



РАЗРАБОТКА КОМПОНЕНТОВ В MS VISUAL STUDIO 2005/2008



РАЗРАБОТКА И ОТЛАДКА
КОМПОНЕНТОВ

КОМПОНЕНТЫ WIN FORMS

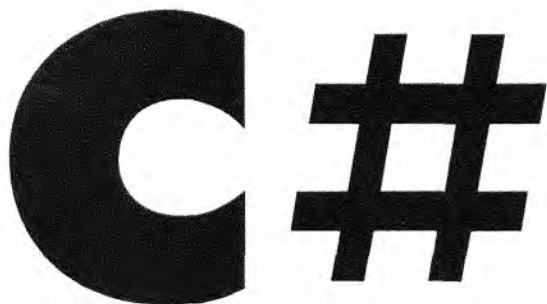
ДИЗАЙНЕРЫ, КОНВЕРТЕРЫ
ТИПА, СЕРВИСЫ,
РЕДАКТОРЫ ТИПОВ

ЛИЦЕНЗИРОВАНИЕ
И РАСПРОСТРАНЕНИЕ

ПРАКТИЧЕСКИЕ ПРИМЕРЫ

PRO
ПРОФЕССИОНАЛЬНОЕ
ПРОГРАММИРОВАНИЕ

Павел Агуров



**РАЗРАБОТКА КОМПОНЕНТОВ
В MS VISUAL STUDIO
2005/2008**

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.068+800.92
ББК 32.973.26-018.1
А27

Агуров И. В.

А27 С#. Разработка компонентов в MS Visual Studio 2005/2008. — СПб.: БХВ-Петербург, 2008. — 480 с.: ил. + CD-ROM — (Профессиональное программирование)

ISBN 978-5-9775-0295-5

Книга содержит всю необходимую информацию для создания полноценных компонентов Win Forms на языке C# в MS Visual Studio 2005/2008, начиная с разработки и отладки и заканчивая лицензированием и распространением. Рассмотрены дизайнеры, конвертеры типа, сервисы, редакторы типов и многое другое. Информация о каждом классе, описанном в книге, сопровождается примером его использования. Весь программный код является авторской разработкой и проверен на практике. На компакт-диске приведены примеры из книги.

Для программистов

УДК 681.3.068+800.92
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Анна Кузьмина</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн обложки	<i>Игоря Цырульниковой</i>
Оформление обложки	<i>Елены Беляевой</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 23.05.08.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 38,7.

Тираж 1500 экз. Заказ № 338

«БХВ-Петербург», 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.002108.02.07 от 28.02.2007 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП «Типография «Наука»
190334, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0295-5

© Агуров И. В., 2008

© Оформление, издательство «БХВ-Петербург», 2008

Оглавление

Введение.....	9
Программные требования	10
Аппаратные требования	10
О программном коде.....	10
Краткое описание глав.....	11
Обозначения	12
Благодарности	13
Обратная связь	13
 Глава 1. Что такое компоненты	15
1.1. Компоненты: достоинства и недостатки.....	15
1.2. Особенности разработки компонентов	17
 Глава 2. Дизайнер MS Visual Studio	19
2.1. Среда Visual Studio.....	19
2.2. Дизайнер формы	20
2.3. Визуальные и не визуальные компоненты в дизайнере формы	23
2.4. Палитра компонентов	23
2.5. Окно свойств	24
2.6. Редакторы свойств	25
 Глава 3. Немного теории	29
3.1. Свойства и события	29
3.2. Интерфейсы	34
3.3. Метаинформация, отражение и атрибуты.....	35
 Глава 4. Начнем с нуля	43
4.1. Общая архитектура	43
4.2. Класс <i>Component</i>	45
4.3. Контейнеры	47
4.4. Хосты	51
4.5. Дескрипторы свойств.....	52
4.6. Определение режима разработки	58

