заполняет контекстные параметры, передаваемые объекту мастера, который запускается методом *LaunchWizard*. С помощью свойства *TemplatePath* вы определяете место для размещения .vsz-файла для Web-сервиса XML и передаете тип проекта как проект на языке C#. Заметьте; этот код не пытается создавать при каждом запуске уникальное имя Web-сервиса XML. Если нужно запускать данный мастер несколько раз, потребуется или удалять Web-сервис XML, созданный ранее на сервере, или изменять значение переменной *serviceName* на уникальное значение.

Результат запуска примера ChainWizard — решение, содержащее проект Web-сервиса XML и проект Windows Forms, использующий функциональность Web-сервиса XML.

### Практикум: расшифровка параметров мастера

Как определить, какие аргументы передаются мастеру при связывании? Это можно сделать, обманув Visual Studio .NET: пусть она вызовет одноразовый мастер, а мы с его помощью просмотрим в отладчике, какие нестандартные и контекстные параметры передаются мастеру. Именно так я обнаружил параметры, которые нужно передать мастеру Web-сервисов ASP.NET (ASP.NET Web Service Wizard) в примере ChainWizard.

Сначала запустите мастер библиотек классов C# (или Visual Basic .NET) из диалогового окна Mew Project. Затем измените проект, чтобы реализовать интерфейс *IDTWizard* и зарегистрируйте эту библиотеку как COM-объект, присвоив коду GUID и ProgID и установив флаг необходимости регистрации как COM-объекта в диалоговом окне страниц свойств проекта так же, как вы это делали для других мастеров, Затем измените .vsz-файл, чтобы он указывал на этот одноразовый мастер; поскольку наш пример связывается с мастером Web-сервиса C#, мы должны изменить .vsz-файл для этого мастера. Найдите в каталоге Visual Studio .NET 2003\VC#\CsharpProjects файл по имени CSharpWebService.vsz и откройте этот файл в Блокноте, Поскольку мы будем изменять этот файл, стоит на всякий случай создать его резервную копию.

В .vsz-файле, открытом в Блокноте, измените ProgID с VsWizard.VsWizardEngine.7.1 на ProgID одноразового мастера, а затем сохраните этот файл. Теперь вернемся в Visual Studio .NET, где у нас открыт проект мастера. Поместите точку прерывания в метод *Execute* мастера и нажмите F5 (перед этим нужно установить Visual Studio .NET или devenv.exe в качестве объекта отладки). В появившемся новом экземпляре Visual Studio .NET откройте диалоговое окно New Project и запустите мастер C# ASP .NET Web Service Wizard. Вместо мастера ASP.NET Web Service Wizard должен запуститься ваш мастер, и, когда произойдет останов в точке прерывания в методе *Execute*, вы можете посмотреть, какие данные передаются мастеру через нестандартные и контекстные параметры.

Не забудьте восстановить корректный .vsz-файл, иначе ваш мастер будет запускаться и тогда, когда вам на самом деле потребуется создать Web-сервис XML.

## Вспомогательная библиотека мастера

Как вы убедились, создать мастер несложно. Достаточно создать COM-объект, реализующий интерфейс *IDTWizard* и поместить .vsz-файл на диск, и готов мастер, который пользователь может запускать из диалогового окна New Project или Add New Item. Однако создать и отобразить пользовательский интерфейс для мастера может оказаться непросто, из-за чего мы до сих пор избегали этой темы. Чтобы создать пользовательский интерфейс мастера, надо создать приложение Windows Form и страницы для мастера. Приложение Windows Form будет отображать страницы мастера, и кнопки Next, Back. Finish и Cancel должны правильно перемещать пользователя между страницами,

Большая часть кода для отображения пользовательского интерфейса мастера одинакова для всех мастеров. Для облегчения создания мастеров с пользовательским интерфейсом мы включили в файлы примеров книги исходный код библиотеки, которая управляет этим пользовательским интерфейсом. Библиотека WizardLibrary реализует интерфейс *IDTWizard*, а также разделяет массив *ContextParams* на переменные, делая процесс создания мастера менее подверженным ошибкам. Чтобы задействовать эту библиотеку, достаточно создать пользовательский элемент управления на каждой странице мастера, написать немного кода, позволяющего библиотеке узнать, какие страницы доступны, и затем реализовать функциональность, специфичную для вашего мастера, например, создание и добавление защищенного кода.

Создадим мастер, генерирующий код для мастеров, — этакий «мастер мастеров». Сначала создадим проект библиотеки классов C# и назовем его WizardBuilder, затем мы сможем добавить ссылку на сборку WizardLibrary (но перед этим, чтобы на код библиотеки можно было сослаться, нужно загрузить и скомпоновать этот проект из исходных файлов примера), а затем в нашем проекте породим класс от *InsideVSNet.WizardLibrary.WizardLibrary* (базового класса, реализующего функциональность библиотеки). После внесения изменений для регистрации объекта как COM-объекта, наш код будет выглядеть примерно так:

```
[GuidAttribute("1EF6B85C-FD5C-4fb4-BA4D-5ED221195DBF"),
ProgIdAttribute("WizardBuilder.Wizard")]
public class Wizard : InsideVSNet.WizardLibrary.WizardLibrary
{
    public Wizard()
    {
    }
}
```

Опираясь на этот начальный код, мы можем определить страницы, которые наш мастер будет отображать для пользователя. Первая страница содержит два параметра, которые может изменять пользователь: параметр создания мастера

добавления нового элемента или мастера нового проекта и параметр, определяющий, где пользователь может запускать мастер. Эти параметры отвечают за создание .vsz-файла для генерируемого с помощью WizardBuilder мастера и размещение его в корректном месте. Вторая страница позволяет задать число страниц кода итогового мастера. Поскольку для реализации каждой страницы нашего мастера библиотека применяет пользовательские элементы управления .NET, мы можем в диалоговом окне Add New Item добавить к нашему проекту два пользовательских элемента управления Page1 и Page2, а затем добавить к этим формам требуемые оконные элементы управления.

Для этих двух пользовательских элементов управления, или страниц, добавленных к нашему проекту, нам нужен способ взаимодействия библиотеки мастеров с каждой из страниц, чтобы позволить ей сгенерировать выходной проект и изменить генерируемые файлы исходного кода. Этот способ — реализация для каждой из страниц интерфейса *InsideVSNet.WizardLibrary.IWizardPage*определенного в данной библиотеке. Вот его сигнатура:

```
public interface IWizardPage
{
    void PerformWork1(WizardLibrary WizardLibrary);
    void PerformWork2(WizardLibrary WizardLibrary);
    string HeadingLabel
    {
        get;
    }
    string DescriptionLabel
    {
        get;
    }
    System.Drawing.Image Icon
    {
        get;
    }
    void ShowHelp();
    void Initialize (WizardLibrary wizardLibrary);
}
```

А вот методы и свойства данного интерфейса.

- ***PerformWork1*** Этот метод вызывается, когда пользователь щелкает кнопку Finish, и данная страница мастера приступает к генерации кода. Каждая страница может генерировать свою часть кода в пределах проекта, и эта работа выполняется в данном методе.
- ***PerformWork2*** Каждая страница мастера имеет метод *PerformWork1,* вызываемый при отображении страницы. Однако иногда выходные результаты одной страницы зависят от выходных результатов другой. Информация может генерироваться в методе *PerformWork1,* сохраняться и извлекаться для последующей обработки во время работы метода *PerformWork2.*После того как были вызваны методы *PerformWork1* каждой из страниц, вызываются методы *PerformWork2* в том порядке, в каком отображались страницы.

- ***HeadingLabel*** Это свойство, доступное только для чтения, позволяет странице мастера вернуть информацию о заголовке, отображаемом в верхней части мастера. В пользовательском интерфейсе страницы мастера надстроек заголовочная информация, или баннер, отображает для пользователя три типа информации: полужирный заголовок, текст описания и значок страницы, Данное свойство возвращает заголовок страницы.
- ***DescriptionLabel*** Это свойство возвращает строку описания, которая будет отображаться в баннере мастера.
- ***Icon*** Это свойство возвращает картинку в формате *System.Drawing.Image*, которая будет отображаться в баннере мастера.
- ***ShowHelp*** Этот метод вызывается при нажатии кнопки Help в левом нижнем углу мастера, когда пользователь запрашивает справку по данной странице,
- ***Initialize*** Этот метод вызывается, после того как данная страница мастера была добавлена к списку страниц, обрабатываемых библиотекой.

Данный интерфейс, реализуемый каждым пользовательским элементом управления, позволяет рассказать библиотеке мастеров о страницах. Сама библиотека реализует для нас интерфейс *IDTWizard* и его метод *Execute*, но при первом запуске мастер вызывает метод, определенный в классе *WizardLibrary* (от которого мы породили наш класс мастера), этот метод объявлен как абстрактный и тоже называется *Execute*. Поэтому в классе, наследующем от класса *WizardLibrary,* должен находиться метод, переопределяющий данный абстрактный метод:

```
public override void Execute(EnvDTE.DTE applicationObject);
```

Эта версия метода *Execute* — то место, где мы даем понять библиотеке мастеров, какие страницы ей доступны. Страницы можно добавлять, последовательно вызывая метод *WizardLibrary.AddPage*, передавая ему экземпляр одного из наших пользовательских элементов управления, реализующих интерфейс *IWizardPage*, причем делать это нужно в том порядке, в каком эти страницы должны появляться в пользовательском интерфейсе мастера. В методе *Execute* мы можем создать и добавить два пользовательских элемента управления, созданных ранее (*Page1* и *Page2*):

```
public override void Execute(EnvDTE.DTE applicationObject)
{
    Title = "Wizard Builder";
    AddPage(new Page1());
    AddPage(new Page2());
}
```

Этот код не только сообщает библиотеке о страницах нашего мастера, но также устанавливает заголовок диалогового окна в *Wizard Builder*. В классе *WizardLibrary* определено свойство *Title*, устанавливающее текст для формы мастера. По выполнении рассмотренного метода *Execute* библиотека имеет необходимую информацию для работы. После этого библиотека отображает диалоговое окно со страницами — пользовательскими элементами управления и осуществляет необходимую навигацию между страницами мастера. Библиотека также управляет параметром *wizardResult* метода *IDTWizard.Execute*, возвращающим *wizardResult*-

*Success* при нажатии кнопки Finish, *wizardResultCancel*— кнопки Cancel, *wizardResultFailure* в случае возникновения исключительной ситуации или *wizardResultBackOut*, если отображается первая страница мастера и пользователь щелкнул кнопку Back, По окончании работы над кодом, при запуске мастера, отображается диалоговое окно с первой страницей мастера (рис. 9-3).

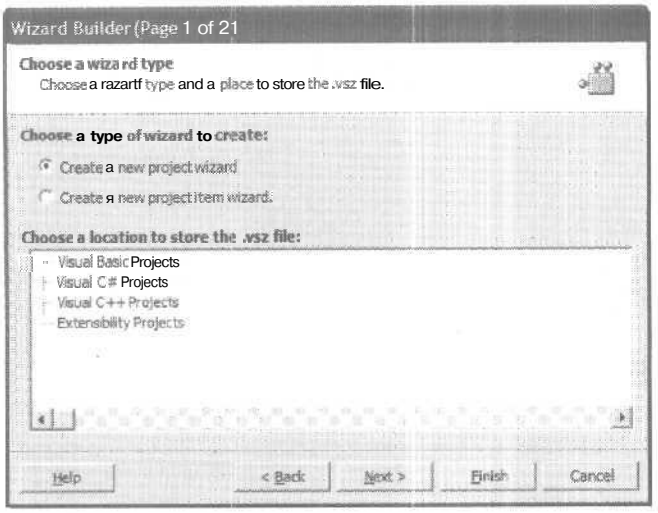



Рис. 9-3. *Первая страница из примера WizardBuilder*

## Переменные мастера

При вызове метода *IDTWizardExecute* информация передается мастеру через аргументы *ContextParams* и *CustomParams*. Но метод *Execute,* реализуемый вашим мастером, не принимает этих аргументов, так как библиотека мастеров извлекает нужные переменные из массива *ContextParams* и сохраняет их как переменные-члены. Значения из *CustomParams* извлекаются не так, как из *ContextParams,* поскольку у вашего мастера свой набор этих переменных и библиотека не имеет предварительных сведений о его составе. В табл. 9-3 и 9-4 перечислены имена переменных, которые вы можете задействовать, и то, как они соотносятся со значениями из табл. 9-1 и 9-2, а также контекст, в котором их можно применять.

**Табл. 9-3. Переменные для мастера нового проекта и соответствующие им значения массива *ContextAttribute***

| Переменная | Соответствующее значение |
|---|---|
| *WizardType* | Тип мастера |
| *NewProjectName* | Имя проекта |
| *NewProjectLocation* | Локальный каталог |
| *VisualStudioInstallDirectory* | Установочный каталог |
| *exclusiveProject* | Взаимоисключение |
| *NewSolutionName* | Имя решения |
| *RunSilent* | Скрытый |

**Табл. 9-4. Переменные для мастера добавления элемента и соответствующие им значения массива *ContextAttribute***

| Переменная | Соответствующее значение |
|---|---|
| *WizardType* | Тип мастера |
| *ProjectName* | Имя проекта |
| *ProjectItems* | **Элементы** проекта |
| *NewItemLocation* | Новое место элемента |
| *NewItemName* | Новое имя элемента |
| *ProductInstallDirectory* | Установочный каталог продукта |
| *RunSilent* | Скрытый |

Значения переменных, перечисленных в табл. 9-3 и 9-4, создаются путем извлечения значений из массива контекстных параметров. Две другие переменные, принадлежащие библиотеке мастеров, доступны и для мастера нового проекта, и для мастера добавления нового элемента.

- ***application*** Объект DTE экземпляра Visual Studio .NET, в котором запущен мастер.
- ***CustomArguments*** Список нестандартных параметров, скопированный дословно из аргументов *CustomParam*, переданных методу *IDTWizard.Exec*.

Библиотека мастеров предоставляет и еще одну переменную, которую может задействовать ваш мастер. До этого мы отмечали, что метод *IWizardPage.PerformWork1* может сохранять информацию для последующего использования ее в методе *IWizardPage.PerformWork2*. Однако одна страница мастера не имеет доступа к данным другой страницы, так как они являются разными объектами. Для хранения информации библиотека мастеров имеет член типа *System.Collections.SpecializedListDictionary*. Этот член позволяет странице мастера сохранить пару «имя – значение» для использования другой страницей. Пример мастера, который мы создаем, применяет значение данного члена *customData* для хранения такой информации, как объект *EnvDTE.Project*, который был создан вызовом метода *CreateProject* из метода *PerformWork1* первой страницы мастера.

## Вспомогательные методы мастера

Библиотека мастеров поддерживает четыре вспомогательных метода, которые мастер может задействовать при генерации итогового проекта или файла. Метод *CreateProject* позволяет создать проект, основанный на шаблоне проекта;

```
public EnvDTE.Project CreateProject(string templatePath)
```

Этот метод создает файл решения, если он необходим, и помещает файл проекта в подходящий каталог на диске, если пользователь указал создать отдельные каталоги для файлов проекта и файлов решения. Этот метод принимает единственный параметр — путь к файлу шаблона проекта. Такие значения, как имя проекта и решения, а также то, должен ли быть закрыт файл решения или новый проект нужно добавить к открытому в данный момент файлу решения, передавать этому методу не требуется, так как эти значения уже известны мастеру. Последние два метода — *DeleteBetweenTokens* и *MakeReplacements* — C#-версии одноименных

макросов, показанных ранее в этой главе. Их можно использовать для модификации создаваемых вами исходных файлов.

## Завершение примера WizardBuilder

Теперь мы можем завершить пример WizardBuilder. Мы уже создали класс, производный от класса *WizardLibrary*, создали метод *Execute*, добавили к проекту две страницы в виде пользовательских элементов управления и реализовали для каждой из страниц интерфейс *IWizardPage*. Осталось сгенерировать выходной код. Мы начнем с создания файлов шаблонов. Создадим библиотеку класса C# под названием WizardTemplate, зададим параметры проекта для регистрации в качестве COM-объекта и изменим код класса следующим образом:

```
namespace %NAMESPACE%
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
     [GuidAttribute("%GUID%"), ProgIdAttribute("%NAMESPACE%.Wizard")]
    public class Wizard : InsideVSNet.WizardLibrary.WizardLibrary
    {
        public Wizard()
        {
            I!
            // TODO: Add constructor logic here
            //
        }

        public override void Execute(EnvDTE.DTE application)
        {
            Title = "My Wizard";
%WIZARDPAGES%
        }
    }
}
```

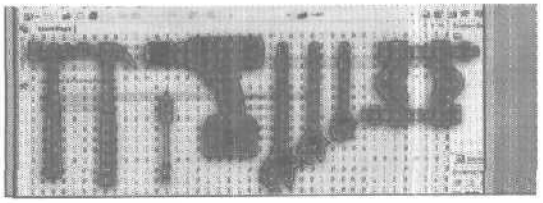
При запуске мастер заменит *%NAMESPACE%* на имя проекта, заданное к диалоговом окне New Project, а *%GUID%* — на новый GUID. Выражение *%WIZARDPAGES%* будет заменено на код, генерируемый для добавления экземпляров страниц к библиотеке. Следующий создаваемый шаблон — шаблон страниц мастера. Мы его сделаем, запустив мастер C# Windows Control Library Wizard, изменив сгенерированный пользовательский элемент управления для реализации *IWizardPage* и изменив пространство имен и вхождения имени класса на выражения *%NAMESPACE%* и *%PAGENAME%* соответственно. Мы скопируем полученный файл с пользовательским элементом управления в каталог шаблонов нашего мастера.

Заключительный шаг — наполнить содержимым методы *PerformWork1* и *PerformWork2* двух страниц нашего мастера. *PerformWork1* первой страницы занимается созданием проекта и выполняет замены в соответствующем файле. *PerformWork2* первой страницы занимается созданием .vsz-файла, помещая копию в указанный пользователем каталог и добавляет ссылку на сборку WizardLibrary.dll.

*PerformWork1* для второй страницы не используется. *PerformWork2* для второй страницы добавляет по одной копии шаблона пользовательского элемента управления для создания того количества страниц, что задал пользователь во время работы нашего мастера. Затем он выполняет подстановки для вновь добавленных страниц. И, наконец, метод *PerformWork2* создает в файле wizard.cs код, заменяя выражения *%WIZARDPAGES%*.

## Что дальше

В этой и предыдущей главах мы имели дело с проектами и их элементами. Далее мы рассмотрим программирование окон в Visual Studio .NET.



ГЛАВА

# 10

# Программирование пользовательского интерфейса

Множество окон в Microsoft Visual Studio .NET — Task List, Solution Explorer, дизайнер Windows Forms и др. — отображает данные. Этими окнами можно манипулировать не только мышью и с клавиатуры, но и посредством объектной модели, применяя макросы и надстройки. В этой главе мы рассмотрим объекты, позволяющие программировать пользовательский интерфейс Visual Studio .NET.

## Основные сведения об окнах

Пользовательский интерфейс разных окон в Visual Studio .NET различен, но все окна имеют базовый набор методов и свойств. Рассмотрим эту общую часть объектной модели.

### Набор *Windows*

Visual Studio .NET содержит несколько окон инструментов и документов, доступ к которым обеспечивает модель автоматизации. В объектной модели каждое из этих окон представлено объектом *Window* из набора *Windows,* доступного через свойство *DTE.Windows.*

Получить объект *Window* из набора *Windows* можно по-разному, например, использовать перечислитель для прохода по списку всех доступных окон:

```
Sub EnumWindows()
    Dim window As EnvDTE.Window
    For Each window In DTE.Windows
```

```
        MsgBox(window.Caption)
    Next
End Sub
```

Можно использовать метод числовой индексации:

```
Sub EnumWindows2()
    Dim window As EnvDTE.Window
    Dim i As Integer
    For i = 1 To DTE.Windows.Count
        MsgBox(DTE.Windows.Item(1).Caption)
    Next
End Sub
```

Однако эти способы неоптимальны, так как обычно требуется найти одно конкретное окно, а на перечисление всех окон тратятся немалые ресурсы процессора. Метод числовой индексации не всегда хорош, поскольку положение окна в наборе различно в разных экземплярах Visual Studio.NET. По сути у вас нет гарантии, что два вызова метода *Item* с применением числового индекса вернут один и тот же объект *EnvDTE.Window*, так как между вызовами метода могут быть созданы новые окна. Кроме того, метод числовой индексации может не найти все доступные окна. Так, создание инструментального окна может быть дорогой операцией. Чтобы увеличить производительность, Visual Studio .NET не создает инструментальное окно, пока к нему не будет явного обращения, а поскольку метод числовой индексации просматривает только уже созданные окна, отдельное окно может быть не найдено.

Простой пример может продемонстрировать, как перебор списка всех инструментальных окон замедляет работу, если созданы еще не все эти окна. По умолчанию окно Server Explorer является пристыкованным и скрытым слева от основного окна Visual Studio .NET. Если перевести указатель мыши на значок этого окна, оно появляется на экране. Если в данном экземпляре Visual Studio .NET это окно еще не показывалось, вы заметите задержку в пару секунд, которая связана с созданием окна. Если при работе макроса *EnumWindows2* требуется создание некоторых объектов *Window,* этот макрос будет работать очень медленно, так как много процессорного времени уйдет на создание окон.

Другой способ поиска окна — индексация набора *Windows* с использованием имени окна. Такой подход демонстрирует следующий макрос, в котором для поиска объекта *Window* для инструментального окна Task List используется его имя:

```
Sub FindTaskListWindow()
    Dim objWindow As EnvDTE.Window
    objWindow = DTE.Windows.Item("Task List")
End Sub
```

Но и это не лучший способ поиска конкретного объекта *Window*. При поиске окна строка, переданная методу *WindowsItem*, сравнивается с заголовком каждого окна, пока не будет найдено совпадение. Щелкнув правой кнопкой Task List и выбрав в контекстном меню Show Tasks | Comment, вы увидите, что заголовок окна станет «Task List - X Comment tasks shown (filtered)», где *X* — число. Поскольку строка *«Task List»*, переданная методу *Item*, не совпадает полностью с заголовком

окна Task List, код *WindowsItem(«TaskList»)* не найдет объект *Window*. Понятно, что метод индексации по заголовку иногда неприменим. Некоторые окна, такие как Properties или Object Browser, имеют неизменяемые имена (если пользователь не меняет язык), и для них метод поиска по заголовку работает, Но и здесь, как и в случае с числовой индексацией, окно может быть не найдено, если оно еще не создано.

Объект *Window* лучше всего искать по уникальному индексу, не зависящему ни от положения в наборе *Windows*, ни от заголовка окна, Каждому окну инструментов назначен постоянный GUID, который можно передавать методу *Item* для поиска нужного окна. Поскольку GUID трудно запомнить, для большинства окон инструментов, создаваемых Visual Studio .NET, определены константы, которые легко запомнить и распознать. Все они начинаются с префикса *vsWindowKind* и являются статическими (или общими, если вы используете Visual Basic .NET) членами класса *EnvDTE.Constants*. Вот макрос для поиска окна инструментов Task List:

```
Sub FindTaskListWindow2()
    Dim objWindow As EnvDTE.Window
    objWindow = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindTaskList)
End Sub
```

Поскольку GUID уникален для каждого окна инструментов и не меняется со временем, вас не должно заботить изменение заголовка или положения искомого окна в наборе *EnvDTE Windows,* Еще одно преимущество применения GUID в том, что Visual Studio .NET создает запрашиваемое инструментальное окно, если его еще нет.

Встречаются окна, для которых константа, соответствующая GUID, не определена. Примером служит окно Favorites. Для них можно использовать GUID в виде строки, указывая ее там же, где и предопределенную константу. Это демонстрирует следующий пример, в котором ищется объект *Window* для окна Favorites:

```
Sub FindTheFavoritesWindow()
    Dim window As EnvDTE.Window
    window = DTE.Windows.Item("{E8B06F43-6D01-11D2-AA7D-  00C04F990343}")
End Sub
```

GUID, передаваемый методу *Item*, можно получить из свойства *ObjectKind*. Вот макрос, в котором данный подход применяется для вывода GUID окна Favorites:

```
Sub  FindTheFavoritesWindow2()
    Dim window As EnvDTE.Window
    ' Прежде чем вызывать этот код,
    ' нужно отобразить окно Favorites!
    window = DTE.Windows.Item("Favorites")
    MsgBox(window.ObjectKind)
End Sub
```

Этот макрос выводит GUID окна Favorites в информационном окне. Затем вы можете определить константу, соответствующую этому GUID, и применять ее в любом коде, которому нужно это окно, Для макроса *FindTheFavoritesWindou* мы определили GUID именно так.

## Свойство *Object*

Многие окна Visual Studio .NET имеют объектную модель, которую вы можете применять для работы с данными, содержащимися в этих окнах. Эти объекты, связанные с конкретным окном, позволяет получить свойство *Object* объекта *Window*. Например, свойство *Object* объекта *Window* для окна Task List возвращает объект *TaskList*, который позволяет перечислять, добавлять и изменять свойства отдельных задач в окне Task List. Следующий макрос получает объект *TaskList:*

```
Sub GetTaskListObject()
    Dim window As EnvDTE.Window
    Dim taskList As EnvDTE.TaskList
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindTaskList)
    taskList = CType(window.Object, EnvDTE.TaskList)
End Sub
```

Есть множество программируемых объектов для различных окон, не только объект *TaskList*, показанный в этом макросе. В табл. 10-1 перечислены константы, соответствующие GUID, которые можно передавать методу *Item* как для поиска объекта *Window,* так и программируемого объекта, соответствующего этому окну.

**Табл. 10-1. Окна и их программируемые объекты**

| Окно | Константа GUID | Тип объекта |
|---|---|---|
| Command Window | *vsWindowKindCommandWindow* | *EnvD TE.CommandWindow* |
| Macro Explorer | *vsWindowKindMacroExplorer* | *EnvDTE.UIHierarchy* |
| Output window | *vsWindowKindOutput* | *EnvD TE.OutputWindow* |
| Server Explorer | *vsWindowKindServerExplorer* | *EnvD TE.UlHierarchy* |
| Solution Explorer | *vsWindowKindSolutionExplorer* | *EnvD TE.UlHierarchy* |
| Task List | *vsWindowKindTaskList* | *EnvDTE.TaskList* |
| Toolbox | *vsWindowKindToolbox* | *EnvDTE.ToolBox* |
| Окно Web-браузера | *vsWindowKindWebBrowser* | *SHDocVw.WebBrowser* |
| Текстовый редактор | <Нет> | *EnvDTE.TextWindow* |
| Дизайнер форм | < Нет > | *System.ComponentModel. Design.IDesignerHost* |
| Дизайнер HTML | < Нет > | *EnvDTE.HTML Window* |

Объектную модель имеют не только окна инструментов Visual Studio .NET, но и многие окна документов. Свойства *Window.Object* текстового редактора, дизайнера форм и дизайнера HTML возвращают объекты для программирования этих объектов-окон. Объект программирования дизайнера форм обсуждается в этой главе, а объекты программирования текстового редактора и дизайнера HTML — в главе 11.

## Основное окно

Каждому окну инструментов и документов Visual Studio .NET соответствует объект *Window.* Однако Visual Studio .NET сама является окном, и ей тоже соответствует объект *Window.* Чтобы его получить, нужно не индексировать набор *EnvDTE.Windows*, а взять свойство *MainWindow* объекта *DTE*:

```
Sub FindTheMainWindow()
    Dim mainWindow As EnvDTE.Window
    mainWindow = DTE.MainWindow
End Sub
```

У объекта *Window* основного окна Visual Studio .NET несколько методов и свойств работает иначе, чем для объектов *Window* окон документов и инструментов:

- методы *Document, Selection, Object, ProjectItem* и *Project* возвращают *null* при использовании C# и *Nothing* в Visual Basic .NET;
- *set*-версии свойств *Caption* и *Linkable* вызывают исключение;
- *IsFloating* и *AutoHides* генерируют исключения при вызове *get*- или *set*-версий этих свойств;
- вызов метода *Close* приводит к исключению.

Некоторые методы и свойства объекта *Window* основного окна не работают, но есть одно свойство, доступное только для основного окна. Когда в надстройке или макросе нужно вывести диалоговое окно, вы должны предоставить родительское окно, чтобы корректно управлять фокусом и «немодальностью» нового окна. Использовать основное окно Visual Studio .NET как родительское позволяет свойство *Window.HWnd*, которое возвращает описатель окна типа *HWND*, определенного в Windows SDK. Это свойство является скрытым; разрабатывая надстройку или макрос, вы не увидите его при автозавершении оператора. Поскольку .NET Framework не применяет значение *HWND* в качестве предка, этот описатель должен быть предварительно вложен в класс, реализующий интерфейс, который .NET может воспринимать в качестве предка. Этот интерфейс — *System.Windows.Forms.IWin32Window* — вы можете реализовать в классе своей надстройки или в отдельном классе в проекте макроса. Единственное свойство интерфейса *IWin32Window* — *Handle* — возвращает значение *System.IntPtr*, которое содержит описатель родительского окна, и в данном случае является значением, возвращаемым свойством *Window.HWnd*. Когда нужно показать форму методом *Form.ShowDialog*, можете передать этому методу в качестве аргумента класс, реализующий *IWin32Window*.

Чтобы реализовать *IWin32Window* в надстройке, сначала нужно добавить его в список интерфейсов этой надстройки:

```
public class Connect : Object, Extensibility.IDTExtensibility2,
System.Windows.Forms.IWin32Window
```

Затем добавляется реализация свойства *Handle*:

```
//Реализация свойства IWin32Window.Handle:
public System.IntPtr Handle
{
    get
    {
        return new System.IntPtr (applicationObject.MainWindow.HWnd);
    }
}
```

Теперь можно вывести форму (предполагается, что в проекте надстройки есть класс формы *Form1*):

```
Form1 form1 = new Form1();
form1.ShowDialog(this);
```

Реализация этого интерфейса в макросе еще проще; проект примера макроса, устанавливаемый вместе с Visual Studio .NET, содержит код класса, реализующего этот интерфейс. Этот класс называется *WmWrapper,* расположен он в модуле Utilities проекта Samples, и вы можете создавать его экземпляры и передавать их любому коду, которому нужно родительское окно, скажем, стандартному диалоговому < жну Open File:

```
Sub ShowFileOpenDialog()
    Dim openFile As New OpenFileDialog
    openFile.ShowDialog(New WinWrapper)
End Sub
```

Вам нужно лишь скопировать класс *WmWrapper* в свой проект макроса, и он готов к применению.

# Окна-проводники и объект *UIHierarchy*

Окна иерархии пользовательского интерфейса (UI-иерархии) — это окна инструментов, использующие для представления данных древовидную структуру- Примерами могут служить окна Solution Explorer, Server Explorer и Macro Explorer, Объект *UIHierarchy* и связанные с ним объекты *UIHierarchyItems* и *UIHierarchyItem* называются так потому, что представляют иерархию объектов, отображаемых в инструментальном окне. Объект *UIHierarchy* активно используется рекордером макросов, позволяя ему записать корректный код для модификации выбранного элемента в окне UI-иерархии. Этот объект также можно задействовать как источник сведений о содержимом инструментальных окон.

## Дерево объекта *UIHierarchy*

Объекты *UIHierarchy, UIHierarchyItems* и *UIHierarchyItem* работают рекурсивно. *UIHierarchy* служит для поиска набора *UIHierarchyItems,* содержащего все корневые элементы в окне UI-иерархии. Каждый корневой элемент дерева представлен объектом *UIHierarchyItem* в наборе *UIHierarchyItems,* а поскольку каждый элемент может в свою очередь содержать подэлементы, свойство *UIHierarchyItem.UIHierarchyItems* возвращает набор *UIHierarchyItems.* Такая структура узлов, возвращающих наборы других узлов, продолжается, пока не заканчивается ветвление дерева Вот макрос, который использует объект *UIHierarchy* для поиска и вывода названия узла верхнего уровня в Macro Explorer:

```
Sub GetTopLevelUIHierItems()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItems As EnvDTE.UIHierarchyItems
    ' Находим окно-проводник макросов
    ' и обьект UIHierarchy для этого окна:
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    ' Получаем набор верхнего уровня:
```

```
    uiHierarchyItems = uiHierarchy.UIHierarchyItems
    ' Выводим имя первого узла в этом наборе:
    MsgBox(uiHierarchyItems.Item(1).Name)
End Sub
```

Здесь осуществляется поиск объекта *UIHierarchy* для окна Macro Explorer и берется набор *UiHierarchyItems*. На экран выводится имя первого элемента набора; в случае окна Macro Explorer это всегда Macros.

Продолжим с этим макросом: узел Macros в окне Macro Explorer содержит несколько проектов макросов. Поскольку этот узел содержит подэлементы, он является контейнером объектов *UIHierarchyItem* и свойство *UIHierarchyItem.UIHierarchyItems* возвращает объект-набор. Этот набор *UiHierarchyItems* содержит список всех проектов макросов, и если мы изменим предыдущий макрос, то сможем проходить по списку проектов макросов:

```
Sub WalkMacroProjects()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItems As EnvDTE.UIHierarchyItems
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    Dim uiHierarchyItem2 As EnvDTE.UIHierarchyItem
    ' Находим окно-проводник макросов
    ' и объект UIHierarchy для этого окна:
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    ' Получаем первый узел в наборе - Macros:
    uiHierarchyItem = uiHierarchy.UIHierarchyItems.Item(1)
    ' Проходим по всем элементам набора,
    ' который является списком проектов:
    For Each uiHierarchyItem2 In uiHierarchyItem.UIHierarchyItems
        MsgBox(uiHierarchyItem2.Name)
    Next
End Sub
```

Чтобы просматривать окна Solution Explorer и Server Explorer, в этом макросе нужно лишь изменить значение, передаваемое методу *WindowsItem*, на *Constants.vsWindowKindSolutionExplorer* или *Constants.vsWindowKindServerExplorer*.

**Примечание** Не кажутся ли вам знакомыми объекты *UIHierarchy*? Работа с объектами *UIHierarchy*, *UiHierarchyItems* и *UIHierarchyItem* для поиска элемента в окне UI-иерархии аналогична работе с *ProjectItems* и *ProjectItem* для поиска элементов проекта. Причина такого сходства в том, что объекты *UIHierarchy* спроектированы так, чтобы соответствовать объектам *ProjectItem* и *ProjectItems*.

## Объект *UIHierarchy*

Поиск конкретного узла в окне UI-иерархии может потребовать большого объема кода, особенно если искомый узел вложен на несколько уровней. Метод *UIHierarchy.GetItem* позволяет сразу найти объект *UIHierarchyItem* нужного узла, а не рыс-

кать по дереву. Так, чтобы получить объект *UIHierarchyItem* макроса *InsertDate*, расположенного в модуле VSEditor проекта Samples, напишите:

```
Sub FindUIHierItemForInsertDateMacro()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    Dim uiHierarchyItems As EnvDTE.UIHierarchyItems
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    uiHierarchyItems = uiHierarchy.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("Macros")
    uiHierarchyItems = uiHierarchyItem.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("Samples")
    uiHierarchyItems = uiHierarchyItem.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("VSEditor")
    uiHierarchyItems = uiHierarchyItem.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("InsertDate")
    MsgBox(uiHierarchyItem.Name)
End Sub
```

Этот код несколько многословен, так что сократим его, применив метод *UIHierarchy.GetItem*:

```
Sub FindUIHierItemForInsertDateMacro2()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    uiHierarchyItem = _
        uiHierarchy.GetItem("Macros\Samples\VSEditor\InsertDate")
    MsgBox(uiHierarchyItem.Name)
End Sub
```

*UIHierarchy.GetItem* принимает строку, которая является путем к элементу, указывающему узел в иерархии. Этот путь получается из имен всех узлов в ветви, разделенных обратным слэшем. Хотя метод *UIHierarchy.GetItem* находит узлы в окнах Macro Explorer и Solution Explorer, ошибка в Visual Studio .NET не позволяет ему находить объект *UIHierarchyItem* в окне Server Explorer.

Свойство *UIHierarchy.SelectedItems* возвращает массив объектов *UIHierarchyItem* для элементов, выбранных в дереве. Как и другие массивы, возвращаемые объектной моделью при использовании языков, поддерживаемых .NET, этот объект состоит из нетипизированных объектов *System.Object*.

```
Sub GetUIHierSelectedItems()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim selectedItems As Object()
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
```

```
    selectedItems = uiHierarchy.SelectedItems
    For Each uiHierarchyItem In selectedItems
        MsgBox(uiHierarchyItem.Name)
    Next
End Sub
```

Чтобы облегчить рекордеру макросов запись движения по иерархии, объект *UIHierarchy* имеет два метода, *SelectUp* и *SelectDown*, имитирующих выбор пользователем узлов в дереве. Оба метода принимают два параметра. Первый имеет тип *EnvDTE.vsUISelectionType* и указывает способ выбора узлов по аналогии с тем, как это делается мышью и с клавиатуры. *EnvDTE.vsUISelectionTypeSele* выбирает один узел в дереве, и все остальные узлы теряют свое состояние выбора. *EnvDTE.vsUI-SelectionTypeExtend* выбирает от последнего выбранного узла до указанного аналогично тому, как это происходит при щелчке нового узла при нажатой клавише Shift. *EnvDTE.vsUISelectionTypeSetCare* не выбирает узел, а перемещает курсор по дереву к указанному узлу. И, наконец, *EnvDTE.vsUISelectionTypeToggle* изменяет состояние выбора узла на противоположное. Второй параметр методов *SelectUp* и *SelectDown* — счетчик. По умолчанию выбирается один элемент, но этим параметром вы можете указать другое число одновременно выбираемых элементов.

У объекта *UIHierarchy* есть также метод *DoDefaultAction*, имитирующий нажатие пользователем клавиши Enter когда в дереве выбран один или несколько узлов. Так, если в Macro Explorer выбран макрос и вызывается метод *UIHierarchy.DoDefaultAction*, этот макрос выполняется.

## Объект *UIHierarchyItems*

Объект *EnvDTE.UIHierarchyItems* является набором объектов *EnvDTE.UIHierarchyItem* и работает, как любой другой объект-набор в объектной модели Visual Studio. Он поддерживает одно свойство, которое не является частью стандартного набора методов и свойств других объектов-наборов, — *Expanded*. Оно имеет тип *Boolean* и указывает, отображаются ли в пользовательском интерфейсе узлы из набора *UIHierarchyItems*. Присваивание этому свойству значения *True* имеет тот же эффект, что щелчок знака «плюс» рядом с узлом дерева, а *False* — знака «минус».

## Объект *UiHierarchyItem*

*EnvDTE.UIHierarchyItem*, будучи элементом набора, поддерживает стандартные для этого типа методы и свойства, такие как *Collection* и *Name*. Он также поддерживает метод *Select*, похожий на методы *UIHierarchy.SelectUp* и *UIHierarchy.SelectDown*, с той разницей, что работает в каждый отдельный момент с единственным узлом — *UIHierarchyItem*, указываемым при вызове *Select*. Поскольку метод *Select* изменяет только текущий *UIHierarchyItem*, он не принимает количество выбираемых элементов.

Свойство *UIHierarchyItem.Object* возвращает объект из модели расширяемости, если он доступен для данного узла. Так, используя Solution Explorer, вы можете получить объекты *EnvDTE.Project* или *EnvDTE.ProjectItem* для узла. Следующий код ищет *UiHierarchyItem* для первого проекта и второй элемент в этом проекте (второй элемент ищется потому, что первым элементом в проектах .NET является узел

References), а также получает объекты *EnvDTE.Project* и *EnvDTE.ProjectItem* для этих узлов:

```
Sub GetUIHierItemObject()
    Dim uihier As EnvDTE.UIHierarchy
    Dim uihierProj As EnvDTE.UIHierarchyItem
    Dim uihierProjItem As EnvDTE.UIHierarchyItem
    Dim project As EnvDTE.Project
    Dim projItem As EnvDTE.ProjectItem
    uihier = DTE.Windows.Item( _
        Constants.vsWindowKindSolutionExplorer).Object
    uihierProj = uihier.UIHierarchyItems.Item(1).UIHierarchyItems.Item(1)
    project = uihierProj.Object
    uihierProjItem = uihierProj.UIHierarchyItems.Item(2)
    projItem = uihierProjItem.Object
End Sub
```

# Окно Toolbox

В Toolbox хранятся элементы управления и фрагменты кода, которые можно переносить в окно дизайнера форм, окна редакторов и любое другое место, допускающее буксировку. Toolbox состоит из набора вкладок, позволяющих группировать элементы по категориям. Элементы вкладок относительно статичны, но одна вкладка, Clipboard Ring, часто меняется и содержит текстовые фрагменты, вырезанные или скопированные из окон редакторов.

## Вкладки и элементы

Чтобы получить окно Toolbox, можно передать константу *vsWindowKindToolbox* методу *Windows.Item*, который вернет объект *Window*, Объект *ToolBox* затем можно получить из свойства *Window.Object* возвращенного объекта:

```
Sub FindTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
End Sub
```

Поскольку окно Toolbox может иметь несколько вкладок, для них предусмотрен набор, позволяющий перечислять все эти вкладки, — *ToolBoxTabs*, который получают из свойства *ToolBox.ToolBoxTabs*. Этот набор позволяет находить каждый объект *ToolBoxTab* в Toolbox и даже создавать новые вкладки для хранения любых компонентов и текстовых фрагментов. Для создания вкладки служит метод *ToolBoxTabs.Add*, который принимает в качестве аргумента имя вновь создаваемой вкладки и возвращает соответствующий ей объект *ToolBoxTab*. Следующий макрос добавляет в Toolbox новую вкладку:

```
Sub AddNewToolBoxTab()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
```

```
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBox.ToolBoxTabs.Add("My commonly used items").Activate()
End Sub
```

Этот код создает вкладку My Commonly Used Items, и она становится выбранной благодаря вызову метода *Activate* объекта *ToolBoxTab*.

Кроме того, что Toolbox является набором вкладок, каждая из вкладок представляет собой набор элементов. Каждый элемент набора представлен в объектной модели объектом *ToolBoxItem* и может быть найден в объекте *ToolBoxItems*, получаемом из свойства *ToolBoxTab.ToolBoxItems*. Все содержимое Toolbox позволяет просмотреть макрос *EnumerateToolBoxContents*:

```
Sub EnumerateToolBoxContents()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As ToolBoxTab
    Dim outputWindow As New _
        InsideVSNET.Utilities.OutputWindowPaneEx(DTE, "Toolbox contents")
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    For Each toolBoxTab In toolBox.ToolBoxTabs
        Dim toolBoxItem As ToolBoxItem
        outputWindow.WriteLine(toolBoxTab.Name)
        For Each toolBoxItem In toolBoxTab.ToolBoxItems
            outputWindow.WriteLine(vbTab + toolBoxItem.Name)
        Next
    Next
End Sub
```

С объектом *ToolBoxItem* особо делать нечего. Можно вызвать его метод *Select*, чтобы сделать элемент активным в Toolbox, можно удалить методом *Delete*, а также получить надпись, отображаемую в пользовательском интерфейсе, из свойства *Name*. Хотя функционально объектная модель *ToolBoxItem* довольно убога, настоящая мощь объектной модели Toolbox скрыта в возможности создавать новые элементы.

## Добавление элементов в Toolbox

Toolbox может содержать разные виды объектов, такие как текст, HTML, COM- и .NET-компоненты. Вы можете добавлять собственные элементы методом *ToolBoxTabAdd*, который принимает три параметра: *Name* (имя добавляемого элемента, отображаемого в пользовательском интерфейсе Toolbox), *Data* (определяет информацию об элементе, хранимую в Toolbox) и *Format* (имеет тип *vsToolBoxItemFormat* и отвечает за форматирование этой информации).

Простейший тип хранимых данных — обычный текст. Строка, передаваемая параметром *Data*, копируется в элемент Toolbox без изменений, и, когда такой текст буксируется в окно, поддерживающее буксировку в формате *text* буфера обмена (скажем, в окно текстового редактора), этот текст копируется в окно. Вот как добавить текстовый фрагмент:

```
Sub AddTextToTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    toolBoxItems.Add("My Text", "This is some text", _
        vsToolBoxItemFormat.vsToolBoxItemFormatText)
End Sub
```

В этом коде сначала просматривается объектная модель в поисках вкладки General окна Toolbox. В конце вызывается метод *ToolBoxItemsAdd*, добавляющий элемент с именем *My Text* и текстом *This is some text,* который имеет формат буфера обмена типа *text.*

Добавление текста в формате HTML аналогично добавлению простого текста — разница в том, что передается не обычный текст, а фрагмент HTML-кода, а формат передаваемых данных устанавливается в *vsToolBoxItemFormatHTML*

```
Sub AddHTMLToTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    toolBoxItems.Add("My HTML", "<b>This is bold HTML</b>", _
        vsToolBoxItemFormat.vsToolBoxItemFormatHTML)
End Sub
```

По выполнении этого кода фрагмент HTML будет помещен в Toolbox, и если вы отбуксируете этот элемент Toolbox в дизайнер HTML, то увидите, что текст имеет полужирное начертание.

---

**Примечание** HTML на самом деле — подмножество XML, соответствующее определенной схеме. Поскольку HTML — это XML, вы можете хранить в Toolbox фрагменты XML как HTML. Visual Studio .NET позволяет буксировать эти фрагменты HTML/XML не только в HTML-, но и в XML-документ.

---

Кроме этих двух текстовых форматов, Toolbox может хранить элементы управления ActiveX, которые можно буксировать в HTML-файлы или Win32-приложения (такие как диалоговые окна MFC), поддерживающие хостинг элементов управления ActiveX. Добавляя ActiveX, вы указываете тип данных *vsToolBoxItemFormatGUID.* Формат аргумента *Data* — CLSID GUID элемента управления ActiveX или (несмотря на название формата) ProgID этого элемента управления. Следующий макрос

добавляет в Toolbox две копии элемента управления Windows Media Player. Первый добавляется с использованием его CLSID, а второй — на основании его ProgID:

```
Sub AddCOMObjectToTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    toolBoxItems.Add("Name", "{22D6F312-B0F6-11D0-94AB- 0080C74C7E95}", ..
        vsToolBoxItemFormat.vsToolBoxItemFormatGUID)
    toolBoxItems.Add("Name", "MediaPlayer.MediaPlayer.1", _
        vsToolBoxItemFormat.vsToolBoxItemFormatGUID)
End Sub
```

Запустив этот код, вы увидите, что параметр *Name* проигнорирован. Дело в том. что Toolbox берет имя из элемента управления.

Последняя разновидность элементов Toolbox — это компоненты .NET Framework. Вы можете использовать формат *vsToolBoxItemFormatDotNETComponent* и передать путь к сборке .NET. В этой сборке ищутся элементы управления, и, если таковые обнаруживаются, они добавляются на вкладку Toolbox. Например, у вас есть сборка WindowsControlLibrary.dll в корне диска C:. Следующий код добавляет все элементы управления из этой сборки в Toolbox:

```
Sub AddDotNetComponentToTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    toolBoxItems.Add("Control", "C:\WindowsControlLibrary.dll", _
        vsToolBoxItemFormat.vsToolBoxItemFormatDotNETComponent)
End Sub
```

Зачастую сборки, содержащие элементы управления, располагаются в глобальном кэше сборок (GAC), что делает их доступными всем программам, работающим под управлением .NET Framework. Поскольку путь к сборке в GAG найти непросто, вы можете загрузить такую сборку, узнать путь к ней и передать этот путь функции добавления в Toolbox. Допустим, у вас есть пользовательский элемент управления в сборке с именем *WindowsControlLibrary*, номером версии 1.0.795.38182 и открытым ключом 6fc70375761725c0. Вы можете загрузить сборку статическим методом *System.Reflection.Assembly.Load*, передав ему имя сборки, номер версии и открытый ключ. Этот метод возвращает объект *System.Reflection.Assembly*, поддерживающий свойство *CodeBase*, которое возвращает путь в формате *file:///*.

```
Sub AddDotNetComponentToTheToolBox2()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    Dim asm As System.Reflection.Assembly
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    asm = System.Reflection.Assembly.Load("WindowsControlLibrary1,
      Version=1.0.795.38182, Culture=neutral, PublicKeyToken=6fc70375761725c0")
    toolBoxItems.Add("Control", asm.CodeBase, _
       vsToolBoxItemFormat.vsToolBoxItemFormatDotNETComponent)
End Sub
```

Так же, как и при добавлении ActiveX, когда вы добавляете в Toolbox компонент .NET, параметр *Name* игнорируется, и имя элемента управления берется из самого элемента. Это может показаться ошибкой в Visual Studio .NET, но добавление в Toolbox сборки отличается от добавления элементов других форматов. Когда вы добавляете текст, HTML или ActiveX, возвращается объект *EnvDTE.ToolBoxItem* для вновь добавленного элемента. Но когда вы добавляете сборку, всегда возвращается *null* или *Nothing* (в зависимости от языка программирования). Дело в том, что сборка может содержать ноль, один или несколько элементов управления. Если сборка не содержит элементов управления, возвращаемое значение *null* или *Nothing* имеет смысл, так как ничего не было добавлено и нет объекта *ToolBoxItem*, который можно бы было вернуть. Несколько элементов не могут быть возвращены, так как метод возвращает единственный объект *EnvDTE.ToolBoxItem*, а не массив. Если в сборке один элемент управления, можно бы было вернуть *EnvDTE.ToolBoxItem*, но было бы странно возвращать элемент в одном случае и не возвращать в других, так что разработчики решили и здесь возвращать *null* или *Nothing*.

# Окно Task List

Как вы уже знаете, программируемым объектом для окна Task List является *EnvDTE.TaskList*. Через объект *TaskList* вы можете добавлять новые элементы задач, предоставляя пользователю сведения о работах, которые нужно выполнить, а также просматривать задачи, добавленные компилятором и другими инструментами.

## Элементы списка задач

Объект *EnvDTE.TaskList* предоставляет доступ к списку задач через свойство *TaskItems*, которое возвращает набор *TaskItems*, содержащий по одному элементу для каждой задачи в окне Task List. Можно просмотреть подмножество элементов Task List, отфильтровав элементы, не принадлежащие к определенной группе или категории, но при этом элементы, скрытые в результате такой фильтрации, будут оставаться элементами набора *EnvDTE.TaskItems*.

Как и любой набор, *EnvDTE.TaskItems* можно индексировать числовыми значениями, получая объект *EnvDTE.TaskItem*. Использовать в качестве индексов строки для этого набора нельзя,

## Добавление новых задач

Добавление новых элементов в Task List позволяет создавать разнообразные новые инструменты. Вот примеры инструментов, использующих Task List.

- Средство анализа кода, обнаруживающее программные ошибки. Вы можете добавлять описание этих ошибок вместе с сообщениями компилятора об ошибках в Task List.
- Средства планирования, которые получают информацию от других программ, таких как Microsoft Project, и создают задачи, извещающие программистов о необходимости выполнения очередной части работ. Если установить флажок, соответствующий задаче, в Project эта задача помечается как выполненная,
- Надстройка, просматривающая ошибки компиляции и по возможности их исправляющая. Помните, что при последней компиляции проекта C# все ошибки были из-за пропуска точек с запятой? Неплохо бы было иметь средство автоматического исправления этих ошибок?
- Макрос, синхронизующий ваш календарь в Microsoft Outlook со списком задач Visual Studio .NET. Помимо прочего, он может напомнить вам купить подарок по дороге с работы.

Добавление подобных средств обеспечивает метод *TaskItems.Add*, предлагающий также их отображение в списке. У него одна из самых сложных сигнатур среди методов Visual Studio .NET:

```
public EnvDTE.TaskItem Add(string Category,
    string SubCategory,
    string Description,
    EnvDTE.vsTaskPriority Priority = vsTaskPriorityMedium,
    object Icon,
    bool Checkable = false,
    string File = "",
    int Line = -1,
    bool CanUserDelete = true,
    bool FlushItem = true)
```

Увидеть эффект применения различных комбинаций параметров этого метода позволяет надстройка-пример AddTaskListItems. Мы же рассмотрим все параметры в следующих разделах.

### *Category* и *SubCategory*

Каждая задача независимо от того, создана она программно или посредством Visual Studio .NET, принадлежит к той или иной категории. Категории применяются для группирования задач и связывания их между собой. Наиболее распространенные типы категорий: ошибки компиляции, пользовательские задачи и ярлыки. Когда пользователь щелкает правой кнопкой Task List и выбирает Show Tasks, могут быть показаны или все задачи, или только те, что принадлежат к определенной категории. Создать новую категорию позволяет параметр *Category* метода *Add*. При вызове этого метода просматривается список существующих категорий в поисках имени, совпадающего с заданным аргументом. Если совпадения не найдено, добавляется новая категория, и новая задача помещается в нее. Если категория существует, новая задача добавляется в нее.

В настоящее время Visual Studio .NET не использует аргумент *SubCategory* метода *Add*; ваши надстройки и макросы должны оставлять этот параметр пустым.

### *Description*

Описание задачи отображается в столбце Description списка задач, и этот столбец заполняется аргументом *Description* метода *Add,* имеющего три обязательных параметра: *Description, Category* и *SubCategory*. Опустив остальные необязательные параметры, создадим наш первый элемент списка Task List:

```
Sub TLAddItems()
    Dim taskList As EnvDTE.TaskList
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
    taskList.TaskItems.Add("Category", "", "Description2")
    taskList.TaskItems.Add("Category", "", "Description1")
End Sub
```

Выполнив этот макрос и выбрав Sort By | Category в контекстном меню Task List. вы увидите список задач (рис. 10-1),




**Рис. 10-1.** *Элементы Task List, добавленные макросом*

### *Priority*

Следующий аргумент, который вы можете передать методу *Add*, — это *Priority,* Он является необязательным, но если он задан, в первом столбце Task List — столбце приоритета — появляется значок, чтобы напомнить пользователю о важности задачи. Для высокоприоритетных задач отображается красный восклицательный знак, для низкоприоритетных — направленная вниз голубая стрелка, а для среднеприоритетных значка нет. Следующий макрос добавляет в Task List новые элементы с разными приоритетами. По выполнении этого макроса окно Task List должно выглядеть, как показано на рис. 10-2.

```
Sub TLAddItemsPriority()
    Dim taskList As EnvDTE.TaskList
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
```

```
    taskList.TaskItems.Add("Category", "", _
        Description1", vsTaskPriority.vsTaskPriorityHigh)
    taskList.TaskItems.Add("Category", "", _
        "Description2", vsTaskPriority.vsTaskPriorityLow)
    taskList.TaskItems.Add("Category", "", _
        "Description3", vsTaskPriority.vsTaskPriorityMedium)
End Sub
```

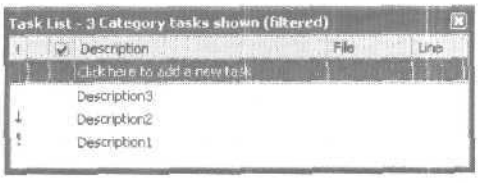



**Рис. 10-2.** *Элементы Task List с разными приоритетами*

### *Icon*

Параметр *Icon* позволяет задавать значок для добавляемой задачи. Пять стандартных значков описаны в табл. 10-2.

**Табл. 10-2. Предопределенные значки для элементов Task List**

| Изображение значка | Константа |
|---|---|
| » | *vsTaskIcon.vsTaskIconComment* |
| | *vsTaskIcon.vsTaskIconUser* |
| } | *vsTaskIcon.vsTaskIconSquiggle* |
| | *vsTaskIcon.vsTaskIconShortcut* |
| | *vsTaskIcon.vsTaskIconCompile* |

В этой таблице не показан значок по умолчанию, *vsTaskIconNone* — пустой значок, который отображается при отсутствии этого параметра.

**Примечание** Если вы передадите методу *TaskItemsAdd* в качестве значка значение *vsTaskIconShortcut*, ярлык создаваться не будет. Значок служит только для описательных целей. Это относится и к другим значкам: *vsTaskIconCompile* не создает ошибку компилятора, *vsTaskIconComment* не добавляет комментарий в исходный файл и т. д.

Вы можете создать и собственные картинки. Вам потребуется растровое изображение 16x16 пикселов с 16 цветами. Пикселы с RGB-цветом 0,255,0 будут прозрачны, т. е. за ними будет виден фон окна Task List. Чтобы установить картинку, сначала нужно загрузить растровое изображение в объект *IPictureDisp* и затем передать его в параметре *Icon*. *IPictureDisp* — это способ работы с растровыми объектами в стиле COM. Чтобы создать объект *IPictureDisp* в .NET-надстройке, придется написать небольшой P/Invoke-код для создания объекта этого типа. (P/Invoke — это технология .NET Framework, применяемая для вызова неуправля-

емого кода из программ .NET.) Системная DLL oleaut32.dll экспортирует метод *OleLoadPictureFile,* который принимает путь к файлу картинки и возвращает объект *IPictureDisp.* Прежде чем вызывать метод *OleLoadPictureFile,* в класс, реализующий надстройку, нужно добавить код:

```
[DllImport("oleaut32.dll", CharSet=System.Runtime.InteropServices.CharSet.Auto,
  SetLastError=true)]
internal extern static int OleLoadPictureFile(object fileName,
[MarshalAsAttribute(UnmanagedType.IDispatch)] ref object ipictrueDisp);
```

который определяет сигнатуру кода, экспортируемого из COM-библиотеки OleAut32.dll. Имея такое объявление P/Invoke-метода, вы сможете вызвать метод *OleLoadPictureFile* с указанием имени файла вашей картинки:

```
object objIPictureDisp = null;
string filename = "C:\SomeImage.eps";
int nret = OleLoadPictureFile(fileName, ref objIPictureDisp);
```

При возврате из этого метода переменная *objIPictureDisp* указывает на объект *IPictureDisp,* который может быть передан параметром *Icon* методу *TaskItemsAdd.*

Если вы попробуете вызвать метод *Add* из макроса и передать в параметре *Icon* объект *IPictureDisp,* возникнет исключение. Это случится потому, что редактор макросов Visual Studio .NET выполняет ваши макросы в отдельном процессе. Когда выполняется метод или свойство объектной модели Visual Studio .NET, все данные должны быть маршализованы из памяти редактора макросов в память программы Visual Studio .NET через границы процессов. Однако такие объекты, как *IPicture* и *IpictureDisp,* не могут быть маршализованы через границы процессов, поэтому, если вы создадите *IPicture* и *IPictureDisp* и передадите их методу *TaskItems.Add* из макроса, будет сгенерировано исключение. Такое ограничение не позволяет создавать задачу с нестандартным значком из макроса, но вы можете это делать из надстроек, так как они загружаются в одном процессе с Visual Studio .NET.

#### *Checkable*

Параметр *Checkable* указывает, нужен ли флажок для задачи. Если параметр равен *true,* флажок будет доступен, *false* — флажок не отображается.

#### *File и Line*

Параметры *File* и *Line* имеют строковый и целочисленный тип соответственно и определяют содержимое колонок File и Line в окне Task List. Значения этих параметров могут быть произвольными — их назначение чисто информационное. Если пользователь выполняет действие по умолчанию для задачи (дважды щелкая выделенную задачу или нажимая клавишу Enter), файл не открывается, а курсор не перемещается на строку, соответствующую аргументу *Line.* Дело в том, что объектная модель не знает о типе данных файла; если это двоичный файл, открытие его и установка указателя на какую-то строку вряд ли имеет смысл. Однако вы можете установить обработчик событий для созданного элемента в списке задач, отслеживать события *TaskNavigate* (см. ниже), а затем вручную открыть файл из программы.

### CanUserDelete

Этот параметр указывает, может ли пользователь удалить задачу, когда она выделена в списке, нажав клавишу Delete. Если этому параметру задано *false*, пользователь не может удалить элемент из интерфейса пользователя, но может сделать это программно, вызвав метод *TaskItem.Delete*.

### FlushItem

Это последний параметр метода *TaskListItems.Add*— булевское значение *FlushItem*. При добавлении каждого нового элемента в окно Task List это окно нужно обновить, чтобы показать новую задачу. Если задач много, перерисовка Task List при добавлении каждой из них снижает производительность. Если параметром *FlushItem* передать *false*, окно Task List не будет обновлено до добавления очередной задачи или до вызова метода *TaskItems.ForceItemsToTaskList*.

## Объект TaskItem

После того как элемент добавлен в Task List — программным способом или самой Visual Studio .NET — и получен из набора *TaskItems*, вы можете анализировать и изменять данные, отображаемые для данного элемента посредством методов и свойств объекта *TaskItem*. Свойства *Category*, *SubCategory*, *Checked*, *Description*, *FileName*, *Line* и *Priority* можно прочитать программно и получить данные соответствующих столбцов списка Task List. Можно также устанавливать эти свойства, поскольку они изменяемые. Некоторые столбцы элементов, создаваемых Visual Studio .NET, помечены как неизменяемые, и модифицировать их нельзя. Проверить, можно ли изменять некоторый столбец, позволяет свойство *IsSettable*, которое принимает как параметр столбец списка задач (значение из перечислителя *vsTaskListColumn*) и возвращает *true*, если столбец изменяемый, и *false* — в противном случае. Так, изменить описание задачи позволяет следующий код, который сначала проверяет, можно ли изменять описание:

```
Sub ModifyTaskDescription()
    Dim taskList As EnvDTE.TaskList
    Dim task As EnvDTE.TaskItem
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
    task = taskList.TaskItems.Item(1)
    If (task.IsSettable(vsTaskListColumn.vsTaskListColumnDescription)) Then
        task.Description = "A new description"
    End If
End Sub
```

*Delete* удаляет элемент, если его удаление возможно. Как мы уже говорили, любой элемент, добавленный через объектную модель, можно удалить независимо от того, установлен ли параметр *CanUserDelete* для него в *true* или в *false* при вызове метода *TaskItems.Add*. Можно ли удалять другие элементы, зависит от того, кто их создал. Например, если задача была создана, когда пользователь щелкнул строку Click Here To Add A New Task Item в окне Task List, она может быть удалена через объектную модель. Если элемент был создан IntelliSense или компилятором, при вызове метода *Delete* возникнет исключение, поскольку единственный способ удале-

ния такой задачи — изменение исходного кода, который вызвал появление такой задачи.

Метод *Navigate* имитирует двойной щелчок или нажатие клавиши Enter, когда задача имеет фокус. Если задача добавлена Visual Studio .NET или является ярлыком, это приведет к открытию целевого файла и переводу курсора в строку, указанную в задаче. Если задача добавлена через объектную модель, ничего не последует, если вы не написали код для поиска соответствующего файла и строки в обработчике события *TaskNavigated*.

## События Task List

При взаимодействии пользователя с Task List возникают события, позволяющие надстройкам и макросам реагировать на действия пользователя. Пожалуй, важнейшее событие для макроса, добавляющего задачу в список, — *TaskNavigate*. Оно возникает, когда пользователь дважды щелкает элемент Task List, нажимает Enter, когда задача имеет фокус или выбирает элементы Next Task или Previous Task из контекстного меню Task List. Чтобы уловить этот момент, нужно подключиться к событию *TaskListEvents.TaskNavigated*. Оно передает объект *TaskItem* того элемента, на который хочет перейти пользователь, а также ссылку на булевское значение *NavigateHandled*, которое позволяет указать Visual Studio .NET, обработал ли ваш код переход на элемент списка. Если параметром *NavigateHandled* передается *false* и больше никто не обрабатывает это событие, Visual Studio .NET выдает звуковой сигнал.

Подключение к этому событию в проекте макроса сводится к открытию модуля *EnvironmentEvents* макроса, выбору элемента TaskListEvents в выпадающем списке в левом верхнем углу окна редактирования этого модуля и выбора события *TaskNavigated* из выпадающего списка в правом верхнем углу. Используя это событие, можно написать макрос-обработчик события *NavigateHandled*, открывающий файл (если он указан), на который ссылается задача, и перемещающий курсор на строку исходного файла, указанную в задаче. Код обработчика этого события может выглядеть так:

```
Private Sub TaskListEvents_ TaskNavigated(ByVal TaskItem As EnvDTE.TaskItem, _
    ByRef NavigateHandled As Boolean) Handles TaskListEvents.TaskNavigated
    ' Если для этой задачи указан аргумент file...
    If (TaskItem.FileName <> "") Then
        Dim fileWindow As EnvDTE.Window
        Dim textWindow As EnvDTE.TextWindow
        Dim textPane As EnvDTE.TextPane

        ' Открываем файл, ищем объекты TextWindow и TextPane objects...
        fileWindow = DTE.ItemOperations.OpenFile(TaskItem.FileName, _
            EnvDTE.Constants.vsViewKindTextView)
        textWindow = CType(fileWindow.Object, EnvDTE.TextWindow)
        textPane = CType(textWindow.ActivePane, EnvDTE.TextPane)

        ' Перемещаем курсор на нужную строку:
        textPane.Selection.MoveTo(TaskItem.Line, 1, False)
        textPane.Selection.SelectLine()
```

```
        NavigateHandled = True
    End If
End Sub
```

Подключение к этому событию из надстройки требует чуть больше кода, чем из макроса. Сначала объявляется переменная для подключения обработчика события. Поскольку мы подключаемся к событиям окна Task List, то используем класс *EnvDTE.TaskListEvents*:

```
private EnvDTE.TaskListEvents taskListEvents;
```

Затем объявляем метод обработчика события, который должен соответствовать сигнатуре, приведенной в Object Browser. Можно перевести код макроса, показанный ранее, на C# для надстройки:

```
public void TaskNavigated(EnvDTE.TaskItem taskItem,
    ref bool navigateHandled)
{
    // Если для этой задачи указан аргумент file...
    if(taskItem.FileName != "")
    {
        EnvDTE.Window fileWindow;
        EnvDTE.TextWindow textWindow;
        EnvDTE.TextPane textPane;

        // Открываем файл, ищем объекты TextWindow и TextPane objects...
        fileWindow = applicationObject.ItemOperations.OpenFile(
            taskItem.FileName, EnvDTE.Constants.vsViewKindTextView);
        textWindow = (EnvDTE.TextWindow)fileWindow.Object;
        textPane = (EnvDTE.TextPane)textWindow.ActivePane;

        // Перемещаем курсор на нужную строку:
        textPane.Selection.MoveTo(taskItem.Line, 1, false);
        textPane.Selection.SelectLine();
        navigateHandled = true;
    }
}
```

В завершение нужно присвоить переменной *taskListEvents* экземпляр объекта *TaskListEvents*, который позволяет получить свойство *Events.TaskListEvents*. Это свойство принимает один аргумент — категорию, используемую как фильтр. Если аргументом передать пустую строку обработчик события будет вызываться при инициации события любой задачей независимо от того, была ли она добавлена программно или самой Visual Studio .NET. Но если указать категорию — ту, что вы передавали первым параметром метода *TaskItems.Add*, — обработчик события будет получать уведомления только от задач этой категории. Такой механизм фильтрации позволяет уменьшить число инициируемых событий, повышая производительность кода. Поскольку мы хотим обрабатывать события от всех задач, мы передаем свойству *Events.TaskListEvents* пустую строку:

```
EnvDTE.Events events = applicationObject.Events;
taskListEvents = (EnvDTE.TaskListEvents)events.get_TaskListEvents("");
```

Последний шаг — установить соответствие между объектом события и его обработчиком. Для этого создается новый объект *EnvDTE._dispTaskListEvents_TaskNavigatedEventHandler* и добавляется в набор *taskListEvents.TaskNavigated* обработчиков события:

```
taskListEvents.TaskNavigated += new
    _dispTaskListEvents_TaskNavigatedEventHandler(this.TaskNavigated);
```

*TaskNavigated* — не единственное событие Task List, способное отслеживать код. События *TaskAdded* и *TaskRemoved* возникают сразу после добавления задачи и после ее удаления соответственно. Последнее событие, *TaskModified*, возникает при изменении одного из столбцов Task List. Например, пользователь может установить/снять флажок для задачи, изменить ее приоритет или описательный текст. Событие *TaskModified* сообщает об изменениях в задаче, передавая *TaskItem* задачи и измененный столбец.

## Маркеры комментариев

Программисты часто оставляют программы незаконченными, собираясь вставить куски кода потом, Эти пропущенные куски обычно отвечают за проверку ошибок, анализ параметров или просто являются дополнительными функциями, вместо которых вставляются какие-то примечания. Разумеется, если специально не поискать эти примечания в тексте программы (визуально или средствами поиска Visual Studio .NET), вы никогда к ним не вернетесь и не внесете запланированные изменения. Между тем, если использовать специальную нотацию, такие примечания может автоматически находить Task List и сообщать о них. Когда вы открываете исходный файл, он сканируется на предмет этих специальных маркеров и, если они встречаются, создаются записи в списке задач. Такие маркеры в исходном тексте начинаются с признака комментария применяемого языка программирования, за которым следует маркер примечания, двоеточие и текст, который будет отображаться в списке Task List. Так, комментарий *//TODO: Fix this later* создает элемент в Task List для языков Visual C++ и C# с описанием *Fix this later*, а *'TODO: Fix this later* делает то же самое для Visual Basic .NET.

Специальные маркеры, которые ищет Task List, определяются в диалоговом окне Options, где можно добавить новые маркеры и удалить или изменить существующие. На рис. 10-3 показаны маркеры HACK, TODO, UNDONE и UnresolvedMergeConflict.

Добавить новый маркер можно, набрав его имя в поле Name, выбрав приоритет и щелкнув кнопку Add. Маркеры можно также добавлять, удалять и изменять через объектную модель. Для этого служит набор *Properties*. Подробнее о наборе *Properties* мы поговорим ниже, а сейчас остановимся только на его особенностях, связанных с маркерами. Первый шаг — получение свойства *CommentTokens* с помощью подобного кода:

```
Sub GetTokensArray()
    Dim tokens As Object()
    Dim prop As EnvDTE.Property
    Dim props As EnvDTE.Properties
    props = DTE.Properties("Environment", "TaskList")
```

```
    prop = props.Item("CommentTokens")
    tokens = prop.Value
End Sub
```

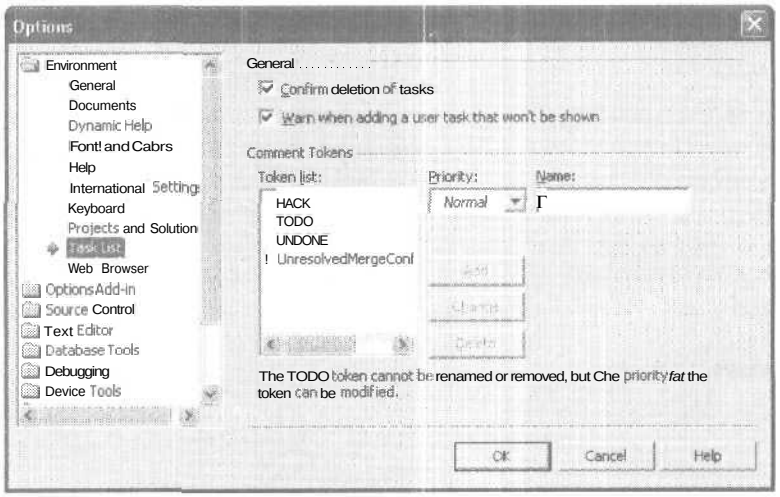



Рис. 10-3. *Параметры маркеров Task List*

Свойство *CommentTokens* возвращает массив строк особого формата — *TokenName:Priority,* где *TokenName* — текст, следующий за признаком комментария языка программирования, а *Priority* — числовой индекс элемента в перечислении *EnvDTE.vsTaskPriority.* В предыдущем макросе строка для маркера TODO — *«TODO:2»,* т. е. искомая строка в текстовом редакторе — *«TODO»,* а приоритет для этого маркера в Task List — *vsTaskPriorityMedium* (чье числовое значение равно 2).

Добавление собственного маркера в список производится в три этапа. Установка списка маркеров очищает текущий список (по крайней мере это касается изменяемых маркеров), поэтому нужно сохранить существующие маркеры. Сначала нужно получить список текущих маркеров в списке Task List, затем добавить собственный маркер к массиву существующих маркеров и, наконец, присвоить свойству расширенный массив. В следующем макросе в список маркеров комментариев добавляется маркер *SECURITY* с высоким приоритетом:

```
Sub AddSecurityToken()
    Dim tokens As Object()
    Dim token As String
    Dim prop As EnvDTE.Property
    Dim props As EnvDTE.Properties
    ' Получаем свойство, хранящее существующие маркеры.
    props = DTE.Properties("Environment", "TaskList")
    prop = props.Item("CommentTokens")
    tokens = prop.Value
    ' Расширяем список существующих маркеров
    ' для хранения нового.
    ReDim Preserve tokens(tokens.Length)
    ' Добавляем новый маркер.
```

```
    tokens(tokens.Length - 1) = "SECURITY:3"
    ' Устанавливаем список известных маркеров.
    prop.Value = tokens
End Sub
```

Удаляет маркер похожий макрос:

```
Sub RemoveSecurityToken()
    Dim tokens As Object()
    Dim newTokens As Object()
    Dim token As String
    Dim i As Integer = 0
    Dim found As Boolean = False
    Dim prop As EnvDTE.Property
    Dim props As EnvDTE.Properties
    props = DTE.Properties("Environment", "TaskList")
    prop = props.Item("CommentTokens")
    tokens = prop.Value
    ' Массив не нужно сжимать, если маркера нет.
    For Each token In tokens
        If token = "SECURITY:3" Then
            found = True
            Exit For
        End If
    Next
    ' Если маркер SECURITY не найден,
    ' удалять нечего и можно закончить.
    If found = False Then
        Exit Sub
    End If
    ' Изменяем размер массива newTokens.
    ReDim newTokens(tokens.Length - 2)
    ' Колируем список маркеров в массив newTokens,
    ' пропуская маркер SECURITY.
    For Each token In tokens
        If token <> "SECURITY:3" Then
            newTokens(i) = token
            i = i + 1
        End If
    Next
    ' Устанавливаем список маркеров.
    prop.Value = newTokens
End Sub
```

Если ваша надстройка генерирует код, помещаемый в текстовый буфер, и вы хотите добавить маркер комментария, информирующий пользователя о том, как изменить этот код, можете получить вставляемый маркер через свойство *TaskListDefaultCommentToken*. Следующий код создает строку, содержащую класс с маркером комментария по умолчанию, который указывает пользователю, где добавить код:

```
Sub InsertTLTokenCode()
    Dim classString As String
```

```
    Dim taskList As EnvDTE.TaskList
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
    classString = "Public Class AClass" + Chr(13)
    classString = classString + Chr(9) + " " + taskList.DefaultCommentToken
    classString = classString + ": Insert your code here" + Chr(13)
    classString = classString + "End Class"
End Sub
```

# Окно Output

В окне Output Visual Studio .NET отображает текстовую информацию, генерируемую компилятором и отладчиком. Окно Output — подходящее место и для вывода информации вашими инструментами. В этой книге примеры макросов и надстроек используют библиотеку классов OutputWindowPaneEx для вывода текста в окно Output.

Окну Output соответствует объект *OutputWindow*, который можно получить так:

```
Sub FindOutputWindow()
    Dim window As EnvDTE.Window
    Dim outputWindow As EnvDTE.OutputWindow
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindOutput)
    outputWindow = CType(window.Object, EnvDTE.OutputWindow)
End Sub
```

## Области окна Output

Пользовательский интерфейс окна Output состоит из нескольких областей (panes), в каждой из которых отображается некоторый текст. Вы можете переключаться между этими областями посредством выпадающего списка имен областей в верхней части окна Output. Для перечисления областей служит объект *OutputWindowPanes:*

```
Sub EnumOutputWindowPanes()
    Dim window As EnvDTE.Window
    Dim outputWindow As EnvDTE.OutputWindow
    Dim outputWindowPanes As EnvDTE.OutputWindowPanes
    Dim outputWindowPane As EnvDTE.OutputWindowPane
    ' Получить объект OutputWindow,
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindOutput)
    outputWindow = CType(window.Object, EnvDTE.OutputWindow)
    ' Получить объект OutputWindowPanes.
    outputWindowPanes = outputWindow.OutputWindowPanes
    ' Выбрать каждый OutputWindowPane.
    For Each outputWindowPane In outputWindowPanes
        MsgBox(outputWindowPane.Name)
    Next
End Sub
```

Объект *OutputWindowPanes* можно также применять для создания новых областей. Метод *Add* имеет единственный аргумент — имя вновь создаваемого окна:

```
Sub CreateOutputWindowPane()
    Dim window As EnvDTE.Window
    Dim outputWindow As EnvDTE.OutputWindow
    Dim outputWindowPanes As EnvDTE.OutputWindowPanes
    Dim outputWindowPane As EnvDTE.OutputWindowPane
    ' Получить объект OutputWindow.
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindOutput)
    outputWindow = CType(window.Object, EnvDTE.OutputWindow)
    ' Получить объект OutputWindowPanes.
    outputWindowPanes = outputWindow.OutputWindowPanes
    ' Добавляем новую область:
    outputWindowPane = outputWindowPanes.Add("My New Pane")
End Sub
```

Этот макрос создает новую область — My New Pane, которую могут заполнять текстом ваши надстройки или макросы. Текст добавляет метод *OutputWindowPane.OutputString*, который принимает строку, добавляемую в конец текста соответствующей области. Между строками, добавляемыми в окно Output, автоматически не вставляется код перевода строки, так что, если вам нужно разделять строки, добавьте код перевода строки сами. Следующий макрос выводит содержимое папки, в которой хранится открытое в данный момент решение; после вывода пути каждого файла добавляется символ перевода строки (ASCII-код 13):

```
Sub DisplaySolutionDirectory()
    Dim files As String()
    Dim file As String
    Dim directoryOutputWindowPane As OutputWindowPane
    Dim fullName As String
    Dim outputWindow As OutputWindow
    outputWindow = DTE.Windows.Item(Constants.vsWindowKindOutput).Object

    ' Находим папку с проектом и все содержащиеся в ней файлы:
    fullName = System.IO.Path.GetDirectoryName(DTE.Solution.FullName)
    files = System.IO.Directory.GetFiles(fullName)

    ' Пытаемся найти область "Solution Directory"; если ее нет, создаем:
    With outputWindow.OutputWindowPanes
        Try
            directoryOutputWindowPane = .Item("SolutionDirectory")
            ' Показываем область:
            directoryOutputWindowPane.Activate()
        Catch
            directoryOutputWindowPane = .Add("Solution Directory")
        End Try
    End With
    ' Очищаем область:
    directoryOutputWindowPane.Clear()
    For Each file In files
        ' Выводим пути с переводом строки после каждого из них.
        directoryOutputWindowPane.OutputString(file + Chr(13))
    Next
End Sub
```

Этот макрос демонстрирует применение нескольких методов и свойств объекта *OutputWindowPane*. Метод *Activate* показывает нужную область в пользовательском интерфейсе; он имитирует выбор этой области из выпадающего списка окна Output. *OutputString* выводит строку в область, а *Clear* удаляет из области весь текст. Еще одно свойство, *TextDocument*, не представленное в макросе, заслуживает особого внимания. Оно возвращает объект *EnvDTE.TextDocument* для неизменяемой области — вы можете получать из нее информацию, но не изменять ее (подробнее см. главу 11).

# Окно дизайнера форм

Визуальное создание пользовательского интерфейса для программ .NET Framework осуществляется при помощи дизайнера форм Visual Studio .NET. При этом вы перетаскиваете на форму нужные элементы управления из Toolbox и устанавливаете их свойства в окне Properties. Дизайнер форм построен из компонентов .NET и для отображения и создания форм использует сборку *System.Windows.Forms*. Поскольку применяется эта сборка, программирование формы в дизайнере аналогично программному управлению ею в период выполнения.

## Интерфейс *IDesignerHost*

Окно дизайнера форм, как и многие другие окна, имеет объектную модель. Иерархия объектов, возвращаемая свойством *Window.Object* окна дизайнера форм, имеет тип *System.ComponentModel.Design.IDesignerHost*. Чтобы просмотреть и изменить форму в дизайнере, нужно получить соответствующий ей объект *System.Windows.Forms.Control*. Это можно сделать, получив свойство *IDesignerHost.RootComponent* и приведя возвращенный объект к *Forms.Control*.

```
System.Windows.Forms.Control control;
System.ComponentModel.Design.IDesignerHost designerHost;
designerHost =(System.ComponentModel.Design.IDesignerHost)
    applicationObject.ActiveWindow.Object;
control = (System.Windows.Forms.Control)designerHost.RootComponent;
```

**Примечание** Класс *System.Windows.Forms.Form* происходит от класса *System.Windows.Forms.Control*. Показанный код демонстрирует работу с пользовательским элементом управления, но этот же код без изменений можно задействовать для программирования Windows Form.

Объект *System.Windows.Forms.Control* позволяет подключать события, чтобы определять моменты изменения формы, получать и изменять ее размеры, а также размещать, удалять и изменять элементы управления на форме.

## Маршалинг

Если вы попытаетесь задействовать интерфейс *IDesignerHost* в макросе, будет сгенерировано исключение *System.Runtime.Remoting.RemotingException*. Это объясняется тем, что элементы пользовательского интерфейса, представленные объектом

*System.Windows.Control*, недоступны через границы процессов. Не забывайте, что Macros IDE и Visual Studio .NET выполняются в разных процессах. Поэтому объектная модель дизайнера может применяться только в надстройках, но не в макросах.

## Добавление элемента к форме

Получив интерфейс *IDesignerHost* дизайнера форм, вы можете легко добавить к форме элементы управления. Для этого нужен объект *System.Type*, описывающий элемент управления. Этот объект позволяет получить статический метод *Type.GetType*, который передает полное имя класса элемента. Так, следующий код добавляет к форме окно со списком:

```
System.Type type;
type = System.Type.GetType("System.Windows.Forms.ListView");
```

Затем можно передать этот объект типа *Type* методу *IDesignerHost.CreateComponent* для создания элемента управления. Этот метод имеет две перегруженные версии, первая из которых принимает два параметра: объект *Type,* который мы только что получили, и имя элемента управления, который мы хотим создать. Это имя должно быть уникальным в классе формы, иначе возникнет конфликт имен, и будет сгенерировано исключение. Вторая версия метода принимает как аргумент лишь объект *Type,* а дизайнер форм просматривает код формы в поисках уникального имени. Обе версии генерируют соответствующий код для создания экземпляра элемента управления указанного типа. Следующий код создает окно со списком с именем *listViewControl*:

```
System.ComponentModel.IComponent component;
component = designerHost.CreateComponent(type, "listViewControl");
```

Если вы добавите этот код к своей надстройке и выполните его, вы не увидите элемент на форме, Дело в том, что при создании экземпляра элемента управления он не «удочеряется» формой и не добавляется к набору *Controls* формы. Чтобы добавить элемент в набор *Controls* формы, нужно присвоить свойству *Parent* элемента форму, которая должна содержать этот элемент. Свойство *Parent* (да и другие свойства элемента управления) устанавливается с применением объекта *System.ComponentModel.PropertyDescriptorCollection*. Этот объект содержит набор свойств элемента управления, и, когда этим свойствам присваиваются значения, в классе формы генерируется код, соответствующий устанавливаемым свойствам. Свойство *Parent* можно установить так:

```
System.ComponentModel.PropertyDescriptorCollection props;
System.ComponentModel.PropertyDescriptor propParent;
// Получаем свойства элемента listViewControl:
props = System.ComponentModel.TypeDescriptor.GetProperties(component);
// Получаем свойство Parent.
propParent = props["Parent"];
// Присваиваем свойству Parent форму;
propParent.SetValue(newControl, designerHost.RootComponent);
```

## Поиск существующих элементов управления

Вы знаете, как создавать новые элементы управления и помещать их на форму. Но как найти существующий элемент на форме? Как мы уже говорили, свойство *IDesignerHost.RootComponent* возвращает объект, который можно привести к *System.Windows.Forms.Control*. Имея этот объект, вы можете вызывать методы и свойства, как в период выполнения. Например, этот код просматривает список элементов управления, содержащихся в объекте *System.Windows.Forms.Control*:

```
System.ComponentModel.Design.IDesignerHost designer;
System.Windows.Forms.Control rootControl;

// Здесь устанавливаем переменную для дизайнера из свойства Window.Object.

rootControl = (System.Windows.Forms.Control)designer.RootComponent
foreach (System.Windows.Forms.Control control in rootControl.Controls)
{
    // Получаем нужную информацию об элементе управления.
}
```

Объект *PropertyDescriptorCollection* можно задействовать для получения свойств элементов управления аналогично установке свойств формы, но при этом вызывается метод *PropertyDescriptor.GetValue*:

```
System.ComponentModel.Design.IDesignerHost designer;
System.ComponentModel.PropertyDescriptor propControls;
System.ComponentModel.PropertyDescriptorCollection props;
System.ComponentModel.IComponent component;
System.Windows.Forms.Form form;
System.Drawing.Size size;
designer = (System.ComponentModel.Design.IDesignerHost)
   applicationObject.ActiveWindow.Object;
component = designer.RootComponent;

// Получаем свойство Size, используя дизайнер форм:
props = System.ComponentModel.TypeDescriptor.GetProperties(component);
propControls = props["Size"];
size = (System.Drawing.Size)propControls.GetValue(component);

// Получаем свойство Size непосредственно из формы:
form = (System.Windows.Forms.Form)component;
size = form.Size;
```

## Пример компоновки формы

В Visual Basic 6 и более ранних версиях было окно Form Layout, в котором проектируемая форма показывалась так, как она должна выглядеть на рабочем столе компьютера. В Visual Studio .NET такой функции нет, но вы можете добавить ее, используя модель автоматизации дизайнера форм. Исходный код этого примера, FormView, вы можете найти среди примеров к этой книге.

При старте эта надстройка создает окно инструментов с виртуальным монитором, представляющим монитор компьютера. Экран этого виртуального м< >нитора соответствует разрешению вашего реального монитора. (Если к компьютеру подключено несколько мониторов, используется разрешение основного.) После подключения к событию *WindowActivated* надстройка ждет, когда активизируется окно дизайнера форм, находит элементы управления на форме и рисует эту форму вместе с элементами управления на виртуальном экране. Например, если вы создаете форму с календарем и кнопкой (рис. 10-4), окно дизайнера форм будет выглядеть, как на рис. 10-5.




Рис. 10-4. *Windows Form с календарем и кнопкой*

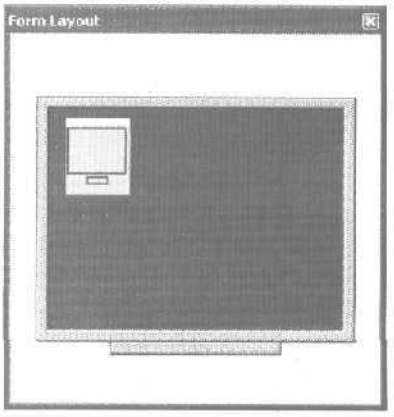



Рис. 10-5. *Окно Form Layout с изображением формы, показанной на рис, 10-4*

## Создание нестандартных окон инструментов

Как вы знаете, большинство окон Visual Studio .NET имеет объектную модель, которую можно использовать для программного управления их содержимым. Однако иногда нужно отображать данные в такой форме, которую стандартные окна инструментов не поддерживают. Чтобы ваши надстройки могли отображать данные подходящим для них способом, объектная модель Visual Studio .NET допускает создание нестандартных окон инструментов.

Все, что нужно для создания инструментального окна, — это элемент управления ActiveX и надстройка, вызывающая метод *Windows.CreateToolWindow*.

```
public EnvDTE.Window CreateToolWindow(EnvDTE.AddIn AddInInst,
    string ProgID,
    string Caption,
    string GuidPosition,
    ref object DocObj)
```

Этот метод возвращает объект *Window,* который ведет себя подобно любому окну инструментов, созданному Visual Studio .NET. Вот аргументы этого метода.

- ***AddInInst*** Объект, представляющий надстройку, который является инициатором окна инструментов. Когда эта надстройка выгружается, все связанные с ней инструментальные окна автоматически закрываются, а ActiveX выгружается.
- ***ProgID*** Идентификатор ActiveX, располагающегося во вновь создаваемом инструментальном окне.
- ***Caption*** Текст, отображаемый в строке заголовка нового инструментального окна.
- ***GuidPosition*** GUID в строковом формате. Как вы знаете, метод *WindowsItem* может индексироваться GUID, который уникально идентифицирует определенное окно. Этот параметр определяет GUID вашего окна инструментов и GUID, передаваемый методу *WindowsJtem,* У всех инструментальных окон должны быть разные GUID; если вы вызываете *CreateToolWindou* несколько раз, для каждого окна нужно указывать свой GUID.
- **DocObject** Большинство элементов управления ActiveX имеет программируемый объект в форме COM-интерфейса *IDispatch,* который при использовании .NET Framework отображается на *System.Object.* Программируемый объект элемента ActiveX возвращается вызывающей стороне через этот параметр, что позволяет программировать элемент управления, как и в случае любого другого окна инструментов. Вы можете получить программируемый объект элемента управления ActiveX и через свойство *Object* объекта *Window* окна инструментов, созданного методом *CreateToolWindow.*

Проект надстройки VSMediaPlayer из примеров к этой книге демонстрирует применение метода *CreateToolWindow.* В этом примере окно инструментов создается с использованием Windows Media Player в качестве ActiveX, а затем проигрывается звуковой файл с применением программируемого объекта этого элемента управления. Код для создания этого инструментального окна выглядит так:

```
void CreateMediaPlayerToolWindow()
{
    EnvDTE.Windows windows;
    EnvDTE.Window mediaPlayerWindow;
    object controlObject = null;
    string mediaPlayerProgID = "MediaPlayer.MediaPlayer";
    string toolWindowCaption = "Windows Media Player";
    string toolWindowGuid = "{AB5E549E-F823-44BB-8161-BE2BD5D698D8}";

    // Создание и отображение инструментального окна,
    // содержащего Windows Media Player:
    windows = applicationObject.Windows;
```

```
    mediaPlayerWindow = windows.CreateToolWindow(addInInstance,
                                                 mediaPlayerProgID,
                                                 toolWindowCaption,
                                                 toolWindowGuid,
                                                 ref controlObject);
    mediaPlayerWindow.Visible = true;

    // Проигрываем системный звук "Tada":
    string mediaFile = System.Environment.GetFolderPath(
        System.Environment.SpecialFolder.System);
    mediaFile += "\\..\\media\\tada.wav";
    MediaPlayer.IMediaPlayer2 mediaPlayer =
         (MediaPlayer.IMediaPlayer2)controlObject;
    mediaPlayer.AutoStart = true;
    mediaPlayer.FileName = mediaFile;
}
```

В этой надстройке-примере метод *CreateMediaPlayerToolWindow* вызывается дважды: в методе *OnConnection* и в методе *OnStartupComplete*. Это объясняется способом загрузки надстройки в Visual Studio .NET. Если для надстройки указано, что она загружается при запуске, она начинает загружаться при запуске Visual Studio .NET. Этот процесс загрузки включает вызов метода *OnConnection*. Но метод *OnConnection* вызывается перед созданием и выводом основного окна Visual Studio .NET. Если вы обращаетесь к методу *CreateToolWindow* из *OnConnection* до вывода основного окна, окно инструментов не будет создано, так как для создания ActiveX нужно, чтобы было видимо родительское окно. Проверить, создано ли основное окно, позволяет аргумент *connectMode*, переданный методу *OnConnection*. Если он равен *ext_cm_AfterStartup* надстройка была загружена диспетчером надстроек или иначе, но не при запуске Visual Studio .NET. Значит, инструментальное окно можно отобразить, когда надстройка загружена реализацией метода *OnConnection* вроде этой:

```
public void OnConnection(object application,
    Extensibility.ext_ConnectMode connectMode, object addInInst,
    ref System.Array custom)
{
    applicationObject = (_DTE)application;
    addInInstance = (AddIn)addInInst;

    // Если надстройка загружена с помощью Add-in Manager,
    // создаем и показываем окно инструментов:
    if(connectMode == Extensibility.ext_ConnectMode.ext_cm_AfterStartup)
    {
        CreateMediaPlayerToolWindow();
    }
}
```

Если флажок загрузки при запуске установлен и вы хотите вывести окно инструментов, когда надстройка загружена, вы можете создать окно в методе *OnStartupComplete*. Он вызывается по завершении инициализации Visual Studio .NET, которая включает создание и вывод основного окна:

```
public void OnStartupComplete(ref System.Array custom)
{
    // Если надстройка загружена при запуске,
    // создаем и выводим инструментальное окно:
    CreateMediaPlayerToolWindow();
}
```

Метод *CreateToolWindow*— мощное средство создания пользовательских интерфейсов надстроек, но с ним связана одна сложность. Как уже говорилось, при создании окна инструментов создается экземпляр элемента управления ActiveX, хостом которого является новое окно; но ActiveX нельзя создать с помощью .NET Framework. Чтобы задействовать пользовательский элемент управления .NET в инструментальном окне, нужен согласователь.

## Согласователи

Чтобы разрешить окну инструментов управлять пользовательскими элементами управления .NET Framework, можно применить специальный элемент управления ActiveX — согласователь (shim control). Согласователь — это ActiveX, написанный на неуправляемом языке (например, на Visual C++), который создает экземпляр CLR .NET. Небольшой код позволит задействовать этот экземпляр CLR для создания пользовательского элемента управления, отображаемого в окне инструментов и связать его с согласователем. Вы можете добавить новый пользовательский элемент управления к существующему проекту, выбрав Project | Add User Control, а можете создать новый проект Windows Control Library, в котором будет создан новый элемент.

Среди примеров к книге есть реализация согласователя (VSUserControlHost), а также пример его применения (CSHostedControl). Создание и размещение экземпляра пользовательского элемента в окне инструментов производится в два этапа. Сначала в инструментальное окно помещается согласователь, для чего вызывается метод *CreateToolWindow*, которому передается ProgID согласователя, как показано в этом фрагменте, взятого из примера CSHostedControl:

```
toolWindow = applicationObject.Windows.CreateToolWindow(addInInstance,
    "VSUserControlHost.VSUserControlHostCtl", "C# Hosted Control",
    "{C4E8F504-E3FB-4828-82F4-DDD1CAE13D39}", ref ObJ);
```

Затем нужно указать согласователю, откуда брать элемент управления и как называется реализующий его класс. Это позволяет сделать интерфейс *VSUserControlHostLib.IVSUserControlHostCtl*, который можно получить путем приведения типа программируемого объекта согласователя:

```
shimControl =
    (VSUserControlHostLib.IVSUserControlHostCtl)toolWindow.Object;
```

Интерфейс *IVSUserControlHostCtl* имеет четыре метода, вызывая которые, можно передать расположение элемента управления, имя класса и другую информацию:

```
System.Object HostUserControl(System.String Assembly, System.String Class);
System.Object HostUserControl2(System.Int32 HWnd);
```

Каждый из этих методов возвращает программируемый объект .NET-элемента, управляемого согласователем. Им передаются следующие аргументы.

- ***Assembly*** Местоположение сборки, содержащей элемент управления, которое можно указать в одном из трех форматов. Первый — полный путь к сборке, например *C:\Assembly.dll*. Второй — URL сборки, расположенной на Web-сервере, например *http://localhost/Assembly.dll*Третий формат — полное имя сборки, расположенной в GAC, например *System.Windows.Forms,Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089,Custom=null* для сборки с пространством имен *System.Windows.Forms*.
- ***Class*** Полное имя класса, реализующего элемент управления. Если в качестве местоположения сборки задано полное имя сборки *System.Windows.Form*,а имя класса — *System.Windows.Forms.Button*создается объект ***Button*** Windows Forms и помещается в инструментальное окно.
- ***HWnd*** Описатель элемента управления .NET, которым будет управлять согласователь. Такая форма определения элемента управления удобна, когда элемент уже создан или его код расположен в той же сборке, что и код надстройки, вызывающей *CreateToolWindow*,так как вы можете создать элемент и передать его описатель окна, а не информацию о местоположении сборки и имя класса.

Разработчики надстроек часто добавляют класс пользовательского элемента управления в надстройку и затем помещают этот элемент в окно инструментов. Сделать это с применением согласователя можно двумя способами. Во-первых, определить местоположение сборки, реализующей надстройку и передать это значение в качестве первого параметра методу *HostUserControl*:

```
VSUserControlHostLib.IVSUserControlHostCtl shimControl;
string assemblyPath;
EnvDTE.Window toolWindow;
object obj = null;
toolWindow = applicationObject.Windows.CreateToolWindow(addInInstance,
    "VSUserControlHost.VSUserControlHostCtl", "Hosted Control",
    "{A71654EC-A72E-40cf-9CD6-63FA3C52C307}", ref obj);
toolWindow.Visible = true;
shimControl =
    (VSUserControlHostLib.IVSUserControlHostCtl)toolWindow.Object;
assemblyPath = System.Reflection.Assembly.GetExecutingAssembly().Location;
shimControl.HostUserControl(assemblyPath, "MyAddin1.UserControl1");
```

Во-вторых, напрямую создать экземпляр элемента управления и передать его описатель окна методу *HostUserControl2*:

```
VSUserControlHostLib.IVSUserControlHostCtl shimControl;
string assemblyPath;
EnvDTE.Window toolWindow;
object obj = null;
toolWindow = applicationObject.Windows.CreateToolWindow(addInInstance,
    "VSUserControlHost.VSUserControlHostCtl", "Hosted Control",
    "{A71654EC-A72E-40cf-9CD6-63FA3C52C307}", ref obj);
toolWindow.Visible = true;
```

```
shimControl =
    (VSUserControlHostLib.IVSUserControlHostCtl)toolWindow.Object;
assemblyPath = System.Reflection.Assembly.GetExecutingAssembly().Location;
UserControl1 uc = new UserControl1();
shimControl.HostUserControl3(uc.Handle.ToInt32());
```

### Практикум: применение Web-сервера с пользовательским элементом для окна инструментов

Первый аргумент метода *HostUserControl* согласователя — это полный путь к пользовательскому элементу управления .NET Framework на диске, полное имя сборки в GAC компьютера пли URL элемента на Web-сервере. Поместив пользовательский элемент управления для окна инструментов на Web* сервере, вы сможете изменять этот элемент, расширяя его функциональность и исправляя ошибки, но пользователю нужно будет подключаться к Интернету, чтобы скачивать элемент управления. Если пользователь не подключен к Интернету или Web-сервер не работает и момент запроса, но элемент управления был предварительно скачан, он загружается из кэша загруженных файлов компьютера.

В качестве тестового сервера можно задействовать Microsoft Internet Information Services (IIS) на своем компьютере. Чтобы настроить свой проект на работу с пользовательским элементом управления на этом Web-сервере, щелкните правой кнопкой проект пользовательского элемента управления, который вы хотите поместить в окно инструментов, и выберите Properties, чтобы вывести окно Property Pages для этого проекта. Выберите узел Configuration Properties | Build, измените значение в выпадающем списке Configuration на Ail Configurations, а затем введите путь к каталогу своего Web-сервера (обычно это C:\Inetpub\wwwroot) в поле Output Path. Чтобы согласователь загружал ваш элемент с Web-сервера, измените путь к элементу на http://localhost/*assemblyname*.dll.

При загрузке элемента с Web-сервера нужно кое о чем помнить. Во-первых, все сборки, на которые ссылается ваш элемент управления (кроме расположенных в GAC компьютера, на котором загружается элемент), должны быть в тойже папке Web-сервера, что и ваш элемент, чтобы .NET Framework могла разрешить эти ссылки. Во-вторых, вы сможете программировать объект элемента управления только с применением метода *TypeInvokeMember*. Это связано с тем, как .NET Framework разрешает типы при обращении к методам и свойствам. В-третьих, прежде чем передавать свою надстройку пользователям, вы должны скопировать пользовательский элемент управления на сервер, к которому пользователи имеют доступ, и изменить URL, передаваемый методу *HostUserControl*, так чтобы он указывал на этот сервер.

## Настройка вкладок в нестандартном инструментальном окне

Когда два или более окон представлены перекрывающимися вкладками, на них показаны картинки, чтобы пользователь мог их легко различить (рис. 10-6).




**Рис. 10-6.** *Картинки на вкладках окон Macro Explorer, Solution Explorer и Properties*

Картинки на вкладке окна инструментов, создаваемого надстройкой, устанавливает метод *Window.SetTabPicture*. Он принимает в качестве аргумента COM-тип *IPictureDisp*, которому в .NET Framework соответствует тип *System.Object*. Для создания объекта *IPictureDisp* можно применять описанную ранее методику: вызывать метод *OleLoadPictureFile* и передавать возвращенный им объект *IPictureDisp* методу *SetTabPicture*.

Картинка, помещаемая на вкладку инструментального окна, должна быть размером 16x16 пикселов и с глубиной цвета 16. Прозрачные пикселы должны иметь RGB-значение 0,254,0. Формат этих картинок такой же, как для кнопок панели команд (см. главу 7); это могут быть одни и те же картинки, используемые в обоих случаях.

Метод *Window.SetTabPicture* можно вызывать только для окна инструментов, созданного методом *Windows.CreateToolWindow*. Окна, определенные Visual Studio .NET, имеют свои стандартные картинки, и, если попытаться их изменить, возникнет исключение. Устанавливать картинку для собственного инструментального окна нужно до присваивания свойству *Visible* окна значения *true*, иначе картинка не отобразится в первый момент. Если картинка для вкладки не установлена, Visual Studio покажет картинку по умолчанию — свой логотип.

## Установка объекта *Selection*

При переходе от одного окна Visual Studio .NET к другому вы видите, что изменяется окно Properties, отображая информацию, относящуюся к новому окну. Так, если вы выберете файл в Solution Explorer, становится доступным набор таких свойств, как путь к файлу, время его изменения и т. д. Для собственных окон инструментов тоже можно задать отображение свойств в окне Properties. Элементы, отображаемые в окне Properties, задает метод *Window.SetSelectionContainer*, принимающий массив элементов типа *System.Object*. Когда окно, для которого был выполнен этот метод, становится активным, эти элементы отображаются в окне Properties. Пример VSMediaPlayerAdv (расширенный пример VSMediaPlayer) показывает набор свойств в окне Properties, вызывая метод *SetSelectionContainer* с объектом Windows Media Player, возвращаемым параметром *DocObj* метода *CreateToolWindow*. Вот как это делается:

```
object []propertiesWindowObjects = {mediaPlayer};
mediaPlayerWindow.SetSelectionContainer(ref propertiesWindowObjects);
```

Вы можете вызывать метод *SetSelectionContainer* только для собственных инструментальных окон. Попытка сделать это, скажем, для объекта *Window* окна Solution Explorer вызовет исключение.

# Диалоговое окно Options

Разработчики — народ привередливый. Они хотят, чтобы Visual Studio .NET во всем потакала их привычкам; если хоть один параметр настроен не так, работа становится непродуктивной. В диалоговом окне Options можно задать массу параметров: от числа пробелов, вставляемых при нажатии Tab, до отображения строки состояния внизу основного окна Visual Studio .NET.

## Изменение существующих параметров

Многие параметры диалогового окна Options управляются моделью автоматизации через объекты *Properties* и *Property,* Чтобы получить набор *Properties,* нужно сначала вычислить категорию и подкатегорию изменяемых параметров. В левой части этого диалогового окна расположено дерево, которое редко имеет глубину более двух уровней. Узлы верхнего уровня, такие как Environment, Source Control и Text Editor, — это категории параметров, которыми вы можете манипулировать. Каждая категория имеет группу связанных страниц Options, содержащих элементы управления, которые служат для настройки среды разработки. Вложенные узлы представляют подкатегории диалогового окна Options: когда вы выбираете один из этих узлов, правая часть окна Options изменяется, и в ней отображаются параметры, соответствующие выбранной категории и подкатегории. Категории и подкатегории, указываемые для получения набора *Properties,* основаны на категориях и подкатегориях, отображаемых окне Options, но их имена слегка отличаются. Для получения списка категорий и подкатегорий служит редактор реестра. Сначала следует найти элемент из окна Options, который вы хотите изменить. В нашем примере мы будем изменять размер отступа в редакторе кода Visual Basic .NET; этот параметр находится в категории Text Editor и подкатегории Basic.

**Примечание** Категория Text Editor отличается от других категорий окна Options тем, что имеет три уровня (третий уровень — подкатегория). Однако в объектной модели подкатегории General и Tabs объединены в одну, которая имеет имя используемого языка программирования.

Запустив regedit.exe, откройте узел *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\AutomationProperties.* В него вложен список категорий свойств, доступных из макросов и надстроек. Нам нужна категория Text Editor. Раздел, название которого больше всего похоже на эту категорию, — TextEditor (без пробела). Раскрыв этот элемент в редакторе реестра, вы увидите список подкатегорий. Одна из этих них, Basic, соответствует подкатегории, отображаемой в окне Tools Options dialog box, она-то нам и нужна.

Имея названия категории и подкатегории для модели автоматизации (TextEditor и Basic), мы можем использовать их в свойстве *DTE.Properties* для получения набора *Properties*:

```
Sub GetVBTextEditorProperties()
    Dim properties As Properties
    properties = DTE.Properties("TextEditor", "Basic")
End Sub
```

Последний шаг для получения объекта *Property* — вызов метода *Item* набора *Properties*. Этот метод принимает в качестве параметра имя свойства, но это имя хранится только в объектной модели. Объект *Properties* — это набор, и поэтому можно перечислить все содержащиеся в нем объекты и их имена. Узнать имена, которые передаются методу *PropertiesItem*, позволяет следующий макрос. Он проходит по всем категориям и подкатегориям, перечисленным в реестре, и затем использует перечислитель набора *Properties* для поиска имен объектов *Property*, содержащихся в этом наборе. Имя каждой категории, подкатегории и свойства помещается в текстовый файл, создаваемый этим макросом:

```
Sub WalkPropertyNames()
    Dim categoryName As String
    Dim key As Microsoft.Win32.RegistryKey
    Dim newDocument As Document
    Dim selection As TextSelection
    ' Открываем новый документ для хранения информации.
    newDocument = DTE.ItemOperations.NewFile("General\Text File").Document
    selection = newDocument.Selection
    ' Открываем раздел реестра, содержащий список категорий:
    key = Microsoft.Win32.Registry.LocalMachine
    key = key.OpenSubKey( _
        "SOFTWARE\Microsoft\VisualStudio\7.1\AutomationProperties")
    ' Перечисляем категории:
    For Each categoryName In key.GetSubKeyNames()
        Dim subcategoryName As String
        selection.Insert(categoryName + vbLf)
        ' Перечисляем подкатегории:
        For Each subcategoryName In _
            key.OpenSubKey(categoryName).GetSubKeyNames()
            Dim properties As Properties
            Dim prop As [Property]
            selection.Insert("  " + subcategoryName + vbLf)
            Try
                ' Перечисляем свойства:
                properties = DTE.Properties(categoryName, subcategoryName)
                For Each prop In properties
                    selection.Insert("    " + prop.Name + vbLf)
                Next
            Catch
            End Try
        Next
    Next
End Sub
```

Просматривая результаты работы этого макроса, мы находим категорию TextEditor и подкатегорию Basic, а затем что-нибудь похожее на Tab Size. Больше всего подходит TabSize. Используя это имя, можно получить объект *Property* для параметра Tab Size текстового редактора Visual Basic .NET:

```
Sub GetVBTabSizeProperty()
    Dim properties As Properties
```

```
    Dim prop As [Property]
    properties = DTE.Properties("TextEditor", "Basic")
    prop = properties.Item("TabSize")
End Sub
```

Осталось лишь получить значение этого свойства из свойства *Property.Value*:

```
Sub GetVBTabSize()
    Dim properties As Properties
    properties = DTE.Properties("TextEditor", "Basic")
    MsgBox(properties.Item("TabSize").Value)
End Sub
```

Макрос выводит значение *4* — именно то, что мы видим в окне Tools Options для параметра Tab Size в подкатегории Basic категории Text Editor. Устанавливается это значение аналогично его чтению:

```
Sub SetVBTabSize()
    Dim properties As Properties
    properties = DTE.Properties("TextEditor", "Basic")
    properties.Item("TabSize").Value = 4
End Sub
```

Изменяя категорию и подкатегорию, передаваемые свойству *DTE.Properties*, и найдя нужное имя в списке имен свойств, сгенерированном макросом *WalkPropertyNames*, вы можете изменить многие параметры, отображаемые в диалоговом окне Tools Options.

### Что скрывается за /s?

При работе с объектной моделью Visual Studio .NET для сравнения объектов можно задействовать оператор *Is* языка Visual Basic .NET или метод *Object.Equals* .NET Framework. Но оператор *Is* и метод *Equals* не всегда возвращают то, что от них ожидают. Если запустить макрос:

```
Sub CompareWindowsObjects()
    Dim window1 As Window
    Dim window2 As Window
    window1 = DTE.Windows.Item(Constants.vsWindowKindTaskList)
    window2 = DTE.Windows.Item(Constants.vsWindowKindTaskList)
    MsgBox(window1 Is window2)
End Sub
```

будет выведено информационное окно со значением *True*. Когда вы запрашиваете объект *Window*, объектная модель проверяет, существует ли объект *Window* для указанного окна; если нет, создается новый объект *Window*, возвращаемый вызывающей программе. Если объект *Window* уже есть, он возвращается вызывающей программе. Это оптимизирует производительность и использование памяти, так как объект не каждый раз создается (что требует памяти) и инициализируется (что требует процессорного времени). Но если вы выполните такой код:

```
Sub ComparePropertyObjects()
    Dim props1 As Properties
```

```
    Dim props2 As Properties
    props1 = DTE.Properties("Environment", "General")
    props2 = DTE.Properties("Environment", "General")
    MsgBox(props1 Is props2)
End Sub
```

в информационном окне появится *False,* поскольку набор *Properties* создается при каждом обращении к свойству *DTE.Properties,* чтобы оно предоставляло актуальную информацию,

Многократные обращения к свойству *DTEProperties* могут привести к перерасходу памяти. Допустим, вы вызываете свойство *DTE.Properties* в цикле; при каждом обращении создается новый набор *Properties,* который затем инициализируется и передается вызывающей программе. Память уходит на COM-объекты, создаваемые Visual Studio .NET, а если используете язык, поддерживаемый .NET Framework, создаются классы-оболочки .NET, обеспечивающие программный доступ к этим объектам. Вы можете увидеть практически неограниченный рост потребления памяти процессом vsmsvr.exe (в нем исполняется экземпляр .NET Framework и код макросов) на вкладке Процессы диспетчера задач Windows при выполнении такого макроса:

```
Sub RepeatedConstruct()
    Dim i As Long
    Dim props As Properties
    For i * 1 To Long.MaxValue
        props = DTE.Properties("Environment", "General")
    Next
End Sub
```

Цикл, выполняемый в этом макросе, не позволяет работать сборщику мусора, так как .NET Framework ориентирована на выполнение вашего кода, а не на поиск и освобождение неиспользуемых объектов. Чтобы ваша программа не тратила больше памяти, чем ей нужно для нормальной работы, убедитесь, что вы не создаете лишние объекты, используя оператор *Is* или метод *ObjectEquals* и проведя оптимизацию. Например, в макросе *RepeatedConstruct* можно вынести за пределы цикла обращение к *DTEProperties:*

```
Sub OptimizedRepeatedConstruct()
    Dim i As Long
    Dim props As Properties
    Dim showStatusbar As Boolean
    props = DTE.Properties("Environment", "General")
    For i = 1 To Long.MaxValue
        showStatusbar = props.Item("ShowStatusBar").Value
    Next
End Sub
```

Ненаучная оценка (основанная на показаниях диспетчера задач до и после запуска макроса) показывает, что простое перемещение одной строки за пределы цикла экономит почти 35 Мб памяти — ваши пользователи оценят это по достоинству.

## Создание нестандартных параметров

Вы можете не только анализировать и изменять существующие параметры, но и создавать собственные для своих надстроек. Чтобы создать для своей надстройки страницу в диалоговом окне Options, нужен ActiveX и некоторые изменения в системном реестре. Visual Studio .NET определяет, какие страницы окна Options нужны надстройке, по содержимому раздела реестра, описывающему надстройку. Когда пользователь открывает диалоговое окно Options, проверяются разделы реестра для всех надстроек, и, если существуют записи для страниц параметров нестандартных окон инструментов, создается экземпляр ActiveX, отображаемый в диалоговом окне Options.

Вы можете создать разделы реестра для страницы Options, модифицируя параметры реестра для проекта установки, который создается при запуске Add-in Wizard. Чтобы сделать нужные изменения, щелкните правой кнопкой проект установки своей надстройки в окне инструментов Solution Explorer и выберите View | Registry, Откроется окно редактирования реестра для проекта установки. Откройте узлы дерева в левой части этого окна и найдите раздел с ProgID надстройки. Если вы выбрали надстройку OptionsAddin, созданную мастером, и приняли для нее параметры по умолчанию за исключением того, что выбрали параметр Yes, Create A «Tools» Menu Item, левая панель редактора реестра должна выглядеть так (рис 10-7):

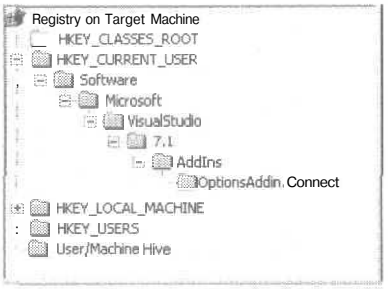



**Рис. 10-7.** Окно *редактора реестра для надстройки OptionsAddin*

Для объявления страницы Options нужно добавить три подраздела к этому выбранному разделу, каждый из которых является подразделом предыдущего. Первый раздел называется Options. Когда Visual Studio .NET отображает диалоговое окно Options, проверяется раздел реестра каждой надстройки, и, если существует подраздел Options, диалоговое окно Options знает, что у надстройки есть страница Options. Для создания раздела Options щелкните правой кнопкой раздел реестра с ProgID надстройки, выберите New | Key и введите **Options.** Следующие два раздела реестра соответствуют категории и подкатегории страницы Options. Для их создания повторите шаги, которые вы прошли для создания раздела Options, так чтобы категория была подразделом раздела Options, а подкатегория — подразделом раздела категории. Названия этих разделов могут быть произвольными, они будут отображаться в дереве диалогового окна Options, Если вы выберете для категории название Options Add-in, а для подкатегории — General, окно редактора реестра будет выглядеть так (рис. 10-8):

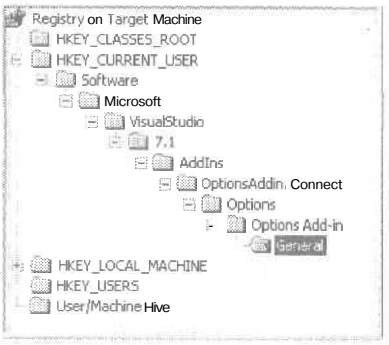



**Рис.** 10-8. *Редактор реестра для надстройки с разделами для страницы Options*

Последний этап в настройке реестра для страницы Options — объявление ProgID элемента управления ActiveX, расположенного в окне Options. Выберите узел подкатегории в левой панели редактора реестра, щелкните правой кнопкой в правой панели, выберите New | String Value и введите строку **Control.** Чтобы установить значение для этого элемента реестра, выделите Control и в окне Properties введите ProgID элемента управления ActiveX в поле Value. Мы пока не создали сам ActiveX, но если вы будете до конца выполнять этот пример, то в качестве ProgID введите *OptionsAddinPage.OptionsAddinPageCtl.*

### Создание ActiveX

Следующий шаг после настройки реестра — создание элемента управления ActiveX. Здесь все не так просто, Когда мы пытались создать окно инструментов методом *Windows.CreateToolWindow* и хотели задействовать в качестве пользовательского интерфейса этого окна пользовательский элемент управления .NET Framework, нам понадобилось создавать ActiveX-согласователь, так как .NET-элемент — это не ActiveX. Проблема с пользовательским элементом управления для страницы Tools Options, во-первых, в том, что это не ActiveX, а во-вторых, нельзя применить согласователь, так как вам нужно иметь возможность выполнить код, уведомляющий согласователь о том, где найти пользовательский элемент управления. Поскольку страницы Tools Options создает Visual Studio .NET, а не вы сами, как в случае инструментального окна, ваш код не сможет работать с согласователем.

По всем этим причинам пользовательский элемент управления не может быть задействован в качестве страницы Options. Остается применить для создания ActiveX неуправляемый язык, скажем, Visual C++. Для создания такого элемента управления нужно сначала добавить проект ATL к проекту вашей надстройки; в нашем примере мы назовем его OptionsAddinPage. Затем вы реализуете составной ActiveX, подсоединяете несколько дополнительных интерфейсов и пишете код привязки к странице. Мы создадим мастер страниц Options, OptionsPageWizard, который будет генерировать весь код, нужный для создания страниц Tools Options. Скомпоновав пример мастера и скопировав файл .vsz в папку Extensibility Projects, как вы это делали в главе 9 для примера WizardBuilder, вы сможете щелкнуть правой кнопкой узел решения в Solution Explorer, ввести имя проекта **OptionsAddin** и запустить мастер для создания начального кода. Единственный параметр, кото-

рый запросит мастер, — это ProgID надстройки, для которой предназначена страница Options,

### Интерфейс *IDTToolsOptionsPage*

Жизненный цикл страницы Options имеет три стадии: создание, взаимодействие и освобождение. Чтобы сообщить странице об этих стадиях, в ее элементе ActiveX можно реализовать интерфейс *IDTToolsOptionsPage*. Вот его сигнатура:

```
public Interface IDTToolsOptionsPage
{
    public void GetProperties(ref object PropertiesObject);
    public void OnAfterCreated(EnvDTE.DTE DTEObject);
    public void OnCancel();
    public void OnHelp();
    public void OnOK();
}
```

Когда пользователь первый раз выводит окно Options, Visual Studio .NET видит в реестре, что вы объявили страницу, и создает экземпляр вашего ActiveX. Если в этом элементе управления реализован интерфейс *IDTToolsOptionsPage,* методу *OnAfterCreated* передается объект *DTE,* соответствующий экземпляру Visual Studio .NET, который создает элемент управления, В реализации этого метода можно выполнять инициализационные действия, например читать реестр, и задействовать полученные данные для настройки пользовательского интерфейса элемента управления. Страница по умолчанию, генерируемая мастером OptionsPageWizard, делает именно это: читает некоторую запись из реестра и, если она равна некоторому значению, устанавливает флажок.

В окне Options три кнопки: OK, Cancel и Help. При щелчке OK вызывается метод *IDTToolsOptionsPage.OnOK* давая вашей странице шанс сохранить в реестре изменения, сделанные на странице. Код, генерируемый нашим мастером, сохраняет состояние флажка в реестре. Следует также освободить ресурсы в методе *OnOK,* так как страница Options закрывается. При щелчке кнопки Cancel вызывается метод *OnCancel.* Изменения, сделанные пользователем, сохранять не следует, однако освободить ресурсы нужно, поскольку страница закрывается, как и в случае нажатия кнопки OK. При щелчке кнопки Help вызывается метод *OnHelp,* позволяя вашей странице отобразить справочную информацию. В отличие от других кнопок Help не закрывает диалоговое окно, так что освобождать ресурсы в этом методе не надо.

Последний метод интерфейса *IDTToolsOptionsPage ~ GetProperties* — позволяет получить объекты *Properties* для параметров на странице.

#### Предоставление объекта *Property*

Как вы уже знаете, многие значения диалогового окна Options программируются через набор *Properties.* Вы тоже можете предоставить доступ к своей странице через набор *Properties* посредством метода *GetProperties.* Этот метод возвращает интерфейс *IDispatch,* оболочкой которого является набор *Properties.* По умолчанию мастер OptionsPageWizard создает одно свойство, *CheckBoxOption, с* интерфейсом *IDispatch.* Оно соответствует флажку на элементе управления. Используя пример

страницы OptionsAddinPage, к этому свойству можно обратиться посредством такого макроса:

```
Sub GetSetCustomOptions()
    Dim properties As EnvDTE.Properties
    Dim prop As EnvDTE.Property
    ' Получаем объект Properties нашей страницы:
    properties = DTE.Properties("Options Add-in", "General")
    prop = properties.Item("CheckBoxOption")
    ' Выводим значение параметра:
    MsgBox(prop.Value)
    ' Изменяем и выводим новое значение:
    prop.Value = True
    MsgBox(prop.Value)
End Sub
```

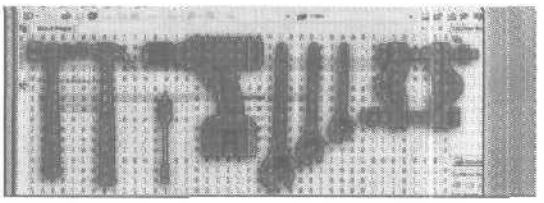
Когда вы хотите получить значение параметра в вашей надстройке, используйте код, подобный GetSetCustomOptions. Например, вы хотите добавить страницу Tools Options для окна инструментов VSMediaPlayer, чтобы позволить пользователю указывать файл, который будет проигрываться при активизации окна. Для получения пути к этому файлу можно написать такую строку:

```
DTE.Properties("VSMediaPlayer", "General").Item("MediaFile").Value
```

Значения, получаемые из объекта *Property*, нельзя кэшировать, так как пользователь может в любой момент их поменять; вы должны заново получать эти значения при активизации окна.

# Что дальше

Вы узнали, как программировать окна Visual Studio .NET. В следующей главе мы покажем, как взаимодействовать с одним конкретным окном — текстовым редактором.



ГЛАВА

# 11

# Объекты и события редактирования текста

Большая часть работы, выполняемой программистом за рабочий день (или рабочую ночь), связана с редактированием текста. Как много успешных проектов выросло из создания усовершенствованной программы Notepad (Блокнот)! И популярность этих редакторов растет прямо пропорционально количеству автоматизируемых ими задач и возможностям настройки параметров. Как вы узнали в главе 3, Visual Studio .NET имеет первоклассный редактор кода и объектную модель, позволяющую создать функции редактирования, которые были бы включены в редактор, если б в Microsoft за это отвечали бы вы.

## Окна редактора

Если вы собираетесь редактировать текст в IDE, вам нужен документ; если документ уже есть, вам требуется окно редактора. В Visual Studio .NET документы и окна редактора подобны Зигфриду и Рою: вместе они волшебники, а по отдельности просто пара парней в расшитых блестками костюмах с домашними тиграми.[1] (Ладно, признаем, что эта аналогия далека от совершенства.) Дело в том, что вы не можете иметь одно без другого и вам нужно знать хотя бы минимум об окнах редактора, даже если редактирование текста — ваша единственная цель, На рис. 11-1 — окна редактора, которые будут интересовать нас в этой главе.

[1] Известные американские иллюзионисты, выступающие с **белыми** голубоглазыми тиграми и прочими экзотическими животными. — *Прим. перев.*

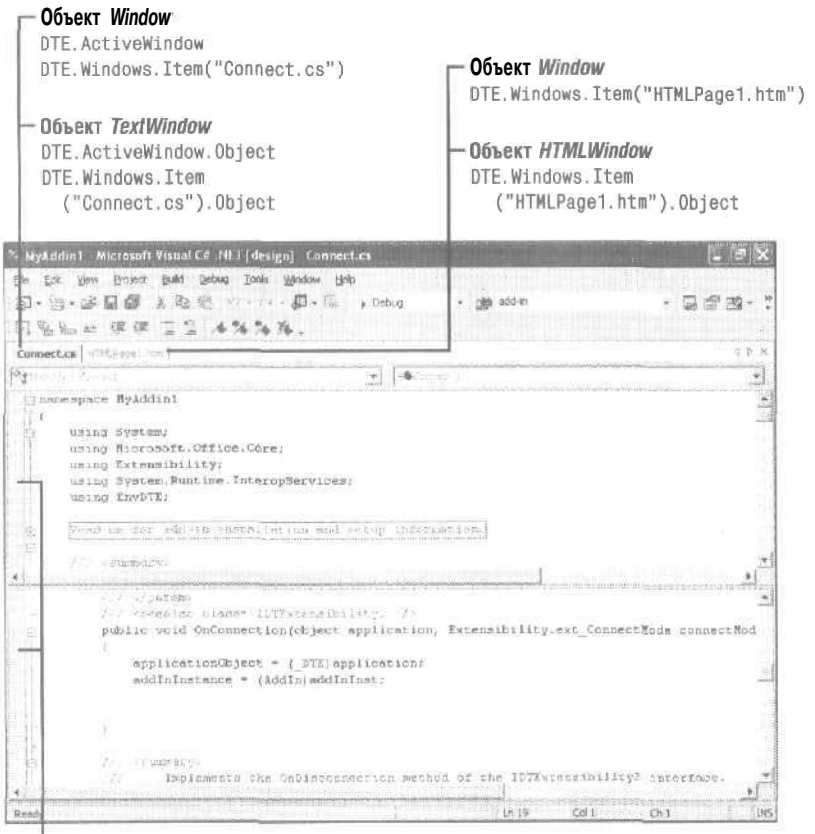



**Рис. 11-1.** *Окна редактора*

## Объект *Window*

Мы не будем много говорить об объекте *Window,* а просто сделаем небольшую остановку на пути к более специализированным окнам. Найти окно довольно просто: если вам нужно окно, имеющее фокус, свойство *DTE.ActiveWindow* возвратит его вам, если же другое и вы знаете его заголовок, используйте *DTE.Windows.Item(<заголовок>).* (На рис. 11-1 показан код для отыскания окон Connect.cs и HTMLPage1.htm.)

Следующим важным свойством для поиска других окон является *Object,* которое возвращает соответствующие объекты *TextWindow* или *HTMLWindow* для окон редактора. Если вы не знаете наверняка, какой тип содержит свойство *Object,* выполните проверку, используя ключевые слова *TypeOf...Is* (в Visual Basic) или *is* (в C#):

```
If TypeOf DTE.ActiveWindow.Object Is TextWindow Then
    H
End If
```

Конечно, если вы не выполните проверку и используете неправильный объект, то получите исключение CLR.

## Объекты *TextWindow* и *HTMLWindow*

Объекты *TextWindow* и *HTMLWindow* представляют окна редактора в IDE, Каждый тип предлагает небольшой набор свойств, предоставляющих доступ к специфичным для окна редактирования функциональным возможностям. В табл. 11-1 перечислены свойства объекта *TextWindow*. Отметим свойства *ActivePane* и *Panes,* которые предоставляют доступ к областям полученного окна редактора.

**Табл. 11-1. Свойства *TextWindow***

| Свойство | Описание |
|---|---|
| *ActivePane* | Возвращает объект *TextPane*, ассоциированный с активной областью окна. |
| *DTE* | Возвращает объект *DTE* верхнего уровня. |
| *Panes* | Возвращает набор *TextPanes*, содержащий области окна. |
| *Parent* | Возвращает родительский объект *Window*. |
| *Selection* | Возвращает объект *TextSelection* для активной области окна (эквивалентно *Parent.Selection*). |

*Объект HTMLWindow* в основном похож на объект *TextWindow*. Свойства объекта *HTMLWindow* таковы (табл. 11-2):

**Табл. 11-2. Свойства *HTMLWindow***

| Свойство | Описание |
|---|---|
| *CurrentTab* | Устанавливает или возвращает выбранную вкладку (HTML или Design). |
| *CurrentTabObject* | Возвращает объект *TextWindow,* если выбрана вкладка HTML или возвращает интерфейс *IHTMLDocument2*, если выбрана вкладка Design. |
| *DTE* | Возвращает объект *DTE* верхнего уровня. |
| *Parent* | Возвращает родительский объект *Window*. |

Свойство *CurrentTab* использует значения из перечисления *EnvDTE.vsHTMLTabs*: *vsHTMLTabsSource* для вкладки HTML и *vsHTMLTabsDesign* для вкладки Design. Свойство *CurrentTabObject* возвращает объект *TextWindow,* если выбрана вкладка HTML, поэтому мы и сказали выше, что объект *HTMLWindow* — это скрытый *TextWindow*. Однако, когда выбрана вкладка Design. *CurrentTabObject* возвращает интерфейс *mshtml.IHTMLDocument2*, предоставляющий доступ к объектной модели Dynamic HTML (DHTML) основного документа. Помните: представления на вкладках Design и HTML не синхронизированы: изменения, сделанные в одном представлении, не распространяются на другое, пока вы не переключите представления. С практической точки зрения это значит, что вы должны применять ссылки только на текущее представление.

**Примечание** Чтобы использовать пространство имен *mshtml*, вам понадобится основная связующая сборка (primary interop assembly) Microsoft.mshtml.dll. Вы найдете ее в папке Program Files\Microsoft.NET\PrimaryInterop Assemblies. Тот, кто пишет надстройки, может добавить ссылку на эту сборку из диалогового окна Add Reference; тот, кто пишет макросы, должен сначала скопировать файл DLL в папку PublicAssembliesVisual Studio .NET, чтобы получить доступ к сборке.

Как вы теперь знаете, чтобы определить, скрывается ли внутри обычного окна текстовое окно, надо сделать несколько шагов. Если вы мечтаете заиметь функцию, которая сделает все это за вас, вам повезло:

```
Function GetTextWindow(ByVal win As Window) As TextWindow
    ' Описание: Возвращает объект TextWindow для полученного окна,
    '           или Nothing, если текстового окна нет,

    Dim txtWin As TextWindow = Nothing

    ' Проверка для TextWindow
    If TypeOf win.Object Is TextWindow Then
        txtWin = win.Object

    ' В ином случае проверяем сначала HTMLWindow, а затем TextWindow
    ElseIf TypeOf win.Object Is HTMLWindow Then
        Dim htmlWin As HTMLWindow = win.Object

        If htmlWin.CurrentTab = vsHTMLTabs.vsHTMLTabsSource Then
            txtWin = htmlWin.CurrentTabObject
        End If
    End If

    Return txtWin
End Function
```

## Объект *TextPane*

Объект *TextPane* представляет область в окне редактора. Каждое окно редактора может быть разделено на две области, что позволяет одновременно видеть два фрагмента текстового файла. Вы можете разделить представление вручную, либо двойным щелчком полосы разделителя (splitter bar) — тонкого прямоугольника над вертикальной полосой прокрутки — либо щелкнув полосу разделителя и перетащив ее на требуемое место. После этого можно вносить изменения в один и тот же документ в любой области окна.

### Поиск объектов *TextPane*

Объектная модель упрощает поиск объектов *TextPane*, если у вас уже есть объект *TextWindow*. просто примените свойство *ActivePane* или выполните перебор элементов набора *Panes*, пока не найдете искомый объект *TextPane*. Увы, alter ego объекта *TextWindow*— *HTMLWindow*— не предлагает подобных свойств напрямую,

поэтому надо использовать логику, подобную приведенной в функции *GetTextWindow* из предыдущего раздела, чтобы извлечь объект *TextWindow* из *HTMLWindow*.

Альтернативный путь поиска *TextPane* — через объект *TextSelection*. Он имеет свойство *TextPane*, возвращающее область окна, которой принадлежит выделенный текст. (*TextPane* имеет зеркальное свойство *Selection*, возвращающее объект *TextSelection* в области окна.) И *TextWindow*, и *HTMLWindow* имеют свойство *Selection*, как и *Window*, а значит, есть обходной путь к *TextPane*, которым могут проходить все объекты окон. Для большинства целей, однако, прекрасно работает поиск *TextPane* с использованием *TextWindow*.

У вас может возникнуть вопрос: открыта ли вторая область в окне редактора. Ответ на него дает следующий код:

```
Function IsSecondPaneOpen(ByVal txtWin As TextWindow) As Boolean
    ' Описание: Определяет, открыта ли вторая область в окне редактора.

    Return (txtWin.Panes.Count = 2)
End Function
```

Набор *TextPanes*, возвращаемый свойством *Panes* содержит один объект *TextPane* для каждой области окна, поэтому, когда открыта вторая область, свойство *Count* набора возвращает 2.

Более интересная задача — найти верхнюю или нижнюю область окна. Проблема была бы действительно сложной, если бы индекс нижней области в наборе *TextPanes* не был всегда равен 1. Применим полученную информацию для функций, которые возвращают соответствующую область:

```
Function GetTopPane(ByVal txtWin As TextWindow) As TextPane
    ' Описание: возвращает верхнюю область текстового окна.

    Dim txtPane As TextPane = Nothing

    If txtWin.Panes.Count = 1 Then
        ' Только одна панель, возвращаем ее.
        txtPane = txtWin.ActivePane
    Else
        ' Индекс верхней панели всегда 2.
        txtPane = txtWin.Panes.Item(2)
    End If

    Return txtPane
End Function

Function GetBottomPane(ByVal txtWin As TextWindow) As TextPane
    ' Описание: Возвращает нижнюю область текстового окна.
    ' Если открыта только одна область, возвращает ее.

    ' Индекс нижней области всегда 1.
    Return txtWin.Panes.Item(1)
End Function
```

Макросы *ActivateTopPane* и *ActivateBottomPane*, включенные в примеры к книге, позволяют проверить предыдущий код на реальных окнах.

И, наконец, последний вопрос: к какой области относится данный объект *TextPane*? Казалось бы, достаточно просто сравнить данный *TextPane* с соответствующим членом набора *TextPanes*, но по причинам, указанным в главе 10 во врезке «Что скрывается за *Is*?», вы не можете сравнивать ссылки *TextPane* и получить прямой ответ. К счастью, вы можете успешно сравнивать ссылки *TextSelection*, и *это* все, что нужно, чтобы написать следующую функцию:

```
Function IsTopPane(ByVal txtPane As TextPane) As Boolean
    ' Описание: определяет, является ли полученный TextPane верхней областью.

    Dim result As Boolean = False

    If txtPane.Collection.Count = 1 Then
        result = True
    Else
        If txtPane.Selection Is txtPane.Collection.Item(2).Selection Then
            result = True
        End If
    End If

    Return result
End Function

Function IsBottomPane(ByVal txtPane As TextPane) As Boolean
    ' Описание: определяет, является ли полученный TextPane нижней областью.

    Dim result As Boolean = False

    If txtPane.Collection.Count = 2 Then
        result = _
            (txtPane.Selection Is txtPane.Collection.Item(1).Selection)
    End If

    Return result
End Function
```

**Головная боль с разбиением**

Ищите сколько угодно, но вы не найдете метода *Split* в объектной модели. Однако Visual Studio .NET предоставляет команду *Window.Split* для активного окна. Для забавы предлагаем вам макрос, который **разбивает** каждое окно IDE, допускающее разделение. (Если вы считаете, что хорошо бы иметь макрос, который отменит разбиение **всех** окон, то это макрос *UnsplitAllWindows* из примеров к книге);

```
Sub SplitAllWindows()
    ' Описание: разбивает все текстовые окна.
```

*см. след. стр.*

```
    Dim win As Window
    Dim txtWin A3 TextWindow
    Dim saveWin As Window = DTE.ActiveWindow

    For Each win In DTE.Windows
        txtWin = GetTextWindow(win)

        If Not txtWin Is Nothing Then
            SplitWindow(txtWin)
        End If
    Next

    saveWin.Activate(>
End Sub

Sub SplitWindow(ByVal txtWin As TextWindow, _
        Optional ByVal restoreActive As Boolean = False)
    ' Описание: разбивает текстовое окно и может
    '  восстановить активное окно по завершении.

    Dim split As Command = DTE.Commands.Item("Window.Split")
    Dim saveWin As Window

    If restoreActive Then
        saveWin = DTE.ActiveWindow
    End If

    txtWin.Parent.Activate()

    If split.IsAvailable And Not IsSecondPaneOpen(txtWin) Then
        DTE.Commands.Raise(split.Guid, split.ID, Nothing, Nothing)
    End If

    If restoreActive Then
        saveWin.Activate()
    End If
End Sub
```

# Документы

Рискуя сообщить очевидное (возможно, глубоко очевидное), скажем, что текстовый редактор Visual Studio .NET оперирует с документами. Программируя, вы по сути меняете файл: загружаете исходный код из файла перед началом редактирования и сохраняете изменения в файле в конце — поэтому естественно предположить, что в промежутке работа ведется также с файлом. Однако файл существует на диске, а документ, с которым вы работаете в текстовом редакторе, — это нечто гораздо менее постоянное, но несравнимо более управляемое. Этот раздел знакомит вас с двумя объектами, захватывающими эти качества документов и делающих их доступными через модель автоматизации: *Document* и *TextDocument*.

## Объект *Document*

Объект *Document* служит универсальной оберткой для текстовых данных; он содержит методы и свойства, предоставляющие вам высокоуровневый контроль над данными и окнами, в которых эти данные представлены.

### Создание и поиск документов

Вы можете программно создать документ, используя методы объектов *ItemOperations* и *ProjectItems* (см. главы 8 и 9). Так, метод *ItemOperations.NewFile* соответствующий команде меню File | New | File, позволяет создать файл, не связанный с конкретным проектом. Приведенный макрос показывает, как создать текстовый файл методом *NewFile*:

```
Sub CreateNewTextFile()
    ' Описание: показывает, как использовать метод ItemOperations.NewFile
    ' для создания нового текстового файла.

    Dim Item As String = "General\Text File"
    Dim Name As String = "MyTextFile"
    Dim ViewKind As String = Constants.vsViewKindPrimary
    Dim win As Window

    win = DTE.ItemOperations.NewFile(Item, Name, ViewKind)
End Sub
```

Особенность параметра *Name* метода *NewFile* в том, что он задает имя файла косвенно. Если файл существует, имя документа и заголовок окна заменяются на имя файла. Однако, если документ создается методом *NewFile,* параметр *Name* служит только как заголовок окна нового документа — сам документ получает имя временного файла, создаваемого Visual Studio .NET для хранения нового документа. Слово <'косвенно>> означает, что, когда вы записываете документ, Visual Studio .NET отображает значение *Name* как предлагаемое по умолчанию имя сохраняемого файла.

У вас есть три способа найти и получить существующий объект *Document:* набор *DTE.Documents*, свойство *Window.Document* и свойство *DTE.ActiveDocument*. Набор *DTE.Documents* содержит ссылку на каждый открытый объект *Document.* Подобно любому другому набору, входящему в объектную модель, вы можете перебирать объекты *Document* в наборе в поисках нужного, или, если вы знаете имя документа, можно обратиться к нему через *Documents.Item*:

```
Dim doc As Document = DTE.Documents.Item("MyFile.cs")
```

Если у вас есть объект *Window*, его свойство *Document* возвращает связанный с окном объект *Document.* Некоторые из примеров этой главы используют следующий макрос для получения объекта *Document,* связанного с объектом *Window;* если окна нет, макрос создает новый текстовый файл с запрошенным заголовком и возвращает его объект *Document:*

```
Function GetDocument(ByVal caption As String) As Document
    ' Описание: возвращает объект Document, связанный
    ' с заданным окном, или создает текстовый файл
```

```
    ' в новом окне и возвращает его объект Document.

    Dim win As Window = DTE.Windows.Item(caption)

    ' Проверим, открыто ли окно.
    If win Is Nothing Then
        win = DTE.ItemOperations.NewFile("General\Text File", caption)
    End If

    Return win.Document
End Function
```

### Управление окнами документа

Между объектами *Document* и окнами существует отношение «один ко многим»: с окном всегда связан один объект *Document,* а объект *Document* может быть открыт во многих окнах. Вы можете открыть новое окно документа методом *Document.NewWindow*, работающим аналогично команде меню Window | New Window. Каждое из окон, связанных с конкретным документом, имеет заголовок, состоящий из имени документа, следующего за ним двоеточия и номера окна (например, «Connect.cs:1,» «Connect.cs:2,» и т. д.). Поскольку со всеми окнами связаны одни и те же данные, изменения, сделанные в одном окне, можно увидеть во всех других связанных окнах.

**Примечание** Файлы Visual Basic не поддерживают *Document.NewWindow* и при вызове этого метода генерируют исключение «не реализовано».

Возможность иметь несколько окон означает, что вы не найдете свойства *Document.Parent*, которое возвращает окно, содержащее данное. (Какое из окон возвращалось бы?) Вместо этого вы можете найти все окна, связанные с отдельным документом, перебрав элементы набора *Document.Windows*:

```
Sub ListDocumentWindows()
    ' Описание: перечисляет все окна, связанные
    ' с каждым открытым документом.

    Dim output As New OutputWindowPaneEx(DTE, title)

    output.Clear()
    output.WriteLine("- ListDocumentWindows -")
    output.WriteLine()

    Dim doc As Document

    For Each doc In DTE.Documents
        Dim win As Window

        output.WriteLine(doc.Name & " windows:")

        For Each win In doc.Windows
```

```
            output.WriteLine("    " & win.Caption)
        Next
    Next
End Sub
```

Чтобы найти активное окно для объекта *Document,* используйте его свойство *ActiveWindow,* которое возвращает активное окно, если это возможно, или самое верхнее окно, связанное с документом, если ни одно из связанных с документом окон не активно.

**Примечание** Б свойстве *DocumentActiveWindow* есть ошибка — оно всегда возвращает первое окно документа независимо от того, какое окно имеет фокус.

### Управление изменением документа

Самый грубый инструмент для управления изменениями объекта *Document* — его свойство *ReadOnly,* позволяющее получить/установить состояние доступности документа только для чтения. Методы, модифицирующие текст документа, генерируют исключение, если документ доступен только для чтения, поэтому стоит проверять свойство *ReadOnly,* прежде чем вносить изменения в текст.

Вы можете отменять и повторять изменения, используя методы *Document.Undo* и *Document.Redo* соответственно. Эти два метода предоставляют те же возможности, что и соответствующие команды меню Edit. Методы *Undo* и *Redo* возвращают логическое значение, показывающее, была ли выполнена операция.

**Примечание** Если вы вызываете метод *Document.Undo* или *Document.Redo* документа, доступного только для чтения, когда соответствующий стек возврата или повтора пуст, метод возвращает *False.* Проблема возникает, если вы из программы изменяете свойство *Document.ReadOnly,* так как, когда вы переключаете документ в состояние с доступом только для чтения, в документе могут быть доступные для отмены (или повтора) изменения. Если вы вызовете метод *Undo* или *Redo* документа, доступного только для чтения, когда стек отмены или повтора не пуст, генерируется исключение.

Свойство *Document.TextSelection* возвращает объект *TextSelection,* связанный с активным окном или самым верхним окном, если ни одно из окон документа не имеет фокуса. Вы можете задействовать бесчисленные методы редактирования и свойства объекта *TextSelection* для автоматизации почти любой задачи редактирования. (Об объекте *TextSelection* см. ниже соответствующий раздел.)

### Сохранение и закрытие документов

Метод *Document.Save* сохраняет документ и позволяет выбрать имя документа и место сохранения. Метод *Save* генерирует исключение, если заданного местоположения не существует; если указано корректное местоположение, но нет имени, метод *Save* использует текущее имя документа. (Ошибка в реализации Visual C++ приводит к тому, что метод *Save* игнорирует любое новое имя файла, заданное

вами.) Сохранить за один вызов каждый открытый документ позволяет метод *SaveAll* набора *Documents*. Выигрыш в удобстве приводит к проигрышу в управлении: вы не можете задать новые имена или местоположения для файлов, как это делается при сохранении файлов методом *Save*.

Метод *Document.Close* закрывает документ и позволяет передать значение *vsSaveChanges*, которое сигнализирует о необходимости сохранить изменения (*vsSaveChanges*), отклонить изменения (*vsSaveChangesNo*) или спросить пользователя о сохранении изменений (*vsSaveChangesPrompt*,значение по умолчанию). У набора *Documents* есть соответствующий метод *CloseAll*, позволяющий закрыть каждый документ и задать значение *vsSaveChanges*, которое будет применяться к каждому документу.

Свойство *Document.Saved* указывает, есть ли в документе несохраненные изменения: значение *False* говорит, что в документе есть несохраненные изменения, на что указывает звездочка в заголовке окна документа. Данное свойство контролирует, будет ли IDE спрашивать вас о сохранении при закрытии документа. Вы можете записать значение в это свойство, но будьте уверены, что не потеряете несохраненные изменения, после того как присвоите свойству *Saved* значение *True*.

## Объект TextDocument

*Объект Document* может представлять любой документ в IDE, а объект *TextDocument* представляет только текстовые документы. Получить объект *TextDocument* позволяет метод *Document.Object*: передайте ему пустую строку или значение «*TextDocument*»; метод возвращает *null* или *Nothing* для нетекстовых документов.

Самые важные свойства и методы объекта *TextDocument* связаны с объектами редактирования *TextPoint, EditPoint* и *TextSelection*. Свойство *TextDocument.Selection* возвращает выделенный фрагмент текстового документа и ведет себя во многом так же, как свойство *Document.Selection*. Свойства *StartPoint* и *EndPoint* возвращают объект *TextPoint*. отмечающие начало и конец буфера текстового документа соответственно. Метод *CreateEditPoint* возвращает объект *EditPoint в* местоположении переданного методу объекта *TextPoint;* передача значения *null* или *Nothing* создаст *EditPoint* в начале документа.

**Примечание** Вызов *TextDocument.CreateEditPoint* с параметром *TextPoint* имеет мало смысла, так как у объекта *TextPoint* уже есть собственный метод *CreateEditPoint*. И все же передача *null* методу *TextDocument.CreateEditPoint* — это единственный способ создать объект *EditPoint* без предварительного создания промежуточного объекта точки.

# Объекты точек

Как вы можете догадаться, объекты точек представляют позицию в текстовом документе. Объектная модель предоставляет вам выбор из трех объектов точек: *TextPoint, VirtualPoint* и *EditPoint*.

## TextPoint

Объект *TextPoint* воплощает фундаментальные атрибуты местоположения в текстовом документе; *VirtualPoint* и *EditPoint* реализуют интерфейс *TextPoint,* так что все объекты точек имеют эти фундаментальные атрибуты. Следующий список дает представление, что это за признаки.

- **Информация о строке** Свойство *Line* возвращает номер строки документа, в которой расположена данная точка.
- **Информация о смещении** Свойства *AbsoluteCharOffset* и *LineCharOffset* возвращают количество символов между точкой и началом документа или началом строки соответственно.
- **Информация о границах** Свойства *AtStartOfDocument, AtEndOfDocument, AtStartOfLine* и *AtEndOfLine* позволяют определить, является ли точка началом или концом документа или строки.

Методы объекта *TextPoint* *LessThan, EqualTo* и *GreaterThan* позволяют определять отношение между точками.

Объект *TextPoint* не предоставляет методы, позволяющие напрямую редактировать текст. Вместо этого данный объект точки либо передается методам редактирования, либо используется для создания в указанном местоположении объекта *EditPoint,* позволяющего редактировать текст. Способы получения объекта *TextPoint* таковы (табл. 11-3):

**Табл. 11-3. Как получить объект *TextPoint***

| Возвращается свойством | Применяется к |
|---|---|
| StartPoint | TextDocument<br>TextPane<br>TextRange |
| EndPoint | TextDocument<br>TextRange |

## VirtualPoint

Объект *VirtualPoint* представляет точку в *виртуальном пространстве,* которое — вынуждены вас разочаровать — не имеет никакого отношения к «Матрице». Виртуальное пространство — это особенность текстового редактора, позволяющая переместить точку вставки на произвольное расстояние за конец строки; ввод символов в указанное место виртуального пространства заставляет редактор автоматически заполнить пространство между концом строки и добавляемыми символами. Чтобы разрешить виртуальное пространство для всех языков, откройте диалоговое окно Tools Options, выберите Text Editor | All Languages | General и установите флажок Enable Virtual Space в группе Settings (рис. 11-2).

В табл. 11-4 перечислены способы получения объекта *VirtualPoint.* Как видите, объект *VirtualPoint* относится к объекту *TextSelection,* что дает ключ к его функциональным возможностям; выделенный фрагмент может быть расширен в виртуальное пространство, поэтому объект *TextSelection* нуждается в объектах *VirtualPoint,* чтобы отслеживать конечные точки, находящиеся вне текстового буфера.

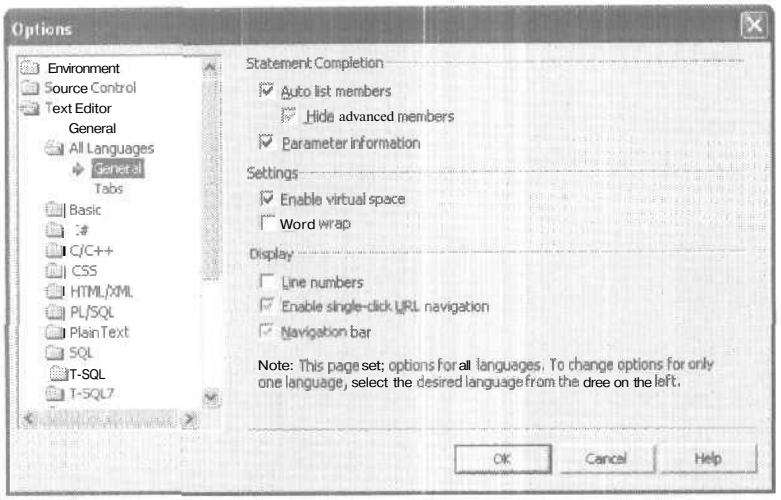



**Рис. 11-2.** *Разрешение виртуального пространства*

**Табл. 11-4. Как получить объект *VirtualPoint***

| Возвращается **свойством** | **Применяется к** |
|---|---|
| *ActivePoint* | *TextSelection* |
| *AnchorPoint* | |
| *BottomPoint* | |
| *TopPoint* | |

Как и для объекта *TextPoint*, одно из основных применений объектов *VirtualPoint* связано с созданием объектов *EditPoint*. Помните, однако, что объект *VirtualPoint* не может создать объект *EditPoint* в виртуальном пространстве: если вы попробуете сделать это, объект *EditPoint* будет создан в конце текущей строки. Вы можете избежать таких ситуаций, применив функцию, сообщающую, когда *VirtualPoint* находится в виртуальном пространстве:

```
Function IsVirtualSpace(ByVal vrtPoint As VirtualPoint) As Boolean
    ' Описание: определяет, находится ли VirtualPoint
    ' в виртуальном пространстве.
    Return vrtPoint.LineCharOffset o vrtPoint.VirtualCharOffset
End Function
```

Объект *VirtualPoint* определяет свойство с именем *VirtualCharOffset*, возвращающее значение, которое указывает, насколько далеко расположена точка от начала строки. Свойство *VirtualCharOffset* всегда имеет то же значение, что и свойство *LineCharOffset*, исключая случаи, когда точка находится в виртуальном пространстве.

### Практикум: исследование виртуального пространства

Лучший способ понять виртуальное пространство и эффекты, оказываемые им на объекты точек, — проверить их самостоятельно. Проведем быстрый эксперимент.

1. Отключите виртуальное пространство в редакторе.
2. Откройте новый текстовый файл и введите «I am я fish», не нажимая Enter.
3. Выделите все предложение мышью, проведя указателем слева направо. Редактор не расширяет выделенный фрагмент за границы.
4. Откройте окно Output и запустите макрос *DisplayTextSelectionEditPoints*. Заметьте: элементы *DisplayColumn* для *TopPoint* и *BottomPoint* равны 1 и 13 соответственно.
5. Выполните макрос *DisplayTextSelectionVirtualPoints*. Элементы *DisplayColumn* для *TopPoint* и *BottomPoint* те же. что и в предыдущем случае,
6. Выполните макрос *DisplayTextSelectionText* и посмотрите, что он выводит «I am a fish.».

Когда виртуальное пространство запрещено, объекты *VirtualPoint* ограничены пределами текстового буфера. Однако если разрешить виртуальное пространство, те же самые объекты *VirtualPoint* могут свободно перемещаться за границы.

1. Разрешите виртуальное пространство в текстовом редакторе.
2. Снова выделите предложение целиком, но на сей раз расширьте выделенный фрагмент за его границу.
3. • Снова выполните макросы *DisplayTextSelectionEditPoints* и *DisplayTextSelectionVirtualPoints*. Заметьте; значение *EditPoint* остается неизменным, но значения *VirtualCharOffset* и *VirtualDisplayColumn* объекта *VirtualPoint* превышают соответствующие значения *LineCharOffset* и *DisplayColumn*
4. Выполните макрос *DisplayTextSelectionText* и убедитесь, что он выводит то же значение, что и прежде.

Последний этап показывает, что виртуальное пространство существует вне текстового буфера, а также что виртуальное пространство не считается частью выделенного текста. Вместо этого виртуальное пространство делает возможным то, что можно было бы назвать WYSINWYG-редактированием — то, что вы видите, не обязательно совпадает с тем, что вы получаете.

## EditPoint

Объект *EditPoint* — рабочая лошадка объектов точек. В дополнение к методам и свойствам *TextPoint* этот обьект имеет методы, позволяющие автоматизировать любое возможное изменение текстового буфера. Вот как можно получить объект *EditPoint* (табл. 11-5):

**Табл. 11-5. Как получить объект *EditPoint***

| Возвращается методом | Применяется к |
|---|---|
| *CreateEditPoint* | *EditPoint* |
| | *TextPoint* |
| | *VirtualPoint* |
| | *TextDocument* |

Мы кратко опишем методы объекта *EditPoint* в разделе «Сравнение *TextSelection* и *EditPoint",*

## Объект *TextSelection*

Объект *TextSelection* совмещает два режима работы: как представление каретки в окне редактора и как представление текущего выделенного текста. (Вы можете думать о каретке как о выделенном фрагменте нулевой длины.) Поскольку в окне редактора может быть только один выделенный фрагмент, в документе может быть только один объект *TextSelection*. На рис. 11-3 показано деление *TextSelection* на составные части.

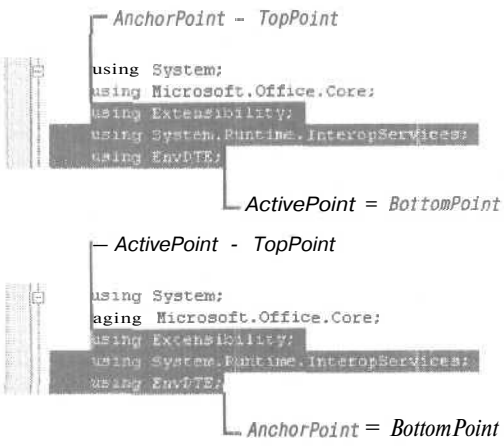



**Рис. 11-3.** *Анатомия* TextSelection

Как видите, объект *TextSelection* определяют четыре свойства: *TopPoint*, *BottomPoint*, *AnchorPoint* и *ActivePoint*. Каждое из этих свойств возвращает объект *VirtualPoint* с одного из концов выделенного диапазона. Свойства *TopPoint* и *BottomPoint* всегда ссылаются на левый верхний и правый нижний края выделенного фрагмента соответственно. Свойства *AnchorPoint* и *ActivePoint* указывают на эквивалент начальной и конечной точек выделенного мышью фрагмента; так, верхний выделенный фрагмент на рис. 11-3 может быть результатом перемещения мыши от начала *using Extensibility;* к концу *using EnvDTE;*. Можно определить ориентацию *TextSelection*, проверив его свойство *IsActiveEndGreater*, которое возвращает *True* если *ActivePoint* равно *BottomPoint*. Если ориентация вас не устраивает, можно поменять ее методом *TextSelection.SwapAnchor*, меняющим местами позиции *AnchorPoint* и *ActivePoint*.

Свойство *TextSelection.IsEmpty* позволяет узнать, есть ли выделенный текст, который можно получить из свойства *Text*. Если такового нет, *Text* возвращает пустую строку. Однако обратное утверждение неверно, поскольку *Text* возвращает пустую строку и для выделенного фрагмента в виртуальном пространстве. Когда выделенный фрагмент занимает несколько строк, свойство *TextRanges* содержит набор объектов *TextRange*, по одному объекту для каждой выделенной строки.

Способы получения объекта *TextSelection* таковы (табл. 11-6):

**Табл. 11-6. Свойства, возвращающие объект *TextSelection***

| Свойство | Применяется к |
|---|---|
| *Selection* | *Document* |
| | *TextDocument* |
| | *TextPane* |
| | *TextWindow* |
| | *Window* |

## Сравнение *TextSelection* и *EditPoint*

Объекты *TextSelection* и *EditPoint* предоставляют замечательный набор методов редактирования (табл. 11-7). Заметьте: *TextSelection* и *EditPoint* совместно используют большую часть методов и имеют лишь несколько незначительных различий вроде тех, что определяют выбор между швейцарским армейским ножом с 52 предметами, комплектуемым ножницами, и швейцарским армейским ножом с 52 предметами, комплектуемым пилкой. Обычно оба ножа работают замечательно — только в те редкие моменты, когда вам нужно собрать дрова для костра или сделать маникюр, вы понимаете, что нельзя резать ветви ножницами или стричь волосы в носу пилкой. Использование объектов редактирования во многом похоже: вы не будете знать, правильно ли выбрали объект, пока он не подведет вас.

**Табл. 11-7. Методы объектов *TextSelection* и *EditPoint***

| Задача | Общие методы | Только *TextSelection* | Только *EditPoint* |
|---|---|---|---|
| Перемещение точки вставки | *CharLeft*, *CharRight*, *EndOfDocument*, *EndOfLine*, *LineDown*, *LineUp*, *MoveToAbsoluteOffset*, *MoveToLineAndOffset*, *MoveToPoint*, *StartOfDocument*, *StartOfLine*, *WordLeft*, *WordRight* | *Collapse*, *GoToLine*, *MoveToDisplayColumn*, *PageDown*, *PageUp* | |
| Поиск и получение текста | *FindPattern* | *FindText* | *GetLines*, *GetText* |
| Выделение текста | | *SelectAll*, *SelectLine* | |
| Модификация текста | *ChangeCase*, *Copy*, *Cut*, *Delete*, *DeleteWhitespace*, *Indent*, *Insert*, *InsertFromFile*, *PadToColumn*, *Paste*, *ReplacePattern*, *SmartFormat*, *Unindent* | *DeleteLeft*, *DestructiveInsert*, *NewLine*, *Tabify*, *Untabify* | *ReplaceText* |

*см. след. стр.*

**Табл. 11-7. Методы объектов *TextSelection* и *EditPoint (окончание)***

| Задача | Общие методы | Только *TextSelection* | Только *EditPoint* |
|---|---|---|---|
| Управление закладками | *ClearBookmark*, *NextBookmark*, *PreviousBookmark*, *SetBookmark* | | |
| Прочие функции | *OutlineSection* | *SwapAnchor* | *Readonly* |

Фундаментальное различие между этими двумя объектами в том, что *TextSelection* базируется на представлении, а *EditPoint* — на буфере. *TextSelection* существует прежде всего для моделирования действий, выполняемых пользователем в редакторе: если вы что-то можете сделать вручную в редакторе, вы можете сделать это с объектом *TextSelection*. (Вы можете видеть подтверждение этого каждый раз, когда записываете макрос: Macro Recorder преобразует изменения, которые вы делаете в текстовом документе, в последовательность команд для *TextSelection*.) Этот акцент на функциональных возможностях WYSIWYG, однако, означает, что глобальное состояние представления может влиять на поведение метода *TextSelection*. Так, если разрешен перенос длинных строк, вы не можете считать, что метод *TextSelection.LineDown* переместит точку вставки на следующую строку текста из буфера: если строка за точкой вставки переносится, то перемещение точки вставки к следующей строке представления приводит только к перемещению точки вставки далее в той же самой строке буфера.

С другой стороны, объект *EditPoint* ничего не знает о представлении и поэтому обладает иммунитетом против эффектов представления. Таким образом, вызов *EditPoint.LineDown* всегда перемещает *EditPoint* к следующей строке буфера независимо от состояния переноса строк. Единственный недостаток такой изоляции от представления в том, что вы не можете задействовать объекты *EditPoint*, чтобы влиять на виртуальное пространство.

Так что, если вы хотите, чтобы ваши надстройки и макросы автоматически учитывали состояние представления, применяйте *TextSelection*; если же вам нужен полный контроль над текстовым буфером, используйте *EditPoint*.

## Контексты отмены

Интерфейс пользователя проделал долгий путь, чтобы воплотить одну из величайших надежд человечества — быть защищенным от собственной глупости. Возможность отмены, которую можно найти в большинстве современных приложений, представляет венец этого стремления. Подобно машине времени отмена позволяет вам прокручивать в обратном направлении ваши последние ошибки — обычно с чувством огромного облегчения — после чего на их месте вы можете начать делать новые. Объектная модель дает вам полный доступ к диспетчеру отмены Visual Studio .NET, разрешая определить собственный набор ошибок, которые могут быть уничтожены щелчком кнопки.

## Автоматические контексты отмены

Основной единицей «отменяемости» является контекст отмены. (Мы используем этот термин как для обозначения отменяемой единицы — именованной сущности, появляющейся в списке отмены, так и для механизма, посредством которого группируются отдельные действия при создании отменяемой единицы.) IDE Visual Studio .NET автоматически создает контексты отмены, позволяющие отменять и восстанавливать изменения в вашем коде. Проведем небольшой эксперимент, чтобы увидеть некоторые контексты отмены, автоматически создаваемые Visual Studio .NET.

1. Откройте пустой текстовый файл в Visual Studio .NET.
2. Введите **spelled backwards is epyT** и нажмите Enter.
3. Скопируйте блок текста из любого документа и вставьте его в текстовый файл,

Закончив, щелкните выпадающий список кнопки Undo, и появится список контекстов отмены (рис. 11-4). (Он представляет *стек отмены* документа, являющийся внутренней структурой данных, в которой сохраняются допускающие отмену изменения.) Каждый из трех контекстов отмены с именами Paste, Enter и Type представляет одно или несколько индивидуальных действий, которые могут быть отменены как единое целое. Бы оцените возможность группировки нескольких действий под одним именем, столкнувшись с операциями вставки больших объемов текста, альтернативой для которых была бы отмена вставки символ за символом.

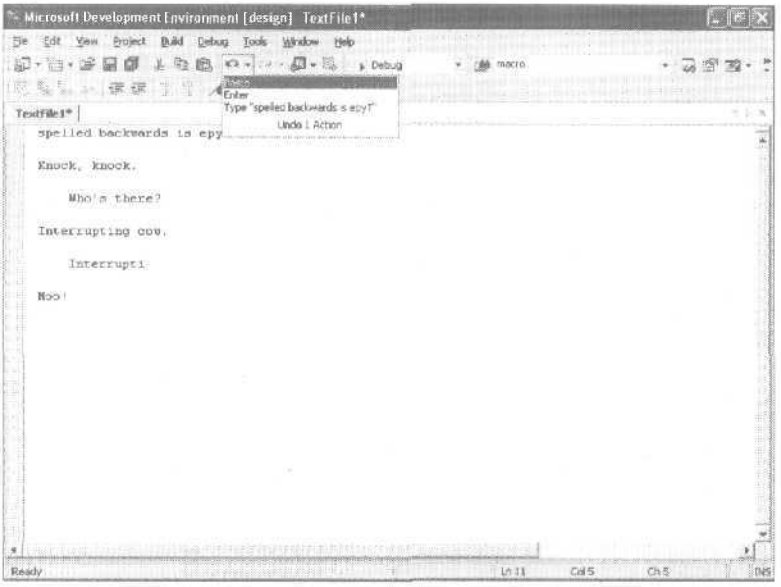



**Рис. 11-4.** *Список контекстов отмены*

## Создание контекстов отмены

Контекст отмены является атомарной транзакцией: вы открываете контекст отмены и даете ему имя, делаете изменения в одном или нескольких документах и затем либо принимаете изменения, закрывая контекст отмены, либо отменяете все

изменения. Принятые изменения могут быть отменены только всей группой. Вы создаете свой контекст отмены, вызывая методы объекта *DTE.UndoContext*:*Open* начинает контекст отмены, *SetAborted* отменяет все изменения, сделанные в текущем контексте отмены, а *Close* принимает изменения и помещает контекст отмены в стек отмены конкретного документа.

Диспетчер отмен в Visual Studio .NET допускает наличие только одного открытого контекста отмены одновременно, и для совместного использования этого контекста отмены надо соблюдать несколько правил. Во-первых, всегда вызывайте *Open* из блока *try,* так как этот метод генерирует исключение, если контекст отмены уже открыт. Хотя вы можете проверить доступность контекста отмены с помощью свойства *UndoContextIsOpen* возвращающего *True,* когда контекст отмены открыт, значение *False* не гарантирует, что контекст отмены останется свободен в тот момент, когда в вашем коде будет выполняться *Open.* Во-вторых, если вы открыли контекст отмены, то по завершении работы вы должны закрыть его методом *Close* или *SetAbort.* (Используйте либо один, либо другой, так как *SetAbort* закрывает контекст отмены за вас и, если вы потом вызовете *Close* для закрытого контекста отмены, будет сгенерировано исключение.) В-третьих, не следует вызывать *SetAbort или Close* для чего-нибудь иного, нежели контекст отмены: ни к чему хорошему это не приведет.

**Важно!** Создатели макросов должны помнить, что диспетчер отмены не верит, что они закроют собственные контексты отмены. Вместо этого он автоматически закрывает любой открытый макросом контекст отмены, когда управление возвращается Visual Studio .NET. (Надстройки могут сохранять контекст отмены открытым, сколько хотят, независимо от последствий.) Если вы находите, что это унижает вас, не доставляйте удовольствия диспетчеру отмены — всегда используйте *SetAbort* или *Close,* чтобы убрать ваши контексты отмены.

Поскольку в отдельный момент может быть открыт только один контекст отмены, если вы не получите контекст отмены, то все сделанные вами изменения будут принадлежать какому-то другому контексту. Если изменения обязательно должны быть в собственном контексте, вы должны опрашивать свойство *UndoContextIsOpen* пока контекст отмены не станет свободным. Листинг 11-1 показывает способ сделать это.

**Листинг 11-1.** Захват контекста отмены, когда он становится свободным

UndoContextTimer

```
<System.ContextStaticAttribute()> Public WithEvents UndoTimer _
    As System.Timers.Timer

Private Const title As String = "Text Editor"

Sub UndoContextTimer()
    ' Описание: если контекст отмены занят, создается таймер,
    ' опрашивающий свойство UndoContext.IsOpen,
```

```
    * пока контекст отмены не станет свободным.

    ' Запуск таймера, если контекст отмены занят.
    If Not AddText() Then
        UndoTimer = New System.Timers.Timer()
        UndoTimer.Interval - 100
        UndoTimer.Enabled = True
    End If
End Sub

private Function AddTextC) As Boolean
    " Описание: добавляет текст в документ "Text File"
    ' внутри контекста отмены "Timer".

    Dim success As Boolean = True

    Try
        ' Открытие контекста отмены.
        DTE.UndoContext.Open("Timer")

        ' Открытие документа "Text File" для редактирования.
        Dim txtDoc As TextDocument = GetTextDocument("Text File")
        Dim editPnt Aa EditPoint = txtDoc.StartPoint.CreateEditPoint

        * Добавление произвольного текста.
        editPnt.Insert("Here's text in the 'Timer' undo context." _
            + System.Environment.NewLine)

        ' Закрытие контекста отмены.
        DTE.UndoContext.Close()
    Catch
        success - False
    End Try

    Return success
End Function

Public Sub UndoTimer_Elapsed(ByVal sender As Object, _
    ByVal e As System.Timers.ElapsedEventArgs) Handles UndoTimer.Elapsed

    ' Проверить, свободен ли контекстотмены.
    If Not DTE.UndoContext.IsOpen Then
        UndoTimer.Enabled = False

        ' Уничтожить таймер, если вызов AddText успешен,
        ' в ином случае повторно разрешить его работу. ,
        If AddText() Then
            UndoTimer.Dispose()
            UndoTimer = Nothing
        Else
```

*см. след. стр.*

```
            UndoTimer.Enabled = True
        End If
    End If
End Sub

Function GetTextDocument(ByVal caption As String) As TextDocument
    " Описание: Возвращает объект TextDocument, связанный
    ' с заданным окном или создает текстовый файл
    ' в новом окне и возвращает его объект TextDocument.

    . Dim doc As Document = GetDocument(caption)

    fietyrndoc.Object("TextDocument")
End Function

Function GetDocument(caption As String) As Document
    ' Описание:  возвращает объект Document,
    ' связанный с заданным окном или создает
    " документ в новом окне.

    Dim win As Window

    Try
        ' Проверить, открыто ли окно.
        win = DTE.Windows.Item(caption)
    Catch
        ' Создание нового текстового файла.
        win = DTE.ItemOperations.NewFile("General\Text File", caption)
    End Try

    Return win.Document
End Function
```

Обычно вам не придется прибегать к мерам, показанным в листинге 11-1, потому что контекст отмены не остается открытым на продолжительный период времени. И все же полезно знать, как это можно сделать.

## Взаимосвязь стеков

Редактируя несколько документов внутри одного контекста отмены, вы рано или поздно столкнетесь с проблемой рассинхронизации стеков отмены. Допустим, вы редактируете Document 1 и Document2 в контексте отмены Link. После закрытия Link он помещается на вершины стеков отмены двух документов. Затем, если вы отменяете Link в Document 1, вы также отменяете Link в Document2, поскольку их редактирование принадлежит одной атомарной операции. Пока все хорошо.

Пусть вы добавили некий текст в Document2. Это *новое* редактирование было помещено на вершину стека отмены документа Document2. Что произойдет теперь, если мы попытаемся отменить Link в Document1? Чтобы сохранить атомарность Link, вы должны отменить Link в Document2, и здесь возникает проблема:

нельзя отменить Link в Document2, не отменив сначала добавление текста. Стеки отмены рассинхронизировались.

Диспетчер отмены решает проблему синхронизации, вводя концепцию взаимосвязи стеков. По умолчанию контекст отмены, связанный более чем с одним документом, имеет нестрогую взаимосвязь стеков, позволяющую разбить атомарный контекст по документам; когда разбиение проходит успешно, в контекст отмены каждого документа добавляются только изменения, относящиеся к нему. Если контекст отмены Link в предыдущем примере был создан с нестрогой взаимосвязью стеков, вы можете отменить Link в Document1, не влияя на Document2. Link исчезнет из стека отмены Document1, но останется на стеке отмены Document2 без тех изменений, что относятся к Document1. С другой стороны, строгая взаимосвязь стеков заставляет контекст отмены оставаться атомарным. Если бы в предыдущем примере использовалась строгая взаимосвязь стеков, диспетчер отмены отклонил бы любую попытку отменить Link в Document1.

Строгость взаимосвязи стеков задается вторым параметром *UndoContext.Open*, причем *True* означает строгую взаимосвязь. Вы можете идентифицировать контексты отмены со строгой взаимосвязью стеков по знаку «плюс» (+) предшествующему их имени в списке отмены,

### Практикум: строгая и нестрогая взаимосвязь стеков

Макрос *UndoContexts.StackLinkage* позволяет проверить различия между строгой и нестрогой взаимосвязью стеков. Этот макрос создает три документа и добавляет в них текст внутри контекста отмены; необязательный параметр типа *Boolean* контролирует, будет ли взаимосвязь стеков контекстов отмены строгой. Чтобы увидеть нестрогую взаимосвязь стеков в действии, сделайте так.

1. В окне Command Window введите **Macros.InsideVSNET.Chapter11.UndoContexts.StackLinkage** и нажмите Enter. Макрос создаст три файла: Nonstrict1, Nonstrict2 и Nonstrict3 — и добавит текст в каждый из них внутри контекста отмены NonstrictLinkage.
2. В любом из файлов щелкните кнопку Undo, а затем Redo, Вы увидите, что отмена и восстановление изменений происходят сразу во всех трех файлах.
3. Добавьте произвольный текст в файл Nonstrict2.
4. Выберите файл Nonstrict3 и щелкните его кнопку Undo,

Изменения в файле Nonstrict3 будут отменены, и его кнопка Undo посереет. Однако список отмены для Nonstrict1 и Nonstrict2 продолжает показывать NonstrictLinkage, а значит, атомарность NonstrictLinkage была разбита. Вы обнаружите, что NonstrictLinkage в Nonstrict1 отменяет изменения в Nonstrict1 и не влияет на Nonstrict2, и наоборот.

Теперь закройте все документы и повторите предыдущие действия, но добавьте параметр True к команде вызова макроса в п. 1. Параметр *True* заставляет *StackLinkage* создать файлы с именами Strict1, Strict2 и Strict3 и добавить в них текст внутри контекста отмены StrictLinkage. На этот раз, попытавшись выполнить п. 4, вы получите сообщение об ошибке «The application cannot undo». Это суть строгой взаимосвязи стеков.

## Событие *LineChanged*

Объектная модель определяет единственное событие, относящееся к редактированию, — *LineChanged*. Иметь в распоряжении только одно событие редактирования текста — это и благословение, и проклятие, подобное обладанию единственной плоской отверткой, когда половина мира держится на Phillips'овских винтах: вы всегда знаете, какой инструмент будете использовать, и, потратив время, вы сможете что-нибудь разобрать, но иногда вам хочется, чтобы инструментов было побольше.

Событие *LineChanged* имеет три параметра, помогающих выяснить, почему событие произошло. Первые два — *StartPoint* и *EndPoint* — отмечают начало и конец изменений в текстовом буфере. Чтобы получить изменения, вы можете использовать эти значения *TextPoint* следующим образом:

```
Dim text As String
text = StartPoint.CreateEditPoint.GetText(EndPoint)
```

И, если вам любопытно, какому документу принадлежат изменения, вы можете подняться по иерархии объектов на пару уровней вверх, чтобы найти родительский документ:

```
Dim doc As Document
doc = StartPoint.Parent.Parent
```

Третий параметр, *Hint*, — это битовая маска, содержащая значения из перечисления *vsTextChanged* (табл. 11-8). Набор флагов в *Hint* — это доказательства преступления, позволяющие восстановить действия, которые привели к возникновению события, [На практике параметр *Hint* не дает достаточно информации, чтобы точно выяснить, какие действия привели к возникновению события. Если бы он это делал, то не назывался бы подсказкой (hint).]

**Табл. 11-8. Перечисление *vsTextChanged***

| Поле | Описание |
|---|---|
| *vsTextChangedMultiLine* | Изменения влияют на несколько строк текста. |
| *vsTextChangedSave* | Изменения были записаны на диск. |
| *vsTextChangedCaretMoved* | Точка вставки переместилась за линию, содержащую изменения. |
| *vsTextChangedReplaceAll* | Весь буфер текста был замещен посредством вставки. |
| *vsTextChangedNewLine* | Введена новая строка. |
| *vsTextChangedFindStarting* | Операция поиска переместила точку вставки за пределы строки, содержащей изменения. |

Событие *LineChanged* не возникает при каждом изменении строки — это значит, что оно не возникает при каждом добавлении или удалении символа. Вместо этого событие возникает, когда изменения так или иначе приняты, например, когда точка вставки переместилась за пределы строки, изменения были записаны на диск или окно документа потеряло фокус. Контекст отмены блокирует это событие до своего закрытия; после этого события возникают, как если бы изменения внутри контекста отмены производились в обычных условиях (перемещение точки вставки

за пределы измененной строки, весь текстовый буфер заменен операцией вставки и т. д.). Обработчик события получает значения *StartPoint* и *EndPoint*, отражающие все неподтвержденные изменения, сделанные как до, так и внутри контекста отмены. Этот эффект демонстрируется листингом 11-2.

**Листинг 11-2.** Проверка эффекта, оказываемого контекстом отмены на событие *LineChanged*

**UndoContexts и событие *LineChanged***

```
Private Const title As String = "Text Editing Events"

<System.ContextStaticAttribute()> Public WithEvents _
    TextEditorEvents1 As EnvDTE.TextEditorEvents

Sub LineChangedAndUndoContexts(Optional ByVal useUndoContext _
        As String = "False")
    ' Описание: Демонстрирует эффект действия
    ' контекста отмены на событие LineChanged.

    TextEditorEvents1 = DTE.Events.TextEditorEvents()

    Dim useUndo As Boolean * ToBoolean(useUndoContext)

    ' Создание нового документа.
    Dint txtDoc As TextDocument = GetTextDocument("LineChanged Test")
    Dim editPnt As EditPoint = txtOoc, EndPoint.CreateEditPoint

    ' Вставка текста. Это не новая строка,
    • поэтому не вызывает событие LineChanged.
    editPnt.Insert("Beginning of document.")

    If useUndo Then
        Try
            DTE.UndoContext.Open("LineChangedAndUndoContexts")

            ' Внутри контекста отмены даже новая строка не вызывает события
            ' LineChanged, пока контекст отмены не будет закрыт.
            editPnt.Insert(Environment.NewLine)
            editPnt.Insert("Here's a new line." + Environment.NewLine)

            DTE.UndoContext.Close()
        Catch
        End Try
    Else
        ' Вне контекста отмены новая строка сразу
        ' вызывает событие LineChanged.
        editPnt.Insert(Environment.NewLine)
        editPnt.Insert("Here's a new line." + Environment.NewLine)
    End If
```

*см. след. стр.*

```
End Sub

' Обработчик события LineChanged.
Public Sub TextEditorEvents1_LineChanged( _
        ByVal StartPoint As EnvDTE.TextPoint, _
        ByVal EndPoint As EnvDTE.TextPoint, _
        ByVal Hint As Integer) Handles TextEditorEvents1.LineChanged
    Dim output As New OutputWindowPaneEx(DTE, title)

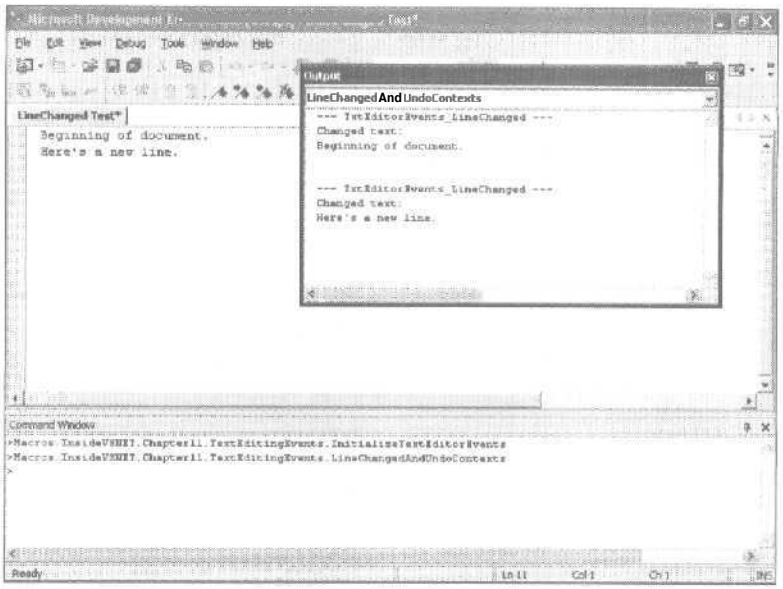
    output.WriteLine("- TextEditorEvents1_LineChanged -")
    output.WriteLine("Changed text:")
    output.WriteLine(StartPoint.CreateEditPoint.GetText(EndPoint))
    output.WriteLine()
End Sub

Function ToBoolean(ByVal val As Object) As Boolean
    Dim bool As Boolean = False

    Try
        ' Преобразование параметра в логическое значение.
        bool = System.Convert.ToBoolean(val)
    Catch
    End Try

    Return bool
End Function
```

Макрос *LineChangedAndUndoContexts* из листинга 11-2 создает новый документ и добавляет в него пару строк. На рис. 11-5 показаны результаты выполнения этого макроса вне контекста отмены: событие *LineChanged* генерируется один раз для каждой новой строки текста, так же, как если бы вы вводили эти строки вручную. Рис. 11-6 показывает, что происходит, когда изменения выполняются в контексте отмены. Поскольку новые строки создаются внутри контекста отмены, все изменения, в том числе сделанные до запуска контекста отмены, объединяются в одно событие *LineChanged*, происходящее после закрытия контекста отмены.




PMC. *11-5. Изменение текста вне контекста отмены генерирует два события* LineChanged

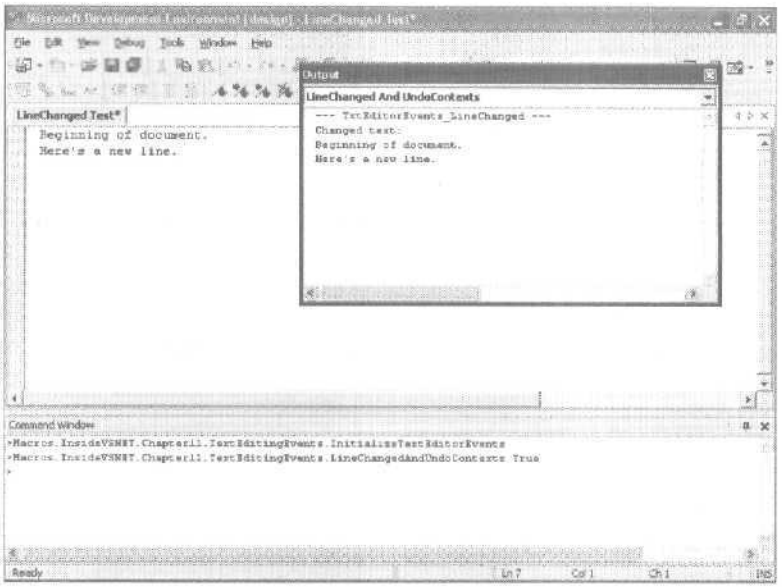



***Рис. 11-6. Изменение текста внутри контекста отмены генерирует одно событие*** LineChanged

## Несколько обработчиков события *LineChanged*

Обработчик события *LineChanged* может модифицировать текстовый буфер, не генерируя событие *LineChanged,* что позволяет вносить изменения, не опасаясь повторного входа в обработчик. Когда вовлечено несколько обработчиков событий, любые изменения, сделанные одним обработчиком, видны всем следующим обработчикам событий; всем обработчикам событий передается один и тот же параметр *Hint,* а параметры *StartPoint* и *EndPoint* изменяются, чтобы охватить любые изменения, сделанные предыдущим обработчиком.

### Практикум: применение нескольких обработчиков события *LineChanged*

Надстройка *MultipleHandlers* из примеров к книге позволяет поэкспериментировать с прохождением изменений по цепочке из нескольких обработчиков события *LineChanged*. *MultipleHandlers* определяет три обработчика события *LineChanged,* имена которых отражают порядок их вызова: *LineChanged1*, *LineChanged2* и *LineChanged3*. (Заметьте обработчики события в надстройке вызываются в том порядке, в каком они подписаны на событие. Однако обработчики событий з макросе вызываются в порядке, обратном порядку их объявления.) Каждый из этих обработчиков события отображает информацию о своих параметрах; кроме того, *LineChanged1* и *LineChanged2* изменяют документ, чтобы вы могли увидеть, как изменения передаются к *LineChanged2* и *LineChanged3* соответственно.

*MultipleHandlers* также определяет команду *ReplaceAll*, принимающую необязательный параметр *Boolean*, контролирующий изменения, производимые *LineChanged2*. Передача параметра *False* (или вызов команды без параметра) команде *ReplaceAll* заставляет *LineChanged2* добавить строку текста, тогда как параметр *True* указывает *LineChanged2* заменить весь текст документа на последние две строки документа. Заметьте, один вызов *ReplaceAll* дает одно событие *LineChanged* — *LineChanged3* отменяет подписку обработчиков событий, чтобы после эксперимента вы случайно не изменили важные файлы.

Вот эксперимент №1.

1. Откройте новый текстовый файл.
2. В окне Command Window введите **MultipleHandlers.Connect.ReplaceAll** и нажмите Enter.
3. В текстовом файле введите строку текста (например, **Here's a line) и** нажмите Enter.

Нажатие Enter генерирует событие *LineChanged*, и область Text Editing Event Output Window отображает вывод обработчиков событий. Вы увидите, как измененный текст увеличивается от *LineChanged 1* к *LineChanged3* по мере того, как обработчики событий добавляют новый текст,

Эксперимент №2 демонстрирует постоянство параметра *Hint*

1. Откройте новый текстовый файл и добавьте в него три строки текста.
2. В окне Command Window введите Mult**ipleHand**lers.Connect.Replac**eAll** и нажмите Enter.

3. Запустите макрос *TextEditingEvents.ReplaceTextWithLastTwoLines*. В области окна Test Editing Event Output Window проверьте, что одна из подсказок — *vsTextChangedReplaceAll*.

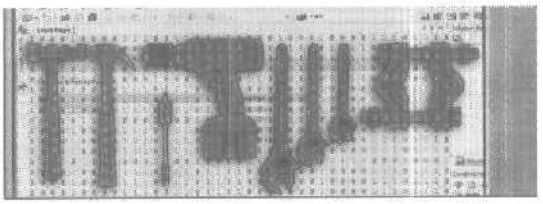
Макрос *ReplaceTextWithLastTwoLines* копирует две последние строки текста в документе и заменяет весь текст документа этими двумя строками. *LineChanged* передает значение *vsTextChangedReplaceAll*, чтобы вы знали, что событие вызвано данной глобальной заменой. Но давайте взглянем, что происходит, когда *MultipleHandlers* выполняет ту же операцию в своем обработчике события *LineChanged2*.

1. Откройте новый текстовый файл и добавьте в него две строки текста.
2. В окне Command Window введите **MultipleHandlers.Connect.ReplaceA** и нажмите Enter.
3. Добавьте третью строку текста и нажмите Enter.

Просмотрев результат, вы увидите, что в качестве подсказки о событии каждый из обработчиков получил значения *vsTextChangedCaretMoved* и *vsTextChangedNewline*, хотя *LineChanged2* выполнил действие, которое в ином случае добавило бы значение *vsTextChangedReplaceAll*. Помните это, если хотите использовать параметр *Hints* в логике обработчика события: если ваш обработчик события не первый в строке, информация в параметре *Hints* может быть подобна бисквиту месячной давности.

## Что дальше

Объектная модель упрощает редактирование текста. А поскольку исходный код — этот тоже текст, объектная модель упрощает редактирование исходного кода, так? И да, и нет. Редактирование исходного кода как большого массива символов годится для многих задач, и вы можете использовать для их решения описанные в этой главе объекты и события, но мы же программисты и знаем, что при более высокоуровневом взгляде на этот массив символов в нем можно найти массу структур. В главе 12 мы покажем вам представление более высокого уровня, обеспечиваемое объектной моделью, — модель кода.



ГЛАВА

# 12

# Модель кода

Модель кода Visual Studio .NET обещает стать ни больше ни меньше, как Универсальным Транслятором для программистов, Вавилонской рыбкой[1] для разработчиков макросов и эсперанто для хакеров. Идея проста: определяется единый API, инкапсулирующий самые распространенные программные конструкции, а все языки Visual Studio .NET реализуют этот API на своем родном диалекте. В итоге получается единый набор объектов — модель кода, которую программисты используют для чтения и записи кода на любом языке Visual Studio .NET.

## Обнаружение кода

Одна из основных функций модели кода — поиск существующих конструкций. Модель кода предоставляет инструмент для перечисления всех конструкций кода в проекте, а также позволяет сосредоточиться на любой конструкции в определенном исходном файле.

### Беглый просмотр исходного текста

Рассмотрим пример типичного исходного файла на C# (листинг 12-1).

В этом листинге определяется пространство имен с самыми разными конструкциями: делегатом, структурой, перечислением, интерфейсом и классом. В интерфейсе и классе определяются их члены; в интерфейсе — метод, а в классе — член-переменная, метод и свойство.

---

[1] В романе «Как путешествовать по Млечному Пути автостопом» фантаста Дугласа Адамса так названо «маленькое желтое существо, похожее на пиявку», которое позволяет человеку понимать любой язык. — *Прим. перев.*

**Листинг 12-1.** Пример исходного файла на C#

```
namespace CMNamespace
{
    delegate void CMDelegate(int delParam);

    struct CMStruct .
    {
        int field;
    }

    enum CMEnum
    {
        Member
    }

    interface CMInterface
    {
        inf  CMInterfaceMethod();
    }

    [ CMAttribute("CMVal") I
    class CMClass
    {
        object memberVar;

        void CMCallbackFunction(int param)
        {

        }

        int CMProperty
        {
            get
            {
                return 0;
            }
            set      -
            {
            }
        }
    }
}
```

Модель кода предлагает различные варианты просмотра этих конструкций. Начнем с самого распространенного — с помощью *CodeElement*. Объект *CodeElement* предоставляет свойства, позволяющие определить конкретную разновидность кодовой конструкции. На рис. 12-1 показано, как кодовые конструкции из листинга 12-1 представлены моделью кода.

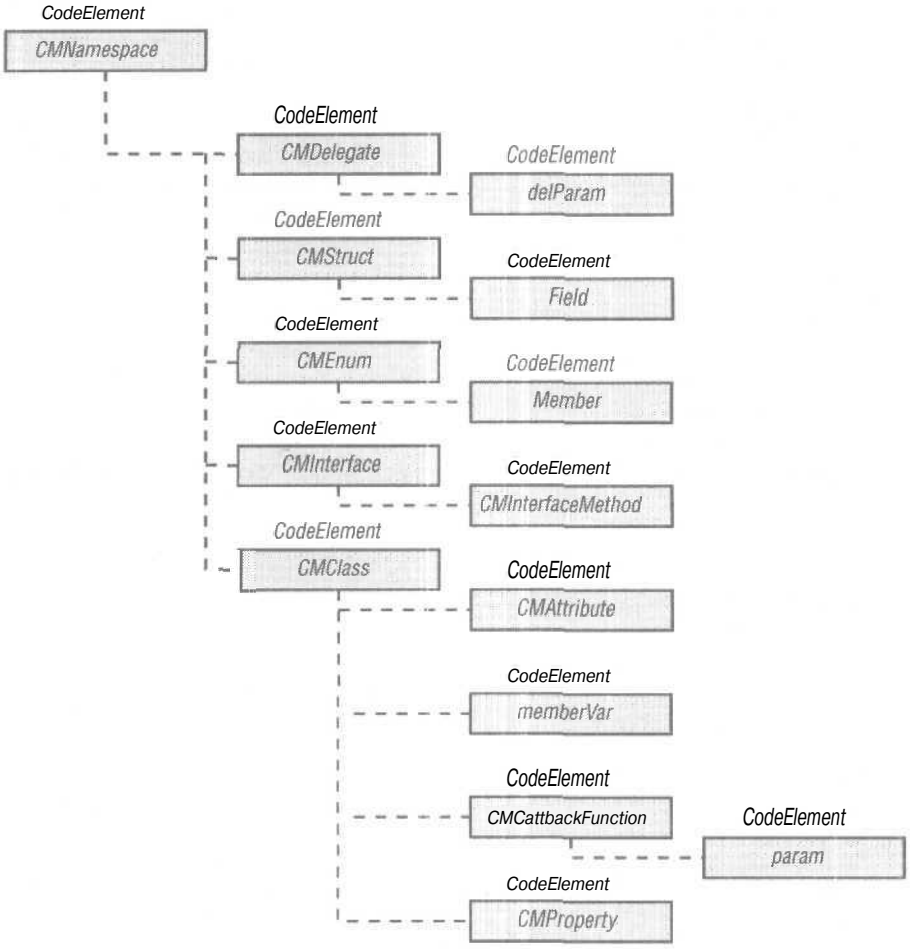



**Рис. 12-1.** *Объекты* CodeElement, *сгенерированные моделью кода для листинга 12-1*

На рис. 12-1 показана иерархическая взаимосвязь между объектами *CodeElement* начиная с *CMNamespace.* Пунктирные линии напоминают, что объекты *CodeElement* не имеют прямых объединяющих их связей: для прохода по иерархии нужно более детализированное представление.

**Примечание** Программисты знают, что современные языки сложны: обещания некоторых книг «обучить языку за 21 день* можно отнести к *cout* << *«Hello, world»;*, но не ко всему языку Visual C++. А теперь представьте сложность модели кода, которая пытается выделить функциональность четырех основных языков — Visual C++, Visual C#, Visual J# и Visual Basic — в один сложный API. Бы правы, если считаете, что в одной главе нельзя описать всю эту модель кода, да мы и не пытаемся это сделать. Эта глава — лишь краткое знакомство с двумя основными применениями модели кода;

обнаружением кода и его генерацией; полное описание модели кода содержится в приложении.

### Проход по иерархии

Чтобы двигаться по иерархии, показанной на рис. 12-1, нужно получить доступ к объекту *CodeElement* верхнего уровня. Объект *FileCodeModel* представляет исходный файл и его кодовые конструкции, а свойство *FiieCodeModeLCodeElements* этого объекта содержит нужные вам объекты *CodeElement* верхнего уровня. В нашем примере единственный объект верхнего уровня — *CMNamespace,* так что вы обнаружите единственный объект *CodeElement* в наборе *FiieCodeModeLCodeElements.* (Другой способ — задействовать набор *CodeModel.CodeElements,* в котором хранятся объекты *CodeElement* для всего проекта, но тогда вам придется просматривать элементы в этом наборе в поисках того, что представляет *CMNamespace.)* Дабы вы смогли начать свое путешествие по модели кода, листинг 12-2 предоставляет вам функции, возвращающие объект *FileCodeModel* или *CodeModel,* связанный с активным окном.

**Листинг 12-2.** Функции *GetFileCodeModel* и *GetCodeModel*

```
Function GetFileCodeModel() As FileCodeModel
    ' Описание: возвращает объект FileCodeModel активного окна.

    Dim txtWin As TextWindow = GetTextWindow(DTE.ActiveWindow)
    Dim fcm As FileCodeHodel

    If Not txtWin Is Nothing Then
        Try
            fcm = txtWin.Parent.ProjectItem.FileCodeModel
        Catch e As Exception
        End Try
    End If
    Return fcm
End Function

Function GetCodeModel() As CodeModel
    ' Описание: возвращает объект CodeModel активного окна.

    Dim txtWin As TextWindow = GetTextWindow(DTE.ActiveWindow)
    Dim cm As CodeModel

    If Hot txtWin Is Nothing Then
        Try
            cm = txtWin.Parent.ProjectItem.ContainingProject.CodeModel
        Catch e As Exception
        End Try
    End If

    Return cm
End Function
```

Мы уже знаем, что переходить от одного объекта *CodeElement* к другому напрямую нельзя, но как же получить оставшиеся объекты *CodeElement* из объекта *CMNamespace?*Нужно получить интерфейс объекта *CodeElement*, соответствующий описываемой кодовой конструкции, а затем использовать этот интерфейс для поиска соответствующих объектов *CodeElement.* Модель кода определяет интерфейсы для всех основных кодовых конструкций: *CodeNamespace*, *CodeStruct*, *CodeInterface,* *CodeClass*, *CodeEnum*, *CodeVariable*, *CodeDelegate*, *CodeProperty*, *CodeAttribute*, *CodeFunction* и *CodeParameter.* Каждый из этих интерфейсов предлагает методы и свойства, специфичные для описываемой конструкции, — например, *CodeFunction* имеет набор *Parameters,* содержащий объекты *CodeParameter* для каждого формального параметра описываемой функции. Свойство *CodeElement.Kind* возвращает значение из перечисления *vsCMElement*, указывающее конкретный тип описываемой кодовой конструкции. Свойство *Kind* объекта *CodeElement* для *CMNamespace* возвращает *vsCMElementNamespace*, а значит, вы можете получить интерфейс *CodeNamespace* из этого объекта *CodeElement.*

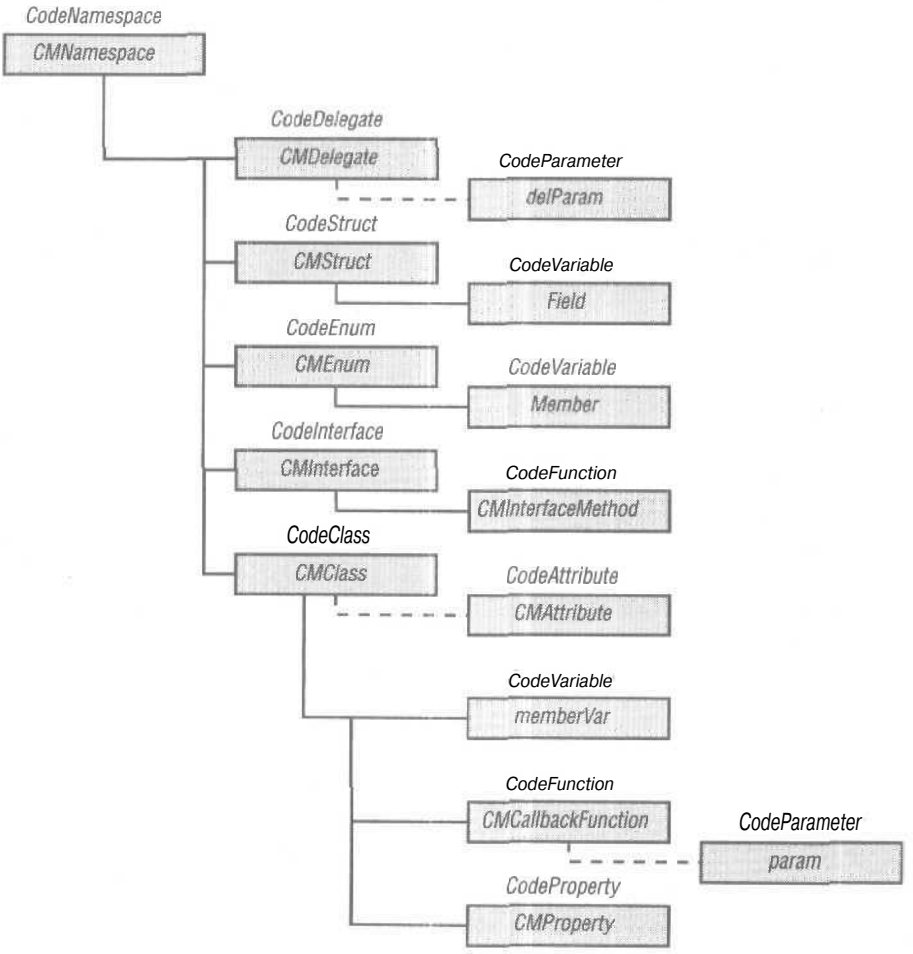



**Рис. 12-2.** *Подробное представление иерархии*

Имея интерфейс *CodeNamespace*, вы можете получить его потомков в иерархии из его набора *Members*. Набор *Members* содержится в большинстве интерфейсов, что позволяет получить их потомки. Дальнейшее движение по иерархии требует итерационного запроса конкретных интерфейсов у объектов *CodeElement* и получения дочерних объектов *CodeElement* из свойств *Members* этих интерфейсов. На рис. 12-2 показано более подробное представление иерархии для нашего примера. Сплошными линиями обозначены связи с дочерними кодовыми элементами, доступными через набор *Members*.

Листинг 12-3 демонстрирует один из способов просмотра иерархии. Набор *CodeElements* передается процедуре *RecurseCodeElements*, которая перебирает этот набор, выводя имена каждого его элемента в окно Output, причем, если текущий элемент содержит набор *Members*, делается это рекурсивно.

**Листинг 12-3.** Рекурсивный просмотр иерархии

```
Sub TestRecurseCodeElements()
    Dim output As New OutputWindowPaneEx(DTE)

    output.Clear()
    output.WriteLine("- TestRecurseCodeElements -")
    output.WriteLine()

    Dim fcm As FileCodeModel = GetFileCodeModel()

    If Not fcm Is Nothing Then
        RecurseCodeElements(fcm.CodeElements, 0) ;
    End If
End Sub

Sub ftecurseCodeElementsfByVal elements As CodeElements, _
        ByVal level As Integer)
    Dim output As New OutputWindowPaneEx(DTE)

    Dim indent As New String(" ", 4 * level)
    Dim members As CodeElements
    Dim elem As CodeElement

    ' Итерация по всем элементам набора CodeElements.
    For Each elem In elements
        ' Выводим имя элемента.
        output.WriteLine(indent 4 elem.Name)

        members = GetMembers(elem)

        If Not members Is Nothing Then
            ' Вызываем макрос рекурсивно.
            RecurseCodeElements(members, level + 1)
        End If
    Next
End Sub
```

*см. след. стр.*

```
Function GetMembers(ByVal elem As CodeElement) As CodeElements
    Din members As CodeElements = Nothing

    If Not elem Is Nothing Then
        ' Определяем тип элемента и получаем его набор Members.
        Select Case elem.Kind
            Case vsCMElement.vsCMElementNamespace
                Dim cdeNamespace As CodeNamespace = elem
                members = cdeNamespace.Members

            Case vsCMElement.vsCMElementClass
                Dim edeClass As CodeClass = elem
                members = cdeClass.Members

            Case vsCMElement.vsCMElementStruct
                Dim cdeStruct As CodeStruct = elem
                members * cdeStruct.Members

            Case vsCMElement.vsCMElementDelegate
                Dim cdeDelegate As CodeDelegate = elem
                members = cdeDelegate.Members

            Case vsCMElement.vsCMElementEnum
                Dim cdeEnum As CodeEnum = elem
                raijmbers = cdeEnum. Members

            Case vsCMElement.vsCMElementInterface
                Dim cdeInterface As CodeInterface = elem
                members = cdeInterface.Members
        End Select
    End If

    Return members
End Function
```

Функция *GetMembers* из листинга 12-3 определяет тип передаваемого ей объекта *CodeElement*, присваивает его переменной соответствующего типа и возвращает набор *Members* этого типа. В Visual Basic вы можете вернуть *Members* из самой переменной *CodeElement*, так как предполагается, что соответствующий объект также реализует свойство *Members*. Однако в языках со строгой типизацией нужно явно приводить переменную *CodeElement* к правильному типу (или интерфейсу — через *Query Interface*) так что мы написали код, пригодный для таких языков.

Функция *GetMembers* демонстрирует некоторую сложность работы с интерфейсами модели кода, Для упрощения работы с интерфейсами модель кода определяет некий обобщенный интерфейс — *CodeType*, который можно получить из любого объекта, поддерживающего один из следующих интерфейсов: *CodeClass. CodeStruct, CodeInterface, CodeEnum* и *CodeDelegate*. (Кстати, *CodeType* определяет свойство *Members,* поэтому его можно найти для объектов, поддерживающих упо-

мянутые интерфейсы.) Если у вас есть объект *CodeElement*, вы можете проверить доступность интерфейса *CodeType* напрямую, используя свойство *CodeElement.IsCodeType*.

Итак, *CodeType* дает еще один способ просмотра иерархии (рис. 12-3). *CodeType* упрощает логику просмотра иерархии кода, что видно из листинга 12-4. Единственный недостаток *CodeType*-подхода— отсутствие поддержки интерфейса *CodeType* объектом *CodeNamespace*. В макросе *RecurseCodeElementsByCodeType* решение этой проблемы требует дополнительного блока *If* для поиска именно элемента *CodeNamespace*.

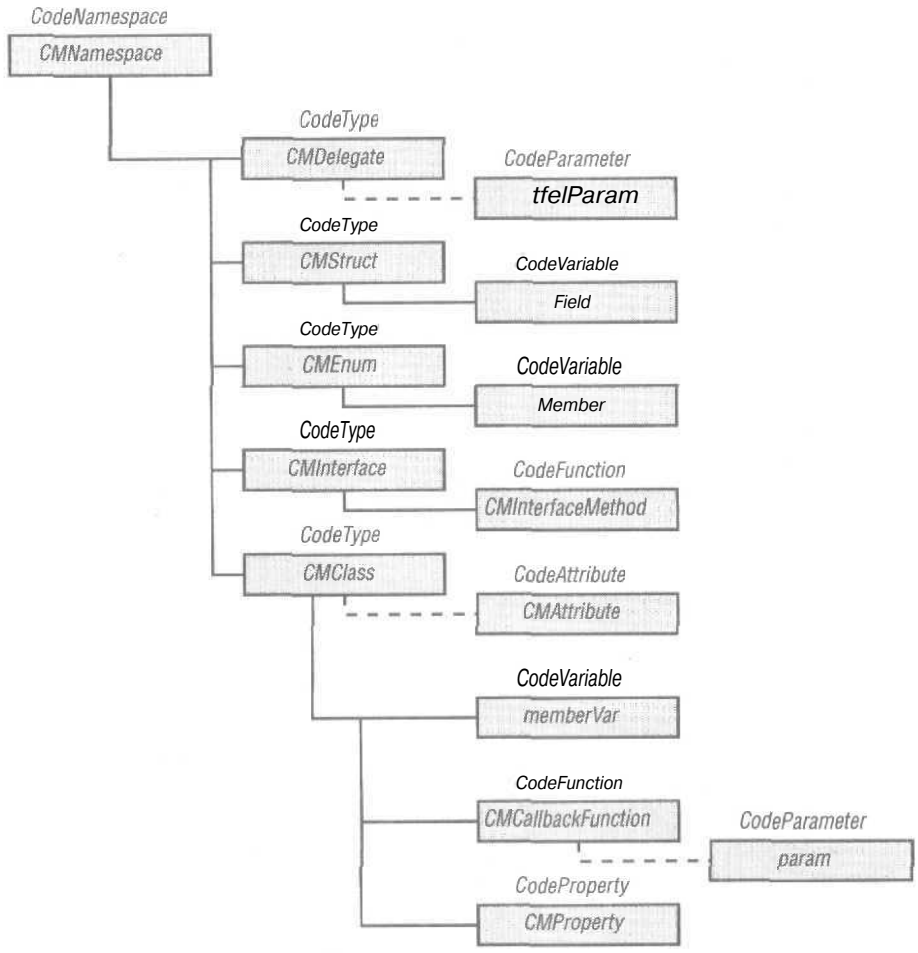



**Рис. 12-3.** *Представление иерархии с применением* CodeType

Мы уже почти у цели. Из рис. 12-2 и 12-3 видно, что наборы *Members* позволяют получить все объекты *CodeElement*, кроме атрибута класса, параметров делегата и члена-функции класса. Интерфейс *CodeClass* определяет набор *Attributes*, содержащий объекты *CodeAttribute*, которые относятся к классу, а также интерфейсы *CodeDelegate* и *CodeFunction*, определяющие набор *Parameters* объектов *Code-*

*Parameter*. **Просмотр** этих **наборов** позволит завершить **наше** путешествие по кодовой **иерархии.**

**Листинг 12-4.** Использование CodeType для просмотра иерархии кода

```
Sub TestRecurseCodeElementsByCodeType()
    Dim output As New OutputWindowPaneEx(DTE)

    output.Clear()
    output.WriteLine("- TestRecurseCodeElementsByCodeType -")
    output.WriteLine()

    81» fcm As FileCodeModel = GetFileCodeModel()

    If Not fcm Is Nothing Then
        RecurseCodeElementsByCodeType(fcm.CodeElements, 0)
    End If
End Sub

Sub RecurseCodeElementsByCodeType(ByVal elements fts CodeElements, _
        ByVal level As Integer)
    Dim output As Hew OutputWindowPaneEx(DTE)

    Dim indent As New String(" ", 4 * level)
    Dim elem As CodeElement

    ' Итерация по всем элементам набора CodeElements.
    For Each elem In elements
        ' Выводим имя элемента.
        output.WriteLine(indent & elem.Name)

        " Проверяем, является ли элемент пространством имен.
        If elem.Kind = vsCMElement.vsCMElementNamespace Then
            Dim cdeNamespace As CodeNamespace = elem

            ' Вызываем макрос рекурсивно.
            RecurseCodeElementsByCodeType(cdeNamespace.Members, level + 1}

        ' Проверяем, доступен ли CodeType.
        ElseIf elem.IsCodeType Then
            Dim cdeType As CodeType - elem

            ' Вызываем макрос рекурсивно.
            RecurseCodeElementsByCodeType(cdeType.Members, level + 1)
        End If
    Next
End Sub . .  ; •; -  :  . ..
```

### Практикум: надстройка Code Model Explorer

Надстройка Code Model Explorer (CME) предоставляет удобный интерфейс для изучения объектов модели кода, представляющих проекты Visual Studio .NET. Вы изучите типичный проект надстройки с помощью надстройки CME, Для начала проделайте следующее.

1. Создайте новый проект надстройки на Visual C++.
2. Для вывода главной формы надстройки CME выберите Tools | Code Model Explorer.
3- Откройте окно Output.
4 Выберите определение *CConnect* (рис. 12-4).




Рис. 12-4. *Окно надстройки CME*

Когда вы выбираете в дереве надстройки CME узел, которому соответствует объект *CodeElement*, надстройка выводит сведения об этом объекте в окне Output. Так, в окне Output (рис 12-4) заголовок «--- - CodeClass Details ---» указывает, что *CConnect* представлен объектом *CodeClass*; в разделе «Bases» показаны три базовых класса *CConnect: CComObjectRootEx, CComCoClass* и *IDispatchImpl*; в разделе «Children» перечислены 14 объектов *CodeElement* из набора *Children* класса *CConnect*.

Когда установлен флажок Show Definition, надстройка CME отображает исходный код для выбранного узла *CodeElement*. Установив флажок Show Definition, раскрыв узел *CConnect* и выбрав узел *OnConnection*, вы увидите исходный код метода *OnConnection*, выделенный в файле Connect.cpp (рис 12-5).

Установка флажка Show Children заставляет CME отображать объекты *CodeElement* из набора *Children*. Для объектов *CodeElement* Visual C++ набор *Children* является надмножеством набора *Members*. Установите Show Children, а затем сверните и заново разверните узел *CConnect* — появятся узлы *Children*, выделенные красным цветом (рис. 12-6). Заметьте: узел *COM* набора *Members*

см. след. стр.

не имеет дочерних узлов, это значит, что красный узел COM имеет свои узлы *Children*. Развернув красный узел *COM*, вы увидите среди его потомков два узла *COM_INTERFACE_ENTRY*. Выбрав первый узел *COM_INTERFACE_ENTRY*, вы увидите окно, показанное на рис 12-7.

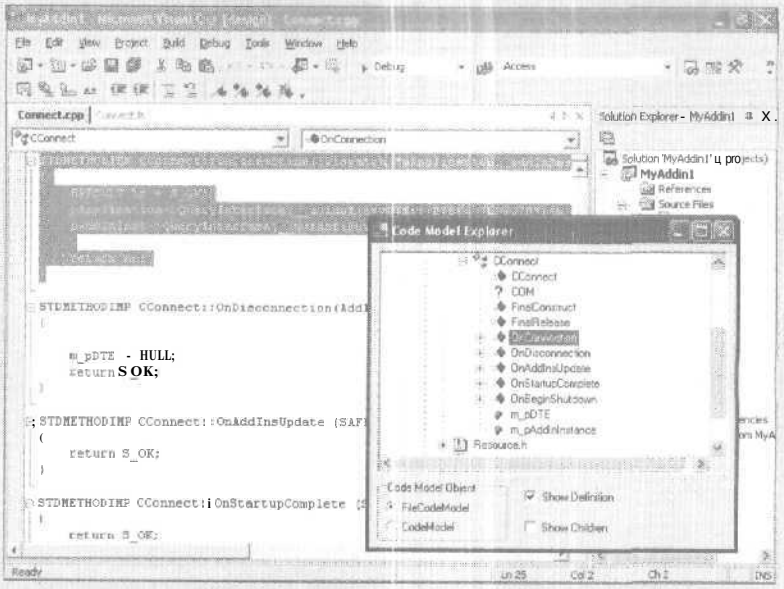



**Рис. 12-5.** *Функция Show Definition надстройки CME*




**Рис. 12-6.** *Функция Show Children*

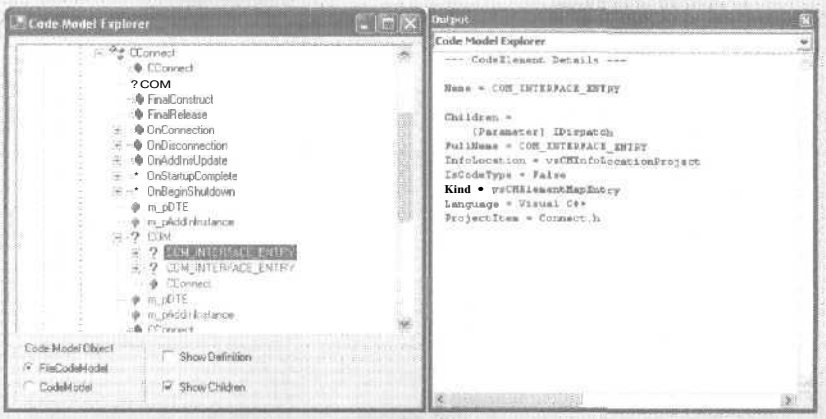



Рис. 12-7. *COM_INTERFACE_ENTRY*

В завершение выберите параметр CodeModel, который показывает объекты *CodeElement*, получаемые из свойства *CodeModel.CodeElements*. Кроме объектов верхнего уровня, получаемых из объекта *FileCodeModel*, вы увидите объекты *CodeElement*, представляющие все сущности: от макросов до библиотек IDL.

## Получение *CodeElement* из объекта точки

Вы видели, как просматривать иерархию объектов *CodeElement* сверху и получать любой потомок. Модель кода также позволяет находить *CodeElement* из объекта точки в исходном файле. Такая функция позволяет добавлять интерактивные возможности.

Модель кода предлагает два способа получения объекта *CodeElement* из точки: метод *CodeElementFromPoint* объекта *FileCodeModel* и свойство *CodeElement* объектов *TextPoint*, *EditPoint* и *VirtualPoint*. Вот прототип *CodeElementFromPoint*.

```
CodeElement CodeElementFromPoint(TextPoint Point, vsCMElement Scope);
```

Метод *CodeElementFromPoint* принимает объект *TextPoint*, который определяет расположение исходного файла, и значение *vsCMElement*, определяющее, какой из вложенных элементов кода вернуть. Например, если в листинге 12-1 есть *TextPoint*, расположенный на параметре *param* функции *CMCallbackFunction*, вызов *CodeElementFromPoint* со значением *vsCMElement*, равным *vsCMElementParameter*, вернет *CodeElement*, представляющий *param*; вызов *CodeElementFromPoint* со значением *vsCMElement*, равным *vsCMElementClass*, вернет *CodeElement*, представляющий *CMClass*.

Свойство *CodeElement* принимает значение *vsCMElement*, назначение которого аналогично параметру *Scope* метода *CodeElementFromPoint*. Может показаться странным, зачем модели кода метод *CodeElementFromPoint*, если у объекта точки есть свойство для получения *CodeElement*. Дело в том, что свойство *CodeElement* реализовано на основе метода *CodeElementFromPoint* и является лишь сокращен-

ным способом вызова *xxxPoint.Parent.Parent.ProjectItem.FileCodeModel.CodeElementFromPoint*.

### Практикум; получение объектов CodeElement из объектов точек

Макросы *TestCodeElementFromPoint* и *TestCodeElementProperty* помогут разобраться, как параметр *Scope* влияет на *CodeElement*, возвращаемый методом *CodeElementFromPoint* и свойством *CodeElement*. При работе с любым да этих макросов откройте окно Output, поместите точку вставки в любом месте исходного файла и запустите макрос. Макрос обращается к *CodeElementFromPoint* или свойству *CodeElement* со всеми возможными значениями *vsCMElement* в параметре *Scope* и отправляет информацию о возвращенном *CodeElement* в окно Output.

Если вы запустите эти макросы для различных кодовых конструкций в исходных файлах на разных языках, то скоро обнаружите, что Visual C++ лучше всего справляется с возвратом запрошенного элемента. Однако «лучше всего» не значит «идеально». Допустим, у нас есть эквивалент листинга 12-1 на Visual C++. Если вы поместите точку ввода на параметре *param* функции *CMCallbackFunction* и запустите макрос *TestCodeElementFromPoint*, Visual C++ правильно вернет *CMCallbackFunction* для *vsCMElementFunction*, *CMClass* для *vsCMElementClass* и *CMNamespace* для *vsCMElementNamespace*, но пропустит *param* при запросе с *vsCMElementParameter*! Тот же тест для других языков даст следующие результаты: Visual C# и Visual J# дадут 2 ответа на 3 запроса (*vsCMElementParameter* и *vsCMElementFunction*), а Visual Basic — 1 на 3 (*vsCMElementParameter*). Но еще хуже ложные результаты — неправильные возвращаемые объекты *CodeElement*. Заказав *vsCMElementParameter*, не удивляйтесь, когда официант принесет вам *vsCMElementFunction*.

Вывод? Вообще доверять *CodeElementFromPoint* и *CodeElement* нельзя. В отдельных ситуациях, когда *вы* знаете, что применяется только определенный язык, можете рассчитывать на успех.

# Генерация кода

Еще одно важное применение модели кода — программная генерация исходного кода. Этот аспект модели кода наиболее ярко показывает ее обещания стать универсальным языком программирования: тот же метод *AddClass*, который генерирует класс Visual C#, когда запускается для файла .cs, генерирует и класс Visual C++ для .cpp-файла, и класс Visual J# для .jsl-файла. Ниже мы покажем, как с помощью модели кода генерировать исходный файл, аналогичный листингу 12-1. Учтите: следующий пример лишь знакомит с предметом (подробнее об объектах и методах, используемых в данном разделе, см. приложение).

## Построение исходного файла

Имена всех методов модели кода, генерирующих код, начинаются с *Add*: *AddNamespace*, *AddClass*, *AddVariable* и т. д. Вызывая *Add*-метод для *CodeElement*, вы создаете новую кодовую конструкцию в этом *CodeElement*. Заметьте: объекты *CodeElement*

не могут «удочерять» другие существующие объекты *CodeElement*— у них могут быть только потомки, порожденные ими самими; следствием из этого является то, что вы можете создавать иерархию кода лишь сверху вниз. Самый верхний элемент в листинге 12-1 — это *CMNamespace*, так что начнем с создания пространства имен:

```
Sub CreateListing_12_1()
    Dim fcm As FileCodeModel = GetFileCodeModel()

    If Not fcm Is Nothing Then
        Dim cdeNamespace As CodeNamespace

        ' Пытаемся создать пространство имен.
        Try
            cdeNamespace = fcm.AddNamespace("CMNamespace")
        Catch e As Exception
        End Try

        ' В случае успеха создаем оставшиеся кодовые элементы.
        If Not cdeNamespace Is Nothing Then

            :

        End If
    End If
End Sub
```

Метод *FileCodeModel.AddNamespace* генерирует исходный код для нового пространства имен верхнего уровня и возвращает ссылку на соответствующий ему объект *CodeNamespace*. Вызов *AddNamespace* делается в блоке *Try/Catch*, так как метод *Add* генерирует исключение, если создать запрашиваемый элемент кода нельзя. Предполагая, что все идет хорошо, вы имеете ссылку, которую используете для создания дочерних элементов.

Первый создаваемый дочерний элемент — *CMDelegate*— имеет целочисленный параметр, и вы можете создавать и сам делегат, и его параметр в одном блоке *Try/Catch*:

```
Try
    Dim cdeDelegate As CodeDelegate

    ' Пробуем создать новый делегат.
    cdeDelegate = cdeNamespace.AddDelegate("CMDelegate", _
        vsCMTypeRef.vsCMTypeRefVoid)

    ' Пробуем добавить новый параметр к делегату.
    cdeDelegate.AddParameter("delParam", vsCMTypeRef.vsCMTypeRefInt)
Catch e As Exception
End Try
```

*AddDelegate* и *AddParameter* принимают параметр, указывающий тип кодового элемента; значение *vsCMTypeRef.vsCMTypeRefVoid* представляет пустой тип, а *vsCM-*

*TypeRef.vsCMTypeRefInt* целый. В предыдущем коде не объявлена переменная для хранения значения, возвращаемого методом *AddParameter*, потому что дальше с ним делать нечего.

Теперь создадим структуру и ее поля:

```
Try
    Dim cdeStruct As CodeStruct

    ' Пытаемся создать новую структуру.
    cdeStruct = cdeNamespace.AddStruct("CMStruct", -1)

    ' Пытаемся добавить поле к структуре.
    cdeStruct.AddVariable("field", vsCMTypeRef.vsCMTypeRefInt)
Catch e As Exception
End Try
```

Теперь давайте в том же ритме: определяем переменную для нового кодового элемента, присваиваем ей значение, возвращаемое *Add*-методом, и вызываем для этой переменной методы, изменяя новый кодовый элемент. Параметр *-1* в *AddStruct* указывает методу на необходимость вставлять исходный код для новой структуры после каждого кодового элемента того же уровня; этот необязательный параметр принимают все *Add*-методы. Вот код для создания перечисления и интерфейса:

```
Try
    Dim cdeEnum As CodeEnum

    ' Пытаемся создать новое перечисление.
    cdeEnum = cdeNamespace.AddEnum("CMEnum", -1)

    ' Пытаемся создать новый элемент перечисления,
    cdeEnum.AddMember("Member")
Catch e As Exception
End Try

Try
    Dim cdeInterface As CodeInterface

    ' Пытаемся создать новый интерфейс.
    cdeInterface = cdeNamespace.AddInterface("CMInterface", -1)

    ' Пытаемся добавить новый метод к интерфейсу.
    cdeInterface.AddFunction("CMInterfaceMethod", _
        vsCMFunction.vsCMFunctionFunction, vsCMTypeRef.vsCMTypeRefInt)
Catch e As Exception
End Try
```

Второй параметр *AddFunction* позволяет задать вид создаваемой функции: конструктор, деструктор, виртуальная функция и т. д. Значение *vsCMFunction.vsCMFunctionFunction* для *CMInterfaceMethod* указывается для создания простой функции. Вот, наконец, код, создающий класс, его атрибуты и члены:

```
Try
    Dim cdeClass As CodeClass

    ' Пытаемся создать новый класс.
    cdeClass = cdeNamespace.AddClass("CMClass", -1)

    Try
        ' Пытаемся добавить новый атрибут к классу.
        cdeClass.AddAttribute("CMAttribute", "CMVal")
    Catch e As Exception
    End Try

    Try
        ' Пытаемся добавить новый член-переменную к классу.
        cdeClass.AddVariable("memberVar", vsCMTypeRef.vsCMTypeRefObject, -1)
    Catch e As Exception
    End Try

    Try
        Dim cdeFunction As CodeFunction

        ' Пытаемся добавить новый член-функцию к классу.
        cdeFunction = cdeClass.AddFunction("CMCallbackFunction", _
            vsCMFunction.vsCMFunctionFunction, _
            vsCMTypeRef.vsCMTypeRefVoid, -1)

        ' Пытаемся добавить новый параметр для функции.
        cdeFunction.AddParameter("param", vsCMTypeRef.vsCMTypeRefInt)
    Catch e As Exception
    End Try

    Try
        ' Пытаемся добавить новое свойство к классу.
        cdeClass.AddProperty("CMProperty", "CMProperty", _
            vsCMTypeRef.vsCMTypeRefInt, -1)
    Catch e As Exception
    End Try
Catch e As Exception
End Try
```

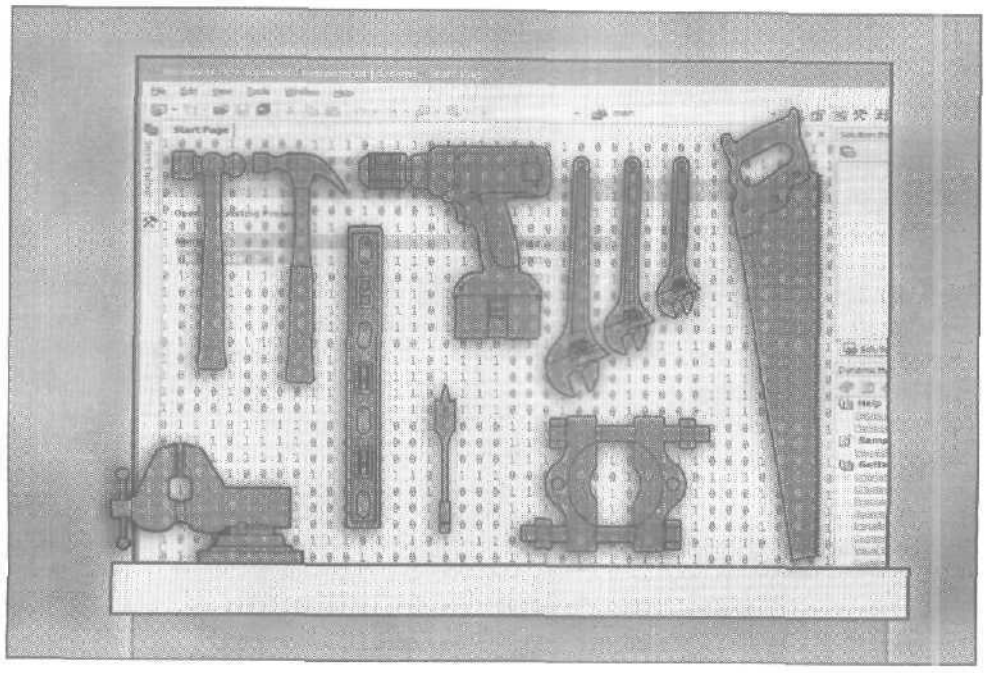
Закончив писать код, пишущий код, узнайте плохие новости: ни один из языков не позволит вам сгенерировать полную аналогию листинга 12-1 то ли потому, что язык не поддерживает каких-то кодовых конструкций, то ли потому, что в языке не реализован нужный *Add*-метод. Убедитесь в этом сами, запустив макрос *CreateListing_12_1* для исходных файлов на разных языках: Visual C# генерирует все, кроме атрибута, Visual C++ — все, кроме делегата и свойства, Visual J# — все, кроме делегата, структуры, перечисления, атрибута и свойства, а Visual Basic и новее ничего не генерирует... Но вот и хорошая новость: модель кода — самая молодая поросль объектной модели автоматизации, и мы вправе ожидать ее существенных улучшений в будущих версиях.

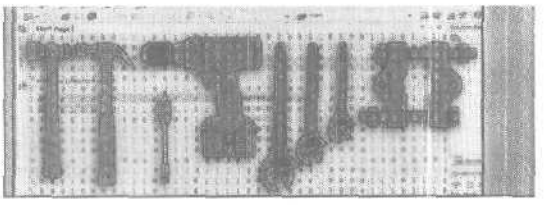
## Что дальше

Модель кода подвела нас к концу второй части книги. Третья часть — территория, не нанесенная на карты другими книгами по программированию: здесь вы узнаете как устанавливать Setup, найдете помощь по Help, включите зажигание движка V12 командной строки, скрытого под непритязательным капотом IDE Visual Studio .NET, и спрячете свой исходный код ото всех на свете, включая себя самого.

ЧАСТЬ III

# РАЗВЕРТЫВАНИЕ, СПРАВКА И РАСШИРЕННЫЕ ПРОЕКТЫ





ГЛАВА

# 13

# Разработка проектов установки

Когда новое приложение уже создано, разработчик обычно думает о том, как поместить его на компьютер пользователя- В этой главе мы исследуем инструменты Microsoft Visual Studio .NET, помогающие создать установочные файлы для применения с Windows Installer.

## История Microsoft Windows Installer (MSI)

Когда-то приложение для Microsoft Windows редко состояло более чем из одного исполняемого файла и, возможно, одной или двух DLL и устанавливалось простым копированием файлов с дискеты на компьютер. Изменение параметров ОС требовалось редко и заключалось во внесении нескольких изменений в файлы win.ini и system.ini, расположенные в папке C:\Windows.

С тех пор многое изменилось: подготовка, необходимая для запуска даже простейшего приложения в современных версиях Windows, требует массы изменений на компьютере пользователя. Процесс установки современного Windows-приложения требует модификации системного реестра для связи файлов с приложением, чтобы файлы автоматически открывались при двойном щелчке. В различных местах на диске надо разместить DLL-библиотеки COM и зарегистрировать их, что также вызывает изменения системного реестра. Следует модифицировать меню Start, чтобы пользователь мог легко найти и запустить приложение. Для Web-приложений и Web-сервисов XMI, установка приложения на Web-сервер требует не только размещения файлов на жестком диске сервера, но и настройки Microsoft Internet Information Services (IIS), чтобы Web-приложение могло обслуживать пользователей.

С ранних дней разработки для Windows непрерывно появлялись инструменты разработки программ установки. Один из самых популярных, который использовался Microsoft, — Acme Setup — применялся в таких программах, как Microsoft Office 95 и Microsoft Visual Studio 6.0. Acme Setup и многие другие технологии установки делали свое дело, но были сложны для разработчика и не давали пользователям опыта, который можно было применять от приложения к приложению. Кроме того, эти технологии установки были опасны для компьютера пользователя. Если в процессе установки возникала проблема (скажем, ошибка в устанавливаемом компоненте) или пользователь прерывал установку, состояние компьютера могло стать нестабильным; некоторые компоненты или параметры реестра могли быть оставлены или некорректно удалены.

Чтобы упростить разработку программ установки и решить связанные с существующими программами установки проблемы, для группы продуктов Office был создан новый тип программы установки — Windows Installer. Когда строится программа установки для Windows Installer, создается файл с расширением .msi; для запуска программы установки пользователю достаточно дважды щелкнуть этот файл. Файлы, составляющие устанавливаемую программу, могут быть сжаты и размещены в файле .msi или располагаться отдельно (на дистрибутивном носителе, таком как дискета, CD или DVD, являющимся отдельным от файла .msi).

Поскольку логика для обработки системного реестра, COM-объектов и (начиная с версии 2.0 Windows Installer, используемой в Visual Studio .NET и .NET Framework) компонентов .NET и Web-приложений .NET встроена в Windows Installer, установка этих компонентов стала простой. Windows Installer также выполняет подготовительные работы, например, проверку наличия достаточного объема свободного дискового пространства, и создает элемент в апплете Add Or Remove Programs Панели управления для удаления программы. И, наконец, Windows Installer поддерживает транзакции, и теперь при установке компонента или раздела реестра состояние компьютера сохраняется. Если при установке возникла проблема, выполняется откат, восстанавливающий состояние компьютера, существовавшее до начала установки.

## Создание пользовательского проекта установки

Инструменты Visual Studio .NET позволяют легко создать проект установки, генерирующий при построении файл .msi. Чтобы найти шаблоны проектов установки, в диалоговом окне New Project выберите узел Setup And Deployment Projects в дереве Project Types. Тип Setup Project служит для установки клиентских программ, таких как приложения .NET или Win32, на компьютер пользователя, а Web Setup Project — для установки Web-приложений или Web-сервисов на сервер. Шаблон Merge Module (см. ниже) предназначен для создания проекта установки компонентов.

Если вы добавите проект установки к существующему решению и затем для построения решения выберете команду меню Build \ Build Solution, проект установки построен не будет, так как в диалоговом окне Configuration Manager проект установки не выбран для построения по умолчанию. Построение проекта установки может потребовать времени, а поскольку обычно во время отладки не нужно

каждый раз заново компилировать файл .msi, для экономии времени проект установки не строится каждый раз при компиляции решения. Когда вы готовы проверить проект установки, щелкните его правой кнопкой в окне Solution Explorer и выберите в контекстном меню команду Build либо установите флажок Build в диалоговом окне Configuration Manager. Если вы создаете проект установки в файле нового решения, флажок Build установлен по умолчанию, так как других проектов в решении нет.

Как только проект установки добавлен к решению, его помогут построить шесть редакторов: File System, Registry, File Types, User Interface, Custom Actions и Launch Conditions. Они применяются, чтобы определить, как ПО будет устанавливаться на компьютер пользователя. Вы можете вывести любое из этих окон редактирования, выбрав проект установки в Solution Explorer и щелкнув соответствующую кнопку на панели команд либо щелкнув правой кнопкой проект в Solution Explorer и выбрав команду View, а затем редактор,

## Редактор File System

Используется для указания в графической форме, куда на диске поместить файлы вашего приложения при установке файла .msi. В редакторе File System можно добавлять папки и файлы, компилируемые проектом для создания структуры каталогов, логичной для вашего приложения.

### Задание установочной папки

Чтобы установить программу, нужно создать структуру каталогов, содержащих программные файлы. В File System доступно много предлагаемых по умолчанию папок для установки, предоставляющих начальную точку для структуры каталогов. Чтобы добавить файл в папку, щелкните требуемую папку правой кнопкой, подведите указатель к команде Add и выберите один из типов файлов: Folder — для создания подпапки, Project Output — для добавления файла, генерируемого другим проектом, входящим в решение, File — для файла на диске или Assembly — для файла сборки. Visual Studio .NET определяет следующие папки, в которые вы можете добавлять свои файлы или папки (в скобках указано возможное местоположение).

- **Common Files Folder** Для файлов, общих для всех установленных на компьютере программ (C:\Program Files\Common Files).
- **Fonts Folder** Для всех файлов шрифтов, установленных на компьютере (C:\Windows\Fonts — по умолчанию).
- **Program Files Folder** Папка, в которой сохраняются устанавливаемые на компьютере программы (C:\Program Files).
- **System Folder** Для сохранения системных компонентов. Содержимое папки изменяется в редких случаях (C:\Windows\System32).
- **User's Application Data Folder** Папка, в которой приложение хранит файлы данных, которыми пользователь не должен управлять (C:\Documents and Settings*пользователь*\Application Data).
- **User's Desktop** Для всех элементов, отображаемых на рабочем столе пользователя (C:\Documents and Settings*пользователь*\Desktop — по умолчанию).

- **User's Favorites Folder** Для избранных ссылок (C:\Documents and Settings*пользователь*\Favorites — по умолчанию).
- **User's Personal Data Folder** Для документов, создаваемых пользователем (C:\Documents and Settings*пользователь*\My Documents).
- User's **Programs Menu** Для отображаемых в меню Start ярлыков программ (C:\Documents and Settings*пользователь*\Start Menu\Programs).
- **User's Send To Menu** Для элементов меню Send To, которое можно вывести, щелкнув файл в Windows Explorer и выбрав команду Send To (C:\Documents and Settings*пользователь*\SendTo).
- **User's Start Menu** Для элементов меню Start (C:\Documents and Settings*пользователь*\Start Menu).
- User's **Startup Folder** Для всех программ (или ярлыков программ), которые запускаются после регистрации пользователя в системе (C:\Document.s and Settings*пользователь*\Start Menu\Programs\Startup).
- **User's Template Folder** Для шаблонов, предназначенных для создания новых файлов. Это исходная папка для хранения элементов подменю New контекстного меню рабочего стола (C:\Documents and Settings*пользователь*\Templates).
- **Windows Folder** Для файлов ОС (обычно C:\Windows).
- **Global Assembly Cache Folder** Для глобального кэша сборок (GAC) компьютера. Любые файлы, помещенные в эту папку, доступны через .NET Framework для всех пользователей компьютера.
- **Application Folder** Папка, где следует хранить большинство файлов и выходных данных проекта, установленного на компьютере пользователя (C:\Program Files*Производитель**Продукт* — по умолчанию; *Производитель* и *Продукт* - значения свойств отображаемых в окне Properties, когда проект установки выбран в Solution Explorer).
- **Web Application Folder** Папка, доступная, только если проект установки является Web Setup Project. Элементы, добавляемые в эту папку, устанавливаются в виртуальную папку IIS и доступны (путем назначения параметров безопасности) пользователям Web-сервера.

Кроме того, вы можете создать на компьютере новую папку, которая не будет дочерней по отношению к какой-либо папке по умолчанию. Чтобы сделать это, щелкните правой кнопкой узел File System On Target Machine в редакторе File System и выберите Add Special Folder | Custom Folder. Имя, которое вы введете, не является путем к создаваемой на компьютере папке, а служит лишь для отображения в редакторе File System. Чтобы задать путь, выберите вновь созданную папку и введите путь для создания папки в свойство *DefaultLocation* в окне Properties. Будьте осторожны, когда создаете папку с жестко заданным путем: если вы укажете диск, отсутствующий на компьютере пользователя, будет сгенерирована ошибка. Ниже мы покажем, как динамически генерировать путь во время установки.

### Выходные данные проекта

Создав программу и структуру каталогов для ее хранения, нужно добавить файлы в редактор File System. Можно построить проект и затем вручную выбирать каж-

дый сгенерированный проектом файл и добавлять его в редактор File System. Однако данный подход приводит к проблемам. Так, если вы меняете тип проекта с Debug на Release, нужно вручную модифицировать проект установки, чтобы гарантировать, что в него скопированы правильные версии файлов. Другая проблема: можно забыть включить файл или добавить ненужный файл, что приведет к неправильно функционирующей программе или созданию файла .msi большего размера, чем надо. Чтобы упростить выбор установочных файлов проекта, можно включить выходные данные проекта в проект установки.

Чтобы добавить выходные данные проекта в редактор File System, щелкните правой кнопкой папку, которая должна содержать их, и выберите Add | Project Output. В открывшемся диалоговом окне Add Project Output Group будут перечислены проекты, генерирующие выходные данные и типы выходных данных, которые можно добавить в проект установки. Вы можете выбрать следующие типы выходных данных проекта.

- **Documentation Files** Этот тип добавляет в редактор File System любые файлы, генерируемые IntelliDoc. Файлы IntelliDoc могут генерироваться только из проектов C#, поэтому данный параметр доступен, только если проект отмечен в диалоговом окне Project как проект C#.
- **Primary Output** Данная группа содержит файлы DLL или EXE, создаваемые при компиляции проекта. Вы должны добавить эти выходные данные в редактор File System, чтобы разрешить пользователю запускать вашу программу.
- **Localized Resources** Если выбран данный тип, в указанную папку редактора File System копируются все сателлитные DLL ресурсы.
- **Debug Symbols** Данный тип выходных данных включает всю отладочную информацию, такую как файлы .pdb, используемые при отладке приложения. Эти данные нужно добавить в редактор File System, если вы хотите, чтобы в случае проблем пользователь мог выполнять отладку приложения. Однако размещение отладочной информации на компьютере пользователя увеличивает файл .msi. Кроме того, это упрощает восстановление исходного кода приложения и делает возможным доступ к вашей интеллектуальной собственности.
- **Content Files** Этот тип выходных данных включает все файлы указанного проекта, добавленные как файлы содержимого.
- **Source Files** Эти выходные данные включают все исходные файлы, используемые при построении проекта.
- **Built Outputs** Этот тип выходных данных доступен, только если выбранный проект является другим проектом установки. Его добавление к проекту установки позволяет установить файл .msi на установочный диск.

Когда выходные данные проекта добавлены к проекту установки, Visual Studio .NET автоматически сканирует их и добавляет в проект любые файлы (сборки, неуправляемые DLL или библиотеки типов), от которых он зависит. При компиляции проекта установки эти DLL упаковываются в файл .msi и устанавливаются вместе с кодом, зависимым от этих DLL. В редакторе File System такие DLL отображаются рядом с файлом или выходными данными проекта, которые зависят от них, и добавляются к Solution Explorer в папку вашего проекта установки Detected Dependencies.

Вы можете определять, какие зависимости будут устанавливаться на компьютер пользователя, исключая ненужные, щелкнув правой кнопкой зависимость и выбрав команду Exclude. Существует масса причин для исключения файла, и самая распространенная в том, что, если на компьютере пользователя уже есть файл, от которого зависит приложение, его не надо упаковывать в файл .msi. Кроме того, когда выходные данные проекта добавлены в редактор File System, зависимости создаются из проекта установки для проекта, генерирующего выходные данные. В результате при запуске построения решения проект, генерирующий данные, добавляемые в проект установки, будет построен до компиляции проекта установки, что гарантирует обновление файлов в проекте.

### Ваши мастера забастовали?

Тестируя проект установки, чтобы убедиться, что он корректно устанавливается и удаляется, вы можете обнаружить, что программы прекратили работать, особенно после отмены установки файла .msi. Это часто случается, когда вы строите программу установки для мастеров или надстроек. Допустим, из проекта надстройки вы ссылаетесь на сборку *VSLangProj*.Когда вы строите проект установки для надстройки, проект видит, что вы ссылаетесь на сборку *VSLangProj* и автоматически добавляет ее к зависимостям проекта установки. Кроме того, поскольку *VSLangProj* — это Primary Interop Assembly (PIA) для библиотеки типов VSLangProj.tlb, библиотека типов будет также добавлена в список зависимостей проекта. Последний файл и является источником проблем. Когда в редактор File System добавляются COM-объекты (включая библиотеки типов), они автоматически помечаются для регистрации после установки.

Если отменяется установка файла .msi с вашей надстройкой или мастером, Windows Installer удаляет все установленные в системе файлы. Поскольку отменяется установка библиотеки типов VSLangProj.tlb, отменяется и *ее* регистрация как библиотеки типов, Другие компоненты, такие как маете] >а Visual Studio .NET, обращатся к ней, и, если она не зарегистрирована, мастер не может работать.

Чтобы решить эту проблему и предотвратить ее повторное возникновение, после добавления выходных данных проекта в редактор File System нужно просматривать проект установки, чтобы обнаружить добавленные как зависимости компоненты, не являющиеся частью вашего проекта. Если вы находите такой файл, щелкните его правой кнопкой и выберите команду Exclude или, если вы знаете, что зависимость локализована в подгружаемом модуле, добавьте этот модуль к проекту установки, Это гарантирует корректную установку компонента на компьютер. (*VSLangProj* не находится в подгружаемом модуле, так что этот вариант не годится для мастеров и надстроек.)

Если вы уже отменили установку файла .msi, содержащего зависимость, которую не надо было устанавливать, и хотите восстановить работоспособность компьютера, есть несколько вариантов. Во-первых, вы можете восстановить приложение, прекратившее работать. Если приложение — это Visual Studio .NET, восстановление будет длительным, и вы можете захотеть

*см. след. стр.*

обойтись без него. Альтернативное решение — найти и вручную зарегистрировать библиотеки типов и COM-объекты, регистрация которых была отменена. Вы сможете сделать это, только если у вас есть инструменты. Третий вариант — создать пустой проект установки, добавить к нему нужные компоненты (такие как VSLangProj.tlb), а затем построить этот проект и установить его, При этом файлы будут зарегистрированы, и все будет прекрасно работать, пока вы не отмените установку этого файла .msi.

## Редактор Registry

Редактор Registry предоставляет интерфейс «укажи и щелкни» для элементов системного реестра, создаваемых при установке файла .msi. В процессе установки созданные в этом редакторе разделы и значения реестра копируются в системный реестр, отражая созданную структуру. Вы можете посмотреть пример использования редактора Registry для справки в параметрах надстроек, запустив мастер Add-in Wizard Цель проекта установки, добавляемого к решению при запуске этого мастера, — не установка файлов (хотя проект установки способен помочь вам и в этом), а создание разделов реестра, нужных Visual Studio .NET для поиска, загрузки и запуска надстройки.

Окно редактора Registry напоминает окно редактора реестра Registry Editor (regedit.exe), устанавливаемого с Windows. Так же, как вы редактируете системный реестр, используя regedit.exe, можно редактировать реестр, используя редактор Registry. Единственное отличие в том, что параметры реестра, объявленные в окне редактора Registry Visual Studio .NET, создаются при установке,

Если вы используете проект для установки COM-объектов, созданных на языке C++, нужно решить: определять разделы реестра для этого COM-объекта в редакторе Registry или позволить COM-объекту регистрироваться при установке самому. Когда выходные данные проекта COM-объекта добавляются в редактор File System, свойство *Register* в окне Properties для этих данных имеет значение *vsdrpCOM*, т. е. COM-объект знает, как регистрировать себя и при его установке и удалении вызываются методы *DllRegisterServer* и *DllUnregisterServer*.Однако, если COM-объект регистрируется сам, созданные им разделы не входят в транзакцию Windows Installer. Если установка завершается неудачно, вызывается метод *DllUnregisterServer*, чтобы попробовать выполнить откат для созданных разделов реестра; если это не удается, некоторые элементы реестра могут остаться. С другой стороны, создание записей реестра для COM-объекта может быть утомительной работой и чревато ошибками. Если б нужно было только скопировать элементы файла .rgs в редактор Registry проблем бы не возникло, но вы должны создать массу разделов и значений реестра для каждого интерфейса, определенного COM-объектом, чтобы DLL посредников и заглушек для этого интерфейса устанавливались корректно. Выбор за вами.

### Куст User/Machine

Редактор Registry позволяет модифицировать параметры реестра в четырех основных корневых разделах (или кустах) реестра: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE и HKEY_USERS. Кроме того, редактор Registry

для проектов установки .msi предоставляет один дополнительный узел, не являющийся корневым разделом системного реестра, — раздел User/Machine Hive. Этот раздел применяется для условной модификации разделов HKEY_LOCAL_MACHINE или HKEY_CURRENT_USER в зависимости от параметров, выбранных пользователем при установке файла .msi. Если вы создадите проект установки, построите его и установите полученный в итоге файл .msi, вторая страница пользовательского интерфейса будет выглядеть, как на рис. 13-1. В верхнем левом углу диалогового окна — два переключателя: Everyone и Just Me. Если пользователь выбирает Everyone, все разделы реестра, созданные в разделе User/Machine Hive помещаются в системный реестр под разделом HKEY_LOCAL_MACHINE, а если Just Me — все эти параметры помещаются в раздел системного реестра HKEY_CURRENT_USER. Если у человека, запустившего файл .msi, ограниченные права, например Guest, эти два параметра не отображаются, и по умолчанию используется Just Me.

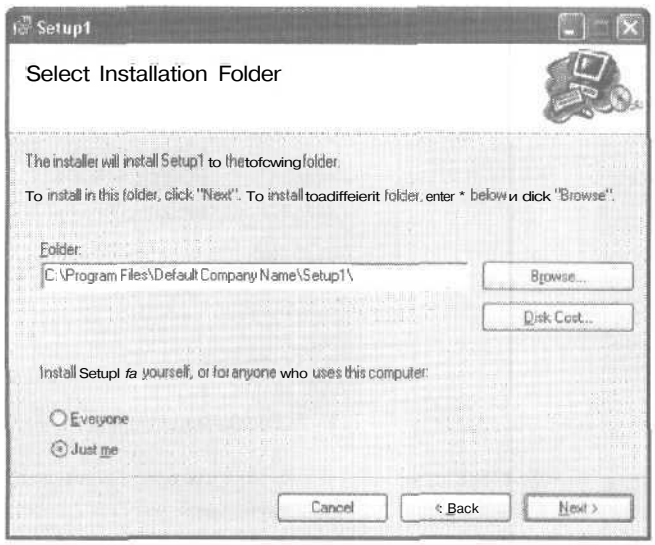



Рис. 13-1. *Страница Select Installation Folder установочного файла .msi*

## Свойства установщика

Хотя редактор Registry предоставляет простой в использовании интерфейс «укажи и щелкни» для создания параметров реестра для ваших программ, все выполняемые модификации реестра являются статическими. То есть имена разделов, имена значений и данные значений копируются в реестр при установке в том же виде, в каком вы их ввели. Однако иногда нужно создать раздел реестра с динамически изменяемым именем или значением, отражающим состояние компьютера пользователя во время установки файла .msi. Такие динамические значения называются свойствами установщика. Чтобы задействовать их, в редакторе Registry достаточно поместить имя свойства установщика в квадратные скобки; когда файл .msi будет запущен, Windows Installer обнаружит эти свойства установщика и заменит их на соответствующие значения.

В табл. 13-1 перечислены наиболее часто используемые свойства установщика из около 200 доступных. Новые свойства добавляются в каждой новой версии Windows Installer. Чтобы получить самый последний список доступных свойств установщика, обратитесь к MSDN. Так, чтобы поместить в реестр дату установки пользователем файла .msi, используйте *[Date]*как имя раздела реестра, имя значения или значение. Вы можете комбинировать свойства установщика, помещая их одно за другим и в случае нужды добавлять строковые данные. Так, значение*[Time]-[Date]* 10 ноября 2002 года в 17:54:35 будет преобразовано в *17:54:35-11/10/2002*. Одно из этих свойств установщика вы можете увидеть в новом проекте установки. Взглянув в редакторе Registry на проект установки, вы увидите автоматически созданные разделы HKEY_LOCAL_MACHINE\Software\[Manufacturer] и HKEY_CURRENT_USER\Software\[Manufacturer]. Это разделы, в которые помещают данные, специфичные для ПО вашей компании; маркер [*Manufacturer*]преобразуется в название вашей компании, которое вы указываете как свойство *Manufacturer* в окне Properties, когда проект установки выбран в Solution Explorer.

**Табл. 13-1. Часто используемые свойства установщика**

| Свойство установщика | Описание | Пример |
|---|---|---|
| *AdminToolsFolder* | Папка, в которой хранятся инструменты для администрирования компьютера. | C:\Documents and Settings\ *пользователь*\Start Menu\ Programs\Administrative Tools\ |
| *AdminUser* | Значение равно 1, если пользователь, выполняющий установку, имеет привилегии администратора, и 0 — если нет. | 1 |
| *AppDataFolder* | Папка, в которой хранятся данные, специфичные для приложения, | C:\Documents and Settings\ *пользователь*\Application Data\ |
| *ARPCONTACT* | Имя контактного лица в службе технической поддержки. Значение устанавливается из свойства Author, отображаемого в окне Properties, когда выбран проект установки. | По умолчанию название компании |
| *Author* | Автор установщика. Значение этого свойства устанавливается в окне Properties, когда проект установки выбран в Solution Explorer. | По умолчанию название компании |
| *CommonApp DataFolder* | Папка, совместно используемая всеми пользователями для хранения специфичных для приложения данных. | C:\Documents and Settings\ All Users\Application Data\ |
| *CommonFilesFolder* | Папка, в которой хранятся компоненты, совместно используемые ПО. | C:\Program Files\Common Files\ |
| *ComputerName* | Сетевое имя компьютера. | CRAIGS4000 |
| *Date* | Дата установки файла .msi. | 11/10/2002 |
| *DesktopFolder* | Папка для установки элементов рабочего стола пользователя. | C:\Documents and Settings\ *пользователь*\Desktop\ |
| *FavoritesFolder* | Папка, в которой хранятся избранные ссылки Internet Explorer. | C:\Documents and Settings\ *пользователь*\Favorites\ |

**Табл. 13-1. Часто используемые свойства установщика** *(продолжение)*

| Свойство установщика | Описание | Пример |
|---|---|---|
| *FontsFolder* | Папка для хранения шрифтов. | C:\Windows\Fonts\ |
| *IntelL* | Если установка запущена на компьютере с процессором Intel, значение используется для идентификации класса процессора: 4 для 486. 5 для Pentium, 6 для P6 и т. д. | 6 |
| *LocalAppDataFolder* | Папка для неперемещаемых данных пользователя, специфичных для приложения. | C:\Documents and Settings\ *пользователь*\ Local Settings\ Application Data\ |
| *LogonUser* | Имя пользователя, запустившего файл .msi. | Craig Skibo |
| *Manufacturer* | Название компании, создавшей файл .msi. | По умолчанию название компании |
| *MyPicturesFolder* | Папка, в которой хранятся пользовательские изображения. | C:\Documents and Settings\ *пользователь*\My Documents\ My Pictures\ |
| *MsiNTProductType* | Тип установленной ОС: 1 — компьютер является рабочей станцией, 2 — контроллером домена, 3 — сервером. | 1 |
| *NetHoodFolder* | Папка сетевого окружения, | C:\Documents and Settings\ *пользователь*\NetHood\ |
| *PersonalFolder* | Папка, в которой хранятся документы пользователя. | C:\Documents and Settings\ *пользователь*\ My Documents\ |
| *PhysicalMemory* | Объем памяти (в Мб) компьютера, на котором запущен файл .msi. | 384 |
| *PrintHoodFolder* | Папка, в которой установлены принтеры. | C:\Documents and Settings\ *пользователь*\PrintHood\ |
| *Privileged* | 1, если установка производится с увеличенными привилегиями пользователя. | 1 |
| *ProductID* | Серийный номер, введенный в поле Serial Number диалогового окна User Information (см. ниже). | 111-7000000 |
| *ProductName* | Название устанавливаемого продукта. Вы задаете это значение, изменяя свойство ProductName в окне Properties Visual Studio .NET, когда выбран проект установки. | Setup1 |
| *ProductVersion* | Версия устанавливаемого файла .msi. Вы устанавливаете это значение в свойстве Version в окне Properties, когда проект установки выбран в Solution Explorer. | 1.0.0 |
| *ProgramFilesFolder* | Папка Program Files. | C:\Program Files\ |

*см. след. стр.*

**Табл. 13-1. Часто используемые свойства установщика *(продолжение)***

| Свойство установщика | Описание | Пример |
|---|---|---|
| *ProgramMenuFolder* | Папка, в которую помещаются ярлыки программ для меню Stare. | C:\Documents and Settings\ *пользователь*\Start Menu\ Programs\ |
| *RecentFolder* | Папка, в которую помещаются ярлыки для недавно использованных документов. | C:\Documents and Settings\ *пользователь*\Recent\ |
| *RemoteAdminTS* | 1, если на компьютере установлены и настроены службы терминалов, | 1 |
| *ROOTDRIVE* | Диск, на который устанавливается программа. | C:\ |
| *ScreenX* | Установленное разрешение по горизонтали (в точках) основного монитора пользователя. | 1024 |
| *ScreenY* | Установленное разрешение по вертикали (в точках) основного монитора пользователя. | 768 |
| *SendToFolder* | Папка, содержащая элементы, показываемые в контекстном меню, когда вы щелкаете правой кнопкой файл в Windows Explorer и выбираете команду Send To. | C:\Documents and Settings\ *пользователь*\SendTo\ |
| *ServicePackLeveL* | Номер версии установленного на компьютере пакета обновлений (service pack). | 1 |
| *ServicePackLevel-Minor* | Младшая цифра номера версии установленного на компьютере пакета обновлений. | 0 |
| *SourceDir* | Папка, содержащая файл .msi. | C:\Documents and Settings\ *пользователь*\My Documents\Visual Studio Projects\ Setup1\Debug\ |
| *StartMenuFolder* | Папка, в которой хранятся ярлыки меню Start. | C:\Documents and Settings\ *пользователь*\Start Menu\ |
| *StartupFolder* | Папка, содержащая ссылки на программы, запускаемые при регистрациипользователя в ОС. | C:\Documents and Settings\ *пользователь*\Start Menu\ Programs\Startup\ |
| *SystemFolder* | Системная папка Windows. | C:\Windows\System32\ |
| *SystemLanguageID* | Идентификатор локализации (LCID) ОС, | 1033 |
| *TARGETDIR* | Папка, в которую устанавливается проект установки. | C:\Program Files\Default Company Name\ Setup1\ |
| *TempFolder* | Папка для временных файлов, | G\ Documents and Settings\ *пользователь*\ Local Settings\Temp\ |
| *TemplateFolder* | Папка, в которой хранятся шаблоны — элементы, отображаемые при выборе команды New в контекстном меню рабочего стола. | C:\Documents and Settings\ *пользователь*\Templates\ |

**Табл. 13-1. Часто используемые свойства установщика *(окончание)***

| Свойство установщика | Описание | Пример |
|---|---|---|
| *Time* | Время начала установки файла .msi в формате ЧЧ:ММ:СС. | 17:54:35 |
| *UserLanguageID* | Идентификатор локализации (LCID), используемой пользователем. | 1033 |
| *USERNAME* | Имя учетной записи пользователя, под которой устанавливается файл .msi. | craigs |
| *VersionNT* | Версия используемой ОС, если система входит в семейство 32-разрядных NT. | 501 |
| *VirtualMemory* | Объем памяти (в Мб), назначенный странице виртуальной памяти. | 576 |
| *WindowsBuild* | Номер построения ОС. | 2600 |
| *WindowsFolder* | Папка, в которую установлена ОС. | C:\Windows\ |
| *WindowsVolume* | Диск, на котором установлена Windows. | C:\ |

Вы можете применять эти значения свойств установщика не только в редакторе Registry, но и в редакторе File System. В предыдущем разделе объяснялось, как определить пользовательскую папку, создаваемую при установке файла.msi, но путь к ней был жестко заданным. Свойства установщика позволяют задать путь к папке пользователя, который будет определен во время установки.

Допустим, вам надо создать папку для хранения специфичных для приложения данных. Свойству установщика *AppDataFolder* обычно задается значение *C:\Documents and Settings*\username*ApplicationData*\, указывающее путь к папке, в которой сохраняются специфичные для пользователя данные программы, такие как параметры настройки. Свойству установщика *ProductName* задается имя устанавливаемого продукта, определяемое по имени проекта установки. Вы можете задать место пользовательской папки, указав путь C:\Documents and Settings\username\Application Data\ProductName, но если пользователь установил ОС на диск D или изменил путь к данным приложений, это не будет правильным местом для хранения данных. Можно скомбинировать свойства установщика *AppDataFolder* и *ProductName* и установить для создаваемой папки свойство *DefaultLocation* в окне Properties равным *[AppDataFolder][ProductName]*. Эти действия приведут к автоматическому созданию папки C:\Documents and Settings\username\ApplicationData\Setup1 (где имя проекта установки — Setup1).

## Редактор File Types

Из всех технологий, встроенных в Windows, чтобы облегчить пользователям работу с компьютером, связывание файлов с приложением, вероятно, самое незаметное. В дни MS-DOS, если вы хотели просмотреть или отредактировать файл данных, нужно было знать, какое приложение может быть использовано для редактирования данного файла и как запустить это приложение для их просмотра. С Windows достаточно знать, как делать двойной щелчок, поскольку, когда вы дважды щелкаете значок в Windows Explorer, приложение, связанное с файлом автоматически запускается и загружает данные из файла. Связь файлов с прило-

жением создается путем создания набора разделов и значений системного реестра, связывающих расширение файла данных с программой просмотра/редактирования данного файла. Чтобы определить связи файлов для вашего проекта установки, можно использовать редактор Registry, но это довольно сложно. Редактор File Types в Visual Studio .NET предоставляет более простой способ задания связей между файлами и приложениями.

Допустим, вы создали новый способ хранения изображений в сжатом формате, который лучше, чем существующие форматы хранения изображений. Вы также создали программу .NET с названием MyImageViewer для просмотра, печати и редактирования файлов этого нового формата. Чтобы напечатать изображение, сохраненное в этом формате, пользователь должен открыть ваше приложение, выбрать команду File | Open, указать изображение (которое хранится в файле с расширением .myimage) и выбрать команду File ) Print. Либо он может позволить Windows проделать всю эту работу, используя связь файлов. Для нашего примера мы оставим теорию о форматах графических файлов другим книгам и задействуем в качестве нашего формата файлов обычные растровые изображения (.bmp), переименованные в файлы с расширением .myimage.

Чтобы создать связь для файлов .myimage из проекта установки, сначала откройте редактор File Types, дважды щелкнув проект установки в окне Solution Explorer и выбрав команду View | File Types. Щелкните правой кнопкой узел File Types On Target Machine и выберите Add File Type. Будет создана связь файлов для команды Open. Команда — это действие, которое вы можете проделать над файлом; команда по умолчанию (выделенная в редакторе File Types полужирным начертанием) — это действие, выполняемое по двойному щелчку значка файла в Windows Explorer. Кроме того, часто используются команды Print и Edit. Для нашего примера добавим обе эти команды.

Чтобы добавить команду Print, щелкните правой кнопкой созданный для нашего типа файла узел, выберите команду Add Action и введите **&Print**. Имени команды предшествует амперсанд, чтобы после отображения контекстного меню при щелчке правой кнопкой значка файла .myimage в Windows Explorer пользователь мог для печати просто нажать клавишу P. Затем надо указать, как команда будет сообщать вашей программе что она выбрана. Выберите &Print и в окне Properties в свойстве *Arguments* введите параметры командной строки для команды Print. Для нашего примера введите параметры командной строки *-print «%1»*; *-print* — это ключ командной строки, сообщающий вашей программе, что она должна печатать файл. Windows Explorer заменяет маркер *%1* на путь к файлу, который должен быть напечатан. Этот маркер нужно заключить в двойные кавычки. Если это не сделано и в пути к файлу есть пробелы, программе вместо одного параметра, содержащего путь к файлу, будут переданы две или более строк, что приведет к сбою программы при обработке команды. Затем в окне Properties введите имя команды **Print** в свойство *Verb*. Для данного примера следует создать еще одну команду, повторив описанные действия, но используя везде Edit вместо Print.

Теперь нужно указать в проекте установки, какая программа будет запускаться при выборе одной из этих команд. Выберите узел с вашим типом файла, созданный под узлом File Types On Target Machine, и откройте окно Properties. Свойство

*Command* определяет программу, которая будет запускаться при вызове команды; вы можете указать в качестве цели команды любой файл, добавленный в редактор File System, включая выходные данные проекта. Чтобы задать расширение, связываемое с программой, введите одно или несколько расширений, разделенных точкой с запятой, в свойство *Extensions*. Для нашего примера введите **myimage**; точка перед расширением не нужна. Если для вашего формата файлов вы хотите применить нестандартный значок, который будет показываться, когда файл отображается в Windows Explorer, добавьте значок в редактор File System и укажите этот значок в свойстве *Icon*.

И последний этап — модифицируем программу, чтобы она могла принимать передаваемые параметры командной строки. Если ваша программа — приложение Windows Forms и главная форма называется Form1, можно добавить в класс *Form1* конструктор:

```
public Form1(string []args)
{
    InitializeComponent();
    if(args.Length == 1)
    {
        pictureBox1.Image = System.Drawing.Bitmap.FromFile(args[0]);
        this.Text = this.Text + " - " +
                    System.IO.Path.GetFileName(args[0]);
    }
    else if (args.Length == 2)
    {
        // Два распознаваемых ключа командной строки:
        String printCommand = "-print";
        string editCommand = "-edit";
        if(System.String.Compare(printCommand, args[0], true) == 0)
        {
            // Нас попросили напечатать изображение. Загружаем его
            // и используем для печати класс PrintDocument:
            pictureBox1.Image = System.Drawing.Bitmap.FromFile(args[1] );
            PrintDocument printDocument = new PrintDocument();
            printDocument.PrintPage += new
                PrintPageEventHandler(printDocument_PrintPage);
            printDocument.Print();
            System.Diagnostics.Process.GetCurrentProcess().Kill();
        }
        else if(System.String.Compare(editCommand, args[0], true) == 0 )
        {
            // Нас попросили редактировать изображение.
            // Вызовем для редактирования MSPaint:
            System.Diagnostics.Process.Start("mspaint.exe", "\""
                                             + args[1] + "\"");
            System.Diagnostics.Process.GetCurrentProcess().Kill();
        }
    }
}
```

Зятем изменим функцию *Main* на:

```
static void Main(string []args)
{
    Application.Run(new Form1(args));
}
```

Когда вызываются команды Edit или Print, командная строка программы имеет вид *–edit filename*или *-printfilename*. Код анализирует параметры и, если указано *-print* или *–edit*, выполняет соответствующие действия: печать изображения или вызов mspaint.exe для отображения изображения, переданного в командной строке, и выход из программы. Если никакая команда не задана и пользователь хочет просмотреть изображение, файл загружается, и изображение отображается на форме. Пример MyImageViewer содержит полный исходный код программы для просмотра файлов .myimage и проект установки, включающий параметры для регистрации расширения файлов.

## Редактор User Interface

Когда устанавливается файл .msi, мастер проводит пользователя через процесс установки. Диалоговые окна мастера установки сообщают пользователю, какую программу он устанавливает, запрашивает имя папки на диске, в которую устанавливается программа, и обеспечивает обратную связь. Кроме того, инструменты установки, встроенные в Visual Studio .NET, позволят вам добавлять диалоговые окна, запрашивающие дополнительную информацию о параметрах установки программы.

Вы можете настроить любое диалоговое окно в проекте установки, модифицируя два свойства в окне Properties: *BannerText* и *BannerBitmap*. Первое задает отображаемый в верхней части диалогового окна текст, который описывает этап, выполняемый мастером установки. Второе задает файл с растровым изображением, показываемый в верхней части диалогового окна; это изображение должно иметь размеры 496x68 точек, и файл с ним должен быть добавлен в редактор File System. Вы можете просмотреть изображение любым способом, но помните, что текст, заданный в свойстве *BannerText,* будет выведен поверх изображения, так что следует использовать цвета, позволяющие увидеть выводимый текст.

В редакторе User Interface две ветви дерева: Install и Administrative Install. Большая часть работы по проектированию пользовательского интерфейса проекта установки выполняется в ветви Install. Диалоговые окна этой ветви образуют пользовательский интерфейс, который будет видеть большинство пользователей; они проводят через процесс установки приложения. Ветвь Administrative Install предназначена для системных администраторов. Если администратор запускает файл .msi с параметром командной строки *–a*, скажем, *msiexec.exe -a msifile.msi*,файлы, содержащиеся в файле .msi, будут установлены так, чтобы пользователи в сети могли выполнять сетевую установку программы. В ветви Administrative Install доступно только подмножество диалоговых окон, доступных в ветви Install. Вносить изменения в ветвь Administrative Install редактора User Interface обычно не требуется; Windows Installer учтет все особенности административной установки за вас.

### Рекламный экран

Страница с рекламным экраном (рис. 13-2) должна просто показать пользователю изображение, идентифицирующее устанавливаемую программу. Это должно быть растровое изображение или файл JPG, размером 480x320 точек. Чтобы задать изображение, отображаемое в этом диалоговом окне, его надо сначала добавить в соответствующую папку в редакторе File System, выбрать диалоговое окно Splash в редакторе User Interface и установить свойство *SplashBitmap* в окне Properties, чтобы оно указывало на файл изображения, который вы добавили в редактор File System.

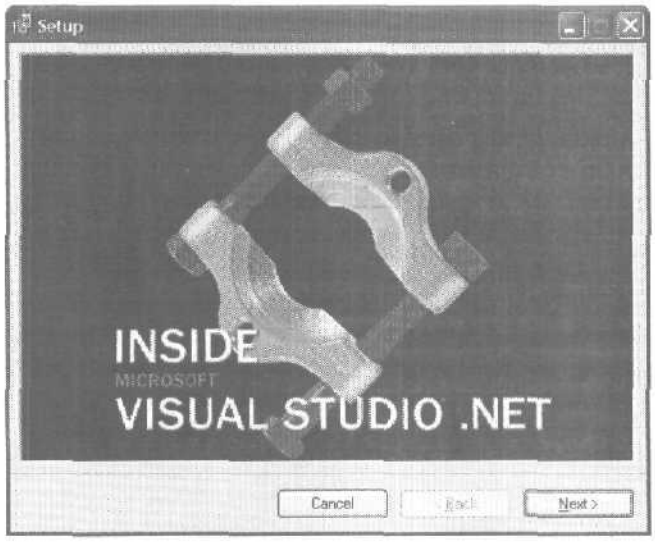



Рис. 13-2. *Рекламный экран проекта установки, отображающий обложку книги «Inside Microsoft Visual Studio .NET»*

### Диалоговые окна выбора

Диалоговые окна выбора — RadioButtons (2 Buttons), RadioButtons (3 Buttons), RadioButtons (4 Buttons), Checkboxes (A), Checkboxes (B) и Checkboxes (C) — позволяют вам предложить пользователю различные варианты установки. Диалоговые окна RadioButtons отображают 2, 3 или 4 переключателя, один из которых может выбрать пользователь; диалоговые окна Checkboxes отображают от 0 до 4 флажков (рис. 13-3).

В окне Properties вы можете задавать параметры для каждого переключателя или флажка этих диалоговых окон. Вы можете определить состояние по умолчанию (выбран/не выбран, отмечен/не отмечен), имя свойства установщика и метку. Наиболее интересно имя свойства установщика, обозначаемое в окне Properties как *ButtonProperty* для диалоговых окон RadioButtons и как *CheckBoxXProperv* (где X — число от 1 до 4) для диалоговых окон Checkboxes. Вы можете использовать значения этих свойств в других редакторах, скажем, в редакторе Registry, как имя ключа, имя значения или значение, так же, как используете свойства установщика из табл. 13-1.

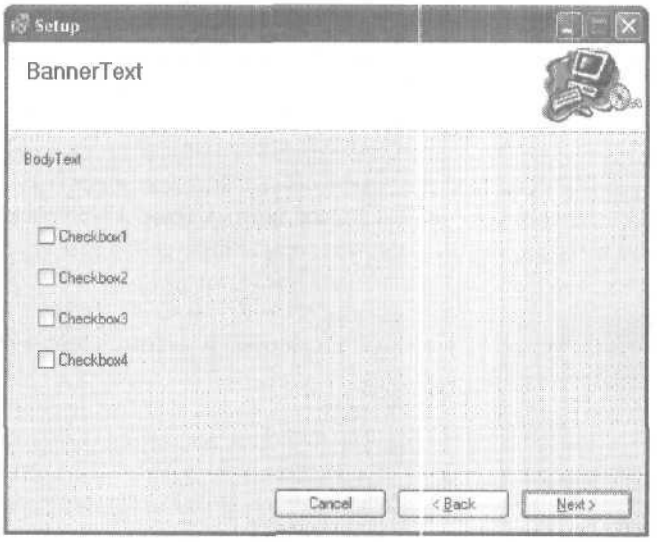



Рис. 13-3. *Одно из диалоговых окон Checkboxes*

Вы можете задействовать значение параметра в диалоговом окне RadioButtons (рис. 13-4) в редакторе Registry так же, как значение флажка. Одновременно может быть выбран только один переключатель, поэтому для каждого диалогового окна доступно только одно свойство; значение *ButtonXValue* применяется, когда установщик заполняет системный реестр. Так, у диалогового окна RadioButtons (2 Buttons) есть два свойства, *Button1Value* и *Button2Value*, которым заданы значения *1* и *2* соответственно. Свойство, задающее имя параметра установщика для этих переключателей, — *BUTTON2*. Таким образом, вы можете в редакторе Registry задействовать значение *[BUTTON2]* для имени раздела реестра, имени значения или значения. Если выбран первый переключатель (*Button1*), значением, помещаемым в реестр, будет *1*; если второй (*Button2*) — *2*.

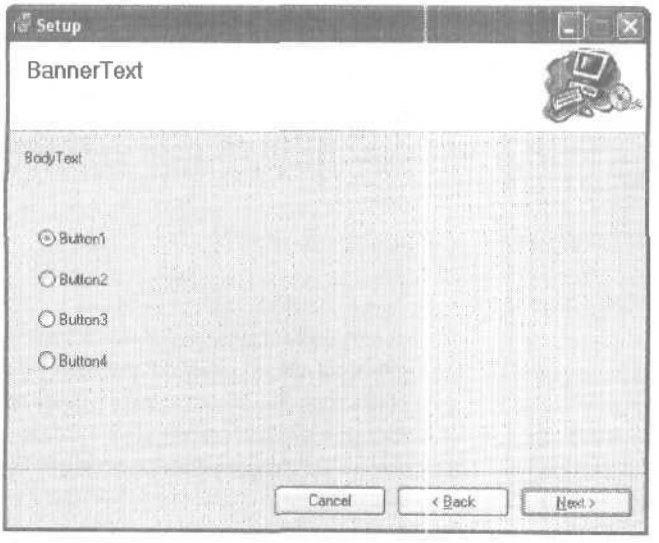



**Рис. 13-4.** *Диалоговое окно RadioButtons (4 Buttons)*

Далее Е этой главе мы обсудим, как использовать настраиваемые диалоговые окна для условной установки разделов реестра и файлов, помещенных в редакторы Registry и File System.

### Диалоговые окна ввода данных

Диалоговые окна ввода данных — Textboxes (A), Textboxes (B), and Textboxes (C) - применяются для получения от пользователя текстовых данных (рис. 13-5). Каждое диалоговое окно ввода данных содержит четыре текстовых поля. Если вам нужно отображать, например, только два текстовых поля, вы можете скрыть остальные текстовые поля, выбрав соответствующее диалоговое окно ввода данных в редакторе User Interface, а затем установив в окне Properties значение свойства *EditXVisible* (где X — номер текстового поля) в *False*. У каждого текстового поля в диалоговом окне есть имя, отображаемое в окне Properties в свойстве *EditXProperty*. Вы можете использовать значение этого свойства, заключенное в квадратные скобки, в редакторе Registry точно так же, как другие свойства установщика. Например, значение свойства *Edit1Properly* для первого текстового поля диалогового окна Textboxes (A) равно *EDITA1*, так что вы можете указать имя ключа реестра, имя значения или значение *[EDITA1]*, чтобы представить текст, введенный пользователем в этом текстовом поле.

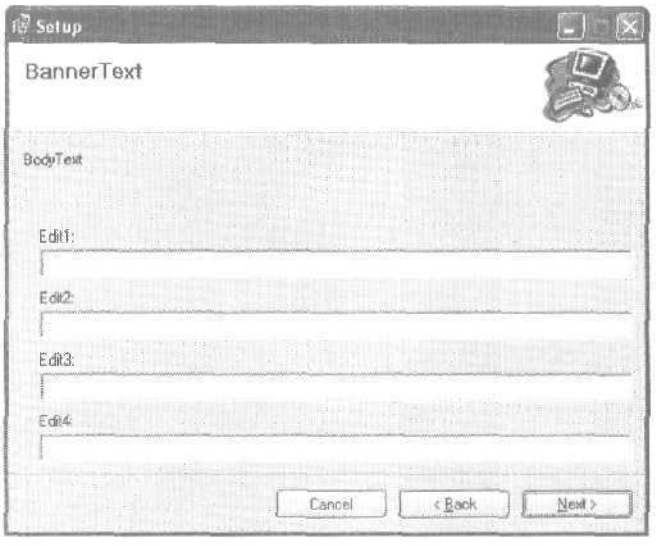



**Рис. 13-5.** *Одно из диалоговых окон TextBoxes*

### Диалоговое окно Customer Information

Диалоговое окно Customer Information служит для сбора данных о пользователе: имени, названия его компании и (необязательно) серийного номера для программы, Для проверки правильности введенного пользователем серийного номера выполняется алгоритм, базирующийся на значении свойства *SerialNumberTemplate* в окне Properties. Значение свойства *SerialNumberTemplate* создает интерфейс пользователя для ввода серийного номера в диалоговом окне Customer Information.

Чтобы определить серийный номер, который пользователь может ввести, создается шаблон, представляющий собой строку, включающую специальные токены. Шаблон заключается в угловые скобки. Между скобками можно разместить произвольное количество символов #, %, ?, ^и -. Символы # и % отмечают местоположение цифр, а ? и ^ — местоположение алфавитно-цифровых символов, причем ^ обозначает символ верхнего регистра. Когда в шаблоне встречается символ «тире», в диалоговом окне создается новое текстовое поле. Правильность введенного номера определяется путем сложения цифр, находящихся на местах, отмеченных в шаблоне знаком %, и деления суммы на 7. (Все остальные символы шаблона игнорируются.) Если остаток от деления равен 0, введенный номер признается правильным, и пользователь может продолжить установку, иначе выводится сообщение об ошибке, и пользователь должен повторно ввести серийный номер или выйти из программы установки. В диалоговом окне на рис. 13-6 применяется шаблон серийного номера <###-%%%%%%>; введенный номер неверен, так как сумма чисел 4, 5, 6, 7, 8, 9 и 9 равна 48 и не делится нацело на 7. После того как серийный номер признан правильным, его значение сохраняется в свойстве установщика *ProductID*.

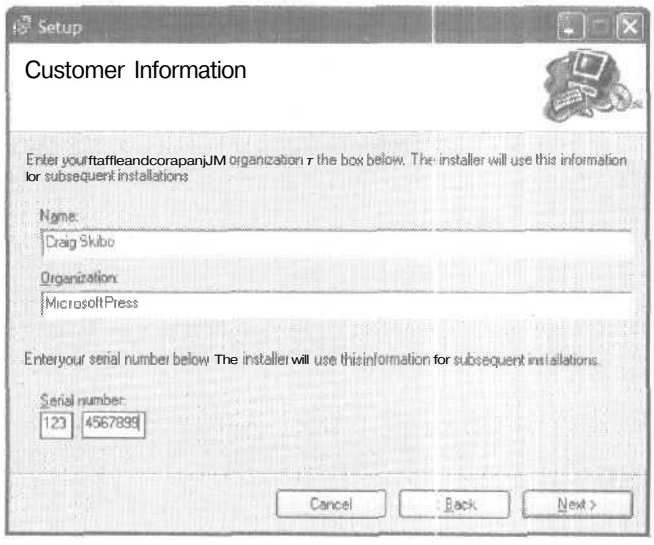



Рис. 13-6. *Диалоговое окно Customer Information*

### Диалоговые окна LicenseAgreement и Read Me

Почти каждая программа поставляется с юридическими ограничениями, определяющими, как она может и как не может быть использована. Эти ограничения Б форме лицензионного соглашения обычно информируют пользователя о запрете нелегального копирования программы, запрете дизассемблирования и т. д. Информацию лицензионного соглашения пользователю представляет диалоговое окно License Agreement; установку ПО нельзя будет продолжить, пока пользователь не укажет, что принимает лицензионное соглашение (рис. 13-7). Если пользователь выбирает переключатель I Do Not Agree option, кнопка Next становится неактивной. Выбор переключателя ] Agree накладывает юридические обязатель-

ства; пользователь, принявший лицензионное соглашение может продолжить установку.

Текст, отображаемый в диалоговом окне License Agreement, создается в RTF-файле в Microsoft Word или WordPad. Созданный RTF-файл нужно поместить в диалоговое окно License Agreement через редактор File System. Для этого сначала добавьте диалоговое окно License Agreement в редакторе User Interface, выберите его, а затем в окне Properties в свойстве *LicenseFile* укажите этот файл RTF.

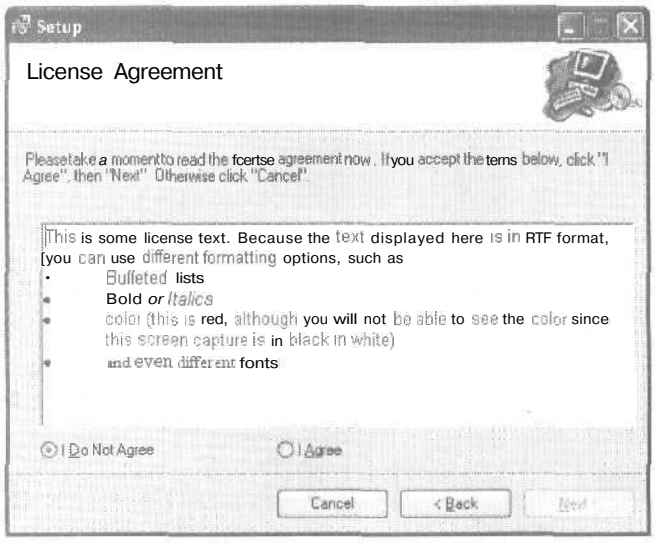



Рис. 13-7. *Диалоговое окно License Agreement с форматированным текстом*

Диалоговое окно Read Me выглядит и работает почти так же, как License Agreement, за исключением отсутствия переключателей I Do Not Agree и I Agree. Текст для диалогового окна Read Me, как и для License Agreement, определяется с помощью файла RTF и может содержать форматирование с использованием цветов, шрифтов и стилей.

### Диалоговое окно Register User

Многие пакеты программ просят указать персональные данные пользователя, такие как имя и адрес электронной почты, чтобы при выходе новых версии или исправлении ошибок производитель программы мог прислать информацию об этих обновлениях. Для сбора такой информации вы можете добавить в ваш проект установки диалоговое окно Register User (рис. 13-8). Оно не содержит элементов, в которых пользователь мог бы ввести имя, адрес, адрес электронной почты или другие сведения; вместо этого оно содержит кнопку, щелчок которой вызывает произвольную программу, добавленную в ваш проект установки в редакторе File System.

Чтобы связать программу регистрации с кнопкой Register Now, вы должны сначала создать программу регистрации и добавить выходные данные ее проекта к проекту установки в редакторе File System. Затем в редакторе User Interface выберите диалоговое окно Register User и в окне Visual Studio .NET Properties укажите исполняемый файл в свойстве *Executable*.

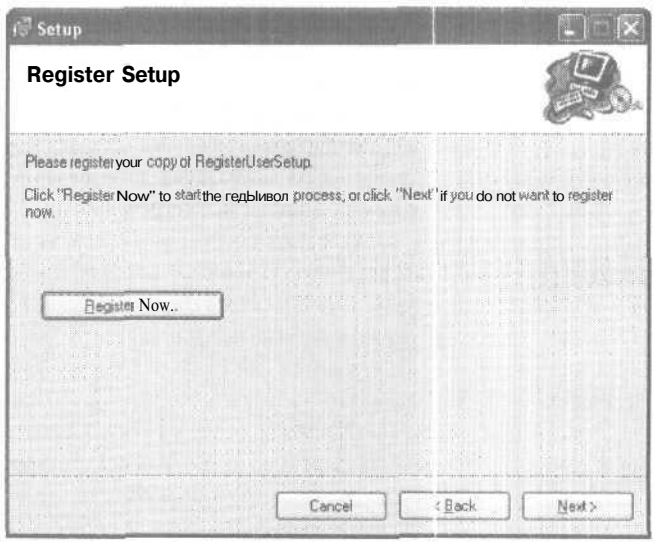



Рис. 13-8. *Диалоговое окно Register User*

Создать программу регистрации нетрудно — просто запустите мастер C# Windows Application Wizard и используйте исполняемый файл. Вы можете найти исходный код проекта регистрации среди примеров к книге. Этот проект, RegisterUser, собирает соответствующую информацию от пользователя. Другой проект, ProductRegistration, является Web-приложением .NET, которое можно установить на Web-сервере. Когда запущена программа RegisterUser и пользователь щелкает кнопку Register Online, введенная информация упаковывается в заголовок запроса HTTP и отправляется Web-приложению ProductRegistration. Web-приложение распаковывает информацию из заголовка запроса и посылает обратно проекту RegisterUser сообщение, показываемое пользователю.

Чтобы использовать примеры RegisterUser и ProductRegistration, установите пример ProductRegistration на Web-сервер и добавьте выходные данные проекта RegisterUser к вашей программе установки. В коде ProductRegistration откройте файл фонового кода для файла ProductRegistration.aspx и найдите комментарий TODO в конце этого файла. Вы должны заменить его на собственный код, сохраняющий информацию о пользователе для дальнейшего применения, скажем, в базе данных. Вы можете также изменить возвращаемое пользователю текстовое сообщение, приспособив его к своим потребностям. Второй этап — модификация файла Form1.cs проекта RegisterUser с целью добавить указание на сервер, содержащий Web-приложение ProductRegistration. Для этого найдите строку *localhost* и измените ее, чтобы указать сервер и виртуальный каталог, в который установлено Web-приложение ProductRegistration,

## Нестандартные действия

Windows Installer предоставляет много возможностей, помогающих установить приложение, но иногда, чтобы разместить программу на компьютере пользователя, в процессе установки нужно выполнить какой-то код. Этот вспомогательный

код называется нестандартным действием. В главе 7 мы кратко ознакомились с нестандартными действиями, позволяющими создать и удалить команды для Visual Studio .NET, но вы можете создавать нестандартное действие, выполняющее любую работу. Для проекта установки можно построить нестандартные действия трех типов: нестандартные действия .NET, сценарии нестандартных действий и нестандартные действия Win32. В примеры к книге включен CustomActions, демонстрирующий создание и использование нестандартных действий всех трех типов.

Нестандартные действия добавляются в редакторе Custom Actions, для чего выходные данные проекта, реализующего нестандартные действия, надо добавить в редактор File System, а затем щелкнуть правой кнопкой соответствующее действие установки в редакторе Custom Actions и указать выходные данные проекта. Определены четыре действия установки: Install, Commit, Rollback и Uninstall. К какому добавляется ваше нестандартное действие, зависит от работы, которое оно выполняет. Нестандартное действие Install вызываются в начале установки файла .msi. Если при установке происходит ошибка, вызывается нестандартные действия из группы Rollback. Нестандартные действия этой группы могут применяться для восстановления состояния компьютера и удаления данных, таких как файлы и разделы реестра, созданные действием Install. Если установка прошла успс шно, вызываются нестандартные действия из группы Commit, позволяющие завершить установку на компьютер. При удалении программы вызываются нестандартные действия из группы Uninstall, предоставляющей вашим нестандартным действиям шанс удалить все данные, которые могут остаться после вашей программы.

### Нестандартные действия .NET

Вы можете построить нестандартное действие, используя .NET Framework с исполняемой программой, такой как приложение Windows Forms или консольное приложение, или с библиотекой классов .NET, производной от базовой библиотеки классов .NET. Выбор типа создаваемого нестандартного действия определяется способом взаимодействия пользователя с нестандартным действием и тем, насколько легко вы можете разработать и отладить это нестандартное действие.

Если вы хотите, чтобы ваше нестандартное действие выводило интерфейс пользователя, лучше всего создать нестандартное действие Windows Forms. Вы просто добавляете приложение Windows Forms к своему решению, добавляете выходные данные этого проекта к проекту установки в редакторе File System, затем добавляете это действие к редактору Custom Actions и выбираете этап установки, на котором надо выполнить нестандартное действие. Когда запустится выбранный этап, будет вызвана программа нестандартного действия, позволяя пользователю взаимодействовать с интерфейсом пользователя, отображаемым программой. Создание нестандартного действия Windows Forms — тоже хороший выбор для упрощения разработки и отладки нестандартного действия. Пси кольку нестандартное действие Windows Forms является программой, вы можете выполнить, проверить и отладить его, не выполняя построение и установку файла .msi, что увеличивает производительность. Создание нестандартного действия с приложением Windows Forms — не лучшее решение, если вам не нужен интерфейс пользователя, так как форма блокирует процесс установки до ее закрытия, а отображать диалоговое окно с сообщением «Щелкни меня, чтобы продолжить установку» — просто плохой стиль.

Второй вариант — создать для нестандартного действия консольное приложение. Такое нестандартное действие предоставляет преимущества нестандартного действия Windows Forms в плане простоты проверки и отладки и отображает интерфейс пользователя. В зависимости от работы, выполняемой нестандартным действием, интерфейс пользователя может быть как благословением, так и проклятием. Пользователи не любят окон консоли — они бедно выглядят и сложны для работы. Если созданное нестандартное действие выполнит свою работу быстро, на экране промелькнет окно консоли, и пользователь может задаться вопросом: что программа установки делает на его компьютере? Однако, если вам нужно вывести текстовую информацию, такую как выходные данные другого консольного приложения, нестандартное действие с консольным приложением — хороший выбор. Если вы создаете нестандартное действие, применяя Windows Forms или консольное приложение и язык, поддерживаемый .NET Framework, нужно выбрать нестандартное действие в редакторе Custom Actions и изменить значение свойства *InstallerClass* в окне Properties с *True* на *False*. Если свойство равно *False*, Windows Installer знает, что должен вызвать нестандартное действие как программу, если же *True* — он ищет программу для класса со специфическим атрибутом, который используется третьим классом нестандартных действий .NET— библиотекой классов .NET.

Библиотека классов .NET подойдет, если ваше нестандартное действие должно выполняться незаметно в фоновом режиме без взаимодействия с пользователем. Нестандартные действия такого типа наиболее сложны для проверки и отладки, так как библиотека классов не может выполняться отдельно, без основного приложения. Чтобы создать нестандартное действие с библиотекой классов, создайте библиотеку классов, используя ваш любимый язык программирования с поддержкой .NET, щелкните этот проект правой кнопкой в Solution Explorer, выберите команду Add | Add New Item и добавьте к проекту элемент Installer Class. Добавленный элемент — это класс, производный от класса *System.Configuration.Install.Installer* и использующий атрибут *RunInstaller*. Когда проект .msi выполняется и запускает нестандартное действие, он проверяет все классы в сборке, реализующей нестандартное действие; если найден класс с атрибутом *RunInstaller*, создается экземпляр класса и вызывается соответствующий метод класса. Класс *System.Configuration.Install.Installer* определяет четыре метода, которые вы можете переопределить и называются соответственно четырем этапам установки: *Commit*, *Install*, *Rollback* и *Uninstall*. Вы можете добавить их к классу из окна Class View.

### Сценарии нестандартных действий

Создание нестандартного действия с использованием кода сценария не требует дополнительных действий, кроме создания файла VBScript или JScript, добавления этого файла в редактор File System и добавление его в качестве нестандартного действия в редактор Custom Actions, Сценарий нестандартного действия — это просто список команд, загружаемых и выполняемых движком ActiveX Scripting. Когда начинается выполнение этого сценария нестандартного действия, создается одна глобальная переменная типа *Session* с именем *Session*, позволяющая получить сведения об устанавливаемом в данный момент файле .msi. Листинг 13-1 демонстрирует простое нестандартное действие VBScript, а 13-2 — его эквивалент на

JScript. Нестандартное действие всего лишь выводит пользователю окно с сообщением о наличии данного нестандартного действия.

**Листинг 13-1.** Исходный код нестандартного действия на VBScript

VBScriptCustomAction.vbs

```
' Можно использовать объект Session для доступа
' к устанавливаемому файлу NSI. Смотрите раздел MSDN
'  ms-help://MS.VSCC.2003/MS.MSDNQTR.2003FEB.1033/msi/vref_8xis.htm
' для получения информации об использовании этого объекта.
msgbox("VBScript Custom Action")
```

**Листинг 13-2.** Исходный код нестандартного действия на JScript

JScriptCustomAction.js

```
// Можно использовать объект Session для доступа
// к устанавливаемому файлу MSI. Смотрите раздел MSDN
//  ms-help://MS.VSCC.2003/MS.MSDNQTR.2003FEB.1033/msi/vref_8xis.htm    ;  -
// для получения информации об использовании этого объекта.
var wshShell = new ActiveXObject("WScript.Shell");
wshShell.Popup("JScript Custom Action");
```

Вы можете выбрать создание сценария нестандартного действия по нескольким причинам. Во-первых, у вас может быть готовая библиотека сценариев для настройки параметров компьютера, написанная на VBScript или JScript. Вместо того чтобы переписывать их для создания нестандартных действий .NET, можно просто добавить сценарий к проекту установки и выполнить его. Другая причина для создания сценария нестандартного действия — простота разработки. Сценарии представляют собой интерпретируемые строки текста, и вам не надо компилировать их. Чтобы создать и проверить сценарий нестандартного действия, достаточно просто открыть любой текстовый редактор (даже Notepad), написать код и затем выполнить код сценария, дважды щелкнув файл в Windows Explorer.

### Нестандартные действия Win32

Если устанавливаемое приложение не обращается к .NET и на компьютере пользователя не установлена .NET Framework, можно выбрать либо сценарий, либо нестандартные действия Win32. Сценарии нестандартных действий просты в написании, но не предоставляют полного доступа к Windows API и поэтому могут не подойти. Единственный остающийся вариант — использовать язык подобный Visual C++ для создания нестандартного действия Win32. Чтобы создать нестандартное действие Win32, нужно сначала на Visual C++ создать Win32-DLL и экспортировать функцию, используя соглашение вызова *__stdcall*. Экспортируемая функция должна возвращать значение типа *unsigned int,* являющееся кодом состояния. Если нестандартное действие возвращает значение, отличное от *ERROR_SUCCESS*, Windows Installer считает, что произошла ошибка, и откатывает установку. Экспортируемая функция получает один аргумент — значение типа *MSIHANDLE*, которое

может применяться для получения информации из проекта установки. Чтобы указать Windows Installer, какие экспортируемые функции DLL должны вызываться при выполнении нестандартного действия, нужно выбрать нестандартное действие в редакторе Custom Actions и задать значение свойства *EntryPoint* в окне Properties. Если бы вы писали нестандартное действие вроде показанного в листинге 13-3, вы должны были бы указать Install как значение свойства *EntryPoint*, так как это экспортируемая функция, обрабатывающая обращение к нестандартному действию.

**Листинг 13-3.** Исходный код нестандартного действия Win32

**Win32CustomAction.cpp**

```
#include "stdafx.h"
#include <tchar.h>
#include <msi.h>
#include <Msiquery.h>

BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
    return TRUE;
}

__declspec(dllexport) unsigned int __stdcall Install(MSIHANDLE hInstal 1)
{
    MessageBox(NULL, "CustomAction", "VC++", MB_OK);
    return ERROR_SUCCESS;
}
```

### Практикум: отладка нестандартных действий

Отладить нестандартные действия, когда они запускаются из файла .msi, не так-то просто. Файл .msi не является исполняемым кодом — это набор сжатых файлов с данными, описывающими, **как** эти файлы должны быть установлены. Для отладки нестандартного действия вы должны запустить установку файла .msi и подключить отладчик точно в нужное время — после того, как *код* начал выполняться, но до того, как выполнится код, нуждающийся в отладке. А подобная синхронизация практически невозможна. Для отладки нестандартных действий я использую один трюк: помещаю в нестандартное действие, как раз перед отлаживаемым фрагментом, код для вывода окна сообщения, после чего выполняю построение и установку файла .msi. Когда выводится окно сообщения, я знаю, что должен для начала отладки подключить отладчик. Выполнение нестандартного действия приостанавливается, пока отображается окно, что позволяет подключить отладчик в правильный момент времени. Но к какой программе подключить отладчик? Это зависит от типа созданного нестандартного действия. Если нестандартное действие — библиотека классов .NET, то, открыв диалоговое окно Processes

отладчика, вы увидите три запущенных процесса msiexec.exe, к которым можно подключиться. В диалоговом окне у одного из этих процессов в колонке Type отображается слово .NET. К этому процессу и нужно подключаться, так как он загрузил .NET Framework и выполняет ваше нестандартное действие.

Если созданное нестандартное действие — приложение .NET Windows Forms или консольное приложение .NET, вы увидите перечисление имен этих программ » диалоговом окне Processes и можете подключиться к ним, не подключаясь к процессу msiexec.exe. Чтобы отладить нестандартное действие Win32, надо найти в *диалоговом* окне Processes процесс msiexec.exe. д. ЕЯ которого в столбце Title отображается тот же заголовок, что я у вашего окна сообщения. В настоящее время нет способа для отладки сценариев нестандартных действий; вы должны применять альтернативные методы отладки, такие как вывод окон сообщений по мере выполнения кода.

По завершении отладки не забудьте удалить окна сообщений, чтобы пользователь не видел их при установке файла .msi.

## Редактор Launch Conditions

Иногда для запуска ПО необходимо выполнение определенных требований. Допустим, вы используете преимущества последних технологий Windows XP, а значит, пользователь может выполнять вашу программу только в этой ОС. Или же ваша программа — надстройка и не может выполняться без Visual Studio .NET. Вы можете написать нестандартное действие, проверяющее во время установки соблюдение этих требований, но лучше позволить файлу .msi убедиться в соблюдении требований до помещения файлов на компьютер пользователя.

Вы можете определить эти требования в проекте установки, задав условия запуска в редакторе Launch Conditions. Чтобы добавить условие запуска откройте редактор Launch Conditions, щелкните правой кнопкой узел Launch Conditions и выберите команду Add Launch Condition. В окне Properties в свойстве *Condition* введите выражение, при вычислении которого должно получаться значение *true*, чтобы файл .msi был установлен. Если значение выражения не равно *true,* пользователь не сможет установить вашу программу и увидит сообщение об ошибке, содержащееся в свойстве *Message.* Если значение свойства *InstallURL* условия отличается от пустой строки и значение выражения равно *false,* пользователю будет предложено перейти на Web-страницу, чтобы получить дополнительную информацию о причинах невозможности установки. Чтобы Windows Installer мог вычислить выражение условия, требуется специальный синтаксис — алгебра условий.

### Алгебра условий

Чтобы описать условие в выражении, нужно применять похожий на Visual Basic .NET синтаксис. Переменные, которые вы можете использовать в выражениях, — это те же свойства установщика из табл. 13-1 или те, что находятся в диалоговых окнах редактора User Interface. Однако, когда вы определяете выражение, не надо

заключать имя свойства установщика в квадратные скобки, как в окнах других редакторов.

В табл. 13-2 перечислены операторы, которые можно использовать в выражениях. Эти операторы в комбинации со строковыми и целочисленными константами (сравнение с плавающей точкой не поддерживается) и именами свойств установщика образуют алгебру, применяемую при создании условий.

**Табл. 13-2. Операторы алгебры условий**

| Оператор | Описание |
|---|---|
| *Not* | Логическое отрицание терма. |
| *And* | *True*, если значения двух термов равны *True*, иначе *False*. |
| *Or* | *True*, если значение хотя бы одного из термов равно *True*. |
| *Xor* | *True*, если значение только одного из термов равно *True*, |
| *Eqv* | Оператор логического равенства. *True*, если значения обоих термов равны *True* или если значения обоих равны *False*. |
| *Imp* | Оператор импликации. *True*, если значение левого терма равно *False* или значение правого термина равно *True*. |
| = | Оператор равенства. *True*, если оба терма равны; иначе *False*. |
| <> | *True*, если два терма не равны. |
| > | *True*, если левый терм больше, чем правый. |
| >= | *True*, если левый терм больше, чем правый, или равен ему. |
| < | *True*, если левый терм меньше, чем правый. |
| <= | *True*, если левый терм меньше, чем правый, или равен ему. |
| >< | Поразрядный оператор *And*. *True*, если любые биты в двух термах совпадают. |
| >< | Оператор сравнения строк. *True*, если левая строка содержит правую строку. |
| << | Оператор поразрядного сравнения. *True*, если старшие 16 бит левого целочисленного терма равны правому целочисленному терму. |
| << | Оператор сравнения строк. *True*, если левая строка начинается с правой строки. |
| >> | Оператор поразрядного сравнения. *True*, если младшие 16 бит левого целочисленного терма равны правому целочисленному терму, |
| >> | Оператор сравнения строк. *True*, если левая строка заканчивается правой строкой. |

Приведем несколько примеров применения этих операторов и свойств установщика в свойстве *Condition*.

- ***Not Privileged*** *True*, если пользователь не работает с повышенными привилегиями.
- ***(VersionNT = 501) And (ServicePackLevel = 1)*** *True*, если файл .msi устанавливается в Windows XP (версия 501) с установленным Service Pack 1.
- ***(VersionNT = 500) Or (VersionNT = 501)*** *True*, если файл .msi устанавливается в Windows 2000 (версия 500) или Windows XP (версия 501).
- ***(ScreenX >= 800) And (ScreenY >= 600)*** *True*, если у пользователя установлено разрешение экрана 800x600 или выше.
- ***PhysicalMemory > 128*** *True*, если на компьютере установлено более 128 Мб оперативной памяти,

- ***PhysicalMemory>= 128*** *True,* если на компьютере установлено 128 Мб оперативной памяти или более.
- ***«Hello World» >< «Hello»*** *True,* если строка справа входит в первую строку.
- ***«Hello World» << «Hello»*** *True,* если строка, расположенная слева, начинается со строки, расположенной справа.
- ***«Hello World» >> «World»*** *True,* если строка расположенная слева, заканчивается строкой, расположенной справа.
- ***Intel > 4*** *True,* если процессор компьютера — Intel Pentium или более поздняя модель. Данное правило полезно, если ваше ПО компилировалось только для процессоров Pentium (или совместимых).
- ***Intel = «5»*** *True,* если процессор компьютера — Intel Pentium. Это выражение похоже на предыдущее за исключением того, что число 5 помещено в кавычки, оператор «больше чем» не может использоваться, так как нельзя применять строки в выражениях, содержащих операторы сравнения чисел.
- ***BUTTON2 = 1*** *True,* если в редакторе User Interface было добавлено диалоговое окно RadioButtons (2 Buttons) и пользователь выбрал в этом окне первый переключатель.

### Определение настраиваемых свойств установщика

Вы видели использование свойств установщика в редакторах Registry, Launch Conditions, User Interface и File System. Однако эти свойства установщика, определенные либо в Windows Installer, либо в диалоговых окнах проекта установки, могут не соответствовать вашим нуждам. Редактор Launch Conditions позволяет создать настраиваемые свойства установщика, которые можно использовать везде, где допускается применение свойств установщика. Чтобы создать настраиваемое свойство установщика, щелкните правой кнопкой узел Search Target Machine в редакторе Launch Conditions и выберите команду File Search или Registry Search.

Свойство установщика File Search выполняет поиск файла на компьютере, и если файл найден, значением свойства становится путь к файлу. Чтобы задать искомый файл, добавьте условие запуска File Search, а затем укажите папку, с которой будет начат поиск и имя искомого файла (в окне Properties в свойствах *Folder* и *FileName* соответственно). Если вы знаете, что искомый файл может находиться в одной из подпапок указанной папки, нужно установить значение свойства *Depth* равным количеству уровней иерархии папок, в которых будет производиться поиск.

Допустим, вам нужно установить в свойстве установщика путь к файлу dte.olb, содержащему библиотеку типов для объектной модели Visual Studio .NET. Поскольку Visual Studio .NET устанавливает этот файл в папку Program Files, задайте свойству *FolderName* условия запуска File Search значение *[ProgramFilesFolder]*, свойству *File* — *dte.olb* и свойству *Depth* — *20* (произвольное число, гарантирующее, что все нужные папки будут просмотрены). Если имя свойства установщика, заданное для условия запуска с поиском файлов, — *FILEEXISTS1* (задаваемое по умолчанию свойство установщика для первого созданного условия запуска с поиском файлов), вы можете задействовать свойство установщика *FILEEXISTS1* в редакторах File System, Registry или в любом другом месте, где вам нужен путь к файлу dte.olb и допускается применение свойств установщика.

Условие запуска Registry Search работает во многом аналогично условию запуска File Search за исключением того, что поиск выполняется в реестре, а не на диске пользователя и свойством установщика становится значение из реестра, а не путь к файлу. Допустим, перед установкой надстройки вы хотите проверить, что на компьютере есть Visual Studio .NET 2003. Это позволяет сделать условие Registry Search. Сначала создайте условие Registry Search, щелкнув правой кнопкой узел Search Target Machine, выбрав команду Add Registry Search и введя имя для условия. В окне Properties укажите в свойстве *RegKey* раздел реестра для Visual Studio .NET (**SOFTWARE\Microsoft\VisualStudio\7.1**), а в свойстве *Value* введите **InstallDir**. Значение реестра *InstallDir* содержит место на диске, куда установлена Visual Studio .NET. Значения других свойств можно не менять, но запомните значение свойства *Property*. Это свойство установщика, которое вам понадобится для создания условия. (По умолчанию для первого условия поиска в реестре это значение равно *REGISTRYVALUE1*.) Затем создайте условие, щелкнув правой кнопкой узел Launch Conditions в редакторе Launch Conditions и выбрав команду Add Launch Condition. Откройте окно Properties, введите **REGISTRYVALUE1 О ""** в свойство *Condition*, а затем — ваше сообщение в свойства *Message* и *InstallURL*. Когда будет запущен файл .msi для надстройки, он создаст свойство установщика с именем *REGISTRYVALUE1* и значением, равным местоположению установленной Visual Studio .NET. Если значение этого свойства отличается от пустой строки, выражение равно *True,* и установка продолжается, иначе выводится сообщение, заданное в свойстве *Message*.

### Установка сборки в папку PublicAssemblies

Как уже не раз говорилось, если вы строите сборку, которую хотите вызывать из макросов, ее надо поместить в определенную папку, чтобы в редакторе Macros можно было добавить ссылку на эту сборку. По умолчанию местоположение этой папки C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\PublicAssemblies, но вы не можете быть уверены в правильности этого пути, так как пользователь мог задать другое место для установки Visual Studio .NET. Можно использовать настраиваемое свойство установщика *Registry Search*, чтобы сначала проверить, установлена ли Visual Studio .NET, а затем установить сборку в требуемую папку, чтобы она могла применяться в макросах.

Чтобы поместить сборку в нужное место, откройте редактор File System, щелкните правой кнопкой узел File System On Target Machine и выберите команду Add Special Folder | Custom Folder. В окне Properties этой пользовательской папки введите имя настраиваемого свойства установщика *Registry Search,* заключенное в квадратные скобки (в данном случае *[REGISTRYVALUE1]*), а затем введите **\PublicAssemblies** в свойство *DefaultLocation*. Затем добавьте выходные данные проекта сборки в эту пользовательскую папку. При установке файла .msi сборка будет помещена в требуемое место, и на нее можно будет ссылаться из редактора макросов,

### Условия

Работая с редакторами File System, Registry, Custom Actions или File Types, вы могли заметить в окне Properties свойство *Condition*. Оно определяет, будет ли выходной файл проекта, раздел реестра или тип файлов установлен на компьютер или будет выполнено нестандартное действие. При установке значения *Condition* используется та же алгебра условий, что и при создании условий запуска. Допустим, вы хотите разрешить пользователю выбирать установку исходного кода вашей надстройки. Чтобы сделать это, запустите мастер Add-in Wizard и после создания исходных файлов откройте редактор User Interface для проекта установки. Добавьте в редакторе User Interface диалоговое окно RadioButtons (2 Buttons), откройте окно Properties и задайте свойству *Button1Label* значение **Yes, install source code,** а свойству *Button2Label* — **No, do not install source code.** Заметьте: имя свойства установщика для этого диалогового окна — *BUTTON2*, а значения для переключателей в диалоговом окне равны *1* и *2*.

Затем откройте редактор File System для проекта установки, щелкните правой кнопкой узел Application Folder, выберите команду Add | Folder и укажите папку для исходного кода. Таким образом, вы создадите папку для хранения исходных файлов надстройки. Чтобы добавить файлы с исходным кодом, щелкните правой кнопкой папку для исходного кода, выберите команду Add | Project Output, а затем — тип выходных данных Source Files. Если, выполнив эти изменения, вы построите и установите файл .msi, исходный код будет устанавливаться всегда, так как условие включения файлов с исходным кодом не связано с переключателем, выбранным пользователем в диалоговом окне RadioButtons (2 Buttons). Чтобы соединить диалоговое окно с созданной папкой исходных файлов, выберите папку исходных файлов в редакторе File System и в окне Properties введите выражение **BUTTON2** = 1 в свойство *Condition*. Теперь при установке файла .msi, если пользователь выбирает первый переключатель, файлы с исходным кодом устанавливаются в соответствующую папку. Но если пользователь выберет второй переключатель (ему соответствует значение *2*) значение выражения *BUTTON2 = 1* будет равно *False,* и папка с исходным кодом не будет создана. Поскольку файлы исходного кода содержатся в этой папке, они тоже не будут установлены. Вы можете поместить условие *BUTTON2 = 1* в выходные данные проекта, чтобы при выборе пользователем второго переключателя файлы с исходным кодом не устанавливались, но на пользовательском компьютере создавалась пустая папка для исходного кода. Примеры к этой книге содержат проект установки для программы SourceFilesAddin, которая демонстрирует реализацию выборочной установки исходных кодов надстройки,

## Подгружаемые модули

При создании Windows-программ разработчики часто помещают код, совместно используемый несколькими приложениями, в библиотеки DLL. При распространении ПО клиентам вы должны установить эти DLL с проектом установки .msi, но это может оказаться далеко не идеальным способом установки кода. Допустим, вы построили библиотеку пользовательских элементов управления .NET, которую хотите продавать; ваши клиенты могут задействовать эту библиотеку для постро-

ения собственных приложений. Но как ваши клиенты будут поставлять эту библиотеку своим клиентам? Вы можете предоставить подробные инструкции, объясняющие пользователям, как включить этот компонент в их установки .msi. Однако этот подход может вызвать проблемы, поскольку, если их программа неправильно установит компонент, другое ПО, использующее этот же компонент, может закапризничать. Вместо этого вы можете создать проект установки, который устанавливает и удаляет вашу библиотеку, но это тоже не лучший вариант, так как ваши пользователи будут вынуждены предоставлять своим клиентам два файла .msi, которые следует устанавливать в определенном порядке.

Чтобы упростить установку вашей библиотеки и снизить риск некорректной установки или удаления компонента, поместите свою библиотеку в подгружаемый модуль (файл .msm). Подгружаемые: модули подобны DLL для проекта установки. Вы можете задействовать DLL для хранения кода, совместно используемого несколькими приложениями; подгружаемый модуль содержит логику установки, совместно используемую несколькими файлами .msi. Проект подгружаемого модуля похож на проект установки за исключением того, что вы не можете работать с редакторами User Interface и Launch Conditions. Еще одно различие между проектом установки и проектом подгружаемого модуля в том, что последний добавляет в редактор File System папки Web Custom Folder и Module Retargetable Folder. Если вы храните в них файлы, клиент, использующий ваш подгружаемый модуль, может указать место на диске, куда будут помещены эти файлы. Пользователь может изменить установочный путь для содержимого подгружаемого модуля, выбрав подгружаемый модуль в Solution Explorer и изменив значение свойства *ModuleRetargetableFolder* в окне Properties. Применение подгружаемого модуля в проекте установки не вызывает трудностей; выберите проект установки в Solution Explorer, а потом — команду Project | Add | Merge Module.

## Установка программ .NET

Допустим, вы испытываете возможности построения программ установки в Visual Studio .NET. Вы построили приложение .NET, добавили выходные данные проекта к проекту установки, построили решение и проверили установку проекта из файла .msi. Все замечательно устанавливается, и вы можете запустить ваше приложение. Желая убедиться, что ваша программа работает во всех случаях, вы устанавливаете тот же файл .msi на чистый компьютер, где установлена только ОС. Запустив проект установки на чистом компьютере, вы видите диалоговое окно (рис. 13-9). Почему же программа перестала работать на чистом компьютере? Visual Studio .NET не может запускаться, если не установлена .NET Framework, а на чистом компьютере ее нет!

Когда вы добавляете выходные данные проекта C# или Visual Basic .NET в редактор File System, Visual Studio .NET автоматически добавляет в редактор Launch Conditions условие, проверяющее, что .NET Framework установлена. Поскольку на чистом компьютере .NET Framework не установлена, условие запуска не выполняется, и пользователю сообщается об ошибке.

Чтобы запустить файл .msi, содержащий компоненты, которые обращаются к .NET Framework, есть два варианта. Выбор зависит от того, как вы планируете распространять ваше ПО. Первый вариант — позволить пользователю щелкнуть

кнопку Yes в диалоговом окне (рис. 13-9) и вручную установить .NET Framework, Годится, если вы распространяете ПО через Интернет или применяете носители небольшой емкости вроде дискет, на которых мало места для размещения файлов .NET Framework,

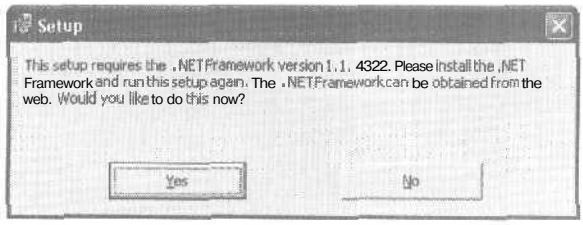



Рис. 13-9. *Сообщение об ошибке, выводимое при попытке установить приложение .NET на компьютер, где не установлена NET Framework*

Если вы поставляете ваши программы на носителях большой емкости, например CD, удобнее второй вариант. Вы помещаете распространяемые файлы .NET Framework на CD и используете программу начальной загрузки, чтобы сначала установить .NET Framework, а затем ваш файл .msi. Программа начальной загрузки — это небольшой фрагмент кода, который заботится о выполнении установки и запуске. Программа начальной загрузки гарантирует, что будет установлена .NET Framework, а затем начнется установка вашего файла .msi. Для установки программы вы должны запустить программу начальной загрузки, а не файл .msi. При построении проекта установки по умолчанию генерируется программа начальной загрузки с именем Setup.exe, которая помещается в выходную папку вашего проекта установки. Эта программа начальной загрузки только проверяет, установлена ли программа Windows Installer, и не проверяет наличие .NET Framework. Если вы хотите устанавливать .NET Framework, не следует распространять программу начальной установки, так как программа установки .NET Framework устанавливает установщик файлов .msi, если его нет на компьютере пользователя. Вы можете отключить генерацию этой программы начальной установки в диалоговом окне Properties проекта установки.

Когда вы впервые добавляете компонент .NET к проекту установки, к нему добавляется зависимость от подгружаемого модуля dotnetfxredist_x86.msm и помечается, чтобы быть удаленной. Вы можете предположить, что достаточно включить этот подгружаемый модуль (а не исключать его), чтобы установить при необходимости .NET Framework, но данный подгружаемый модуль этого не делает. Он содержит сборки, являющиеся частью .NET Framework (такие как System.XML, System.Web и т. д.), и должен быть на компьютере пользователя, если там установлена .NET Framework. Этот модуль не содержит файлов CLR, так что, включив его в свой проект установки, вы включите массу сборок, которые уже должны быть установлены у пользователя, но не добавите ничего, что нужно пользователю для запуска программ .NET.

Чтобы упростить установку .NET Framework с вашими программами, используйте программу Bootstrapper из примеров к книге. Bootstrapper, написанная на Visual C++, выполняет при первом запуске ряд действий. Сначала проверяет, нет ли уже запущенного другого экземпляра программы начальной загрузки, поскольку,

если запущено несколько экземпляров, могут возникнуть ошибки. Чтобы гарантировать, что другой экземпляр не запущен, программа начальной загрузки создает мьютекс; мьютексы совместно используются вне границ процесса, поэтому, если мьютекс уже создан другим экземпляром программы начальной установки, генерируется ошибка. В этом случае пользователь получает сообщение об ошибке, и выполнение программы прекращается. Затем программа начальной установки читает файл Setup.ini, который должен быть расположен в той же папке, что и исполняемый файл программы начальной загрузки. Файл, помимо прочего, указывает программе установки, где она может найти распространяемые файлы .NET Framework. Вот пример файла Setup.ini:

```
[Setup]
InstallName=Setup
FrameworkVersionRequired=v1.1
UseLocaleForFindingRedist=1
FrameworkRedistName=DotNetfx
FrameworkInstallPath=FrameworkRedist
MSIFilePath=Setup.msi
```

Все значения в файле INI являются необязательными; если какое-то из значений не найдено, будет использовано значение, показанное в этом примере. Опишем смысл этих значений.

- ***InstallName*** Имя устанавливаемого продукта. Если имя не задано, в качестве значения по умолчанию используется *Setup*. Эта строка используется для отображения названия вашей программы в пользовательском интерфейсе программы начальной установки.
- ***FrameworkVersionRequired*** Версия .NET Framework, которую использует устанавливаемая программа. Если версия соответствует значению, заданному в этой строке, установка .NET Framework пропускается. Значение может быть *v1.0* (номеру версии должна предшествовать буква *v*, и нельзя опускать заключительный *0)* или *v1.1*. *v1.0* — это версия .NET Framework, устанавливаемая с Visual Studio .NET 2002, а *vl.l* — это версия .NET Framework, устанавливаемая с Visual Studio .NET 2003.
- ***FrameworkRedistName*** Имя исполняемого файла (без расширения .exe), который содержит распространяемые файлы .NET Framework. Имя этого исполняемого файла всегда DotNetfx. так что в большинстве случаев это значение можно не указывать.
- ***UseLocaleForFindingRedist*** Если равно *1,* программа начальной загрузки пытается найти на CD и установить локализованную версию .NET Framework. На данный момент имеются локализованные версии .NET Framework для английского, французского, немецкого, итальянского, японского, испанского, китайского традиционного, китайского упрощенного и корейского языков. Если при установке требуется локализованная версия, программа начальной загрузки определяет язык ОС и использует его или локализацию при поиске распространяемых файлов. Если это значение равно *0,* программа начальной загрузки не учитывает язык ОС при поиске распространяемых файлов. Если ваша программа не локализована на разные языки, вы должны распространять ло-

кализованную версию .NET Framework, которую могут задействовать ваши клиенты, и установить значение *0*.

- ***FrameworkInstallPath*** Относительный путь от программы начальной загрузки к распространяемым файлам .NET Framework. Допустим, код программы начальной загрузки размещен в папке D:\MyProgramSetup и для параметра *FrameworkInstallPath* применяется значение по умолчанию, *FrameworkRedist*. В этом случае программа начальной загрузки для установки .NET Framework будет искать файл D:\MyProgramSetup\FrameworkRedist\DotNetfx.exe. Если значение *UseLocaleForFindingRedist* равно *1*, между значениями *FrameworkInstallPath* и *FrameworkRedistName* в путь вставляется идентификатор языка. Используемые идентификаторы для различных языков см. в табл. 13-3; так, для английского языка будет искаться D:\MyProgramSetup\FrameworkRedist\9\DotNetfx.exe. Если при использовании пути с идентификатором языка распространяемые файлы не найдены или язык ОС не поддерживается, для поиска используется путь без идентификатора языка. Если распространяемые файлы не были найдены, выдается сообщение об ошибке, и выполнение установки прекращается.
- ***MSIFilePath*** Относительный путь от программы начальной загрузки к устанавливаемому файлу .msi. Если программа начальной загрузки расположена в папке D:\MyProgramSetup, а для данного параметра оставлено значение по умолчанию *Setup.msi*, устанавливается файл D:\MyProgramSetup\Setup.msi.

**Табл. 13-3. Языки и их идентификаторы**

| Язык | Идентификатор языка |
|---|---|
| Английский | 9 |
| Французский | *12* |
| Немецкий | *7* |
| Итальянский | *16* |
| Японский | *17* |
| Испанский | *10* |
| Китайский | Если ОС использует традиционный китайский язык — идентификатор языка 1228. Идентификатор упрощенного китайского языка — 2052. |
| Корейский | *18* |

**Примечание** Может показаться, что выбор языка .NET Framework влияет только на вашу программу, но на самом деле он более важен, так как пользователи будут вынуждены жить с установленной вами версией .NET Framework, пока сами не удалят ее и не установят версию с поддержкой другого языка.

### Создание установочного компакт-диска

Сегодня большинство компьютеров оборудовано приводами для записи CD. и вы можете найти чистые CD у вашего поставщика по цене менее 50 центов за штуку. Доступность приводов для записи CD, дешевые носители и инструменты Visual Studio .NET, делают простым распространение ПО. Чтобы создать CD с вашим

файлом .msi, нужно объединить программу начальной загрузки, файл .msi для установки вашей программы и ряд других файлов.

**Примечание** Если вы хотите поместить распространяемые файлы .NET Framework на CD с вашей программой или сделать их доступными через Web-сайт, не относящийся к Microsoft, вы должны перед распространением .NET Framework прочитать и принять лицензионное соглашение конечного пользователя (EULA) Microsoft .NET Framework SDK.

Windows упрощает установку ПО с CD благодаря функции AutoPlay. Когда CD помещается в привод, Windows проверяет корневую папку CD и, если находит в ней файл autorun.inf, читает его, анализирует и выполняет указанные действия. Вот пример файла autoruainf, который вы можете положить в корневую папку CD для автоматического запуска программы начальной загрузки после помещения CD в Привод:

```
[autorun]
open=setup.exe
icon=setup.ico
label=My Setup
shell\launch\command=Setup.exe
shell\launch=&Install this program
```

Рассмотрим назначение записей в файле.

- ***open*** Путь к файлу (без указания буквы диска), который должен быть запущен после помещения CD в привод. Если вы используете программу начальной загрузки для установки .NET Framework и вашей программы, укажите путь к программе начальной загрузки; в противном случае укажите относительный путь к файлу .msi.
- ***icon*** Путь к находящемуся на диске файлу со значком (без указания буквы диска), который будет отображаться в Windows Explorer для CD. Путь может указывать на файл .ico либо на исполняемый файл или DLL, путь может завершаться запятой, за которой следует индекс значка в ресурсах файла.
- ***label*** Текст, который будет отображаться как метка привода CD в Windows Explorer.
- ***shell*\verbname** Если пользователь щелкает правой кнопкой значок привода CD в Windows Explorer, в контекстном меню отображается текст, следующий за этим элементом. Текст *verbname* является произвольным и может содержать только алфавитно-цифровые символы. Текст, следующий за этим ключом, может содержать любые символы и амперсанд, используемый для отметки акселератора. Если вы хотите включить амперсанд в строку, просто поставьте его дважды.
- ***shell*\verbname*command*** Если пользователь щелкает правой кнопкой значок привода CD в Windows Explorer и выбирает команду, заданную в строке *shell*\verbname, запускается программа, на которую указывает заданный путь. Строка *verbname* здесь также произвольна, но должна соответствовать имени, указанному в предыдущей строке. Файл autoruainf может содержать любое число

(в том числе и не одну) таких пар *shell\verbname*, с каждым элементом контекстного меню привода CD, но каждая пара элементов должна использовать уникальное соответствие.

После создания файла autorun.inf нужно собрать все файлы, которые будут помещены на CD; возможно, вам потребуется загрузить распространяемые файлы .NET с Web-сайта Microsoft. Поскольку объем распространяемых файлов около 20 Мб, этот файл (или файлы, если вы хотите предоставлять вашим пользователям локализованные версии) не включен в примеры к книге, На рис. 13-Ю показана структура CD с типичными компонентами, необходимыми для установки вашей программы. Эта структура (без распространяемых файлов .NET Framework) находится среди примеров к книге в папке SetupCD. Чтобы построить установочный CD, просто скопируйте свой файл .msi в эту папку и запишите ее содержимое на чистый CD. Бы создали профессиональную программу установки вашего продукта!

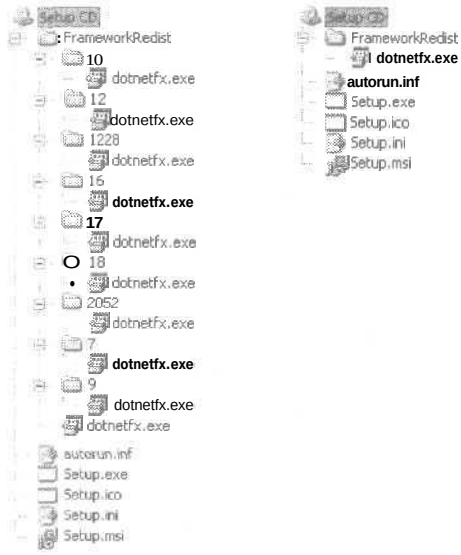



**Рис. 13-Ю.** *Структура каталогов установочного CD при поставке локализованных версий NET Framework (слева) и для нелокализованной установки (справа)*

### Установка примеров к книге

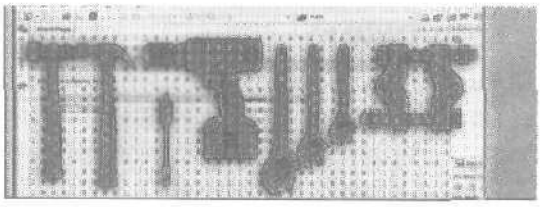
Если вы загрузили и установили файлы примеров к книге, вы запускали файл .msi, созданный с помощью инструментов Visual Studio .NET. При создании этого установочного файла .msi применялись многие из обсуждаемых в главе концепций, включая свойства установщика, нестандартные действия, редактор Registry, проект Web-установки и т. д. Исходный код для проекта установки и используемых нестандартных действий лежит в папке InsideVSNetSetup. Подробнее о том, как построить файл .msi Inside Visual Studio .NET 2005 samples, см. файл Readme.htm в папке InsideVSNetSetup.

## Что дальше

Все пользователи независимо от их опыта иногда для завершения задачи нуждаются в помощи. Visual Studio .NET предоставляет полнофункциональную справочную систему для отображения содержимого MSDN. В следующей главе вы увидите, как настроить эту систему для включения собственных тем.

ГЛАВА

# 14



# Справочнаясистема Visual Studio .NET

MSDN Library, поставляемая с Microsoft Visual Studio .NET, содержит массу данных, которые разработчики могут использовать при создании решений в интегрированной среде разработки (IDE). Но она содержит столько информации, что бывает трудно найти то, что ищешь. К счастью, справочная система Visual Studio .NET обладает чрезвычайной гибкостью. Ее можно настраивать для повышения производительности поиска и получения точной информации по интересующему вопросу. Ее также можно задействовать для публикации контекстно-зависимой справки собственных библиотек, сборок и надстроек, предоставляя эти данные всем разработчикам. Вы даже вправе добавлять собственную документацию в окна Table of Contents, Search и Index справочной системы Visual Studio .NET, максимально облегчая другим программистам доступ к вашей информации.

В этой главе мы посвятим некоторое время изучению настройки справочной системы Visual Studio .NET и созданию собственных наборов справочной информации (Help collections) в Visual Studio .NET.

## Работа со справочной системой

Справочная система Visual Studio .NET — одна из лучших составляющих интегрированной среды разработки. Контекстно-зависимая, с подробным указателем и всеобъемлющим поиском, она покрывает всю библиотеку MSDN Library, которая включает около 1,5 Гб данных. Справочная система настолько велика, что стоит отдельно рассказать, как настроить ее для получения именно тех данных, что вам нужны.

В ходе многоступенчатого процесса установки Visual Studio .NET вы можете установить документацию MSDN, поставляемую в составе этого продукта. Установка

этой документации осуществляется на отдельном этапе, так как позволяет разработчикам установить последнюю версию MSDN Library от Microsoft. (Если вы подписаны на MSDN, проверьте наличие новой версии MSDN Library при установке Visual Studio .NET 2003 вскоре после ее выпуска.)

Если вы установили несколько наборов справочной информации, выбрать нужный можно в диалоговом окне Options. По умолчанию в Visual Studio .NET 2003 используется набор Visual Studio .NET 2003 Combined Help Collection. Он рекомендован к использованию, поскольку содержит дополнительную документацию по некоторым особенностям надстроек и разнообразным деталям данного продукта.

На рис. 14-1 показана страница Help в папке Environment диалогового окна Options. Заметьте: кроме используемого по умолчанию набора справочной информации, можно установить основной язык справки (preferred language). Чтобы у этого параметра появились дополнительные значения, на компьютере должна быть установлена документация на других языках.

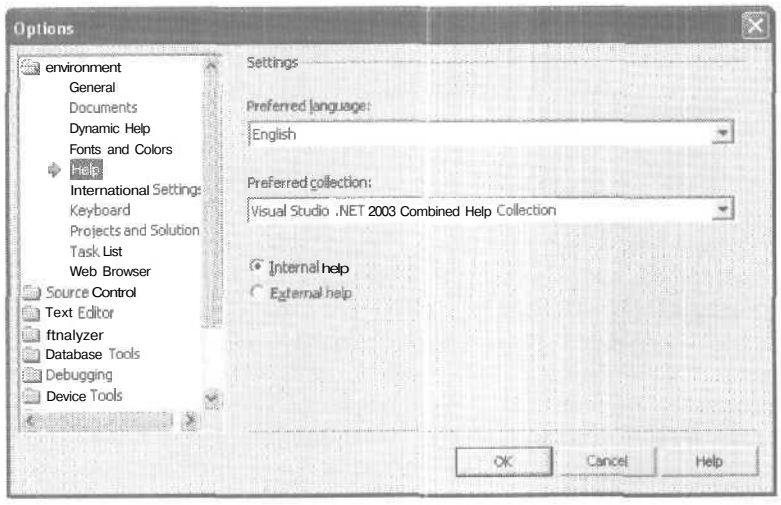



**Рис. 14-1.** *Диалоговое окно Options позволяет выбрать основной набор справочной информации*

Последний параметр позволяет выбрать внутренний (internally) или внешний (externally) способ отображения справки. В первом случае справочная информация будет отображаться внутри IDE, во втором — появится в окне за пределами IDE. Внешнее отображение особенно удобно при использовании нескольких мониторов: вы можете открыть справку на одном экране, оставив IDE на другом.

## Окна справочной системы

Независимо от того, каким способом отображения справки вы пользуетесь, вам надо уметь работать с тремя организационными окнами. Окно Contents отображает иерархическую структуру тем справочной информации. Окно Index содержит список терминов, встречающихся в наборе. А окно Search позволяет осуществлять поиск определенного термина или фразы по всей справочной системе. Полученная в ходе поиска информация отображается в окне Search Results. В ре-

зультатах поиска возвращается не более 500 ссылок, поэтому стоит максимально уточнять сведения для поиска везде, где это возможно. На рис. 14-2 показано окно Microsoft Document Explorer со всеми описанными выше элементами. Чтобы открыть его, щелкните ссылку Microsoft Visual Studio .NET 2003 Documentation в папке Microsoft Visual Studio .NET 2003 из меню Пуск.

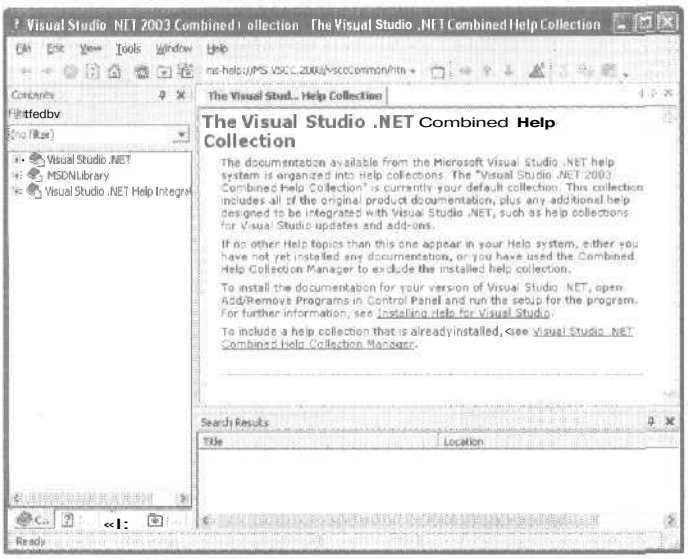



**Рис. 14-2.** *Справочная система Visual Studio .NET 2003 в окне Document Explorer*

Функции Document Explorer не требуют подробных объяснений. Они очевидны и существуют в Visual Studio и MSDN многие годы. Однако некоторые приемы и параметры помогут вам повысить эффективность поиска нужных сведений.

## Параметры поиска и указателя

Значение параметра Filtered By в окне Search (рис. 14-3) серьезно влияет на выдаваемую информацию при указании строки в поле Look For. Так, если внести **CString** и включить фильтр «Visual Studio Macros», первым в окне результатов поиска будет возвращен элемент DataSet Class из библиотеки классов .NET Framework, При включенном фильтре «Visual C++» первой будет возвращена тема CString Operation Relating To C-Style Strings из раздела Visual C++ Concepts. Если не включать фильтры вовсе, в списке появятся оба эти элемента без контекстной информации. Оба результата корректны, но фильтр помогает уточнить интересующую вас тему.

Параметр Search In Titles Only ограничивает возвращаемые результаты терминами, найденными только в заголовках документов. Он пригодится для поиска основной документации по заданной теме. Параметр Match Related Words включает в состав искомых вариации введенных терминов. Например, поиск слова «link» с включенным параметром Match Related Words среди результатов может содержать «linked», «linking» и «linker». Параметр Search In Previous Results ограничивает область поиска результатами предыдущего поиска, что позволяет углубиться в список результатов.

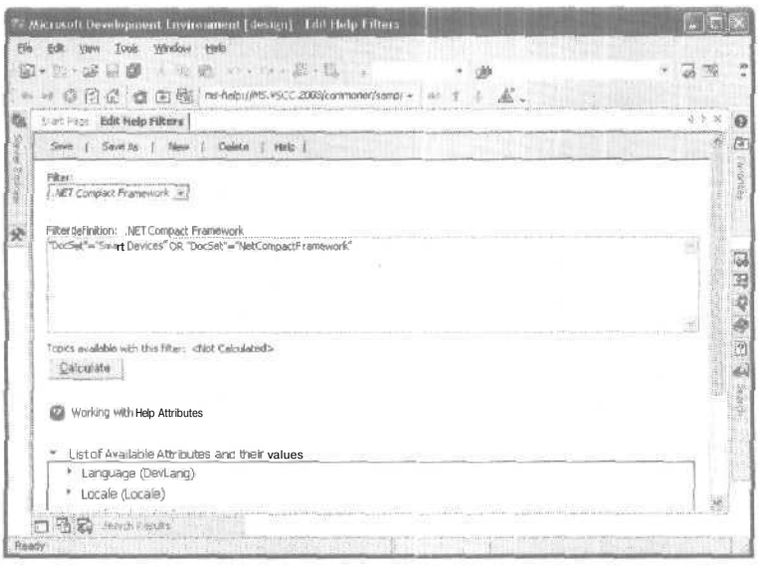



**Рис. 14-4.** *Собственные фильтры справочной системы создаются в окне Edit Help Filters*

Чтобы создать простое подмножество COM/.NET, можно объединить два атрибута *DocSet* с двумя атрибутами *Technology*:

```
("DocSet"="NETFramework" OR "DocSet" = "NETFrameworkExtender")
or("Technology"="COMt" OR "Technology"="NETt" )
```

Для создания фильтров имеется большое количество различных атрибутов. В основном вам, вероятно, потребуется группировать наборы документов, языки и технологии, но есть и другие атрибуты. Если щелкнуть отдельный атрибут в окне List Of Available Attributes, он автоматически будет добавлен к открытому набору.

Создав список используемых атрибутов, щелкните кнопку Calculate, чтобы увидеть, сколько тематических разделов окажется в вашем новом списке. По числу тем можно решить, не чересчур ли расплывчат или конкретен составленный атрибут,

## Поиск из окна Command Window

Тем, кто привык к редактору Vim, понравится возможность поиска справочной информации в окне Command Window. По умолчанию для доступа к справочной информации из окна Command Window служит команда *HelpF1Help*Она принимает один аргумент — строку, содержащую ключевое слово Fl (F1 keyword) для поиска. Если искомый термин является ключевым словом F1, вы получите соответствующий раздел справочной информации, набрав в окне Command Window команду **>help *term,*** где *term* — искомое слово. Если искомый термин не является ключевым словом F1, он помещается в окно Указатель, где для него отображаются соответствующие разделы.

Если вам нужна справка по именованной команде Visual Studio .NET ключевым словом Fl для этой команды скорее всего будет ее имя, и, если оно вам известно, вы без труда найдете нужный раздел справочной информации. Так, чтобы полу-

чить справку по команде Find, введите **>help Edit.Find** в окне Command Window. Попробуйте проделать это с любой из именованных команд, встретившихся в этой книге, и вы сразу получите нужный раздел. В число терминов для поиска могут входить:

- *Edit.ReplaceInFiles;*
- *Help.Index;*
- *File.NewFile;*
- *Debug.Start.*

Если вы начнете привыкать к этой функции и заинтересуетесь, как определять ключевые словя F1 для разных разделов справочной информации, включите отладку в окне Dynamic Help, добавив элемент в раздел реестра

`HKCU\Software\Microsoft\VisualStudio\7.1\Dynamic Help`

Чтобы включить отладку, добавьте раздел *«Display Debug Output in Retail»,* если такого нет, и задайте ему значение *«Yes».* На рис. 14-5 показано окно Dynamic Help после включения отладки. Помните, что таким образом действуют только разделы, сопоставленные помощи F1. Другие термины открываются в окне Index.




**Рис. 14-5.** *Включение отладки в окне Dynamic Help*

# Настройка окна Dynamic Help

Настроить способ представления данных в окне Dynamic Help можно, во-первых, выбрав отображаемые данные на странице Dynamic Help диалогового окна Options (рис. 14-6). По умолчанию для окна Dynamic Help включены все варианты; изменяя эти параметры, вы не получите дополнительной информации, но сможете сосредоточиться только на нужных данных. Кроме того, поскольку размеры окна Dynamic Help невелики, вы сможете увидеть важную информацию, не прокручивая содержимое вверх и вниз. Например, сброс флажка Samples перемещает раздел Getting Started под раздел Help (рис. 14-7),

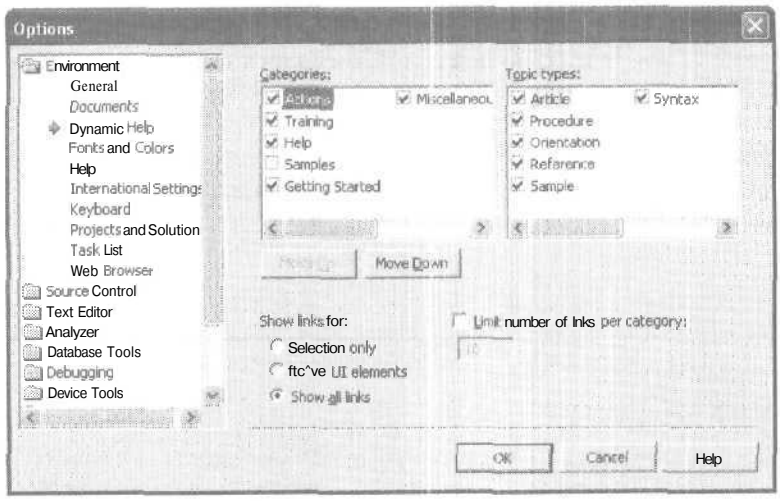



**Рис. 14-6.** *Установка параметров для Dynamic Help в диалоговом окне Options*




**Рис. 14-7.** *Окно Dynamic Help с убранным разделом Samples*

Второй способ настройки Dynamic Help — добавление собственных ссылок в окно Dynamic Help через сервис XML Help Provider — мы обсудим ниже,

## Использование сервиса XML Help Provider

Динамические ссылки позволяют открывать любые типы справочной информации, включая обычный текст, HTML и даже документы Microsoft Word, так что они могут стать эффективной альтернативой созданию более сложных справочных систем, о которых речь пойдет далее.

Чтобы добавить собственные ссылки в окно Dynamic Help, надо создать XML-файл и поместить его в определенный каталог. Открытый затем экземпляр Visual Studio .NET 2003 отобразит вашу информацию при совпадении указанных контекстных критериев. Чтобы ваши данные отображались корректно, необходимо соответствие всего нескольким требованиям. Обсудим их по одному,

### Размещение

XML-файл может иметь любое имя, но должен размещаться в папке C:\Program Fiies\Microsoft Visual Studio .NET 2003\Common7\IDE\HTML\XMLLinks\1033. Если вы установили Visual Studio .NET в другую папку или используете Visual Studio 2002, то путь будет немного отличаться, но важна лишь его последняя часть: Common7\IDE\HTML\XMLLinks\1033. Файл должен иметь расширение .xml.

**Примечание** В Windows число 1033 обозначает код языка U.S. English. Если ваша копия Visual Studio .NET локализована для другого региона, то у папки, в которую следует скопировать ваш XML-файл, будет другой номер.

В папке 1033 вы найдете минимум один XML-файл — Context.xml, который содержит группы ссылок, предварительно установленных для Visual Studio .NET. Вы вправе добавлять собственные группы ссылок, редактируя этот файл или указывая их в своем XML-файле,

### Схема

Для корректной работы в интегрированной среде справке XML, добавляемой в окно Dynamic Help, требуется специальная схема:

```
<DynamicHelp xmlns="http://msdn.microsoft.com/vsdata/xsd/vsdh.xsd"
xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
xsi:schemaLocation="http://msdn.microsoft.com/vsdata/xsd/vsdh.xsd">
```

#### Группы ссылок

Группы ссылок — это сгруппированные разделы в окне Dynamic Help. Вы вправе создавать собственные группы ссылок или добавлять свои справочные разделы в существующие группы. Группы ссылок определяются тэгом *LINKGROUP*. Давайте создадим новую группу ссылок с идентификатором *«InsideVSNET»* и заголовком *«Inside Visual Studio NET»*. Значение идентификатора будет использовано при создании ссылок на какую-либо информацию,

```
<LINKGROUP ID="InsideVSNET" Title="Inside Visual Studio .NET"
  Priority="300">
  <GLYPH Collapsed="1" Expanded="2" />
</LINKGROUP>
```

Значение атрибута *Priority* сравнивается с другими такими атрибутами и определяет положение ссылки в списке ссылок окна Dynamic Help. Эти значения мс >жно увидеть в отладочной информации на рис. 14-5.

Тэг *GLYPH* позволяет выбрать значок, отображаемый для вашей группы ссылок, Можно указать значок для свернутого (collapsed) и развернутого (expanded) представления группы. В таблице 14-3 приведены встроенные значки.

**Табл. 14-3. Значки для групп ссылок**

| Индекс | Значок |
|---|---|
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |

### Confexf

Тэг *Context* включает основной текст информации, размещаемой в Dynamic Help. Он содержит все другие контекстные тэги для этой информации и по существу является основной частью представляемой информации. Все остальные тэги, которые мы рассмотрим в этом разделе, размещаются внутри тэга *Context.*

Внутри *Context* два тэга определяют показываемую в Dynamic Help информацию. Тэги *Keywords* и *Attributes* содержат вложенные тэги *KItem* и *AItem* соответственно, которые позволяют указать, когда тот или иной раздел вносится в контекст. Значимость тэгов в этих разделах определяется на основе таких факторов, как тип открытого в данный момент проекта или окна, владеющего фокусом ввода. Некоторая информация об этих тегах и вычислении их значимости содержится в онлайновой документации, но мы предлагаем для изучения этих вопросов привлечь отладочную информацию.

Приведем простой пример. Поскольку мы хотим, чтобы информация была доступна пользователю при любых обстоятельствах, то в тэге *KItem* мы используем значение *«VSAmbient»*, которое предписывает отображать раздел в любом случае в отличие от таких значений, как *«VSSolutionExplorer»*, которое отображает раздел, только когда фокусом ввода владеет Solution Explorer. Вот код для этого элемента:

```
<Keywords>
  <!- Тэги KItem содержат ключевые слова для разделов ->
```

```
  <KItem Name="VS.Ambient" />
</Keywords>
```

Последнее, что надо сделать, — это создать несколько ссылок. Они размещаются внутри тэга *Links* с помощью тэгов *LItem*:

```
<Links>
  <!- Тэги LItem содержат ссылки на отображаемые разделы ->
  <LItem URL="file:///C:\InsideVSNET\Info.htm"
  LinkGroup="InsideVSNET">Macro Information</LItem>
  <LItem URL="http://www.microsoft.com/mspress/books/6425.asp"
  LinkGroup="InsideVSNET">Web Page</LItem>
  <LItem URL="http://www.microsoft.com/mspress/support/"
  LinkGroup="InsideVSNET">Support</LItem>
</Links>
```

Каждый тэг *Lltem* имеет несколько атрибутов. Атрибут *URL* содержит ссылку на раздел справочной информации, который нужно отобразить. В атрибуте *LinkGroup* указывается идентификатор группы ссылок, в которой нужно отобразить эту ссылку. В данном случае мы создали три ссылки: первая ссылается на файл с диска C:, вторая и третья — на Web-страницу этой книги и Web-сайт Microsoft Press Support.

В листинге 14-1 показан файл InsideVSNET.xml, размещенный, как сказано выше, в папке 1033.

**Листинг 14-1.** Добавление ссылок в Dynamic Help

**InsideVSNET.xml**

```
<?xml version="1.0" encoding="utf-8" ?>
<!- Эта схема нужна для Dynamic Help ->
<DynamicHelp xmlns="http://msdn.microsoft.com/vsdata/xsd/vsdh.xsd"
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
  xsi:schemaLocation="http://msdn.microsoft.com/vsdata/xsd/vsdh.xsd">
  <!- Создаем группу ссылок, в которой будет
       отображаться наша информация ->
  <LINKGROUP ID="InsideVSNET" Title="Inside Visual Studio .NET"
    Priority="500">
    <GLYPH Collapsed="1" Expanded="2" />
  </LINKGROUP>
  <Context>
    <Keywords>
      <!- Тэги KItem содержат ключевые слова для разделов ->
      <KItem Name="VS.Ambient" />
    </Keywords>
    <Links>
      <!- Тэги LItem содержат ссылки на отображаемые разделы ->
      <LItem URL="file:///C:\InsideVSNET\Info.htm"
        LinkGroup="InsideVSNET">Macro Information</LItem>
      <LItem URL="http://www.microsoft.com/mspress/books/6425.asp"
        LinkGroup="InsideVSNET">Web Page</LItem>
      <LItem URL="http://www.microsoft.com/mspress/support/"
```

*см. след. стр.*

```
        LinkGroup="InsideVSNET">Support</LItem>
    </Links>
  </Context>
</DynamicHelp>
```

На рис. 14-8 показана Visual Studio .NET после подключения к Web-странице этой книги из окна Dynamic Help.




**Рис. 14-8.** *Подключение к пользовательской информации из окна Dynamic Help*

## Создание собственных файлов справки

Этот раздел представляет введение в создание справки. Для работы надо установить Visual Studio .NET Help Integration Kit (VSIK) 2003, который к моменту выпуска Visual Studio .NET 2003 должен быть доступен для скачивания на сайте MSDN.

В этом примере мы создадим простой файл справки и встроим его в Visual Studio .NET 2003- Создание справки во многом похоже на использование нового языка программирования, так что мы опустим некоторые детали создания и уделим больше внимания процессу интеграции.

После установки Visual Studio .NET Help Integration Kit 2003 в диалоговом окне Visual Studio .NET New Project вы обнаружите новую категорию проектов. Категория Help Projects предоставляет шаблоны для новых проектов справки, преобразования и декомпиляции файлов справки. Новые шаблоны проектов показаны на рис. 14-9.

Help Integration Kit позволяет работать с проектами справки из Visual Studio .NET, а значит, можно задействовать в работе все преимущества редакторов и справочной системы IDE.

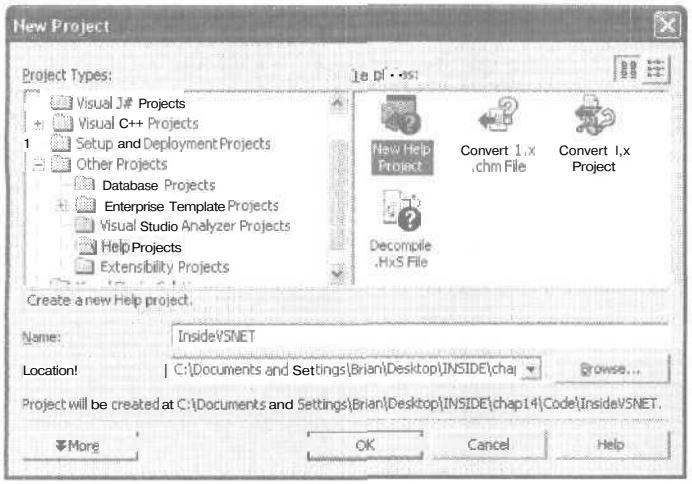



**Рис. 14-9.** *Шаблоны проектов справки в Visual Studio .NET 2003*

В этом разделе мы используем справочную систему версии Microsoft Help *2*. В среднем проекте Help *2* применяется несколько файлов. В основном они имеют расширение .Hx?, где ? заменяется на букву, отражающую назначение файла, Так. файл определения набора справочной информации (Help collection definition) имеет расширение .HxC, а файл включений (Help include) — ,HxF, В Visual Studio .NET проекты Help *2* имеют расширение .HWProj. Проект справки можно добавлять в любое создаваемое решение. В табл. 14-4 приведены расширения файлов, которые вам следует знать.

**Табл. 14-4. Расширения файлов, используемые в Microsoft Help *2***

| Расширение | Описание |
|---|---|
| .HxS | Скомпилированный файл справки |
| .HxC | Файл набора |
| .HxF | Файл включений |
| ,HxT | Файл содержания |
| .HxK | Файл указателя |
| ,HxA | Файл атрибутов |
| ,HxE | Файл определения примера |
| .html, .htm, .txt, and others | Файлы тематических разделов и содержимого справки |

При создании нового проекта вы начинаете работу с двух файлов: .HxC (файл набора) и .HxF (файл включений). Если бы вы создавали эти файлы вручную, вам пришлось бы редактировать их самостоятельно, но в Visual Studio .NET это можно сделать через диалоговое окно Property Pages проекта (рис. 14-10),

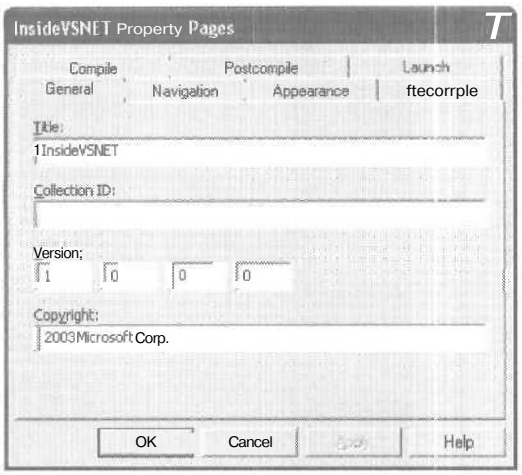



**Рис. 14-Ю.** *Диалоговое окно Property Pages проекта справки*

Шаблоны для типов файлов, которые можно добавить в проект, доступны через диалоговое окно Add New Item (рис. 14-11). Обычно требуется добавить минимум один .HxT-файл (содержание) и один .HxK-файл (указатель).

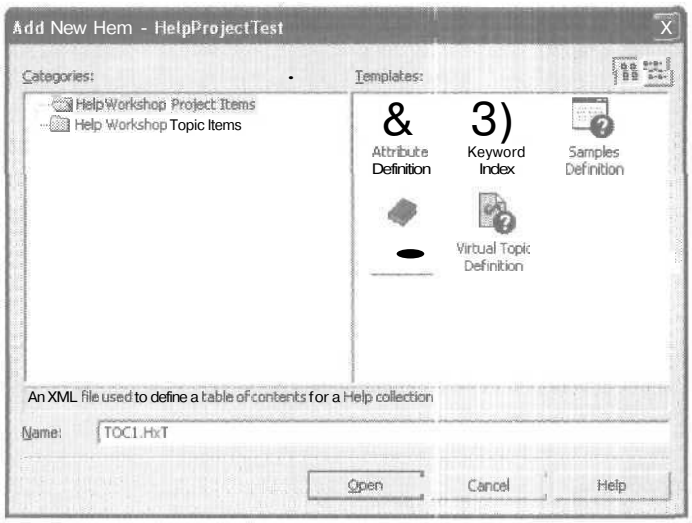



**Рис. 14-11.** *Диалоговое окно Add New Item*

Добавив в проект файлы .HxT и .HxK, следует установить несколько важных параметров в диалоговом окне Property Pages. Перейдя на вкладку Navigation, выберите из соответствующих раскрывающихся списков файлы содержания (TOC) и указателя (Index), которые вы хотите задействовать в проекте (рис. 14-12). Теперь вы сможете добавить в проект некоторую информацию.

Далее через диалоговое окно Add New Item мы добавим .htm-файл с именем Intro.htm. Этот файл будет играть роль начальной страница проекта. В редакторе на рис. 14-13 открыта .htm-страница, а все созданные файлы перечислены Р окне

Project Explorer. С этого момента мы можем добавить к проекту практически любую информацию и внести ее в Содержание.

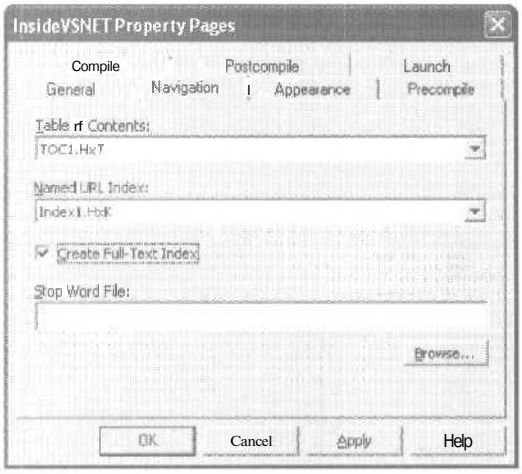



**Рис. 14-12.** *Установка параметров навигации для проекта*

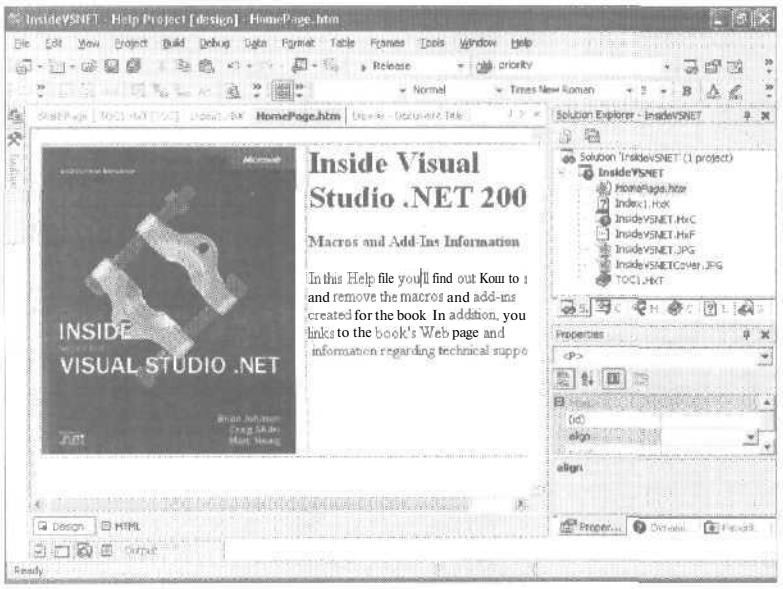



**Рис. 14-13.** *Visual Studio NET с открытым для редактирования разделом справки*

Создав и сохранив несколько файлов с содержимым, вы можете внести их в Содержание операцией буксировки. На рис. 14-14 показан открытый файл Содержания. Перетащите файлы с содержимым на дерево и установите нужный порядок при помощи стрелок на панели инструментов или контекстного меню каждого элемента. В нашем примере мы добавили вторую страницу содержания к общему содержанию «Global TOC».

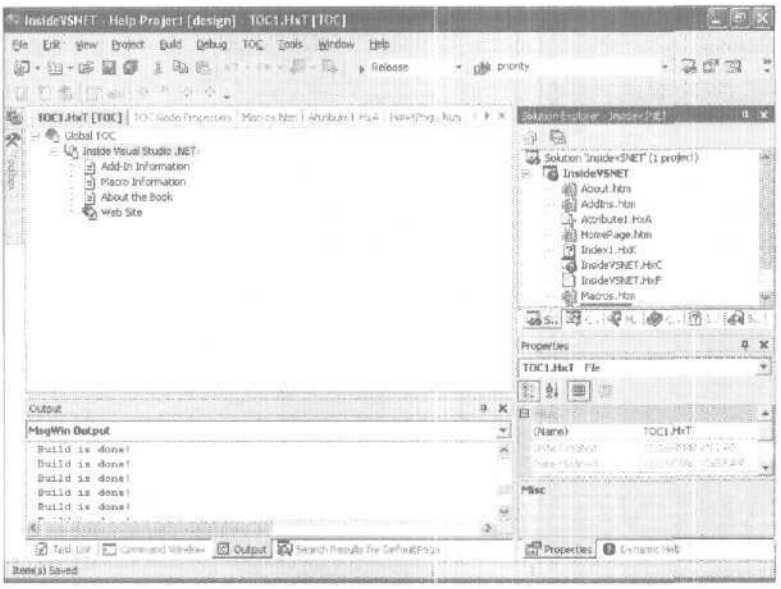


**Рис. 14-14.** *Работа с Содержанием в Visual Studio .NET*

Кроме ссылок на скомпилированные файлы справки, в Содержание можно добавлять ссылки на внешние Web-сайты и элементы файловой системы. В нашем примере мы добавили ссылку на Web-страницу этой книги и задали свойству Icon значение 22 для отображения значка Web. Назначить значок для любого раздела Содержания позволяет окно свойств (Properties) каждого раздела. Всего в Содержании 45 значков, поэтому вы можете довольно точно подобрать значок в соответствии с информацией раздела. Чтобы увидеть их полный список, найдите раздел Default TOC Icons в справочной системе VSHIK.

Теперь нужно добавить в файл немного кода для выполнения определенных задач. Во-первых, нужно, чтобы при открытии нашего файла справки отображалась начальная страница. Если не установить начальную страницу, откроется Web-страница, используемая Microsoft Internet Explorer по умолчанию, чего мы совсем не хотим. Кроме того, надо добавить в Указатель соответствующие разделы справки. Для всего этого потребуется ввести определенный код в .HxK-файл.

Сначала поговорим о том, что нужно для установки начальной страницы. Чтобы в справке отображалась начальная страница, надо в файле указателя создать элемент *Keyword* с нужным ключевым словом и указать в нем файл начальной страницы. Для этого служит элемент *Keyword* с ключевым словом «*HomePage*». Вот код для установки этого ключевого слова в .HxK-файле:

```
<Keyword Term = "HomePage">
  <Jump Url = "HomePage.htm"/>
</Keyword>
```

Чтобы добавить разделы в Указатель, используйте описанный выше синтаксис *Keyword* или добавьте к вашим разделам данные XML, и они будут внесены в Указатель автоматически. Разбираемый пример невелик, поэтому мы добавили в файл

указателя лишь несколько элементов. Вот файл Index1.HxK полностью (листинг 14-2):

**Листинг 14-2.** Файл указателя

**IndexI.HxK**

```
<?xml version="1.0"?>
<!DOCTYPE HelpIndex SYSTEM "ms-help://hx/resources/HelpIndex.DTD">

<HelpIndex DTDVersion="1.0" Name="K">
  <!- Место для ключевых слов »>
  <Keyword Term = "Macros">
    <Jump Url = "Macros.htm"/>
  </Keyword>

  <Keyword Term = "Web Site">
    <Jump Url = "http://www.microsoft.com/mspress/books/6425.asp"/>
  </Keyword>

  <Keyword Term - "About">
    <Jump Url - "About.htm"/>
  </Keyword>

  <Keyword Term = "Add-Ins">
    <Jump Url = "AddIns.htm"/>
  </Keyword>

  <Keyword Term * "HomePage">
    <Jump Url = "HomePage.htm"/>
  </Keyword>

</HelpIndex>
```

Запустив проект, вы получите полный набор файлов справки (рис. 14-15). Чтобы скомпилировать и запустить набор справки, введите в окне Command Window команду *DebugStart* или нажмите F5. Запустится компилятор, который создаст .HxS-файл, а затем откроет копию Document Explorer (Dexplorer.exe), используя аргументы, заданные на странице Launch диалогового окна Project Properties.

## Регистрация набора справки

Перед обращением к разделу справочной информации из Dexplorer его следует зарегистрировать. Рекомендуемый способ регистрации — установить пакет Windows Installer. Он включает ряд довольно сложных этапов, требующих редактирования модуля слияния (merge module) с помощью инструмента Orca из Windows SDK. Этот способ описан в документе *VisualStudio .NETHelp Integration*, поставляемом с Help 2 SDK. Но мы расскажем об альтернативном способе регистрации набора справки с помощью утилиты для тестирования HxReg.exe.

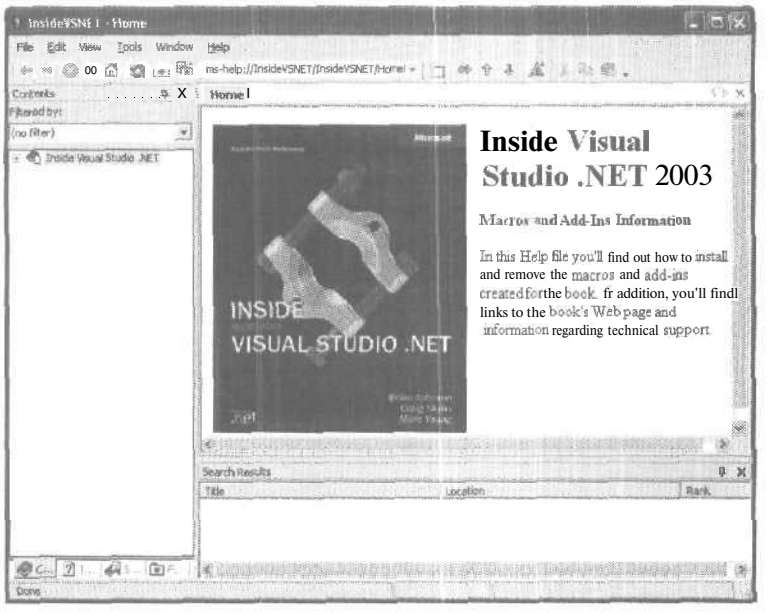



**Рис. 14-15.** *Скомпилированный набор справки в Document Explorer*

Чтобы зарегистрировать созданный набор справки, нужно запустить утилиту HxReg.exe, указав пространство имен набора (*-n*), имя файла набора (*-c*) и описание пространства имен (*-d*). Для этого вам, вероятно, придется добавить Help 2 SDK к переменной PATH (C:\Program Files\Microsoft Help 2.0 SDK). Чтобы выполнить регистрацию, перейдите в папку с .HxS-файлом и запустите следующие команды из окна Command Window. Первая регистрирует пространство имен для набора справки:

```
HxReg.exe -n MS.InsideVSNET.1033 -c InsideVSNET.HxC
-d "Inside Visual Studio .NET Help Sample"
```

Затем регистрируется скомпилированный файл справки:

```
HxReg.exe -n MS.InsideVSNET.1033 -i InsideVSNET
-s InsideVSNET.HxS -1 1033
```

Заметьте; мы заключили имя пространства имен между *MS.* и *.1033-* Соглашение о пространствах имен в данном случае предписывает такой порядок: (*Компания*).Пространство_имен.(*Язык*).

Наконец, чтобы создать ссылку на зарегистрированный файл справки, создайте ярлык, содержащий в текстовом поле Target такой путь:

```
"C:\Program Files\Common Files\Microsoft Shared\Help\dexplore.ex e"
/helpcol ms-help://MS.InsideVSNET. 1033
```

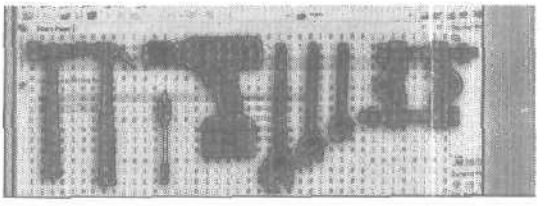
Разделы вашего файла справки можно просматривать и из Dynamic Help. Для этого просто добавьте URL нужного раздела в тэг *Item*, о котором говорилось выше. Для раздела Macro из нашего примера XML со ссылкой будет выглядеть как-то так:

```
<LItem URL="ms-help://MS.InsideVSNET.1033/InsideVSNET/Macros.htm"
  LinkGroup="InsideVSNET">Macro Information</LItem>
```

Описанные операции не помогут встроить набор справки в Visual Studio .NET. Для этого придется создать программу установки и отредактировать соответствующий модуль слияния. Но теперь у вас хватает информации для того, чтобы проверить разделы справки из панели инструментов Web или из Dynamic Help.

## Что дальше

В следующей — последней — главе этой книги мы вернемся к проектам и решениям и расскажем об использовании Visual Studio .NET из командной строки и защите исходных кодов с помощью Microsoft Visual SourceSafe.



ГЛАВА

# 15

# Усложненные проекты в Visual Studio .NET

Понимание концепции проектов *и* решений — ключ к успеху в Microsoft Visual Studio .NET. Мы познакомились с ними в главе 2, а затем перешли к обсуждению других функций IDE и способов расширения IDE через макросы и надстройки. В этой главе вы узнаете о настройке и использовании проектов и решений в усложненных сценариях, произведете компоновку проектов и решений из командной строки и научитесь управлять исходным кодом с помощью Microsoft Visual SourceSafe (VSS).

## Visual Studio .NET из командной строки

Компоновка проектов и решений Visual Studio .NET из командной строки во многом похожа на то, как в прошлом для сборки make-файлов применялась NMake.exe. Работа с Visual Studio .NET из командной строки — удобный способ компоновки проектов, не открывая IDE. Очевидное преимущество такого подхода — в автоматизации процесса компоновки с помощью сценариев и командных файлов.

Простейший способ применения Visual Studio .NET из командной строки - разместить путь к Devenv.exe в переменной окружения *PATH* или работать из окна Command Prompt Visual Studio .NET 2003. По умолчанию переменные окружения *PATH*, *INCLUDE*, *LIBPATH* и *LIB* устанавливаются в IDE, но можно настроить Visual Studio .NET на использование текущих переменных, указав в командной строке параметр */useenv*.

### Компоновка проектов и решений из командной строки

Давайте разберемся с параметрами, которые следует указать для корректного выполнения компоновки решения или проекта из командной строки. Чтобы со-

брать решение из командной строки, откройте окно командной строки, внеся в переменную *PATH* путь к Devenv.exe, и перейдите в *папку,* содержащую файл решения. Затем выполните команду Devenv.exe, указав файл решения и ключ */build,* за которым следует имя нужного типа компоновки. Обычно это имя debug или release. Если вы создали собственные типы компоновки (см. главу 2), можно указать имя одного из них. Типичная команда для отладочной компоновки (debug build) решения выглядит примерно так:

```
devenv SCTestVCS1.sln /build debug
```

Информация о компоновке выводится в командном окне, но ее можно направить и в журнал компоновки (build log). Для этого укажите параметр */out с* именем файла. Чтобы просмотреть информацию о процессе компоновки, откройте файл журнала в Блокноте или выполните команду Type (рис. 15-1).

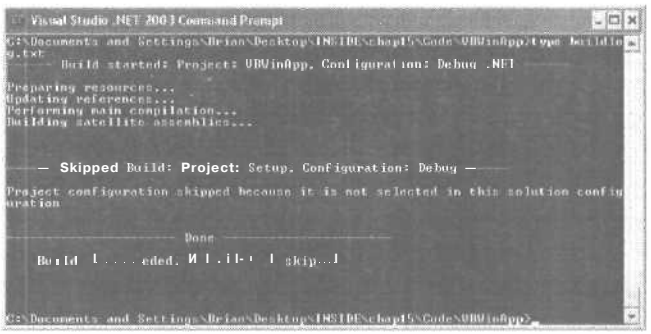



**Рис. 15-1.** *Команда Type позволяет взглянуть на созданный журнал компоновки*

Для сборки проекта в решении служит ключ */project.* Команда для компоновки одного проекта из решения может выглядеть так:

```
devenv SCTestVCS1.sln /project JSConsoleApp /build debug
```

В данном случае решение содержит несколько проектов, но собирается только указанный файл проекта. Если вы хотите открыть решение в IDE с определенной конфигурацией проекта, укажите конфигурацию в ключе */projectconfig.* Открыв IDE таким способом, вы сможете перейти прямо к нужному проекту и конфигурации в решении.

Работая с Devenv.exe из командной строки, обратите внимание на две особенности. Во-первых, файл проекта или решения следует указывать в команде до других ключей. Иногда можно сначала указать ключи, но это не распространяется на все ключи, поэтому проще не учитывать спецификации решения вовсе. Во-вторых, все пути, содержащие пробелы, должны заключаться в кавычки.

Ключи для компоновки действуют так же, как именованные команды в IDE. Ключ */rebuild* очищает целевую папку перед компоновкой или повторно компонует все целевые файлы в зависимости от языка. Ключ */clean* запускает в решении команду *Build.CleanSolution.* Поведение команды Clean зависит от языкового процессора. Она хорошо работает в проектах Visual C++, но Visual Studio .NET 2003 не поддерживает очистку других включенных типов проектов. Как и в команде Build, следует указать тип компоновки, который нужно очистить из командной строки:

```
devenv SCTestVCS1.sln /clean debug /out clean.txt
```

Вот результат работы этой команды для решения, содержащего проекты разных типов (в данном случае команду Clean выполняет лишь один проект):

```
--- Clean started: Project: CPPConsoleNet,
Configuration: Debug Win32 ---

Deleting intermediate files and output files for project
'CPPConsoleNet', configuration 'Debug|Win32'.


---------- Done ----------

    Clean: 1 succeeded, 0 failed, 0 skipped
```

Для запуска целевого файла, создаваемого решением или проектом, служит параметр */run*. Он открывает IDE и запускает вашу программу, как при использовании команды *Debug.Start*. Если вы установили точку прерывания, встречающуюся в ходе работы программы, выполнение останавливается на этой точке. Это свойство может пригодиться, если вам часто приходится открывать и закрывать IDE и вы хотите возвращаться прямо туда, откуда начали отладку.

Взглянуть на вывод определенного решения позволяет параметр */runexit*. Он открывает IDE в свернутом состоянии и запускает ваш исполняемый файл. После выхода из этого процесса свернутая IDE закрывается автоматически. Отладка в сценарии */runexit* не предусмотрена, поэтому, если в коде установлены точки прерывания, они игнорируются.

Командная строка для решения, которое нужно запустить в IDE, будет выглядеть как-то так:

```
devenv VBWinApp.sln /run
```

Если решение содержит проект для развертывания (deployment project), для сборки решения и запуска развертывания можно применить параметр */deploy*. Он требует соответствия двум требованиям. Во-первых, в решении должен быть проект для развертывания. Во-вторых, проект для развертывания должен быть добавлен в конфигурацию, используемую при компоновке. По умолчанию для новых проектов развертывания, добавляемых в решения, флажок Build сброшен. Чтобы выполнить команду */deploy*, следует открыть диалоговое окно Configuration Manager, выбрав команду Configuration Manager из меню Build. На рис. 15-2 в этом диалоговом окне соответствующий флажок установлен для конфигурации Release решения.

Чтобы собрать и развернуть такое решение, закройте IDE и введите команду с ключом */deploy*:

```
devenv VBWinApp.sln /deploy release
```

Это приведет к запуску созданной вами программы развертывания.

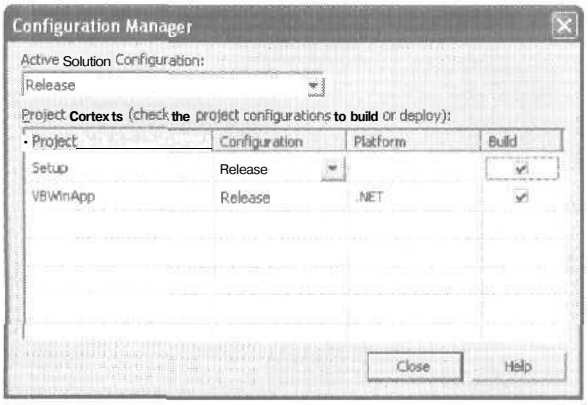



**Рис. 15-2.** *Выбор проекта Setup в диалоговом окне Configuration Manager*

Ключ */command* позволяет запускать именованную команду при открытии IDE. Так, если при запуске IDE вы хотите открыть надстройку Code Model Add-in из главы 12, просто введите команду:

```
devenv /command CMEAddIn.Connect.CMEAddIn
```

Запомнить название *CMEAddIn.Connect.CMEAddin* не так просто, но полезная особенность этого ключа в том, что вместо именованной команды он позволяет задействовать псевдоним, что облегчает ввод команды, Чтобы создать псевдоним для строки *CMEAddIn* (предполагая, что она установлена), находясь в IDE, откройте окно Visual Studio .NET Command Window, нажав Ctrl+Alt+A, и введите:

```
alias cme CMEAddIn.Connect.CMEAddIn
```

Теперь запустить надстройку из командной строки Windows можно так:

```
devenv /command cme
```

Если команда принимает аргументы, можно заключить ее в кавычки, и в большинстве случаев она будет выполнена корректно. Следующая строка запускает псевдоним *help* с аргументом *command:*

```
devenv /command "help command"
```

*Ключ/command* одинаково успешно запускает макросы и их псевдонимы, а это дает доступ к обширной функциональности. Этот ключ можно применять для тестирования надстроек и макросов, выполнения операций над решениями и установки командных файлов, дающих доступ к собственным параметрам IDE.

Ключ */debugexe* открывает исполняемый файл в отладчике Visual Studio .NET в виде нового решения. Запустить исполняемый файл и подключиться к процессу позволяет команда *DebugStart.* На этом этапе вам будет предложено сохранить новое решение, и в отладчике запустится исполняемый файл. Затем можно прерваться и перейти к исходному коду или дизассемблированию исполняемого файла в зависимости от того, отлаживаете вы окончательную или отладочную версию программы, а также от того, может ли Visual Studio .NET найти исходные коды для исполняемого и зависимых файлов. Командная строка для открытия исполняемого файла в Visual Studio .NET может выглядеть так:

```
devenv /debugexe VBWinApp.exe
```

Если последней компоновкой этого файла была отладочная компоновка, Visual Studio .NET, вероятно, не встретит проблем в переходе к исходному коду файла. Если выполнялась финальная компоновка, вы можете перейти только к дизассемблированному коду. Тип компоновки обычно несколько проще определить в Visual C# и Visual C++, так как конечные файлы размещаются в папках debug или release в зависимости от типа сборки.

Ключ */useenv* используется только в Visual C++. Он позволяет приказать IDE задействовать переменные среды, установленные для командного сеанса, из которого запущен Devenv.exe. Параметр полезен для создания компоновок с особыми версиями библиотек периода выполнения. Например, вам могут потребоваться файлы из определенной версии Windows SDK или вы захотите протестировать бета-версии библиотек DirectX. Ключ */useenv* позволяет сделать это, не изменяя параметры в IDE.

Вот параметры Devenv.exe для компоновки из командной строки (табл. 15-1):

**Табл. 15-1. Параметры Devenv.exe для компоновки из командной строки**

| Параметр | Описание |
|---|---|
| */build* <сборка> | Указывает тип компоновки. |
| */clean* <сборка> | Для типов проектов, поддерживающих эту функцию, удаляются результаты компоновки указанного типа. |
| */command* <команда> | Указывает именованную команду, макрос или псевдоним Command Window для выполнения при запуске IDE, |
| */debugexe* <exe-файл> | Указывает исполняемый файл для отладки в IDE, |
| */deploy* | Для решений, включающих проект развертывания, приложение развертывается после компоновки. |
| */out* <имя_файла> | Указывает файл журнала для вывода информации о компоновках из командной строки. |
| */project* <проект> | Указывает проект из решения для открытия или компоновки. |
| */projectconfig* <конфигурация_проекта> | Указывает конфигурацию проекта при открытии проекта из решения. |
| */rebuild* <сборка> | Проекты, поддерживающие команду Clean, перед компоновкой очищаются. В противном случае все целевые файлы проекта собираются повторно. |
| */run* | Результат компоновки решения запускается в IDE. |
| */runexit* | IDE открывается в свернутом виде, и запускается результат компоновки решения. |
| */useenv* | Для компоновок в Visual C++ этот параметр позволяет приказать IDE использовать переменные среды, установленные для командного сеанса. |

## Установка параметров графического пользовательского интерфейса из командной строки

Параметры графического пользовательского интерфейса (GUI) позволяют менять некоторые установки IDE из командной строки. Какие-то из них могут показать-

ся бесполезными, но их можно задействовать для тестирования окон надстроек при различных параметрах GUI.

Ключи */mdi* и */mditabs* позволяют переходить из многодокументного интерфейса (MDI) в режим вкладок в окнах документов IDE. Помните: ваш выбор фиксируется в IDE, так что если вы экспериментируете с параметром */mdi* и забудете запустить Devenv.exe с ключом */mditabs*, то при следующем запуске IDE вкладки исчезнут.

Два ключа предназначены для работы со шрифтами, но влияют они только на IDE, не затрагивая окон документов: */fs* позволяет указать в пунктах размер текста, отображаемого в IDE, */fn* — указать имя нужного шрифта. Эти параметры также фиксируются, поэтому используйте их осторожно; указав бессмысленные значения, можно привести IDE в негодность. Вот результат присвоения параметру */fn* значения 14 (рис. 15-3):

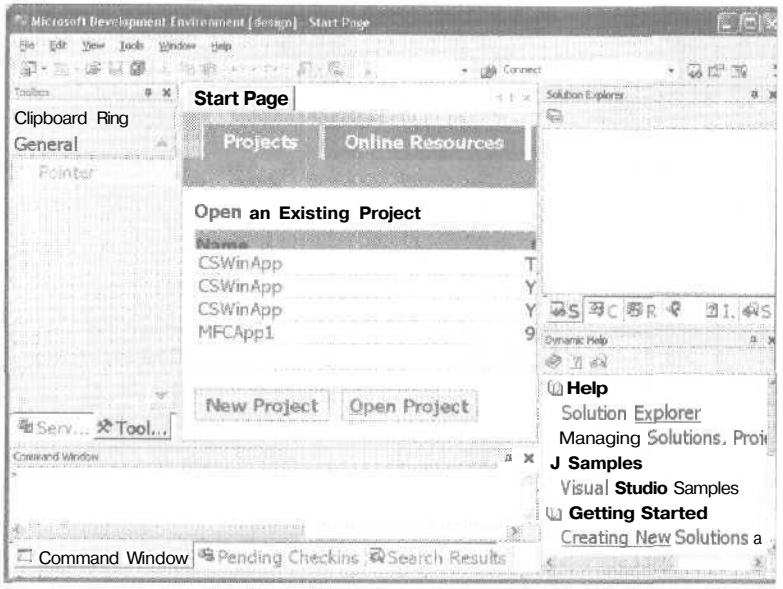



**Рис. 15-3.** *Не пытайтесь повторить это дома:* Devenv.exe /fn 14

Ключ */LCID* позволяет запустить IDE с указанным языком. Он может пригодиться при локализации надстроек. На машине, где запускается IDE, должна быть установлена версия Visual Studio .NET для нужного языка. Параметр можно применять для тестирования надстроек на разных языках без установки разных тестовых машин. Подписчики MSDN Universal получат доступ к различным версиям Visual Studio .NET.

Параметры командной строки для управления GUI таковы (табл. 15-2):

**Табл. 15-2. Параметры Devenv.exe для управления GUI из командной строки**

| Параметр | Описание |
|---|---|
| */mditabs* | Открывает IDE с вкладками в окнах документов. |
| */mdi* | Открывает IDE с MDI-окнами. |
| */fn* <*имя_шрифта*> | Использует в IDE указанный шрифт, |

*см. след. стр.*

**Табл. 15-2.** *(окончание)*

| Параметр | Описание |
|---|---|
| */fs* | Использует в IDE указанный размер шрифта. |
| */migratesettings* | Переносит некоторые пользовательские параметры из предыдущих версий Visual Studio .NET. |
| */LCID <ID_языка>* | Открывает IDE с использованием указанного языка. |

### Параметры VSIP

Остальные параметры позволяют партнерам Microsoft Visual Studio Integration Program (VSIP) проверять свои решения VSIP. Они не принесут вам пользы без доступа к этой программе, но для полноты материала мы их приведем (табл. 15-3):

**Табл. 15-3. Параметры Devenv.exe для управления средой**

| Параметр | Описание |
|---|---|
| */noVSIP* | Отключает номер лицензии VSIP (VSIP license key) для тестирования. |
| */safemode* | Загружает стандартную стабильную среду для тестирования, |
| */resetskippkgs* | Включает пакеты, отмеченные в IDE как пропускаемые. |

## Управление исходным кодом с помощью Visual SourceSafe

Visual Studio .NET поставляется с поддержкой VSS, а версии Visual Studio .NET Enterprise Developer и Enterprise Architect поставляются с необходимыми компонентами для установки VSS-сервера.

В этом разделе мы поговорим об установке и начале работы с VSS-сервером, а также об организации контроля версий и защиты исходных кодов проектов и помощи в управлении командной разработкой.

### Установка VSS

Даже если вы самостоятельный разработчик, попробуйте установить VSS-сервер на отдельной машине. При этом даже при отказе винчестеров вашей рабочей станции у вас сохранится одна из последних версий проекта в системе контроля исходных кодов на другом сервере.

### Размещение файлов в системе контроля исходных кодов

Для использования VSS в Visual Studio .NET нужно установить версию VSS, поставляемую с Visual Studio .NET. Далее мы будем предполагать, что вы правильно установили базу данных VSS и хотите работать с ней, используя встроенные инструменты Visual Studio .NET 2003.

При первом запуске Visual Studio .NET на рабочей станции вы увидите, что элементы меню Source Control затенены. Чтобы задействовать преимущества функций VSS, встроенных в Visual Studio .NET, на рабочей станции должен иметься клиент VSS. Обычно его можно установить из общего ресурса на сервере: найдите в общей папке VSS на сервере файл Netsetup.exe и запустите его.

Клиент VSS позволяет провести начальную установку VSS. Сначала откройте базу данных VSS — она понадобится для управления кодом. После установки клиента VSS в папке Microsoft Visual SourceSafe из меню Пуск вы найдете ссылку на исполняемый файл. Откройте клиент и нажмите Ctrl+P, чтобы отобразить диалоговое окно Open SourceSafe Database (рис. 15-4). Перейдите в общую папку с базой данных VSS и откройте файл srcsafe.ini.

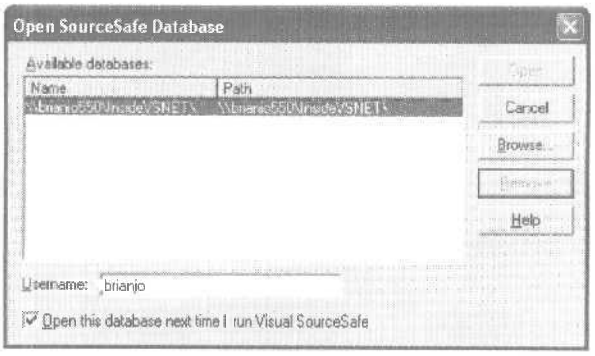



**Рис. 15-4.** *Диалоговое окно Open SourceSafe Database после выбора нужной базы данных*

В ответ на приглашение введите установленные вами или администратором имя пользователя и пароль. На рис. 14-5 показана база данных VSS, установленная для помощи в управлении кодами и главами для этой книги. Можете закрыть клиентское приложение VSS. Далее мы будем работать в основном из Visual Studio .NET.

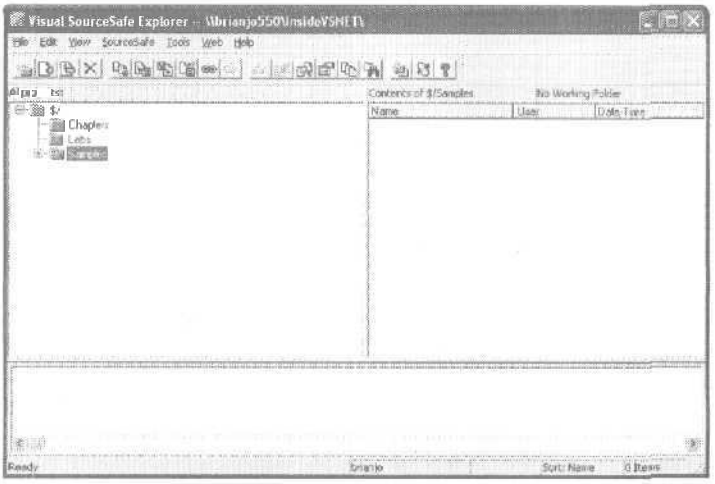



**Рис. 15-5.** *База данных VSS, открытая в клиентском приложении VSS*

Установив базу данных и получив к ней доступ с рабочей станции, можно начать размещение решений и проектов в системе контроля исходных кодов, Откройте решение в IDE или создайте новое решение. В Visual Studio .NET выберите из меню File команду Source Control, а затем — Add Solution To Source Control (рис. 15-6).

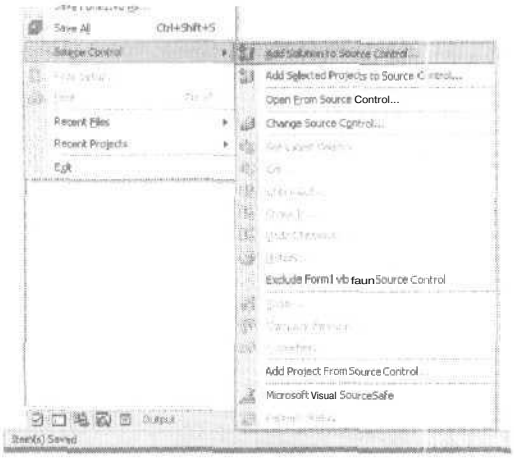



**Рис. 15-6.** *Добавление решения в VSS через меню Source Control в Visual Studio NET*

Вы увидите диалоговое окно Visual SourceSafe Login, а за ним — диалоговое окно Add To SourceSafe Project (рис. 15-7), в котором можно выбрать или создать папку для проекта.

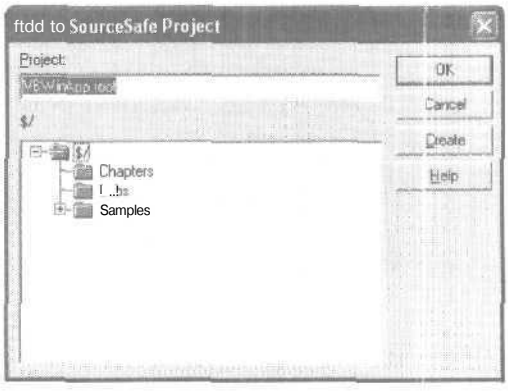



**Рис. 15-7.** *Диалоговое окно Add To SourceSafe Project*

При размещении решения в системе контроля исходных кодов оно добавляется на VSS-сервер. Решение, с которого вы начали, теперь рассматривается как рабочая копия (working copy), и для внесения изменений в файлы их надо получить (check out) из VSS. На рис. 15-8 — окно Solution Explorer для проекта с контролем исходных кодов. Значки замков в Solution Explorer показывают, что файлы скопированы на локальный компьютер, но не получены для редактирования. Рядом с именем полученных файлов отображается галочка.

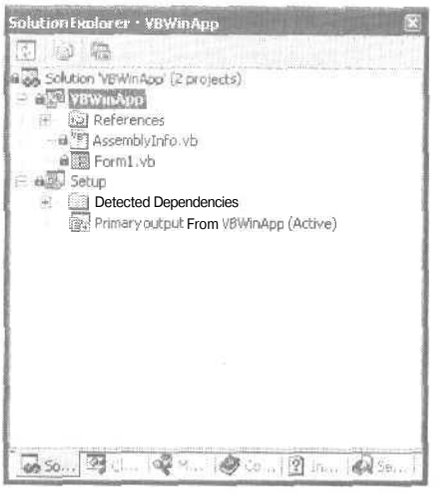



**Рис.** 15-8. *Решение с контролем исходных кодов*

## Работа с файлами при контроле исходных кодов

Поместив решение в систему контроля исходных кодов, вы и члены вашей команды можете получать к нему доступ из Visual Studio .NET. На машине, где было создано решение, добавленные в VSS файлы становятся рабочими копиями данного конкретного разработчика. Вы можете удалить локальную папку и открыть решение из VSS, скопировав его в новое место на рабочей станции.

Чтобы открыть проект с VSS-сервера, шелкните в меню File команду Source Control и затем выберите Open From Source Control или введите в Command Window команду **File.OpenFromSourceControl**. В открывшемся диалоговом окне Visual SourceSafe Login вы сможете ввести ваши учетные данные и выбрать базу данных VSS, к которой нужно получить доступ. Зарегистрировавшись, вы увидите диалоговое окно Create local project from SourceSafe (рис. 15-9).

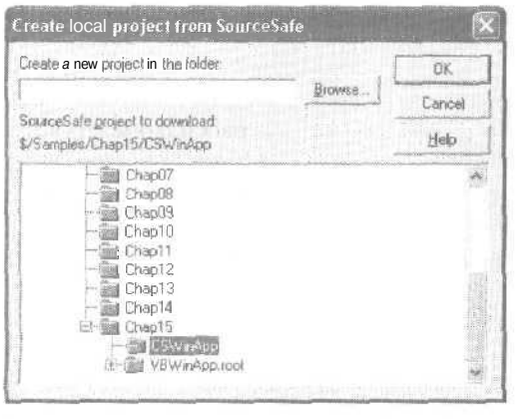



**Рис.** 15-9. *Выбор проекта из базы данных*

Выберите проект и щелкните OK. Вы увидите диалоговое окно Browse Folder, где можно выбрать рабочую папку для решения. Программа добавит выбранную папку в поле Create A New Project In The Folder. Щелкните OK еще раз. Файлы решения будут скопированы на локальный компьютер, и откроется диалоговое окно Open Solution с *.sln-файлом* решения. Выбрав .sln-файл в этом диалоговом окне, вы сможете начать работу с локальной копией решения.

Примечание В качестве рабочей папки для решения можно выбрать общий сетевой ресурс. В VSS часто бывает проще назначить сетевому ресурсу букву диска и использовать ее при работе со стандартными диалоговыми окнами старого типа.

Работа с файлами при контроле исходных кодов вполне очевидна. Когда вы начинаете редактировать файл, он блокируется на сервере, и в это время другой разработчик не может вносить изменения. Это и есть «контроль» исходных кодов.

Посмотрим, что происходит, когда два программиста работают над одним решением, используя контроль исходных кодов, Оба открыли решение через VSS в локальной папке или общем сетевом ресурсе и трудятся над проектами в Visual Studio .NET. Скажем, Марк добавляет кнопку в диалоговое окно. При попытке редактирования файла появляется диалоговое окно Check Out For Edit (рис. 15-10), позволяя оставить комментарий о том, что он делает, прежде чем получить файл для редактирования.

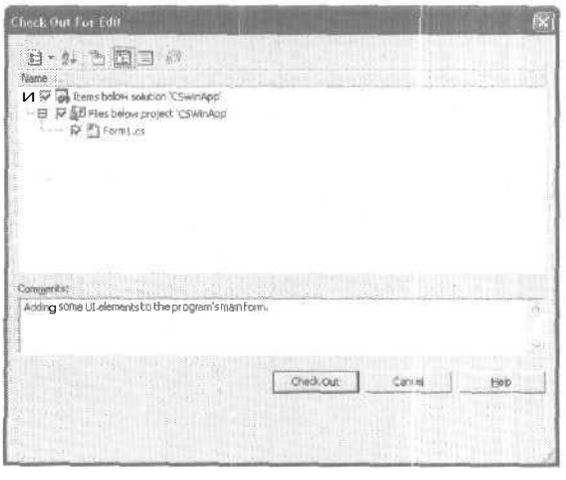



Рис. 15-Ю. *Получение файла из VSS для редактирования*

Помните: оба программиста непрерывно работают в Visual Studio .NET, и при этом им предлагается получать файлы для редактирования. Почти вся работа с базами данных VSS в Visual Studio .NET выполнятся автоматически, поэтому вам не придется охотиться на файлы с помощью клиента VSS.

Тут Брайан захотел внести изменения в форму, которую получил Марк. Марк еще не вернул файл с изменениями и владеет монопольным доступом к нему. Это значит, что Брайан не сможет изменить файл, пока он не будет возвращен. Когда

Брайан пытается получить этот файл для редактирования, он видит запрещающее диалоговое окно (рис. 15-11).

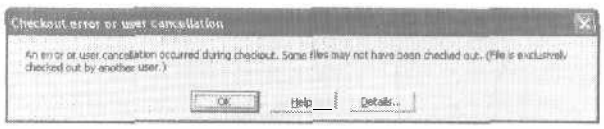



**Рис. 15-11.** *Сообщение, появляющееся при попытке получить для редактирования файл, уже полученный другим пользователем*

Если Брайану действительно нужен этот файл, он может попросить Марка вернуть его. Маленькие группы могут успешно работать по такой схеме, но в более крупных командах вам, вероятно, не захочется вмешиваться друг к другу в код. И все же, как видите, функции контроля исходных кодов позволяют узнать, кто контролирует файл в каждый момент времени, и не допускают изменение этого файла, пока человек, контролирующий файл, не вернет его в базу данных.

Скажем, Марк внес изменения в файл и готов вернуть его. Он может вернуть файлы, щелкнув правой кнопкой полученные файлы в Solution Explorer и выбрав из контекстного меню команду Check In. Откроется диалоговое окно Check In (рис. 15-12). Здесь он может ввести комментарий о возвращаемых файлах. Этот комментарий можно направить руководителю проекта или другим разработчикам.

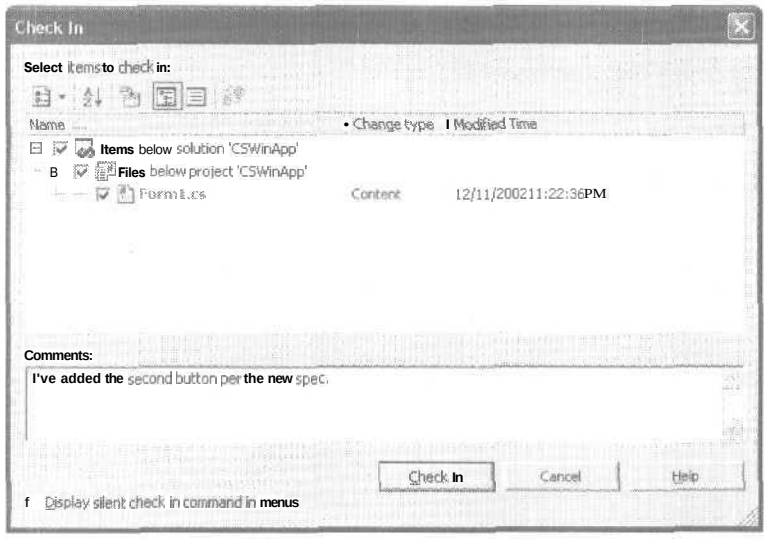



**Рис. 15-12.** *Возврат файла в VSS*

С этого момента Брайан работает с устаревшими копиями исходных файлов. Что происходит, если он пытается редактировать старую форму? Предоставляя файл для редактирования, VSS заменяет устаревшую версию файла на машине Брайана обновленным файлом из базы данных.

Для замены фалов с локального компьютера на последние версии из базы данных служит команда *File.GetLatestVersion*.Она также доступна из контекстного меню Solution Explorer. Если вы внесли изменения в полученный для редактирования

файл, вы можете занести их в базу данных или продолжить работу, не занося изменения в базу. В любом случае изменения, занесенные в базу другими разработчиками, применяются к исходным файлам в вашем решении.

Стоит отметить еще пару функций VSS, доступных из Visual Studio .NET. Первая — это команда History (*FileHistory*). Она работает в пофайловом режиме и доступна через подменю Source Control меню File или окно Command Window. Команда History открывает диалоговое окно, в котором перечислено, кто и когда получал файл для редактирования (рис. 15-13). Двойной щелчок любой версии из списка открывает диалоговое окно History Details, содержащее все примечания, добавленные при возвращении файла (рис. 15-14).

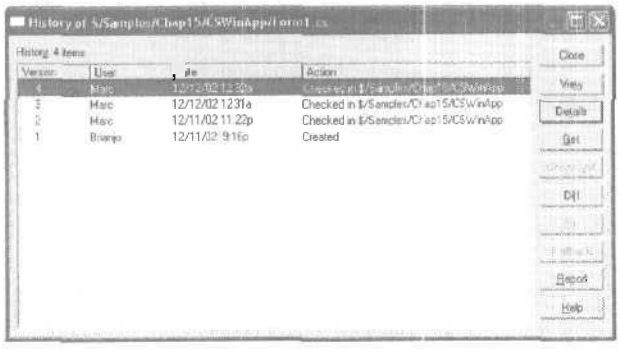



**Рис. 15-13.** *Просмотр хронологии файла в VSS*




**Рис. 15-14.** *Подробности к определенной версии файла*

Еще одна интересная особенность — окно инструментов Pending Checkins (рис. 15-15). Это стыкуемое окно доступно, только когда вами получены файлы для редактирования. Его можно вызвать через подменю Source Control из меню File или контекстное меню Solution Explorer. Это окно инструментов предлагает некоторые функции, которые не так просто найти в других диалоговых окнах.

Например, щелкнув кнопку Option на панели инструментов Pending Checkins, вы сможете удерживать файл полученным для редактирования, занося изменения в базу данных. Таким образом, можно предоставлять сделанные изменения другим разработчикам, в то же время сохраняя контроль над файлом.

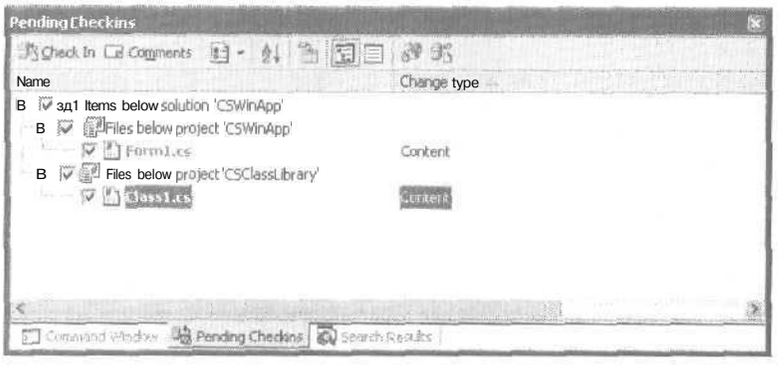
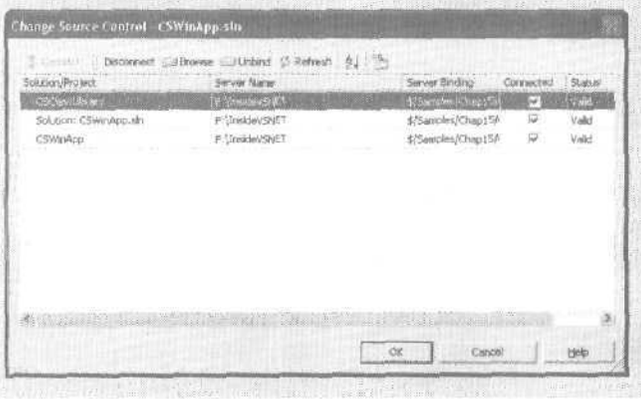



**Рис. 15-15.** *Окно инструментов Pending Checkins*

### Практикум: отключение контроля исходных кодов

Скачав решение, которое кто-то закачал в Web, вы можете обнаружить сообщения об ошибках, говорящих, что решение находилось в системе контроля исходных кодов. Открывать заархивированное решение, к которому кто-то применял контроль исходных кодов, — занятие не из приятных, но контроль исходных кодов легко удалить из решения через подменю Source Control из меню File в Visual Studio .NET.

Откройте решение CSWinApp из папки Chapter 15 сопроводительного материала книги. Вы увидите несколько диалоговых окон, сообщающих о том, что файлы решения находились под контролем исходных кодов. Чтобы избавиться от этих сообщений, надо отключить решение и проекты от контроля исходных кодов, под которым они находились. Дна этого нажмите Ctrl+Alt+A, чтобы открыть окно Command Window, и введите **File.Change-SourceControl**. Эта команда открывает диалоговое окно Change Source Control:




*см. след. стр.*

Чтобы отключить решение, выделите перечисленные компоненты и щелкните кнопку Unbind. Вы получите сообщение о том, что после отключения решения эти файлы больше не будут находиться под контролем исходных кодов. Щелкните OK, сохраните решение, и раздражающие сообщения исчезнут. Это можно проделать для любого решения, которое вы найдете в Web. Предоставляя свои решения для совместного использования, не забудьте отключить контроль исходных кодов перед публикацией. Разработчики, которые будут пользоваться вашим кодом, оценят это.

## Тщательное планирование решений

Работая в VSS со сложными решениями, содержащими много проектов, важно тщательно планировать решения, прежде чем внести их в систему контроля первый раз. Это избавит вас от многих проблем в будущем. Главное, перед первым занесением файлов надо помнить, что файл решения должен находиться в родительской папке, содержащей отдельные папки для каждого проекта в решении, Например, вы хотите создать решение, содержащее три отдельных компонента. В нем есть графический пользовательский интерфейс, написанный на Visual Basic .NET. Есть библиотека классов, написанная на Visual C#, и COM-компонент на Visual C++. (В главе 2 мы говорили о сочетании этик типов проектов в решении.)

Чтобы совместить все эти типы проектов в одном решении, нужно создать папку для хранения решения, создав пустое решение в любом удобном месте. Затем надо добавить новые проекты в пустое решение, поместив папки для этих проектов в папку с главным файлом решения.

Разместив все проекты в решении, можно вносить его в VSS описанным выше способом. Теперь системе контроля будет проще управлять ссылками в проекте, а разработчикам, занимающимся разными компонентами решения, — управлять собственными проектами с контролем исходных кодов.

**Примечание** Для планирования объемных работ больших команд в VSS Microsoft создала прекрасный документ, содержащий массу полезной информации о планировании решений, компоновок и контроля исходных кодов, — «Team Development with Visual Studio .NET and Visual SourceSafe» он доступен в MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/ tdlg_rm.asp.

## Не нарушайте компоновку

Контроль исходных кодов помогает выполнять регулярные компоновки, позволяя постоянно запускать и тестировать продукт. Такие компоновки следует выполнять как можно чаще, учитывая объем и масштаб всего проекта и размер команды. Поэтому, прежде чем добавить новый код в систему контроля, убедитесь, что он успешно собирается.

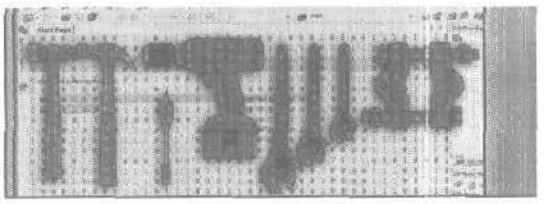
Идея контроля исходных кодов — в сохранении работоспособной версии продукта на протяжении всего времени. Если код, который вы внесли в базу данных VSS, нарушает компоновку, он может повлиять на работу других команд, уча-

ствующих в проекте. Поэтому вносите свой код, только когда он сможет компилироваться в составе продукта, не вызывая проблем. В сценариях со многими разработчиками не следует применять контроль исходных кодов как хранилище резервных копий текущей работы. Сохраняйте резервные копии проектов на своей рабочей станции и вносите их в систему контроля, только когда они готовы к запуску.

**Примечание** Разумеется, для одного пользователя VSS представляет прекрасное решение для резервного копирования.

## Что дальше

Б этой книге мы обсудили ряд вопросов использования и настройки Visual Studio .NET. По мере изучения этого удивительного инструмента и объектной модели автоматизации вы начнете видеть совершенно новые способы настройки и автоматизации IDE. Мы искренне надеемся, что вы создадите решения, в которых в той или иной степени вам помогут идеи, изложенные в этой книге.



ПРИЛОЖЕНИЕ

# Справочник по модели кода

В 12 главе мы обсуждали модель кода Visual Studio .NET, но некоторые детали мы оставили за скобками. Здесь все эти детали собраны в справочник, отражающий современное состояние дел по всем объектам, свойствам и методам модели кода.

## Объекты модели кода

В данном разделе объясняется назначение объектов модели кода и их свойства. Изучение методов модели кода мы отложим до раздела «Генерация кода».

### *FileCodeModel* и *CodeModel*

Объекты *FileCodeModel* и *CodeModel* — два пути, ведущих в модель кода; каждый из этих объектов включает набор элементов кода верхнего уровня и методы, позволяющие добавлять, удалять или изменять эти элементы кода. Свойства объектов класса *FileCodeModel* таковы (табл. A-1):

**Табл. A-1. Свойства класса *FileCodeModel***

| Свойство | Описание |
|---|---|
| *CodeElements* | Возвращает набор объектов *CodeElement* верхнего уровня, определенных в соответствующем исходном файле. |
| *DTE* | Возвращает объект *DTE*. |
| *Language* | Возвращает GUID, представляющий язык программирования исходного файла. |
| *Parent* | Возвращает объект *ProjectItem* соответствующего исходного файла. |

Объект *FileCodeModel* всегда принадлежит определенному исходному файлу проекта. Можно получить объект *FileCodeModel* через свойство *FileCodeModel* объек-

та *ProjectItem,* служащего оберткой для исходного файла; получив объект *FileCodeModel,* можно через его свойство *Parent* вернуться обратно к родительскому объекту *ProjectItem,* Важнейшим свойством объекта *FileCodeModel, с* точки зрения модели кода, является *CodeElements,* открывающее доступ к элементам кода верхнего уровня соответствующего исходного файла.

Свойство *Language* возвращает GUID, представляющий язык программирования исходного файла. Перечисление *EnvDTE.CodeModelLanguageConstants*определяет константы для GUID языков Visual Basic, Visual C#и Visual C++ (в перечислении не приводятся GUID для Visual J#: E6FDF8BF-F3D1-11D4-8576- 0002A516ECE8). Если вам тяжело запомнить, какой GUID какому языку соответствует, воспользуйтесь следующим кодом, который переводит эти GUID на английский:

```
string LanguageFromGUID(string langGuid)
{
    string language = String.Empty;

    switch (langGuid.ToUpper())
    {
        case CodeModelLanguageConstants.vsCMLanguageCSharp:
            language = "Visual C#";
            break;

        case CodeModelLanguageConstants.vsCMLanguageVB:
            language = "Visual Basic";
            break;

        case "{E6FDF8BF-F3D1-11D4-8576-0002A516ECE8}":
            language = "Visual J#";
            break;

        case CodeModelLanguageConstants.vsCMLanguageVC:
            language = "Visual C++";
            break;

        default:
            language = "Other";
            break;
    }

    return language;
}
```

**Важно!** Вы не можете просто открыть произвольный исходный файл в Visual Studio .NET и работать с его элементами посредством модели кода (скажем, вы не сможете получить объект *FileCodeModel*из исходного файла в каталоге Solution Items). Модель кода для исходного файла будет создана, только если вы включите этот файл в проект на соответствующем языке.

В табл. А-2 перечислены методы объектов класса *FileCodeModel*. Методы *Addxxx* и *Remove* мы рассмотрим в разделе «Генерация кода»; полное описание метода *CodeElementFromPoint* см. в разделе «Получение *CodeElement* из объекта точки» в главе 12.

**Табл. А-2. Методы класса *FileCodeModel***

| Метод | Описание |
|---|---|
| *AddAttribute* | Создает новый атрибут верхнего уровня. |
| *AddClass* | Создает новый класс верхнего уровня. |
| *AddDelegate* | Создает новый делегат верхнего уровня. |
| *AddEnum* | Создает новое перечисление верхнего уровня. |
| *AddFunction* | Создает новую функцию верхнего уровня. |
| *AddInterface* | Создает новый интерфейс верхнего уровня. |
| *AddNamespace* | Создает новое пространство имен верхнего уровня, |
| *AddStruct* | Создает новую структуру верхнего уровня. |
| *AddVariable* | Создает новую переменную верхнего уровня, |
| *CodeElementFromPoint* | Возвращает элемент кода, содержащий переданный объект *TextPoint*. |
| *Remove* | Удаляет указанный элемент кода. |

Объект *CodeModel*, возвращаемый свойством *Project.CodeModel*, предоставляет более подробный граф, чем объект *FileCodeModel* Поскольку объект *CodeModel* действует на уровне проекта, а не на уровне элемента проекта, в его наборе *CodeElements* можно ожидать большее количество элементов кода верхнего уровня, чем их можно увидеть в типичном наборе *FileCodeModel.CodeElements*. В зависимости от типа проекта объект *CodeModel* также предоставляет информацию, не включаемую объектом *FileCodeModel*, например, атрибуты уровня сборки и внешние пространства имен. Свойства класса *CodeModel* таковы (табл. А-3).

**Табл. А-3. Свойства класса *CodeModel***

| Свойство | Описание |
|---|---|
| *CodeElements* | Возвращает набор объектов *CodeElement* верхнего уровня, определенных в соответствующем проекте. |
| *DTE* | Возвращает объект *DTE*. |
| *IsCaseSensitive* | Указывает, является ли язык программирования чувствительным к регистру. |
| *Language* | Возвращает GUID, представляющий язык программирования проекта, |
| *Parent* | Возвращает родительский объект *Project*, |

Большинство свойств объекта *CodeModel* аналогично свойствам объекта *FileCodeModel*, но они представляют информацию уровня проекта. Эта информация высокого уровня иногда требуется, например, аналог *CodeModel.Language* может рассказать о языке программирования проекта, даже если проект не имеет файлов (и, следовательно, объекта *FileCodeModel*). Свойство *IsCaseSensitive* присутствует только у объекта *CodeModel* и позволяет определить, чувствителен ли язык программирования к регистру, что очень важно знать при генерации имен новых элементов кода.

Методы объектов класса *CodeModel* перечислены в табл. А-4. Исследование методов *Addxxx* и *Remove* мы отложим до раздела «Генерация кода*.

**Табл. А-4. Методы класса *CodeModel***

| Метод | Описание |
|---|---|
| *AddAttribute* | Создает новый атрибут верхнего уровня. |
| *AddClass* | Создаст новый класс верхнего уровня, |
| *AddDelegate* | Создает новый делегат верхнего уровня. |
| *AddEnum* | Создает новое перечисление верхнего уровня. |
| *AddFunction* | Создает новую функцию верхнего уровня. |
| *AddInterface* | Создает новый интерфейс верхнего уровня. |
| *AddNamespace* | Создает новое пространство имен верхнего уровня. |
| *AddStruct* | Создает новую структуру верхнего уровня. |
| *AddVariable* | Создает новую переменную верхнего уровня. |
| *CodeTypeFromFullName* | Возвращает объект *CodeType*, представляющий данный элемент кода. |
| *CreateCodeTypeRef* | Возвращает объект *CodeTypeRef*, представляющий тип полностью квалифицированного имени. |
| *IsValidID* | Возвращает, допустимо ли указанное имя как программный идентификатор для текущего языка программирования. |
| *Remove* | Удаляет указанный элемент кода из исходного файла. |

Метод *CodeTypeFromFullName* позволяет получить объект *CodeType* для определенного элемента кода по его полностью квалифицированному имени.

Метод *CreateCodeTypeRef позволяет* создать объект *CodeTypeRef* на основе полностью квалифицированного имени или значения перечисления *vsCMTypeRef* (табл. А-5).

И, наконец, метод *IsValidID* позволяет проверить, допустим ли данный идентификатор в некотором языке. Этот метод реализован во всех языках.

**Табл. А-5. Перечисление *vsCMTypeRef***

| Константа | Описание |
|---|---|
| *VsCMTypeRefOther* | Тип данных не из этой таблицы |
| *VsCMTypeRefCodeType* | *CodeType* |
| *VsCMTypeRefArray* | Массив |
| *VsCMTypeRefVoid* | Пустой |
| *VsCMTypeRefPointer* | Указатель |
| *VsCMTypeRefString* | Строка |
| *VsCMTypeRefObject* | Объект |
| *VsCMTypeRefByte* | Байт |
| *VsCMTypeRefChar* | Символ |
| *VsCMTypeRefShort* | Короткое целое |
| *VsCMTypeRefInt* | Целое |
| *VsCMTypeRefLong* | Длинное целое |
| *VsCMTypeRefFloat* | С плавающей точкой |

*см. след. стр.*

**Табл. А-5. Перечисление *vsCMTypeRef* *(окончание)***

| Константа | Описание |
|---|---|
| *VsCMTypeRefDouble* | С плавающей точкой двойной точности |
| *VsCMTypeRefDecimal* | Десятичное |
| *VsCMTypeRefBool* | Логическое |
| *VsCMTypeRefVariant* | Универсальный |

## CodeElement

*CodeElement* — это нечто вроде «базового класса» для остальных объектов модели кода. Он представляет ряд свойств, общих для всех типов программных конструкций. Свойства объектов класса *CodeElement* таковы (табл. А-6):

**Табл. А-6. Свойства класса *CodeElement***

| Свойство | Описание |
|---|---|
| *Children* | Возвращает набор всех объектов *CodeElement,* относящихся к данному элементу кода. |
| *Collection* | Возвращает родительский набор объектов *CodeElements.* |
| *DTE* | Возвращает объект *DTE.* |
| *EndPoint* | Возвращает объект *TextPoint,* отмечающий конец определения данного элемента кода. |
| *Extender** | Возвращает запрошенный объект-расширитель. |
| *ExtenderCATID* | Возвращает идентификатор (ID) категории расширителя. |
| *ExtenderNames* | Возвращает имена доступных объектов-расширителей, |
| *FullName* | Возвращает полностью квалифицированное имя данного элемента кода. |
| *InfoLocation* | Возвращает значение *vsCMInfoLocation,* которое описывает, где определен данный элемент кода. |
| *IsCodeType* | Указывает, относится ли данный элемент кода к объектам *CodeType.* |
| *Kind* | Возвращает значение *vsCMElement,* описывающее определенный тип элементов кода. |
| *Language* | Возвращает язык программирования, использованный для создания данного элемента кода. |
| *Name* | Задает или возвращает имя данного элемента кода. |
| *ProjectItem* | Возвращает объект *ProjectItem,* содержащий данный элемент кода. |
| *StartPoint* | Возвращает объект *TextPoint,* помечающий начало определения данного элемента кода. |

* C# не позволяет ссылаться на это свойство в операторе присваивания, поскольку его аксессор *get* имеет параметр. Используйте явный вызов аксессора *get.*

Три важнейших свойства объектов *CodeElement: Name, FullName* и *Kind. Name,* доступное для чтения и записи, позволяет программно получить и присвоить имя элементу кода. [В Visual Basic для свойства *Name* возможность записи (присвоения) не поддерживается.] Следует, однако, заметить, что не всегда можно присвоить элементу кода другое имя — например, нельзя изменить имя конструктора, так как имя конструктора должно совпадать с именем его родительского класса.

*FullName* возвращает полностью квалифицированное имя элемента кода. *Kind* возвращает значение из перечисления *vsCMElement*, идентифицирующее конструкцию кода, которой является данный элемент кода. Перечисление *vsCMElement* содержит 40 констант, представляющих всевозможные конструкции кода, которые могут вам понадобиться (а также несколько невразумительных конструкций, которые, мы надеемся, вам не понадобятся).

Свойство *Children* возвращает набор *CodeElements,* содержащий все элементы кода, относящиеся к данному элементу. Языкам не обязательно поддерживать это свойство, и большинство языков его не поддерживают. Из четырех языков программирования, включенных в коробочную поставку Visual Studio .NET, только Visual C++ использует свойство *Children*.

**Примечание** Когда вы узнаете о модели кода побольше, вы увидите, что Visual C++ принес с собой массу своих идиосинкразии. Когда Visual Studio .NET пригласила разные языковые группы присоединиться к своей модели кода, Visual C++ появился с собственной моделью кода. По большей части Visual C++ выполняет свою работу по реализации модели кода Visual Studio .NET (конечно, опираясь на собственную модель кода), но в результате в модели кода Visual Studio .NET были сделаны уступки, обеспечивающие более широкий доступ к «родной» модели кода Visual C++. Это и чужеродные свойства, такие как *CodeElement.Children*, и прочие вкрапления Visual C++ в разных частях модели кода, например, среди констант перечисления *vsCMElement*.

Свойство *InfoLocation* возвращает значение, принадлежащее перечислению *vsCMInfoLocation,* позволяющее узнать, где найти данную конструкцию кода. В табл. А-7 представлены константы из *vsCMInfoLocation*.

**Табл. А-7. Константы перечисления *vsCMInfoLocation***

| Константа | Описание |
|---|---|
| *vsCMInfoLocationProject* | Элемент кода в файле проекта. |
| *vsCMInfoLocationExternal* | Элемент кода во внешнем файле. |
| *vsCMInfoLocationVirtual* | Эта константа не используются в Visual Studio .NET 2003. |
| *vsCMInfoLocationNone* | Модель кода не может определить местонахождение данного элемента кода. |

Когда значение свойства *InfoLocation* равно *vsCMInfoLocationProject* свойства *StartPoint* и *EndPoint* возвращают объекты *TextPoint,* определяющие границы данного элемента кода в исходном файле.

Свойства *Extender, ExtenderCATID* и *ExtenderNames* позволяют обращаться к объектам-расширителям, относящимся к данному элементу кода (объекты-расширители позволяют добавить, скрыть или заменить свойства элемента кода, к которому они относятся, когда вы просматриваете эти свойства в IDE).

Свойства *Language* и *ProjectItem* эквивалентны свойствам *FileCodeModel.Language* и *FileCodeModel.Parent* соответственно. Свойство *Collection* возвращает родительский набор *CodeElements.* Свойство *IsCodeType* возвращает *true,* когда объект *CodeElement* также поддерживает интерфейс *CodeType* (о нем см. ниже). Вот методы объектов класса *CodeElement* (табл. А-8);

**Табл. А-8. Методы класса *CodeElement***

| Метод | Описание |
|---|---|
| *GetEndPoint* | Возвращает объект *TextPoint*, отмечающий конец определения данного элемента кода. |
| *GetStartPoint* | Возвращает объект *TextPoint,* отмечающий начало определения данного элемента кода. |

Методы *GetStartPoint и GetEndPoint* возвращают объект *TextPoint*, который отмечает соответственно начало и конец некоторого аспекта определения данного элемента кода. Интересующий аспект передается методу как одна из констант перечисления *vsCMPart* (табл. А-9). Методы *GetStartPoint* и *GetEndPoint* гибче, чем их двойники *StartPoint* и *EndPoint*, фактически *CodeElement.StartPoint и CodeElement.EndPoint* эквивалентны *CodeElement.GetStartPoint(vsCMPartWholeWithAttributes)* и *CodeElement.GetEndPoint(vsCMPartWholeWithAttributes)* соответственно. Чтобы убедиться в гибкости методов *GetStartPoint* и *GetEndPoint,* можете запустить макрос *Chapter12\CodeDiscovery\TextFromStartAndEndPoints*, который вызывает эти два метода с каждым из значений из перечисления *vsCMPart* и отображает результаты.

**Табл. А-9. Перечисление *vsCMPart***

| Константа | Возвращает |
|---|---|
| *VsCMPartName* | Имя данной конструкции языка. |
| *VsCMPartAttributes* | Применяемые к данной конструкции кода атрибуты за вычетом ограничителей атрибутов. |
| *VsCMPartHeader* | Заголовок конструкции кода, |
| *VsCMPartWhole* | Конструкция кода полностью |
| *VsCMPartBody* | Тело конструкции кода, за вычетом ограничителей тела. |
| *VsCMPartNavigate* | Место в исходном коде, куда переместится каретка, когда вы дважды щелкнете элемент в Class View. |
| *vsCMPartAttributesWithDelimiter* | Применяемые атрибуты вместе с ограничителями атрибутов. |
| *vsCMPartBodyWithDelimiter* | Тело данной конструкции кода вместе с ограничителями тела. |
| *vsCMPartHeaderWithAttributes* | Заголовок конструкции кода и его атрибуты. |
| *vsCMPartWholeWithAttributes* | Вся конструкция полностью вместе с атрибутами. |

## Специализированные объекты модели кода

Ниже описаны объекты, относящиеся напрямую к определенным конструкциям кода. Каждый из этих объектов, помимо членов *CodeElement*, имеет собственные специфические члены, что образует очень длинный список членов в файлах справки. Мы не приводим общих членов для всех *CodeElement*, чтобы сосредоточиться на членах, специфичных для каждого типа. Мы также не приводим свойств, включенных в табл. А-10: не будучи свойствами *CodeEletnent,* они являются общими для всех типов, рассматриваемых в данном разделе.

**Табл. A-10. Свойства, общие для всех типов модели кода**

| Свойство | Описание |
|---|---|
| *Comment* | Присваивает или возвращает комментарии, связанные с элементом кода. |
| *DocComment* | Присваивает или возвращает документируемые комментарии элемента кода. |
| *Parent* | Возвращает родительский набор *CodeElements*. |

Свойство *DocComment* позволяет создавать документируемые посредством XML комментарии для языков, их поддерживающих (скажем, C# и J#). Строка, передаваемая свойству *DocComment,* должна содержать допустимый XML-текст, заключенный в тэги *<doc></doc>*.

Свойство *Comment* создает обычные комментарии (C# и J#, однако, своеобразно реализуют функциональность записи свойства *Comment,* преобразуя обычную строку в документируемый XML-комментарий *<summary>*).

### CodeNamespace

Объекты класса *CodeNamespace* относятся к конструкциям пространства имен .NET (*package* в J#, *namespace* в C# и C++, а также *Namespace* в Visual Basic). Объекты *CodeNamespace* имеют следующие свойства (табл. A-11):

**Табл. A-11. Свойства класса *CodeNamespace***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. А-6. |
| Прочие общие свойства | См. табл. А-10, |
| *Members* | Возвращает элементы кода верхнего уровня, входящие в данное пространство имен. |

Как видно из табл. A-11. единственным специфическим свойством класса *CodeNamespace* является *Members*, которое позволяет получить доступ к элементам кода верхнего уровня, входящим в данное пространство имен.

Табл. A-12 показывает методы класса *CodeNamespace,* которые мы обсудим в разделе «Генерация кода».

**Табл. A-12. Методы класса *CodeNamespace***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. A-7 |
| *AddClass* | Создает в данном пространстве имен новый класс. |
| *AddDelegate* | Создает в данном пространстве имен новый делегат. |
| *AddEnum* | Создает в данном пространстве имен новое перечисление |
| *AddInterface* | Создает в данном пространстве имен новый интерфейс. |
| *AddNamespace* | Создает в данном пространстве имен новое пространство имен, |
| *AddStruct* | Создает в данном пространстве имен новую структуру. |
| *Remove* | Удаляет заданный элемент кода. |

### CodeType

Класс *CodeType* — это обобщающий класс, как и *CodeElement*. В качестве объекта такого класса можно рассматривать все, что является типом в .NET Framework. Таким образом, можно рассматривать конкретные типы модели кода *(CodeClass, CodeStruct, CodeDelegate, CodeInterface* и *CodeEnum)* как объекты одного типа — *CodeType*. Свойства объектов класса *CodeType* (табл. A-13) таковы:

**Табл. A-13. Свойства класса *CodeType***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. A-6. |
| Прочие общие свойства | См. табл. A-10. |
| *Access* | Задает или возвращает модификаторы доступа. |
| *Attributes* | Возвращает набор атрибутов. |
| *Bases* | Возвращает набор базовых типов. |
| *DerivedTypes* | Возвращает набор производных типов. |
| *IsDerivedFrom** | Возвращает, имеет ли этот тип в качестве базового другой тип. |
| *Members* | Возвращает набор элементов кода верхнего уровня, входящих в этот тип. |
| *Namespace* | Возвращает объект *CodeNamespace*, представляющий родительское пространство имен. |

* C# не позволяет ссылаться на это свойство в операторе присваивания, поскольку его аксессор *get* имеет параметр. Используйте явный вызов аксессора *get*.

Свойство *Access* задает или возвращает значения из перечисления *vsCMAccess*, определяющие доступ к данному элементу кода (public, private и т. д.). Помните, что возможность записи в свойство *CodeTypeAccess* отсутствует в Visual C++ и Visual Basic. Вот константы перечисления *vsCMAccess* (табл. A-14):

**Табл. A-14. Перечисление *vsCMAccess***

| Константа | Описание |
|---|---|
| *vsCMAccessPublic* | Открытый доступ (public) |
| *vsCMAccessPrivate* | Закрытый доступ (private) |
| *vsCMAccessProject* | Доступ из проекта |
| *vsCMAccessProtected* | Защищенный доступ (protected) |
| *vsCMAccessProjectOrProtected* | Комбинация доступа из проекта и защищенного доступа |
| *vsCMAccessDefault* | Доступ по умолчанию |
| *vsCMAccessAssemblyOrFamily* | Доступ из сборки или семейства |
| *vsCMAccessWithEvents* | Доступ *WithEvents* |

Свойство *Attributes* возвращает набор объектов *CodeElement*, имеющих тип *CodeAttribute*, по одному на каждый атрибут, применяемый к данному объекту *CodeType*. Свойство *Bases* возвращает набор объектов *CodeElement*, имеющих тип *CodeClass*; каждый *CodeClass* представляет собой базовый класс для данного объекта *CodeType*.

Свойство *IsDerivedFrom* позволяет определить, есть ли в текущем проекте другой элемент кода, который являлся бы одним из базовых для данного объекта *CodeType*. В настоящее время реализация Visual C++ работает корректно и для базовых классов, и для базовых интерфейсов, реализации C# и J# работают корректно только для классов, а реализация Visual Basic не работает вообще.

Свойство *DerivedTypes* возвращает набор объектов *CodeElement*, обозначающих прочие конструкции кода из текущего проекта, которые наследуют от данного объекта. Сейчас ни один из языков не поддерживает это свойство.

Свойство *Members* возвращает набор элементов *CodeElement*, которые представляют конструкции кода верхнего уровня, содержащиеся в данном объекте *CodeType*. И, наконец, свойство *Namespace* возвращает родительское пространство имен.

Табл. A-15 показывает методы класса *CodeType*, а их описание см. в разделе «Генерация кода».

**Табл. A-15. Методы класса *CodeType***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. A-7, |
| *AddAttribute* | Создает новый атрибут. |
| *AddBase* | Добавляет новый базовый тип. |
| *RemoveBase* | Удаляет базовый тип, |
| *RemoveMember* | Удаляет член — элемент кода. |

### *CodeClass* и *CodeStruct*

Объекты классов *CodeClass* и *CodeStruct* представляют соответственно классы и структуры. В семействе языков программирования C, классы и структуры очень близки — книги по C++ для начинающих часто представляют конструкцию class, как структуру, чьи члены по умолчанию имеют доступ private. Модель кода также рассматривает *CodeClass* и *CodeStruct* как близкие; объекты этих классов имеют одинаковые свойства и методы, и именно поэтому мы рассматриваем их вместе.

**Табл. A-16. Свойства классов *CodeClass* и *CodeStruct***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. A-6. |
| Свойства *CodeType* | См. табл. A-11. |
| Прочие общие свойства | См. табл. A-10. |
| *ImplementedInterfaces* | Возвращает набор реализованных интерфейсов, |
| *IsAbstract* | Задает или возвращает, является ли данный элемент абстрактным. |

Свойство *ImplementedInterfaces* возвращает набор *CodeElements*, содержащий интерфейсы, реализованные в данном классе или структуре. В C#, J# и Visual Basic это свойство поддерживается корректно; в Visual C++ реализованные интерфейсы включаются в набор *Bases*, а свойсгво *ImplementedInterfaces* всегда возвращает пустой набор.

Свойство *IsAbstract* указывает, является ли данный объект абстрактным классом или структурой. Во всех языках это свойство можно прочитать, но только в C# и J# в него можно записать.

Табл. A-17 перечисляет методы классов *CodeClass* и *CodeStruct,* их описание см. в разделе «Генерация кода».

**Табл. A-17. Методы классов *CodeClass* и *CodeStruct***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. A-7. |
| Методы *CodeType* | См. табл. A-12. |
| *AddClass* | Создает новый класс внутри данного класса или структуры. |
| *AddDelegate* | Создает новый делегат внутри данного класса или структуры, |
| *AddEnum* | Создает ноше перечисление внутри данного класса или структуры. |
| *AddFunction* | Создает новую функцию внутри данного класса или структуры. |
| *AddImplementedInterface* | Добавляет реализуемый интерфейс к классу или структуре. |
| *AddProperty* | Создает новое свойство в данном классе или структуре, |
| *AddStruct* | Создает новую структуру в данном классе или структуре. |
| *AddVariable* | Создает новую переменную в данном классе или структуре. |
| *RemoveInterface* | Удаляет реализуемый интерфейс из класса или структуры, |

### CodeInterface

Объект класса *CodeInterface* инкапсулирует конструкцию кода — интерфейс. В табл. A-18 показаны свойства класса *CodeInterface;* как видите, он не определяет специфических интерфейсных свойств.

**Табл. A-18. Свойства класса *CodeInterface***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. A-6. |
| Свойства *CodeType* | См. табл. A-11. |
| Прочие общие свойства | См. табл. A-10. |

Методы класса *CodeInterface* приведены в табл. A-19. Пояснения к методам *Addxxx* см. в разделе «Генерация кода».

**Табл. A-19. Методы класса *CodeInterface***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. A-7. |
| Методы *CodeType* | См. табл. A-12. |
| *AddFunction* | Создает новую функцию в интерфейсе, |
| *AddProperty* | Создает новое свойство в интерфейсе. |

### CodeEnum

Объект класса *CodeEnum* представляет перечисление. Как видно из табл. A-20, класс *CodeEnum* не определяет специфических для перечисления свойств.

**Табл. A-20. Свойства класса *CodeEnum***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. A-6, |
| Свойства *CodeType* | См. табл. A-11. |
| Прочие общие свойства | См. табл. A-10. |

Методы класса *CodeEnum* перечислены в табл. A-21, пояснения к ним см. в разделе «Генерация кода».

**Табл. A-21.** Методы **класса *CodeEnum***

| **Свойство** | Описание |
|---|---|
| Методы *CodeElement* | См. табл. A-7. |
| Методы *CodeType* | См. табл. A-12. |
| *AddMember* | Создает новую константу в перечислении. |

### CodeDelegate

Объект класса *CodeDelegate* представляет конструкцию кода — делегат. Свойства класса *CodeDelegate* таковы (табл. A-22):

**Табл. A-22. Свойства класса *CodeDelegate***

| **Свойство** | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. A-6. |
| Свойства *CodeType* | См. табл. A-11. |
| Прочие общие свойства | См. табл. A-10. |
| *BaseClass* | Возвращает объект *CodeClass*, представляющий базовый класс делегата. |
| *Parameters* | Возвращает набор *CodeElements* содержащий параметры делегата. |
| *Prototype*[1] | Возвращает прототип делегата, |
| *Type* | Задает или возвращает объект *CodeTypeRef* представляющий тип делегата. |

[1] C# не позволяет ссылаться на это свойство в операторе присваивания, поскольку его аксессор *get* имеет параметр. Используйте явный вызов аксессора *get*.

Свойство *BaseClass* всегда возвращает объект *CodeClass* представляющий *System.Delegate*. Это свойство поддерживается всеми языками, кроме Visual C++.

Свойство *Parameters* возвращает набор объектов *CodeElement*, по одному для каждого параметра делегата. Visual C++ не поддерживает это свойство.

Свойство *Prototype* возвращает прототип делегата в виде строки и получает значение из перечисления *vsCMPrototype*, определяющего, какие аспекты прототипа нужно возвратить. Константы перечисления *vsCMPrototype* (табл. A-23) являются битовыми флагами; их можно комбинировать, настраивая состав строки, возвра-

щаемой данным свойством. Так, комбинация *vsCMPrototypeFullnamevsCMPrototype-Type*, *vsCMPrototypeParamNames*и *vsCMPrototypeParamTypes*приведет к возврату полной информации о данном делегате. Все языки, кроме Visual C++, поддерживают это свойство.

**Табл. А-23. Перечисление *vsCMPrototype***

| Константа | Описание |
|---|---|
| *VsCMPrototypeFullname* | Возвратить полностью квалифицированное имя. |
| *VsCMPrototypeNoName* | Опустить имя. |
| *vsCMPrototypeClassName* | Возвратить имя и префикс класса. |
| *vsCMPrototypeParamTypes* | Возвратить типы параметров. |
| *vsCMPrototypeParamNames* | Возвратить имена параметров. |
| *vsCMPrototypeParamDefaultValues* | Возвратить значения по умолчанию для параметров. |
| *vsCMPrototypeUniqueSignature* | Возвратить уникальную строку, основанную на прототипе. |
| *VsCMPrototypeType* | Возвратить тип. |
| *vsCMPrototypeInitExpression* | Возвратить выражение инициализации. |

И, наконец, свойство *Type* позволяет читать и записывать значения *CodeTypeRef*, представляющие тип делегата. В насгоящее время возможность записи этого свойства нигде не реализована, а возможность чтения работает только в C#, *Jit* и Visual Basic.

Методы класса *CodeDelegate* показаны в табл. А-24, их описание см. в разделе «Генерация кода».

**Табл. А-24. Методы класса *CodeDelegate***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. А-7. |
| Методы *CodeType* | См. табл. А-12. |
| *AddParameter* | Создает новый параметр делегата. |
| *RemoveParameter* | Удаляет параметр делегата. |

## CodeVariable

Объект класса *CodeVariable* представляет описание переменной. В табл. А-25 перечислены свойства класса *CodeVariable*.

**Табл. А-25. Свойства класса *CodeVariable***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. А-б. |
| Прочие общие свойства | См. табл. А-10. |
| *Access* | Задает или возвращает модификаторы доступа к переменной. |
| *Attributes* | Возвращает набор атрибутов переменной, |
| *InitExpression* | Задает или возвращает код инициализации переменной. |
| *IsConstant* | Задает или возвращает, является ли эта переменная неизменяемой. |

**Табл. А-25. Свойства класса *CodeVariable*** *(окончание)*

| Свойство | Описание |
|---|---|
| *IsShared* | Задает или возвращает, является ли эта переменная общей переменной класса. |
| *Prototype** | Возвращает объявление переменной |
| *Type* | Задает или возвращает объект *CodeTypeRef* представляющий тип переменной. |

* C# не позволяет ссылаться на это свойство в операторе присваивания, поскольку его аксессор *get* имеет параметр. Используйте явный вызов аксессора *get*.

Свойство *Access* похоже на свойство *CodeTypeAccess* и принимает те же значения *vsCMAccess* (табл. А-14). Во всех языках реализована возможность чтения этого свойства, записи — только в C# и J#.

Свойство *Attributes* возвращает набор *CodeElements*, содержащий по одному объекту типа *CodeAttribute* для каждого атрибута данной переменной.

Свойство *InitExpression* позволяет читать или задавать выражение инициализации для переменной. Все языки поддерживают возможность чтения этого свойства. И все, кроме Visual Basic, поддерживают запись в него. Выражение, передаваемое свойству *InitExpression,* не должно включать оператор присваивания, поскольку корректный оператор будет подставляться самим языком. Учитывайте, что правильность передаваемого этому свойству выражения не проверяется.

Свойство *IsConstant* позволяет читать или задавать, является ли данная переменная неизменяемой. Все языки поддерживают чтение значения этого свойства, но реализация J# всегда возвращает *False.* Все языки, кроме Visual Basic, поддерживают запись в это свойство.

Свойство *IsShared* позволяет читать и записывать, является ли эта переменная общей переменной класса (значение *True)* или переменной экземпляра класса *(False).* Возможность чтения значения этого свойства реализована для всех языков, записи — в C# и J#.

Свойство *Prototype* возвращает строку, содержащую объявление переменной. Это свойство работает так же, как свойство *CodeDelegate.Prototype*: вы передаете ему комбинацию значений из перечисления *vsCMPrototype* (табл. А-23), и оно возвращает запрошенные вами части объявления. В отличие от свойства *InitExpression* свойство *Prototype* вместе с выражением инициализации возвращает и оператор присваивания (если ему передается значение *vsCMPrototypeInitExpression*). Все языки реализуют это свойство.

Свойство *Type* задает или возвращает объект *CodeTypeRef* представляющий тип переменной. Все языки реализуют возможность чтения этого свойства, но записи — только Visual C++.

В табл. А-26 перечислены методы класса *CodeVariable.* Пояснения к ним см. в разделе «Генерация кода».

**Табл. А-26. Методы класса *CodeVariable***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. А-7. |
| *AddAttribute* | Создает новый атрибут для переменной. |

## CodeProperty

Объект класса *CodeProperty* представляет свойство. В табл, А-27 приведены свойства класса *CodeProperty*. Заметим, что реализация J# не опознает объявление свойства — вместо этого J# интерпретирует методы *get_xxx* и *set_xxx* свойства как объекты *CodeFunction*. Соответственно пояснения к табл. А-27 не касаются J#.

**Табл. А-27. Свойства класса *CodeProperty***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. А-6. |
| Прочие общие свойства | См. табл. А-10. |
| *Access* | Задает или возвращает модификаторы доступа свойства. |
| *Attributes* | Возвращает набор атрибутов свойства. |
| *Getter* | Задает или возвращает объект *CodeFunction*, представляющий функцию-получатель (getter) свойства. |
| *Prototype* | Возвращает прототип свойства. |
| *Setter* | Задает или возвращает объект *CodeFunction,* представляющий функцию-установщик (setter) свойства. |
| *Type* | Задает или возвращает *CodeTypeRef* представляющий тип свойства. |

Свойство *Access* позволяет читать/записывать модификаторы доступа свойства (табл. А-14). Все языки поддерживают возможность чтения этого свойства, но лишь C# реализует запись в него, Visual Basic возвращает «not implemented», а Visual C++ допускает для свойств только открытый (public) доступ и всегда возвращает «read-only».

Свойство *Attributes* возвращает набор *CodeElements*, содержащий по элементу типа *CodeAttribute* для каждого атрибута, применяемого к свойству.

Свойства *Getter* и *Setter* позволяют читать/задавать для свойства получатель и установщик соответственно. *Getter* и *Setter* принимают или возвращают объект типа *CodeFunction,* который представляет соответствующий получатель или установщик свойства. C# и Visual Basic поддерживают возможность чтения этих свойств. Ни один из языков не поддерживает их запись, но для некоторых языков можно добиться того же результата, извлекая объект *CodeFunction*, соответствующий получателю или установщику и внося изменения через методы *CodeFunction,*

Свойство *Prototype* возвращает прототип свойства в виде строки. Свойство *Prototype* принимает комбинацию значений из перечисления *vsCMPrototype* (табл. А-23) и возвращает запрошенную информацию о прототипе. Заметим, что *CodeProperty.Prototype* возвращает информацию только об объявлении свойства и не включает сведений о прототипах получателя и установщика свойства. Свойство *Prototype* реализовано во всех языках,

Свойство *Type* позволяет читать и записывать значения типа *CodeTypeRef* представляющие тип свойства. Все языки поддерживают возможность чтения его значения, и ни один — записи. Однако, хотя нельзя изменить тип свойства в Visual C++ напрямую, это можно сделать окольным путем — изменив типы функций получателя и установщика данного свойства.

В табл. А-28 перечислены методы класса *CodeProperty,* пояснения к ним см. в разделе «Генерация кода».

**Табл. А-28. Методы класса *CodeProperty***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. А-7. |
| *AddAttribiile* | Создает для свойства новый атрибут, |

### CodeAttribute

Объект класса *CodeAttribute* относится к атрибуту, как он определяется в .NET Framework. В табл. А-29 показаны свойства класса *CodeAttribute.*

**Табл. А-29. Свойства класса *CodeAttribute***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. А-б. |
| Прочие общие свойства | См. табл. А-10. |
| *Value* | Задает или возвращает значение атрибута, |

Свойство *Value* позволяет читать/записывать значение атрибута. Все языки поддерживают возможность чтения, но J# возвращает ошибочную информацию, если атрибут имеет более одного параметра. Только Visual C++ реализует возможность записи этого свойства.

Табл. А-30 показывает методы класса *CodeAttribute* пояснения к которым см. в разделе «Генерация кода»,

**Табл. А-30. Методы класса *CodeAttribute***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. А-7. |
| *Delete* | Удаляет данный атрибут. |

### CodeFunction

Объект класса *CodeFunction* представляет функцию или процедуру. В табл. А-31 перечислены свойства класса *CodeFunction.*

**Табл. А-31. Свойства класса *CodeFunction***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. А-б. |
| Прочие общие свойства | См. табл. А-10. |
| *Access* | Задает или возвращает модификаторы доступа к функции. |
| *Attributes* | Возвращает набор атрибутов функции. |
| *CanOverride* | Задает или возвращает возможность переопределения функции. |
| *FunctionKind* | Возвращает значение из перечисления *vsCMFunction,* обозначающее, к какому виду относится функция. |
| *IsOverloaded* | Возвращает, может ли функция быть перегружена. |

*см. след. стр.*

**Табл. A-31. Свойства класса *CodeFunction* (окончание)**

| Свойство | Описание |
|---|---|
| *IsShared* | Задает или возвращает, определена ли данная функция как статическая. |
| *MustImplement* | Задает или возвращает, является ли функция абстрактной |
| *Overloads* | Возвращает набор *CodeElements*, содержащий перегрузки данной функции. |
| *Parameters* | Возвращает набор *CodeElements*, содержащий параметры функции |
| *Prototype*[1] | Возвращает объявление функции. |
| *Type* | Задает или возвращает объект *CodeTypeRef* представляющий тип функции. |

[1] C# не позволяет ссылаться на это свойство в операторе присваивания, поскольку его аксессор *get* имеет параметр. Используйте явный вызов аксессора *get*.

Свойство *Access* позволяет читать и записывать модификаторы доступа к функции (табл. A-14). Все языки реализуют возможность чтения, и все, кроме Visual Basic, — записи значения данного свойства.

Свойство *Attributes* возвращает набор *CodeElements*, содержащий по одному элементу типа *CodeAttribute* для каждого атрибута, примененного к функции.

Свойство *CanOverride* позволяет читать и записывать, является ли функция переопределяемой. Все языки реализуют возможность чтения этого свойства, но Visual C++ всегда возвращает *True*. C# и J# реализуют запись; однако J# позволяет только изменить непереопределяемую *(final)* функцию на переопределяемую.

Свойство *FunctionKind* возвращает одно из значений из перечисления *vsCMFunction* (табл. A-32). Все языки реализуют данное свойство.

**Табл. A-32. Перечисление *vsCMFunction***

| Константа | Описание |
|---|---|
| *vsCMFunctionOther* | Вид функции не описан в данной таблице |
| *vsCMFunctionConstructor* | Конструктор |
| *vsCMFunctionPropertyGet* | *get*-аксессор |
| *vsCMFunctionPropertyLet* | fef-аксессор |
| *vsCMFunctionPropertySet* | *set*-аксессор |
| *vsCMFunctionPutRef* | *put*-ссылка |
| *vsCMFunctionPropertyAssign* | Присваивание свойству |
| *vsCMFunctionSub* | Процедура *Sub* |
| *vsCMFunctionFunction* | Функция |
| *vsCMFunctionTopLevel* | Функция верхнего уровня |
| *vsCMFunctionDestructor* | Деструктор |
| *vsCMFunctionOperator* | Оператор |
| *vsCMFunctionVirtual* | Виртуальная функция |
| *vsCMFunctionPure* | Чисто виртуальная функция |
| *vsCMFunctionConstant* | Константа |
| *vsCMFunctionShared* | Общая функция |
| *vsCMFunctionInline* | Встраиваемая функция |

Свойство *IsOverloaded* возвращает, является ли данная функция перегружаемой. Когда значение этого свойства *True*, свойство *Overloads* возвращает набор, содержащий перегруженные функции для данной функции. Отметим, что свойство *Overloads* для C# и J# содержит только перегруженные функции, а это же свойство для Visual C++ и Visual Basic вместе с перегруженными функциями содержит и данную.

Свойство *IsShared* позволяет прочитать или записать, является ли функция-член общей функцией класса *(True)* или функцией экземпляра класса *(False)*. Возможность чтения этого свойства реализована во всех языках, записи — в Visual C++, C# и J#.

Свойство *MustImplement* позволяет прочитать или записать, является ли функция абстрактной. Все языки поддерживают чтение этого свойства. Visual C+- реализует возможность записи полностью, а C# и J# позволяют только преобразовать абстрактную функцию в реальную.

Свойство *Parameters* возвращает набор, содержащий по элементу для каждого параметра функции. Все языки поддерживают это свойство,

Свойство *Prototype* возвращает строку, представляющую прототип функции. Это свойство получает комбинацию констант из перечисления *vsCMPrototype* (табл. А-23) и возвращает запрошенные части объявления функции. Все языки поддерживают это свойство.

Свойство *Type* позволяет читать и записывать значение *CodeTypeRef*, представляющее тип результата, возвращаемого функцией. Все языки реализуют возможность чтения данного свойства, но лишь в Visual C++ — записи.

В табл. А-33 приведены методы класса *CodeFunction,* их описания см. в разделе «Генерация кода*.

**Табл. А-33. Методы класса *CodeFunction***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. А-7, |
| *AddAttribute* | Создает новый атрибут. |
| *AddParameter* | Создает новый параметр. |
| *RemoveParameter* | Удаляет параметр. |

### CodeParameter

Объект класса *CodeParameter* представляет параметр функции, процедуры или делегата. В табл. А-34 приведены свойства класса *CodeParameter,*

**Табл. А-34. Свойства *CodeParameter***

| Свойство | Описание |
|---|---|
| Свойства *CodeElement* | См. табл. А-6. |
| Прочие общие свойства | См. табл. А-10. |
| *Attributes* | Возвращает набор атрибутов параметра, |
| *Type* | Задает или возвращает *CodeTypeRef* представляющий тип параметра. |

Свойство *Attributes* возвращает набор *CodeElements*, содержащий по одному элементу на каждый атрибут, относящийся к данному параметру. Это свойство реализовано в C#, J#, и Visual Basic.

Свойство *Type* позволяет читать или записывать значение типа *CodeTypeRef*, представляющее тип параметра. Во всех языках реализована возможность чтения данного свойства, записи — только в Visual C++.

В табл. А-35 перечислены методы класса *CodeParameter*, пояснения к ним см. в разделе «Генерация кода*.

**Табл. А-35. Методы *CodeParameter***

| Метод | Описание |
|---|---|
| Методы *CodeElement* | См. табл. А-7. |
| *AddAttribute* | Создает новый атрибут. |

## CodeTypeRef

Объект *CodeTypeRef* представляет тип функции, делегата, свойства, переменной, или параметра. В табл. А-36 приведены свойства класса *CodeTypeRef*.

**Табл. А-36 Свойства класса *CodeTypeRef***

| Свойство | Описание |
|---|---|
| *AsFullName* | Возвращает полностью квалифицированное имя типа. |
| *AsString* | Возвращает ключевое слово языка для данного типа. |
| *CodeType* | Задает или возвращает *CodeType*, связанный с данным типом. |
| *DTE* | Возвращает объект *DTE*. |
| *ElementType* | Задает или возвращает тип элементов массива, |
| *Parent* | Возвращает родительский *CodeElement* |
| *Rank* | Задает или возвращает число измерений массива. |
| *TypeKind* | Возвращает базовый тип. |

Свойство *AsFullName* возвращает полностью квалифицированное имя для данного типа. Все языки поддерживают это свойство, но его реализация в Visual C++ возвращает то же значение, что и свойство *AsString*.

Свойство *AsString* возвращает ключевое слово языка, относящееся к данному типу (так, в C# свойство *AsString* вернет *int* для типа *System.Int32*.) Все языки поддерживают это свойство, но его реализация в Visual Basic возвращает то же значение, что и часть имени типа, возвращаемого свойством *AsFullName*.

Свойство *CodeType* возвращает объект *CodeType*, представляющий тип. C#, J# и Visual Basic поддерживают это свойство, хотя Visual Basic всегда возвращает значение *vsCMElementOther*.

Когда *CodeTypeRef* представляет массив, свойство *ElementType* задает или возвращает тип элементов массива, а свойство *Rank* задает или возвращает число его измерений. В настоящее время ни одно из этих свойств не реализовано в языках.

Когда *CodeTypeRef* представляет *CodeElement* из исходного файла, свойство *Parent* возвращает этот *CodeElement*. Все языки реализуют данное свойство.

Свойство *TypeKind* возвращает значение из перечисления *vsCMTypeRe*(табл. А-5), обозначающее тип этого *CodeTypeRef*. Все языки реализуют данное свойство.

# Генерация кода

В этом разделе мы исследуем активную сторону модели кода — генерацию кода. Как правило, обращения к модели кода, обследование кода — не самоцель: найдя какой-либо элемент исходного кода, вы совершаете лишь первый шаг по изменению этого элемента кода нужным вам образом. Концептуально генерация кода с помощью модели кода очень проста: вы находите элемент кода, который требуется изменить, а затем вызываете подходящий метод *Addxxx,* чтобы добавить к нему дочерний элемент кода, или метод *Removexxx,* чтобы удалить его. Однако дьявол кроется в деталях, и в каждом языке в реализации методов генерации кода есть свои особенности. Вся оставшаяся часть данного раздела представляет собой описание того, что и где работает или не работает.

---

Примечание В документации заявлено, что все языки модели кода распознают полностью квалифицированные имена типов, в которых применяется оператор «точка» (.); в реальности реализация Visual C++ понимает только имена типов, использующие оператор разрешения области видимости — «двойное двоеточие» (::). Обычно, если вы передаете имя тина, применяющее оператор «точка», в методы модели кода *Addxxx,* то Visual C++ добавляет это имя как оно есть, что приводит к синтаксическим ошибкам в исходном коде.

---

## Общие параметры

Большинство методов *Addxxx* имеет несколько общих параметров. Чтобы не повторять их каждый раз в следующих разделах, мы перечислим здесь наиболее часто используемые параметры и дадим для каждого общее описание. Ниже мы будем возвращаться к этим параметрам, только если будет появляться какая-либо дополнительная информация, специфическая для обсуждаемых методов.

- ***Name*** Строка, представляющая имя конструкции кода. Реализация Visual C++ проверяет допустимость применяемого имени, в остальных языках имя используется как есть, и вы сами должны обеспечить его правильность. Проверить корректность имени для некоторого языка позволяет метод *CodeModel.IsValidID.*
- ***Type*** Представляет новый тип конструкции кода. Значение этого параметра может быть или объектом типа *CodeTypeRef,* или значением из перечисления *vsCMTypeRef* а иногда — строкой, содержащей полностью квалифицированное имя типа. Помните, что некоторые реализации подставляют псевдоним вместо передаваемого вами типа. Так, метод *AddFunction,* даже если вы укажете тип возвращаемого функцией значения как *System.Int32* или *System.Boolean,* C# сгенерирует возвращаемый тип как *int* или *bool* соответственно.
- ***Position*** Определяет, где в исходном файле будет размещен новый элемент относительно элементов кода того же уровня. Значение 0 показывает, что новый элемент появится перед всеми остальными элементами этого уровня, а -1 — что новый элемент будет помещен после всех элементов кода этого уровня. Чтобы разместить новый элемент после определенного элемента кода того же. уровня, нужно или передать индекс в наборе *CodeElements* этого определен-

ного элемента, или передать объект *CodeElement*, его представляющий (в текущей версии Visual Studio .NET ни один из языков не поддерживает передачу *CodeElement* в параметре *Position)*.

- ***Access*** Управляет доступностью нового элемента кода. Он принимает значение из перечисления vsCMAccess (табл. A-14). Разные реализации не проверяют корректность передаваемого значения — эта обязанность лежит на вас.

#### Разновидности *CodeModel*

В справке по Visual Studio .NET методы *Addxxx* приводятся в двух вариантах: Variant1 и Variant2. Лучше было бы первый набор назвать «варианты *CodeModel*» — объект *CodeModel* не связан ни с каким определенным исходным файлом, поэтому методы *Addxxx* в этом варианте имеют дополнительный параметр *Location*, указывающий, какой файл получит генерируемый код.

Имя, передаваемое параметру *Location* не обязательно должно ссылаться на существующий файл; если файл не удается найти, он будет создан (если, конечно, это имя — допустимое имя файла). Самое важное, что параметр *Location* должен задавать корректный исходный файл в соответствии с используемым языком программирования (например, *myfile.cp*для Visual C++, *myfile.cs* для C# и т, д.). Б последующих разделах мы будем игнорировать варианты *CodeModel.*

## Основные *Add*-методы

Ниже описано поведение методов модели кода, создающих основные конструкции кода: пространства имен, классы, функции и т. п.

#### AddNamespace

Метод *AddNamespace* применяется к объектам *CodeModel*, *FileCodeModel* и *CodeNamespace* и имеет следующий прототип:

```
CodeNamespace AddNamespace(
    string Name,
    object Position
);
```

Вложенные пространства имен можно создавать, только используя объект *CodeNamespace*. Объекты *CodeModel* и *FileCodeModel* могут создавать только пространства имен верхнего уровня. Про этот метод больше сказать нечего — это самый простой из методов модели кода. Скажите спасибо.

#### *AddClass* и *AddStruct*

Методы *AddClass* и *AddStruct* применяются к объектам *CodeModel*, *FileCodeModel*, *CodeNamespace*, *CodeClass* и *CodeStruct* и имеют прототип:

```
CodeClass AddClass(
    string Name,
    object Position,
    object Bases,
    object ImplementedInterfaces,
    vsCMAccess Access
);
```

```
CodeStruct AddStruct(
    string Name,
    object Position,
    object Bases,
    object ImplementedInterfaces,
    vsCMAccess Access
);
```

Параметр *Bases* позволяет задать один или более базовых типов для нового класса или структуры путем передачи индексируемого с единицы массива объектов *CodeClass*, представляющих базовые типы (реализация Visual C++ также позволяет передать массив строк с именами типов). Конечно, CLR не поддерживает множественное наследование, и, если вы генерируете управляемый код, придется ограничиться одним базовым типом. В настоящее время только реализация Visual C++ поддерживает параметр *Bases,*

Параметр *ImplementedInterfaces* позволяет добавить один или несколько интерфейсов к списку наследуемых типов класса или структуры. Этот параметр принимает индексируемый с единицы массив объектов *CodeInterface*, представляющих реализуемые интерфейсы. Как и в случае с параметром *Bases*, реализация Visual C++ также принимает массив строк с именами типов. Параметр *ImplementedInterfaces* поддерживается только в Visual C++.

### AddInterface

Метод *AddInterface* применяется для объектов *CodeModel*, *FileCodeModel* и *CodeNamespace*:

```
CodeInterface AddInterface(
    string Name,
    object Position,
    object Bases,
    vsCMAccess Access
);
```

Параметр *Bases* добавляет один или более интерфейсов к новому списку интерфейсов. Он принимает индексируемый с единицы массив объектов *CodeInterface* или строк, представляющих реализуемые интерфейсы. Ни в одной из реализаций этот параметр не поддерживается, но тех же результатов в некоторых реализациях можно достичь, используя метод *CodeInterface.AddBase*.

### AddDelegate

Метод *AddDelegate* применяется для объектов *CodeModel*, *FileCodeModel*, *CodeNamespace*, *CodeClass* и *CodeStruct:*

```
CodeDelegate AddDelegate(
    string Name,
    object Type,
    object Position,
    vsCMAccess Access
);
```

В настоящее время этот метод поддерживается только в C#.

### AddFunction

Метод *AddFunction* применяется для объектов *CodeModel, FileCodeModel, CodeClass, CodeStruct* и *CodeInterface;*

```
CodeFunction AddFunction(
    string Name,
    vsCMFunction Kind,
    object Type,
    object Position,
    vsCHAccess Access,
);
```

Параметр *Name* задает имя функции, кроме случаев конструкторов и деструкторов (принимающих имя класса, к которому они принадлежат).

Параметр *Kind* определяет тип создаваемой функции и принимает одну или более констант из перечисления *vsCMFunction* (табл. А-32). Однако не любая комбинация этих констант является осмысленной, и некоторые реализации, если попросить, не задумываясь создают конструктор-получатель (getter constructor), так что проверка корректности передаваемой комбинации лежит на вас.

### Add Property

Метод *AddProperty* применяется к объектам *CodeClass, CodeInterface* и *CodeStruct:*

```
CodeProperty AddProperty(
    string GetterName,
    string PutterName,
    object Type,
    object Position,
    vsCMAccess Access,
    object Location
);
```

В настоящее время этот метод работает только в C#. Параметры *GetterName* и *PutterName* представляют имена получателя и установщика свойства соответственно. В C# для свойства, доступного и для чтения, и для записи, значения *GetterName* и *PutterName* должны совпадать. Чтобы создать свойство только для чтения, параметру *PutterName* нужно указать значение *null;* чтобы создать свойство только для записи, параметру *GetterName* надо присвоить *null.*

Параметр *Location* позволяет указать новый файл, в который будет помещено создаваемое свойство; он имеет ту же семантику, что и параметр *Location* варианта *CodeModel* (см. раздел «Варианты *CodeModel*»). Реализация C# игнорирует этот параметр.

### AddEnum

Метод *AddEnum* применяется к объектам *CodeModel, FileCodeModel, CodeNamespace, CodeClass* и *CodeStruct:*

```
CodeEnum AddEnum(
    string Name,
    object Position,
    object Bases,
```

```
    vsCMAccess Access
);
```

Параметр *Bases* принимает индексируемый с единицы массив объектов *CodeType* или строк, представляющих тип перечисления. Ни в одной из реализаций параметр *Bases* не используется.

### AddVariable

Метод *AddVariable* применяется к объектам *CodeModel*, *FileCodeModel*, *CodeClass* и *CodeStruct*:

```
CodeVariable AddVariable(
    string Name,
    object Type,
    object Position,
    vsCHAccess Access
);
```

Метод *AddVariable* работает во всех реализациях, но так как C# и J# не допускают объявления переменных верхнего уровня, то методы *CodeModel.AddVariable* и *FileCodeModel.AddVariable* для этих языков вызовут возникновение исключительной ситуации.

### AddAttribute

Метод *AddAttribute* применяется к объектам *CodeModel*, *FileCodeModel*, *CodeClass*, *CodeDelegate*, *CodeEnum*, *CodeFunction*, *CodeInterface*, *CodeParameter*, *CodeProperty*, *CodeStruct*. *CodeVariable* и *CodeType* (мы ничего не упустили?):

```
CodeAttribute AddAttribute(
    string Name,
    string Value,
    object Position
);
```

В настоящее время C# и J# не реализуют метод *AddAttribute*. В Visual C++ объекты *CodeDelegate* и *CodeProperty* доступны только для чтения, и к ним нельзя добавить атрибут. Среди объектов *Codexxx*, поддерживаемых Visual C++, доступных для записи, только объект *CodeParameter* не поддерживает метод *AddAttribute*.

## Прочие *Add*-методы

В этом разделе описаны оставшиеся *Add*-методы модели кода, позволяющие расширить базовые конструкции, созданные *Add*-методами предыдущего раздела.

### AddBase

Метод *AddBase* применяется для объектов *CodeClass*, *CodeStruct*, *CodeDelegate*, *CodeInterface*, *CodeEnum* и *CodeType*:

```
CodeElement AddBase(
    object Base,
    object Position
);
```

Параметр *Base* представляет новый базовый тип для данного объекта; в случае с C#-объектом *CodeEnum* параметр *Base* представляет тип перечисления (в настоящее время реализация метода *CodeEnumAddBase* в C# не работает).

Каждый из языков реализует метод *AddBase* по-своему; в табл. А-37 сведены воедино комбинации языка, объекта и базового типа (поскольку добавление нового базового типа к делегату не имеет смысла, не приводится реализация *CodeDelegateAddBase* ни для одного из языков).

**Табл. А-37. Правильные типы для параметра *Base* метода *AddBase***

| | Visual C++ | Visual C# | Visual J# |
|---|---|---|---|
| ***CodeClass.AddBase*** | | | |
| *CodeClass* | Нет | Да | Да |
| *CodeInterface* | Нет | Нет | Нет |
| *CodeTypeRef* | Нет | Да | Да |
| *CodeType* | Нет | Да | Да |
| Полностью квалифицированное имя | Да | Да | Да |
| ***CodeStruct.AddBase*** | | | |
| *CodeClass* | Нет | Нет | Неприменим |
| *CodeInterface* | Нет | Да | Неприменим |
| *CodeTypeRef* | Нет | Нет | Неприменим |
| *CodeType* | Нет | Нет | Неприменим |
| Полностью квалифицированное имя | Да | Да | Неприменим |
| ***CodeInterface.AddBase*** | | | |
| *CodeClass* | Нет | Нет | Неприменим |
| *CodeInterface* | Нет | Да | Неприменим |
| *CodeTypeRef* | Нет | Да | Неприменим |
| *CodeType* | Нет | Да | Неприменим |
| Полностью квалифицированное имя | Да | Да | Неприменим |
| ***CodeEnum.AddBase*** | | | |
| *CodeClass* | Неприменим | Неприменим | Неприменим |
| *CodeInterface* | Неприменим | Неприменим | Неприменим |
| *CodeTypeRef* | Неприменим | Неприменим | Неприменим |
| *CodeType* | Неприменим | Неприменим | Неприменим |
| Полностью квалифицированное имя | Неприменим | Неприменим | Неприменим |

### AddImplementedInterface

Метод *AddImplementedInterface* применяется для объектов *CodeClass* и *CodeStruct:*

```
CodeInterface AddImplementedInterface(
    object Base,
    object Position
);
```

Параметр *Base* принимает или объект *CodeInterface или* строку с полностью квалифицированным именем, представляющие интерфейс, который будет реализован в классе или структуре. Этот метод поддерживается только в J#.

### AddMember

Метод *AddMember* применяется к объектам *CodeEnum*:

```
CodeVariable AddMember(
    string Name,
    object Value,
    object Position
);
```

Этот метод используется для добавления нового члена к перечислению. Параметр *Name* определяет имя нового члена, а параметр *Value* задает его начальное значение. Числовое значение параметру *Value* можно передавать либо как строку, либо как объект, а метод подставит подходящий оператор инициализации; па вас лежит обязанность проверки правильности передаваемого значения.

### AddParameter

Метод *AddParameter* применяется к объектам *CodeDelegate* и *CodeFunction*:

```
CodeParameter AddParameter(
    string Name,
    object Type,
    object Position
);
```

Метод *AddParameter* не позволяет создавать необязательные параметры.

### CreateArrayType

Метод *CreateArrayType* применяется к объекту *CodeTypeRef*.

```
CodeTypeRef CreateArrayType(
    int Rank
);
```

Этот метод возвращает объект *CodeTypeRef*, который можно использовать для создания массива. Параметр *Rank* задает число измерений массива, описываемого новым объектом *CodeTypeRef*. Ниже приведен пример использования этого метода для создания нового массива (предполагается, что в переменной *cdeModel* хранится ссылка на *CodeModel*, а в переменной *cdeClass* — на *CodeClass*):

```
Dim cdeVariable As CodeVariable
Dim cdeTypeRef As CodeTypeRef
Dim cdeTypeRefArray As CodeTypeRef

' Создать ссылку на целый тип.
cdeTypeRef = cdeModel.CreateCodeTypeRef(vsCMTypeRef.vsCMTypeRefInt)

' Превратить эту ссылку в массив целых чисел.
cdeTypeRefArray = cdeTypeRef.CreateArrayType()

' Создать внутри класса новую переменную: массив целых чисел.
cdeVariable = cdeClass.AddVariable("myIntArray", cdeTypeRefArray)
```

В настоящее время этот метод не поддерживается ни в одном из языков.

## Методы для удаления

В данном разделе описаны методы модели кода, позволяющие вам очистить исходный код от ненужных конструкций кода.

### Delete

Метод *Delete* применяется к объекту *CodeAttribute*:

```
void Delete();
```

Ни один из языков не поддерживает этот метод.

### Remove

Метод *Remove* применяется к объектам *CodeModel*, *FileCodeModel* и *CodeNamespace*:

```
void Remove(
    object Element
);
```

Параметр *Element* может быть объектом класса *CodeElement* или *CodeType*. Реализации C# и J# удаляют соответствующий исходный код из файла; реализация Visual C++ просто помещает этот код внутрь комментариев (закомментирует) — исключением является объект *CodeParameter*, который Visual C++ удаляет.

В C# и J# удаляется только идентификатор *CodeVariable*; может, не очень удобно, когда все, что остается от вашего объявления, — это *object;*.

### RemoveBase

Метод *RemoveBase* применяется к объектам *CodeClass*, *CodeDelegate*, *CodeEnum*, *CodeInterface*, *CodeStruct* и *CodeType:*

```
void RemoveBase(
    object Element
);
```

Метод *RemoveBase* является обратным к методу *AddBase*. Параметр *Element* принимает объект *CodeElement*, представляющий удаляемую базу (реализация C# также принимает в качестве параметра объект *CodeType)*. Заметим: *RemoveBase* не позволит вам удалить *System.Object*, если он является единственным базовым классом.

Для перечислений в C# *CodeEnum.RemoveBase* работает, но просто. Как и в случае метода *AddBase*, метод C# *CodeEnum RemoveBase* принимает индексируемый с единицы массив объектов *CodeType* или *CodeTypeRef* представляющих типы перечисления.

### RemoveInterface

Метод *RemoveInterface* применяется к объектам *CodeClass* и *CodeStruct:*

```
void RemoveInterface(
    object Element
);
```

Параметр *Element* принимает объект *CodeInterface*, представляющий удаляемый из набора *ImplementedInterfaces* интерфейс. Только J# поддерживает этот метод,

### RemoveMember

Метод *RemoveMember* применяется к объектам *CodeClass*, *CodeDelegate*, *CodeEnum*, *CadeInterface*, *CodeStruct* и *CodeType*:

```
void RemoveMember(
    object  Element
);
```

Параметр *Element* принимает объект *CodeElement*, представляющий дочернюю конструкцию кода, которую нужно удалить. Метод *RemoveMember* не имеет смысла для объекта *CodeDelegate*, поскольку делегат не имеет элементов в своем наборе *Members*,

### Remove Para meter

Метод *RemoveParameter* применяется к объектам *CodeDelegate* и *CodeFunction*:

```
void RemoveParameter(
    object  Element
);
```

Параметр *Element* принимает объект *CodeParameter*, представляющий параметр, который нужно удалить из данного делегата или функции. Этот метод реализован только в C# и J#.

# Предметный указатель


.NET Framework 3
— архитектура 3
— библиотека классов 3, 4
— нестандартные действия 349
.NET Framework SDK 17

A

Active Template Library *СМ.* ATL
ActiveX 3, 249, 270, 279
Add New Item wizard *см.* мастер, добавления нового элемента
add-in *см.* надстройка
Add-in Manager 125, 128
Add-in Wizard *см.* мастер, надстроек
API 4, 8, 19
ASP.NET 4, 32
ASP.NET Web Service Wizard *см.* мастер, Web-сервисов ASP.NET
ATL (Active Template Library) 3, 7, 101
— сервер 7

B

build *см.* компоновка

C

C 2, 4
C# 187, 283
C# Class Library Wizard *см.* мастер, библиотеки классов C#
C++ 2, 4, 7, 10
CLR (common language runtime) 3, 4, 8, 284
CLS (Common Language Specification) 4
CME (Code Model Explorer) 319
COBOL 4
Code Editor *см.* код, редактор
Code Model Explorer *СМ.* CME
code-behind *см.* код, фоновый
COM 3
Command Mode *см.* окно, командное, командный режим
common language runtime *см.* CLR
Common Language Specification *СМ.* CLS
common type system *СМ.* CTS
Common User Accessibility *СМ.* CUA
CRT 7
CTS (common type system) 4
CUA (Common User Accessibility) 71
CUA (Common User Accessibility) 51
Custom wizard *см.* мастер, нестандартный

D

Design Time Extensibility *СМ.* DTE
Development Tools Extensibility *СМ.* DTE
DHTML 7
DirectX 3
DLL 4, 36, 100, 109, 130
— сателлитная 142, 145
DTE (Design Time Extensibility) 132
DTE (Development Tools Extensibility) 19

F

FUID (for-the-most-part unique identifier) 125

G

GAC (global assembly cache) 190, 250
GUID 125, 148, 191, 216, 240, 268, 388

I

ID 148, 160
IDE 2, 5, 7, 8, 12, 16, 17, 35, 48, 49, 53, 70, 78, 81, 82, 84, 91, 92, 100, 101, 125, 138, 147, 212, 284, 365

IDL (Interface Definition Language) 7
Immediate Mode *см.* окно, командное, режим непосредственного выполнения
IntelliSense *см.* код, автозавершение
Interface Definition Language *см.* IDL
ISAPI 7

J

Java 3

L

LIBID 191

M

Macros IDE 10, 12, 78, 79, 82, 84, 91, 101, 109, 125
Margin Indicator Bar *см.* код, редактор, полоса индикаторов
MDI *см.* интерфейс, многооконный
merge module *см.* модуль, слияния
MFC (Microsoft Foundation Classes) 2, 3, 7
Microsoft CLR Debugger 110
Microsoft Foundation Classes *CM.* MFC
Microsoft Interface Definition Language *CM.* MIDL
Microsoft Intermediate Language *CM.* MSIL
Microsoft Visual C++ 6.0 78
Microsoft Windows Installer *CM.* MSI
MIDL (Microsoft Interface Definition Language) 38
MSDN Library 365
MSI (Microsoft Windows Installer) 328
MSIL (Microsoft Intermediate Language, IL) 4, 8, 32

N

named command *см.* именованная команда
New Project wizard *см.* мастер, нового проекта

O

OptionsPageWizard *см.* мастер. страниц Options
outlining *см.* код, структуризация
output type *см.* проект, тип результата

P

PE-файл (portable executable) 4
project *см.* проект
Property Page *см.* страница свойств

R

references node *см.* проект, узел ссылок

S

SDK 147, 153
shim control *см.* согласователь
solution *см.* решение
splitter bar *см,* разделитель, полоса
STL (Standard Template Library) 7

T

Text Editor *см.* редактор, текста
TlbImp (Type Library Importer) 132

U

UI-иерархия *см.* окно, иерархии пользовательского интерфейса

V

VBScript (Visual Basic Scripting Edition) 78
Visual Basic 3, 4, 12, 283
Visual Basic .NET 4, 23, 78
Visual Basic 6 10
Visual Basic Scripting Edition *CM.* VBScript
Visual C .NET 7
Visual C# 4, 5, 8, 23, 87
Visual C++ 2, 3, 4, 5, 12, 87, 279
Visual C++ .NET 5, 7, 23
Visual C++ 6 10
Visual InterDev 3, 10
Visual J# 23
Visual J# .NET 4
Visual J++ 3

Visual SourceSafe *см.* VSS
Visual Studio 3
Visual Studio .NET 2, 147
— IDE 8
— библиотека классов 7
— быстрые клавиши 52, 54, 71
— вид при запуске 12
— дизайнер 12
— — Web Forms 12
— - Windows Forms 12
— - XML 12
— — компонентов 12
— клавишы быстрого доступа 10
— мастер 214 *см. также* мастер
— модель кода 310
— отладчик 6
— параметры окон 9
— переменные окружения 18
— профиль 9
— размещение файлов 16
— редактор 12
— системные переменные 18
— справка 65, 365
— — динамическая 68
— — фильтр 11, 369
— стандартные изображения 160
— стартовая страница 9
— языки программирования 4
Visual Studio 6 3
VSIP (Visual Studio Integration Program) 9, 19, 21, 40, 390
VSS (Visual SourceSafe) 390

W

Web 3
Web Forms 12, 13, 14, 23
Web Services Description Language *CM.* WSDL
Web-сервис 229
Web-ссылка 191
wildcards *см.* групповые символы
Win32 3, 147, 153, 351
Windows Forms 12, 13, 14, 23, 229
Windows-приложение 4, 23
WSDL (Web Services Description Language) 191
WYSIWYG (What You See Is What You Get) 12

X

XML 2, 7, 229
— определение схемы 12
XSD (XML Schema Definition) 12

А

атрибут 7

Б

буфер обмена кольцевой 53
быстрые клавиши 51

Г

глобально уникальный идентификатор *см.* GUID
глобальный кэш сборок *см.* GAC
групповые символы 72, 73

Д

делегат 114
дизайнер 8, 48
— форм 264

Ж

журнал событий 14

И

идентификатор библиотеки типов *см.* LIBID
идентификатор ресурса *см.* ID
именованная команда 15, 48. 69, 70, 71
именованное значение
— *AboutBoxDetails* 144
— *AboutBoxIcon* 144
— *CommandLineSafe* 140, 141
— *CommandPreload* \ 38
— ***Description*** 144
— *FriendlyName* 144
— *LoadBehavior* 140
— *PreloadAddinState* 138
— *SatelliteDLLName* 142
- *SatelliteDLLPath* 142
интегрированная среда разработки *см.* IDE
интерфейс
— *BuildEvents* 114

— *CodeAttribute* 314
— *CodeClass* 314
— *CodeDelegate* 314
— *CodeEnum* 314
— *CodeFunction* 314
— *CodeInterface* 314
- *CodeModelEvents* 119
— *CodeNamespace* 314
— *CodeParameter* 314
— *CodeProperty* 314
— *CodeStruct* 314
— *CodeType* 316
— *CodeVariable* 314
— *CSharpBuildManagerEvents* 119
— *CSharpProjectItemsEvents* 119
— *CSharpProjectsEvents* 119
— *CSharpReferencesEvents* 119
— *DTEEvents* 117
— *EnvDTE.DTE* 131
- *EnvDTE.Events* 112
- *EnvDTE.IDTWizard* 215
- *EnvDTE.Project* 189
~ *ExtensibilityIDTExtensibility2* 121
— *IDesignerHost* *264*
— *IDispatch* 280
— *IDTCommandTarget* 102, 107, 153, 154, 155
- *IDTExtensibility2* 20, 102, 121, 122, 126, 127, 130, 153, 183
- *IDTToolsOptionsPage* 280
— *IDTWizard* 215, 227, 231, 233
— *InsideVSNet.WizardLibrary.IWizardPage* 232
— *IOleCommandTarget* 153
— *ITDWizard* 20
- *IWizardPage* 236
— *ProjectItems* 187
— *VBBuildManagerEvents* 119
— *VBImportsEvents* 119
— *VBProjectItemsEvents* 1\9
— *VBProjectsEvents* 119
— *VBReferencesEvents* 119
— *VCProjectEngineEventsObject* 119
- *VSLangProj.VSProject* 189
— *VSUserControlHostLib.IVSUserControlHostCtl* 270
— *Win32Window* 242
— многооконный 51
— пользовательский 13, 238
исключение
— *COMException* 133
— *System.Runtime.Remoting.RemotingException* 264
источник данных 14

## K

класс
— *Basic* **121**
— *Connect* 102
— *CString* 7
- *EnvDTE.Constants* 240
- *EnvDTE.TaskListEvents* 258
— *GuidAttribute* 125
— *InsideVSNet.WizardLibrary.WizardLibrary* 231
— *Installer* 166
— *System.Windows.Forms.Form* 264
- *WinWrapper* 243
— *WizardLibrary* 233, 236
— обозреватель 51
— решения 13
ключевое слово
— *For Each* 148
— *foreach* 148
— *is* 283
— *TypeOf...Is* 283
код
— автозавершение 66
— генерация 322
— изменение цвета 59
— модель 310
— неуправляемый 5, 7, 8
— нумерация строк 48, 62
— оптимизация 5
— редактор 48, 49
— — индикатор структуры 50
— — полоса выделения 50
— — полоса индикаторов 50
— — полоса навигации 50
— структуризация 48, 64
— точка прерывания 50
— управляемый 8
— фоновый 32
— форматирование 48, 56
команда 147
- GUID 149
- ID 149
— восстановление 161
— выполнение 150
— главная панель 157
— для надстроек 151
— добавление панели 158

— запуска макросов 151
— имя 149, 155
— комбинация клавиш 169
— маршрутизация 147
— обработка 153
— — вызова 153
— обработчик 155
— объектная модель панели 156
— параметры 168
— поиск 148
— пользовательский интерфейс 156
— состояние 154, 155
— удаление 165
компилятор 5
компоновка 34, 35
— — событие 39
константа
— *EnvDTE.Constants.vsProjectKind-Unmodeled* 198
— *EnvDTE.Constants.vsWizard-AddItem* 216
— *EnvDTE.Constants.vsWizardNew-Project* 216
— *ext_cm_AfterStartup* 132
— *ext_cm_CommandLine* 132
— *ext_cm_External* 132
— *ext_cm_Solution* 132
— *ext_cm_Startup* 132
— *ext_cm_UISetup* 132
— *ext_dm_HostShutdown* 138
— *ext_dm_SolutionClosed* 138
— *ext_dm_UISetupComplete* 138
— *ext_dm_UserClosed* 138
— *prjKindCSharpProject* 220
— *prjKindVBProject* 220
— *vsInsertFlagsCollapseToEnd* 87
— *vsInsertFlagsCollapseToStart* 87
— *vsInsertFlagsContainNewText* 87
— *vsInsertFlagsInsertAtEnd* 87
— *vsInsertFlagsInsertAtStart* 87
— *vsMiscFilesProjectUniqueName* 196
— *vsSolutionItemsProjectUnique-Name* 197
— *vsTaskIcon.vsTaskIcon-Comment* 254
— *vsTaskIcon.vsTaskIconCompile* 254
— *vsTaskIcon.vsTaskIconShortcut* 254
— *vsTaskIcon.vsTaskIconSquiggle* 254
— *vsTaskIcon.vsTaskIconUser* 254
— *vsWindowKindCommand-Window* 241
— *vsWindowKindMacroExplorer* 241
— *vsWindowKindOutput* 241
— *vsWindowKindServerExplorer* 241
— *vsWindowKindSolution-Explorer* 241
— *vsWindowKindTaskList* 241
— *vsWindowKindToolbox* 241, 247
— *vsWindowKindWebBrowser* 241
контекст отмены 298
куст 125, 140

## M

макрос 16, 19, 39, 70, 78
— запись 79
— запускающая кнопка 85
— команда 81
— — *Tools.Edit* 82
— — *Tools.LoadMacroProject* 81
— — *Tools.MacrosIDE* 81
— — *Tools.Newmacro* 82
— — *Tools.NewMacroProject* 81
— — *Tools.Newmodule* 82
— модуль 79
— обработчик события 116
— обращение 85
— преобразование в надстройку 95
— проект 81
— псевдоним 85
— редактирование 82
— с параметрами 169
— событие 91
— совместное использование 94
— экспорт
— — модулей 94
— — проектов 94
маркер комментариев 259
маршалинг 264
мастер 20, 214
— Web-сервисов ASP.NET 230
— библиотек классов C# 230
— библиотека 235
— — вспомогательная 231
— библиотеки классов C# 224
— вспомогательный метод 235
— добавления нового элемента 214, 216, 227
— запуск 219, 228

-надстроек 100, 101, 121, 158, 183, 225, 334 *см. также* надстройка
— нестандартный 214, 227
— нового проекта 214, 216, 223
— объект 215
— переменные 234
— расшифровка параметров 230
— связывание 229
— страниц Options 279
— шаблон 221, 227
метаданные 32
метод
— *Activate* 112, 201, 248, 264
- *Add* 112, 190, 255, 262
— *AddClass* 322
— *AddExistingItem* 187
— *AddFromDirectory* 187, 223
— *AddFromFile* 187, 222, 223
— *AddFromFileCopy* 187, 223
— *AddFromTemplate* 187, 222, 223
— *AddIns.Item* 133
— *AddNamedCommand* 152, 154, 155
— *AddWebReference* 192
- *Close* 242
— *CodeElementFromPoint* 321
— *Command.AddControl* 158
— *Command.Delete* 165
— *CommandBar.ShowPopup* 160
— *Commands.AddCommandBar* 159
— *Commands.AddNamed-Command* 160
— *ConfigurationManager.AddConfigurationRow* 208
— *ConfigurationManager.ConfigurationRow* 207
— *ConfigurationManagePlatform* 207
— *ConnectSolutionEvents* 182
— *Count* 112
— *Create* 175
- *CreateEditPoint* 292, 296
— *CreateMediaPlayerToolWindow* 269
— *CreateProject* 235
- *CreateToolWindow* 269, 270, 273
— *Delete* 112
— *DeleteBetweenTokens* 235
— *DllRegisterServer* 334
— *DllUnregisterServer* 334
— *Document* 242
— *Document.Close* 292
— *Document.Redo* 291
— *Document.Save* 291
— *Document.Undo* 291
— *DoDefaultAction* 246
— *DTE.ExecuteCommand* 151, 155, 168
— *DTE.LaunchWizard* 228
— *EqualTo* 293
— *Events.GetObject* 200
— *Exec* 107, 153
— *Execute* 215, 218, 228, 233, 236
— *ExecuteCommand* 151, 168, 169
— *FileCodeModel.AddNamespace* 323
— *FindProjectItem* 186
— *Form.ShowDialog* 242
— *GreaterThan* 293
— *HostUserControl* 271
— *HostUserControl2* 271
— *IDesignerHost.CreateComponent* 265
— *IDTCommandTarget.Exec* 168 169
— *IDTCommandTarget.QueryStatus* 152
— *IDTToolsOptionsPage.GetProperties* 280
— *IDTToolsOptionsPage.OnOK* 280
— *IDTWizard.Execute* 216, 227, 233, 234
— *Initialize* 233
— *Insert* 87
— *Install* 166
— *Item* 112, 118, 148, 205, 240
— *ItemOperations.NewFile* 197, 289
— *IWizardPage.PerformWork1* 235
— *IWizardPage.PerformWork2* 235
— *LessThan* 293
— *MakeReplacements* 235
— *MsgBox* 88
— *MsgWin* 89
— *My Macros.RecordingModule.TemporaryMacro* 79
— *Navigate* 257
— *NewFile* 289
— *NewTextFile* 85
— *Object* 242
— *Object.Equals* 276, 277
— *OleLoadPictureFile* 255, 273
— *OnAddinsUpdate* 107

- *OnAddInsUpdate* 102, 121
- *OnAfterCreated* 280
- *OnBeginShutdown* 102, 107, 121
— *OnConnection* 102, 107, 121, 122, 131, 153, 183, 269
- *OnDisconnection* 102, 107, 121, 122, 183
- *OnStartupComplete* 102, 107, 121, 269
— *OutputWindowPane.OutputString* 263
— *PerformWork1* 232, 236
— *PerformWork2* 232, 236
— *Project* 242
- *ProjectItem* 242
— *Properties.Item* 275
— *PropertyDescriptor.GetValue* 266
— *QueryStatus* 107, 154, 155
— *References.Add* 190
— *Reference.AddProject* 191
— *Remove* 112
- *SatelliteDllPath* 143
- *Select* 246
- *SelectDown* 246
— *Selection* 242
- *SelectUp* 246
— *SetSelectionContainer* 273
— *ShowHelp* 233
— *Solution.Create* 224
— *SolutionItem* 176
— *Solution.Open* 175
— *Solution.ProjectItemsTemplatePath* 221
— *Solution.SaveAs* 175
— *SolutionBuild.Build* 201
— *System.Console.Write* 91
— *System.Console.WriteLine* 91
— *System.Reflection.Assembly.Load* 250
- *TaskItem.Delete* 256
- *TaskItem.Add* 255
- *TaskItems.ForceItemsToTaskList* 256
- *TextSelection* 87
- *ToolBoxItems.Add* 249
- *ToolBoxTab.Add* 248
- *ToolBoxTabs.Add* 247
- *Type.GetType* 265
- *UIHierarchy.GetItem* 244, 245
— *UIHierarchy.SelectDown* 246
— *UIHierarchy.SelectUp* 246
— *Uninstall* 166
- *Window.SetSelectionContainer* 273
- *Window.SetTabPicture* 273
- *Windows.CreateToolWindow* 267
- *Windows.Item* 239, 244, 247
- *WizardLibrary.AddPage* 233
модель событий 91
модуль
— подгружаемый 357
— слияния 381

## Н

надстройка 19, 78 *см. также* мастер надстроек
— администратора 126
— архитектура 121
— диспетчер 184
— для решений 182
— жизненный цикл 128
— загрузка 107
— именованные значения 138
— команда 151, 155
— компиляция 122
— обратный вызов 153
— отладка 109
— пользователя 126
— предварительная загрузка 138
— проект 102
— производительность 165
— регистрация 125, 126
— — как COM-объекта 123
— событие 127
— создание 121
— установка 107

## О

обработчик события 92, 114, 178, 308
общая система типов *см.* CTS
общеязыковая исполняющая среда *см.* CLR
общеязыковая спецификация *см.* CLS
объект
— *AddIn* 107, 112, 133
— *AddInInst* 268
— *BuildDependencies* 205
— *BuildDependency* 206
- *CMNamespace* 314
— *CodeElement* 311, 321

— CodeFunction 111
— CodeInterface 111
— CodeNamespace 111, 323
— CodeVariable 111
— COM 101, 123, 191
— Command 111, 158
— CommandBarControl 156
— CommandWindow 88, 111
— CommandWindowPaneEx 90
— ConfigurationManager 200, 207
— Debugger 111
— Document 86, 111, 289, 292
— DTE 111, 131, 174, 241
— DTE.AddIn 131
— DTE.AddIns 131
— DTE.Commands 152
— DTE.Events 112
— DTE.SolutionAddIns 131
— DTE.UndoContext 300
— EditPoint 111, 295, 297
— EnvDTE.BuildDependency 205
— EnvDTE.Project 189, 205
— EnvDTE.Properties 209
— EnvDTE.Solution 174
— EnvDTE.TaskItem 251
— EnvDTE.TaskList 251
— EnvDTE.ToolBoxItem 251
— EnvDTE.UIHierarchyItems 246
— EnvDTE.Window 239
— FileCodeModel 111, 321
— Globals 212
— HTMLWindow 283, 284
— IPicture 255
— IpictureDisp 255
— IPictureDisp 254, 255, 273
— ItemOperations 175, 187, 289
— MenuBar 158
— Microsoft.Office.Core.Command-Bar 156
— Microsoft.Office.Core.Command-Bars 156
— OutputWindow 88, 111, 262
— OutputWindowPane 91, 264
— OutputWindowPaneEx 90, 91
— OutputWindowPanes 262
— Process 111
— Program 111
— Project 111, 189
— ProjectItem 111, 184, 186
— ProjectItems 184, 186, 289
— Properties 274
— Property 274, 280
— PropertyDescriptorCollection 266
— Selection 273
— Solution 111, 175, 176, 186, 219
— SolutionBuild 200
— SolutionConfiguration 200, 201
— SolutionContext 200
— SolutionEvents 177, 182
— StackFrame 111
— System.ComponentModel.PropertyDescriptorCollection 265
— System.Reflection.Assembly 250
— System.String 150
— System.Type 265
— System.Windows.Control 265
— TaskItem 256
— TaskList 88, 111, 241
— TaskListEvents 258
— TextDocument 86, 111
— TextPane 285
— TextPoint 86, 111, 293
— TextSelection 86, 87, 111, 291, 296, 297
— TextWindow 88, 283, 284
— Thread 111
— ToolBox 88, 111, 247
— ToolBoxItem 248
— ToolBoxTab 112, 248
— Type 265
— UIHierarchy 243, 244, 246
— UIHierarchyItem 243, 244, 246
— UIHierarchyItems 243
— VirtualPoint 293, 294
— VSLangProjectReferences 190
— VSProject 189, 190, 192
— Window 88, 118, 238, 240, 241, 242, 247, 268, 276, 283
— мастера 215
объектная модель автоматизации 111
окно
— браузера объектов 49
— дизайнера 51
— дизайнера форм 247
— документа 8, 290
— иерархии пользовательского интерфейса 243
— инструментов 8, 12. 49, 267
— интерфейс 238
— командное 52, 69
— — командный режим 69

— — очистка 70
— — режим непосредственного выполнения 69
— обозревателя
— — классов 51
— — решений 51
— проводник 243
— редактор кода 51
— редактора 282
— управление 88
оператор
— *Imports* 86, 193
*-Is* 276, 277
— *using* 193
ошибка периода выполнения 6

П

пакет языков 8
переключатель
— */case* 76
— */doc* 76
- */GL* 5
— */GS* 6
— */hidden* 76
— */markall* 76
— */O* 6
— */open* 76
— */options* 76
— */proc* 76
— */regex* 76
- */RTCc* 6
— */RTCn* 6
- */RTCs* 6
— */sel* 76
-*/up* 76
—*/wild* 76
— */word* 76
переменная
— *addInsCollection* 136
— *addInsList* 136
— *exclusiveProject* 234
— *NewItemLocation* 235
— *NewItemName* 235
— *NewProjectLocation* 234
— *NewProjectName* 234
— *NewSolutionName* 234
— *objlPictureDisp* 255
— *ProductInstallDirectory* 235
— *ProjectItems* 235
— *ProjectName* 235
— *RunSilent* 234, 235
— *serviceName* 230
— *taskListEvents* 258
— *VisualStudioInstallDirectory* 234
- *WizardType* 234, 235
переполнение буфера 6
перечисление
- *EnvDTE.vsHTMLTabs* 284
— *EnvDTE.vsTaskPriority* 260
— *ext_ConnectMode* 122
— *ext_DisconnectMode* 122, 137
— *Extensibility.ext_ConnectMode* 132
— *vsCommandBarType* 159
— *vsTextChanged* 304
поиск текста 71
— быстрые клавиши 71
— групповые символы 72
— из командного окна 75
— именованная команда 71
— инкрементный 76
— регулярные выражения 72, 73
приложение управляемое 33
программа интеграции Visual Studio *см.* VSIP
проект 22, 23, 31, 48, 81
— Deployment 40
— Visual C# 40
- Visual C++ 40
— Visual J# 40
— COM-взаимодействие 130
-Visual Basic .NET 40
- Visual C++ 36
— библиотеки классов 35
— внемодельный 197
— вспомогательный тип 195
— выходные данные 331
— добавление ссылки 45
— зависимость 45, 203
— исходный файл 40
— компиляция 200
— компоновка 46, 200, 384
— конфигурация 208, 209
— надстройки 102
— объект 189
— параметры 32, 207
— перебор элементов 176. 184
— пользовательская конфигурация 33
— порядок сборки 46
— свойства
— — конфигурации 34
—— общие 33

— система управления 174
— событие 198
— сохранение информации 212
— список 45
— ссылка 190
— стартовый 202
— тип 40, 101
— — результата 130
— узел ссылок 189
- файл 31
— — параметров пользователя 43
— шаблон 222
— элемент
— — добавление 187
— — удаление 187
пространство имен
— *DTE* 86
— *DTE.Windows* 88
— *EnvDTE* 106, 130
— *Extensibility* 106, 122, 130
— *Microsoft.Office.Core* 107
— *System.Runtime.InteropServices* 162
— *System.XML* 194
— *Utilities* 130
псевдоним 70, 85

P

раздел реестра
- HKEY_CLASSES_ROOT 334
- HKEY_CLASSES_ROOT\<ProgID> 123
- HKEY_CLASSES_ROOT\<ProgID>\-CLSID 123
- HKEY_CLASSES_ROOT\CLSID\-<CLSID> 123
- HKEY_CLASSES_ROOT\CLSID\InprocServer32 123
- HKEY_CURRENT_USER 334
- HKEY_CURRENT_USER\Software\-Microsoft\VisualStudio\7.1\Add-Ins\<ProgID> 125
- HKEY_CURRENT_USER\Software\-Microsoft\VSA\7.1\AddIns\-<ProgID> 125
- HKEY_LOCAL_MACHINE 334
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\-7.1\AddIns\<ProgID> 125
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VSA\7.1 \Add-Ins\<ProgID> 125
— HKEY_USERS 334
расширение 19
расширяемость периода проектирования *см.* DTE
регулярные выражения 72-75
редактор 8. 48
— текста 49
решение 13, 22, 23, 24, 48, 174
- HTML-файл 24
- XML-файл 24
— выгрузка 175
— загрузка 175
— значок 24
— компоновка 46
— конфигурация 200, 207
— — отладочная 200
— — финальная 200
— надстройка 182
— обозреватель 51
— параметры 26
— проводник 22
— проект 24
— — начальный 26
— — параллельный запуск 26
— — порядок построения 27
— растровая картинка 24
— свойства
— — конфигурации 27
— — общие 26
— событие 177
— создание 175
— схема 24
- точка прерывания 31
— управление параметрами 200
- файл 23, 24, 223
— — исходный 27
— — параметров пользователя 30
— — символьный 27
— шаблон 24

C

сборка 22, 45, 78, 90, 250
свойство
— *ActivePane* 284, 285
— *ActivePoint* 294, 296
— *AdminToolsFolder* 336

— AdminUser 336
— AnchorPoint 294, 296
— AppDataFolder 336
— ARPCONTACT 336
— Author 336
— AutoHides 242
— Bindings 171
— BottomPoint 294, 296
— BuildEvents 113
— ButtonProperty 343
— Caption 242
— Category 256
- CheckBoxOption 280
— CheckBoxXProperty 343
— Checked 256
— CodeBase 250
— CodeElement 321
— Collection 112
— Command.Bindings 169
— Command.IsAvailable 155
— Command.Name 151
— CommandEvents 113, 118
— CommentTokens 259, 260
— CommonAppDataFolder 336
— CommonFilesFolder 336
— ComputerName 336
— Configuration.Properties 209
— ConfigurationName 201
— Connected 128, 136
- CurrentTab 284
— CurrentTabObject 284
-Date 336
— DebuggerEvents 113
— Description 256
— DescriptionLabel 233
- DesktopFolder 336
— DocumentParent 290
— Document.Saved 292
— Document.TextSelection 291
— DocumentEvents 113, 118
— DTE 111, 198, 284
— DTE.ActiveDocument 86
— DTE.ActiveSolutionProjects 177
- DTE.ActiveWindow 283
— DTE.AddIns 183
— DTE.CommandBars 156, 157
— DTE.Events 112
- DTEProperties 274, 277
— DTE.Windows 238
— DTEEvents 113
— EndPoint 86, 292
— Events.SolutionEvents 177
— Events.TaskListEvents 258
— FavoritesFolder 336
— FileName 256
— FindEvents 113
— FontsFolder 337
- FullName 112
— Globals 212
— GUID 149
— HeadingLabel 233
— Icon 233
— ID 149
— IDesignerHost.RootComponent 264, 266
— Intel 337
— IsHoating 242
— IsSettable 256
— ItemOperationsPromptToSave 175
— Kind 398
— Line 256
— Linkable 242
— LocalAppDataFolder 337
— LogonUser 337
— MainWindow 241
— Manufacturer 337
— MiscFilesEvents 113
- MsiNTProductType 337
— MyPicturesFolder 337
— Name 198
— NetHoodFolder 337
— Object 189, 241, 283
- ObjectKind 240
— Optimize 209
— OutputWindowEvents 113, 118
— Panes 284
— Parent 112, 265, 284
— PersonalFolder 337
— PhysicalMemory 337
— PrintHoodFolder 337
— Priority 256
— Privileged 337
— ProductID 337
— ProductName 337
— ProductVersion 337
— ProgramFilesFolder 337
— ProgramMenuFolder 338
— Project.ConfigurationManager 200
— Project.ProjectItems 184
— Project.UniqueName 176
— Projects 176
— PromptToSave 175

— *Readonly* 291
— *RecentFolder* 338
— *RemoteAdminTS* 338
— *ROOTDRIVE* 338
— *ScreenX* 338
— *ScreenY* 338
— *Selection* 86, 284, 297
— *SelectionEvents* 113
— *SendToFolder* 338
— *ServicePackLevel* 338
— *ServicePackLevelMinor* 338
— *ShouldBuild* 202
— *Solution* 174
— *SolutionAddIns* 183
— *Solution.SolutionBuild* 200
— *SolutionBuildBuildDependencies* 205
— *Solution Configuration.SolutionContexts* 201
— *SolutionContexts* 201
— *SolutionEvents* 113
— *SolutionItemsEvents* 113
— *SourceDir* 338
— *StartMenuFolder* 338
— *StartPoint* 86, 292
— *StartupFolder* 338
— *SubCategory* 256
— *SystemFolder* 338
— *SystemLanguageID* 338
— *TARGETDIR* 338
— *TaskItems* 251
— *TaskListEvents* 113, 118
— *TempFolder* 338
— *TemplateFolder* 338
— *TemplatePath* 219, 230
— *TextDocumentSelection* 292
— *TextEditorEvents* 113, 118
— *TextSelectionIsEmpty* 297
— *Time* 339
— *Title* 233
— *ToolBoxTab.ToolBoxItems* 248
— *TopPoint* 294, 296
— *UIHierarchy.SelectedItems* 245
— *UIHierarchyItem.Object* 246
— *UIHierarchyItem.UIHierarchyItems* 244
— *UndoContext.IsOpen* 300
— *UserLanguageID* 339
— *USERNAME* 339
— *VersionNT* 339
— *VirtualCharOffset* 294
— *VirtualMemory* 339
— *Visible* 112
— *VSProjectReferences* 190
— *Window.Object* 247, 264
— *WindowEvents* 113, 118
— *WindowsBuild* 339
— *WindowsFolder* 339
— *WindowsVolume* 339
— *ToolBox.ToolBoxTabs* 247

событие
— *AfterClosing* 113
— *AfterExecute* 113
— *BeforeClosing* 113
— *BeforeExecute* 113
— *DocumentClosing* 113
— *DocumentOpened* 113
— *DocumentOpening* 113
— *DocumentSaved* 113
— *FindDone* 113
— *IDTExtensibility2* 128
— *ItemAdded* 113, 199
— *ItemRemoved* 113, 199
— *ItemRenamed* 113, 199
— *LineChanged* 113, 304, 306
— *ModeChanged* 113
— *NavigateHandled* 257
— *OnAddInsUpdate* 128, 133, 134, 137
— *OnBeginShutdown* 113, 128, 137
— *OnBuildBegin* 113, 210
— *OnBuildDone* 113, 210, 212
— *OnBuildProjConfigBegin* 113, 210
— *OnBuildProjConfigDone* 113, 210
— *OnChange* 113
— *OnConnection* 111, 128
— *OnDisconnection* 128, 137
— *OnEnterBreakMode* 113
— *OnEnterDesignMode* 113
— *OnEnterRunMode* 113
— *OnExceptionNotHandled* 113
— *OnExceptionThrown* 113
— *OnMacrosRuntimeReset* 113, 117
— *OnStartupComplete* 113, 117, 128, 133
— *Opened* 113
— *PaneAdded* 113
— *PaneClearing* 113
— *PaneUpdated* 113
— *ProjectAdded* 113

— *ProjectRemoved* 113
— *ProjectRenamed* ИЗ
— *QueryCloseSolution* 113, 178
— ***Renamed*** 113, 182
— *SolutionEvents* 199
— *TaskAdded* 113
- *TaskModified* 113
— *TaskNavigate* 255
— *TaskNavigated* 113, 257, 259
- *TaskRemoved* 113
— *WindowActivated* 113, 267
- *WindowClosing* 113
— *WindowCreated* 113
- *WindowMoved* 113
— автоматизации 112
— интерфейс 119
— компоновки 114, 210
— надстройки 127
— переменная 116
— позднего связывания 119
— проекта 198
— решения 177
— фильтрация 117
согласователь 270
страница свойств 7

Ф

форма 14
— добавление элемента 265
— компоновка 266
функция
- *CMCallbackFunction* 321
— *GetMembers* 316
— ***Main*** 215, 342
— *MsgWin* *92*
— встраиваемая 5
— обратного вызова 114

Ч

частично уникальный идентификатор *см,* FUID

Э

элемент управления 247, 249. 266, 270, 279
— размещение 14

Я

язык определения интерфейса *см.* IDL

# Об авторах



Фото Джейсона Е. Фиша

*Слева направо: Брайан Джонсон, Крэйг Скибо и Марк Янг*

Брайан Джонсон редактирует материалы по программированию в Microsoft Press, Живет в Рэдмонде, шт. Вашингтон. с женой Кэтрин и тремя детьми: Уиллом. Хантером и Баффи. Брайан работал техническим редактором *журнала Microsoft Office and VBA Developer* и печатается в журналах *MSDN* и *Microsoft Visual J++ informant.* Он закончил университет шт. Висконсин и был военным корреспондентом корпуса морской пехоты.

Крэйг Скибо, проживающий в Рэдмонде, шт. Вашингтон, вот уже семь лет занимается в Microsoft разработкой Visual Studio. Часто выступает на отраслевых конференциях с докладами о модели автоматизации Visual Studio, а также консультирует разработчиков по вопросам использования новейших технологий Visual Studio. Закончил государственный университет Пенн.

Марк Янг около десяти лет редактирует материалы по программированию в Microsoft Press. Работал над книгами, связанными с технологиями Microsoft: от VGA-систем в MS-DOS до серверных компонентов .NET. Марк живет в Сиэтле со своей прекрасной женой, прелестным сынишкой и двумя котами-бандитами.

**Джонсон Брайан, Скибо Крэйг, Янг Марк**

Основы Microsoft Visual Studio .NET 2003

Перевод с английского под общей редакцией *В. Г. Вшивцева*

Переводчики *А. Г. Королев, К. А. Меньшов*

Технический редактор *О. В. Дергачева*

Компьютерная верстка *В. Б. Хильченко*

Дизайнер обложки *Е. В. Козлова*

Оригинал-макет выполнен с использованием
издательской системы Adobe PageMaker 6.0

TypeMarketFontLibrary
легальный пользователь

пользователь
Para(-)Type
IN LEGAL USE

Главный редактор *А. И, Козлов*

Подготовлено к печати издательством «Русская Редакция»
e-mail: info@rusedit.ru, http://www.rusedit.ru

РУССКАЯ РЕДАКЦИЯ

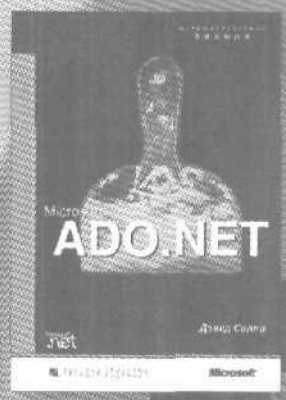
Подписано в печать 08.08.03 г. Тираж 3 000 экз.
Формат 70x100/16. Физ. п. л. 29

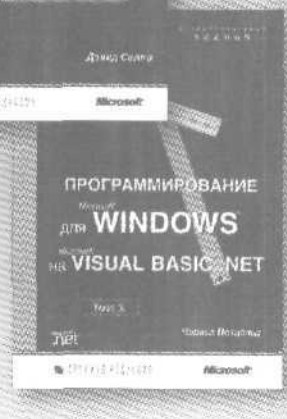
Отпечатано в ОАО «Типография «Новости»,
105005, г. Москва, ул. Фр. Энгельса, 46

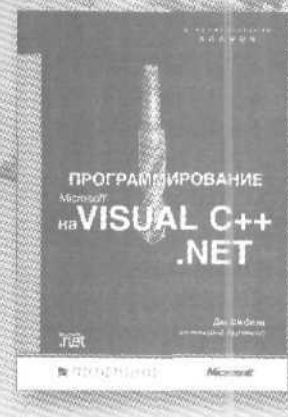
Издательство «Русская Редакция»
представляет серию книг

# Фундаментальные знания







Маститые авторы и ведущие специалисты Microsoft в области разработки — Чарльз Петцольд, Джеффри Рихтер, Джеф Просиз, Дэвид Сеппа и др. — познакомят вас с флагманской платформой Microsoft .NET. Каждая книга серии — это полное, обстоятельное руководство по .NET Framework, .NET Enterprise Servers, Microsoft Visual Studio .NET и др. Основа серии «Фундаментальные знания» — книги Microsoft Press со статусом **Core Reference** — ведущие издания от разработчиков для разработчиков.

издательство компьютерной литературы

РУССКАЯ РЕДАКЦИЯ

e-mail: [illegible]usedit.ru, http:// www.rusedit.ru

# Оперативная информация
## для профессионалов в журнале «MSDN Magazine/Русская Редакция»




**Своей** *популярностью этот* ***журнал обязан содержанию: читатель получает самую*** *свежую* ***и актуальную информацию как от*** **ведущих** ***разработчиков корпорации Microsoft, так и от известных специалистов в сфере*** *разработки* ***приложений. Глубокие и содержательные статьи*** *оказывают* **неоценимую помощь** *профессиональному разработчику,* ***регулярно использующему средства разработки Microsoft.***

*Из статьи* Ольги *Дергуновой, главы представительства Microsoft* в СНГ *(MSDN Magazine/Русская Редакция, спецвыпуск № 1, апрель 2002 г.)*

**Журнал содержит:**

**постоянные рубрики**

**Web: вопросы и ответы** — вопросы и ответы, связанные с Web-разработкой.

**XML** — основные спецификации XML (XMLSchema, пространство имен VS-Security, Web-сервисы и т. д.). Ответы на вопросы читателей.

ASP — все аспекты программирования с использованием ASP и ASP.NET.

**Доступ к данным** — проблемы разработки эффективных приложений для взаимодействия с СУБД, повышение производительности и обеспечение безопасности, а также программирование с использованием ADO.NET.

**На переднем крае** — передовые технологии разработки (SOAP, Web-сервисы, .NET Framework и др.).

**.NET** — Джеффри Рихтер об основных концепциях и понятиях .NET и о приемах программирования.

**Основы и тонкости** — актуальные проблемы программирования, интересующие в основном начинающих программистов.

**Отладка и оптимизация** — эффективные приемы и подходы к решению задач отладки и оптимизации приложений.

**Под капотом** — особенности функционирования системных компонентов и исполняющих сред, создание эффективного программного обеспечения. Для системных программистов и разработчиков.

**Изощренный код** — необычные приемы, трюки программирования. Для системных программистов и разработчиков.

**C++: *вопросы* и *ответы*** — вопросы и ответы, связанные с программированием на языке C++.

**тематические статьи**

— глубоко и подробно раскрывают проблемы разработки ПО.

*Информационная поддержка* журнала «MSDN Magazine/Русская Редакция» — **www.microsoft.com/rus/msdn/magazine**

**Оформить подписку и приобрести журнал можно *в* Интернет-магазине**

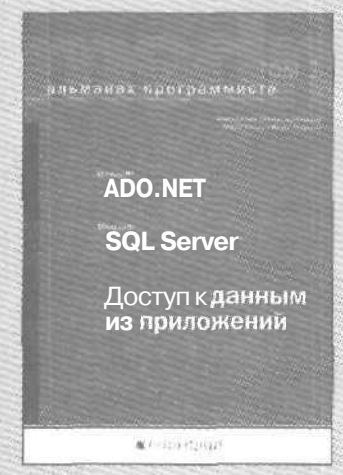
**www.ITbook.ru**

**Альманах** (от араб. аль-манах — календарь) — непериодический сборник, содержащий сведения из различных областей общественной деятельности, обычно с указанием литературных новинок, научных достижений, законодательных изменений и т. д.

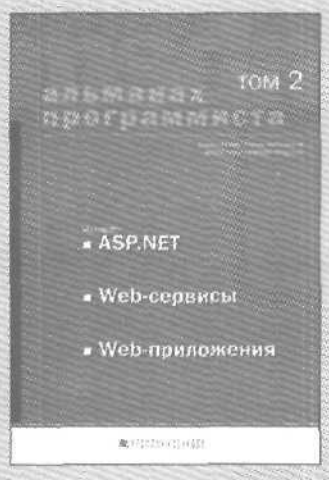
*Большая Советская Энциклопедия*

*Представляем тематический сборник*

# «АЛЬМАНАХ ПРОГРАММИСТА»




**Альманах программиста, том 1:** MicrosoftADO.NET, Microsoft SQL Server, доступ к данным из приложений.
400 стр., апрель 2003 г.




**Альманах программиста, том 2:** MicrosoftASP.NET, Web-сервисы, Web-приложения.
400 стр., сентябрь 2003 г.

***Это уникальное издание адресовано профессионалам в области современных информационных технологий.***
***Каждый том представляет собой тематический сборник статей из журнала «MSDN Magazine» и Microsoft MSDN Library по наиболее актуальным и перспективным технологиям разработки программного обеспечения.***

Планируется выпуск альманахов по базовым механизмам .NET Framework (модели защиты, отражение, удаленное взаимодействие, сервисы взаимодействия с неуправляемым кодом и т. д.), специфике языков программирования, поддерживающих .NET, отладке/тестированию и другим темам.

Если вас интересует специфическая тематика или определенные материалы из «MSDN Magazine» и MSDN Library, обращайтесь на сайт издательства *www.rusedit.ru* или по адресу ***almanah@rusedit.ru.*** Мы постараемся учесть ваши пожелания в будущих выпусках альманаха.


издательство компьютерной литературы
РУССКАЯ РЕДАКЦИЯ

**Продажа книг**
**оптом:** e-mail: sale@rusedit.ru;
интернет-магазин: http:// www ITbook.ru

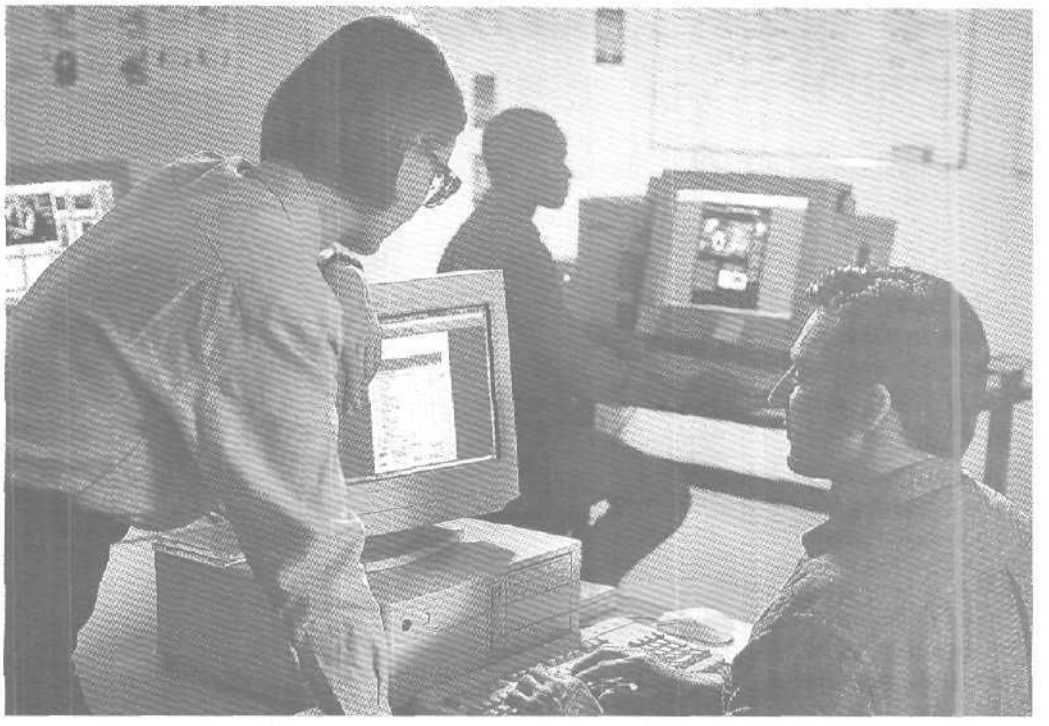
# HARD 'n' SOFT

www.hardnsoft.ru



МНОГОГРАННЫЙ
КОМПЬЮТЕРНЫЙ
ЖУРНАЛ





# Учебный центр SoftLine

softline®
education

Microsoft
CERTIFIED
Technical Education Center

## Ваш курс начинается завтра!

Подготовка сертифицированных инженеров и администраторов Microsoft

**Авторизованные и авторские курсы по:**

- Windows 2000 / XP
- Sun Solaris S
- Visual Studio .NET

да Электронной коммерции

* Безопасности информационных систем

и еще более 40 курсов по самым современным компьютерным технологиям.

**Дневные и вечерние занятия.**

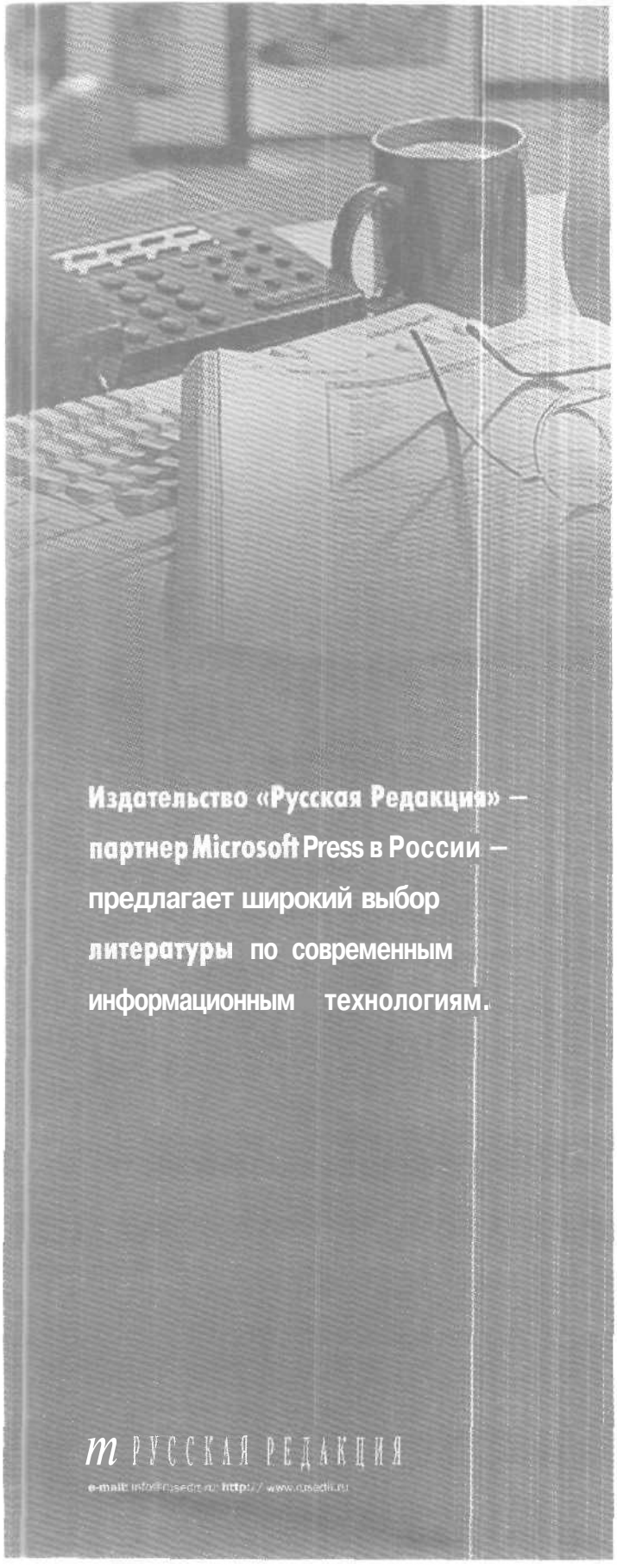
**Опытные преподаватели.**

**Индивидуальные консультации.**

**Учебный центр SoftLine**

119991 г. Москва, ул. Губкина, д. 8
тел.: (095) 232 00 23
e-mail: educ@softline.ru
http://education.softline.ru

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
softline
ЛИЦЕНЗИРОВАНИЕ • ОБУЧЕНИЕ • КОНСАЛТИНГ
www.softline.ru • 232 0023 - info@softline.ru



**Издательство «Русская Редакция» — партнер Microsoft Press в России — предлагает широкий выбор литературы по современным информационным технологиям.**

m РУССКАЯ РЕДАКЦИЯ

e-mail: info@rusedit.ru http://www.rusedit.ru

Наши **книги Вы можете приобрести**

• **в Москве:**

Специализированный магазин «Компьютерная и деловая книга» Ленинский проспект строение 3В. тел : (095) 778-7269

«Библио-Глобус» ул. Мясницкая, 6, тел.: (095) 928-3567

«Московский дом книги» ул. Новый Арбат 3. тел.: (095) 290-4507

«Дом технической книги» Ленинский пр-т 40. тел.: (095) 137-6019

«Молодая гвардия» ул. Большая Полянка, 28, тел.: (095) 238-5001

«Дом книги на Соколе» Ленинградский пр-т, 78, тел.: (095) 152-4511

«Дом книги на Войковской» Ленинградское Ш. 13, стр. 1. тел.: (095) 150-6917

«Мир печати» ул. 2-я Тверская-Ямская, 54, тел.: (095) 978-5047

Торговый дом книги «Москва» ул. Тверская, 8, тел.: (095) 229-6483

• **в Санкт-Петербурге:**

СПб Дом книги, Невский пр-т., 28 тел.: (812) 318-6402

СПб Дом военной книги, Невский пр-т., 20 тел.: (812) 312-0563, 314-7184

Магазин "Подписные издания", Литейный пр-т., 57. тел.: (812) 273-5053

Магазин «Техническая книга», уп. Пушкинская, 2, тел.: (812) 164-6565, 164-1413

Магазин «Буквоед», Невский пр-т., 13. тел.: (812) 312-6734

ЗАО «Торговый Дом «Диалект», тел.: (812) 247-1433

Оптово-розничный магазин «Наука и техника», тел. (812) 567-7025

• **в Екатеринбурге:**

Магазин «Дои книги», ул. Валека, 12. тел : (3432) 59-4200

• **в Великом Новгороде:**

«Наука и техника», уп БольшаяСанкт-Петербургская, 44, Дворец Молодежи, 2-й этаж

• **в Новосибирске:**

ООО «Топ-книга», тел : (3832) 36-1026

• **в Алма-Ате (Казахстан):**

ЧП Болат Амреев, моб. тел.: 8-327-908-28-57, (3272) 76-1404

• **в Киеве (Украина):**

000 Издательство «Ирина-Пресс», тел.: (+1038044) 269-0423

"Техническая книга на Петровке", тел (+1038044) 268-5346

*Эта книга посвящается моим родителям — Мерседес и Джону В. Коннелл. Они всегда со мной и никогда не отказывают в поддержке в любых моих начинаниях. Два моих сына, Гради и Гарретт, которыми я безмерно горжусь, а также моя младшая сестра Патрисия тоже рядом и поддерживают мой дух прежде всего своим юмором. Спасибо, ребята, без вас я бы ничего не сделал.*