4.7. Компоненты или элементы управления?	62
4.8. Выбор базового класса	63
4.9. Отладка компонентов в режиме разработки	65
Глава 5. Создаем первый компонент	69
5.1. Создаем проект	69
5.2. Добавляем информацию о свойствах	72
5.3. Свойство, событие и значение по умолчанию	74
5.4. Дополнительные атрибуты свойств	76
5.5. Реакция на изменение свойств	78
5.6. Описание и значок компонента	79
5.7. События	81
5.8. Установка компонентов	82
5.9. Первые итоги	84
Глава 6. Дизайнеры	85
6.1. Что такое дизайнер	85
6.2. Добавление дизайнера в проект	87
6.3. Привязка дизайнера к компоненту	88
6.4. Смарт-теги	89
6.5. Контекстное меню компонента	100
6.6. Скрытие элементов из редактора свойств	103
6.7. Виртуальные свойства	104
6.8. Дополнительная отрисовка компонентов	107
6.9. Обработка событий мыши в режиме разработки	109
6.10. Линии взаимного расположения компонентов	111
6.11. Действие по умолчанию	112
6.12. Прямая обработка очереди сообщений	113
6.13. Встроенные компоненты	114
6.14. Слои и маркеры	117
6.15. Реализация дополнительного меню с помощью маркера	125
6.16. Правила изменения размера и положения компонента	129
6.17. Расположение компонентов друг на друге	131
6.18. Сетка дизайнера	132
6.19. Инициализация компонентов	133
Глава 7. Сервисы	135
7.1. Сервисы режима разработки	135
7.2. Управление окном смарт-тега	138
7.3. Управление слоями и маркерами	138
7.4. Обработка изменений компонентов и их свойств	139
7.5. Управление выделенными компонентами	145
7.6. Расширение контекстного меню формы	149
7.7. "Горячие" команды формы	150
7.8. Работа с именами и типами компонентов	154
7.9. Управление событиями компонента	158

7.10. Доступ к параметрам дизайнера	161
7.11. Нотификации о смене дизайнера	164
7.12. Пользовательские данные времени разработки	168
7.13. Получение информации о проекте	171
7.14. Вывод отладочной информации	175
7.15. Встроенные компоненты	176
7.16. Взаимодействие с палитрой компонентов	179
7.17. Контекстная подсказка компонента	188
7.18. Добавление и удаление свойств, событий и атрибутов компонентов	190
7.19. Работа с файлами ресурсов	195
7.20. Создание диалоговых окон в режиме разработки	196
7.21. Управление окнами Visual Studio	200
7.22. Сервис <i>IPropertyValueUIService</i>	201
7.23. Добавление специальной области в редактор формы	206

Глава 8. Конвертеры типов.....209

8.1. Сложное свойство	210
8.2. Конвертеры типов	214
8.3. Конвертеры типов .NET Framework 2.0	215
8.4. Реализация собственного конвертера типа	220
8.5. Стандартный набор значений	224
8.6. Класс <i>SimplePropertyDescriptor</i>	233
8.7. Редактирование флагов	234
8.8. Динамическое управление набором свойств	239
8.9. Стандартное сложное свойство	246
8.10. Стандартный набор значений-классов	247
8.11. Определение порядка отображения свойств	254

Глава 9. Редактор типа

255

9.1. Класс <i>UTypeEditor</i>	255
9.2. Классы стандартных редакторов типов	256
9.3. Реализация собственного редактора типа	258
9.3.1. Выпадающий диалог	259
9.3.2. Модальный диалог	267
9.3.3. Изображение значений	270
9.3.4. Использование стандартных редакторов типов	272
9.3.5. Редактор с параметрами	294

Глава 10. Сериализация времени разработки.....299

299

10.1. Провайдеры	299
10.2. Преимущества модели провайдеров	300
10.3. Принцип работы провайдеров	301
10.4. Управление сериализацией	305
10.5. Динамическое управление сериализацией	312
10.6. Локализация	313

10.7. Настраиваемые свойства	316
10.8. Управление кодом сериализации	318
Глава 11. Рендеринг, расположение и поведение компонентов.....	323
11.1. Рендеринг компонентов.....	323
11.2. Стандартный рендеринг	324
11.3. Управление отрисовкой и поведением компонентов.....	326
11.4. Компонент, невидимый в режиме выполнения	328
11.5. Управление расположением элементов	328
Глава 12. Копаем глубже	333
12.1. Компоненты-расширители и глобальные свойства.....	333
12.2. Транзакции дизайнера	347
12.3. Добавление закладки в редактор свойств	349
12.4. Обобщения.....	352
12.5. Потоки.....	357
12.6. Взаимодействие с Win32 API.....	365
Глава 13. Компоненты работы с данными	381
13.1. Несколько слов о привязке данных	381
13.2. Стандартная привязка данных	386
13.3. Интерфейсы источника данных	390
13.4. Разработка компонентов работы с данными	393
13.5. Связь свойств между собой.....	409
Глава 14. Распространение компонентов.....	411
14.1. Три составляющие и позднее связывание.....	411
14.2. Где Visual Studio ищет сборки	413
14.2.1. Подпись сборок.....	414
14.2.2. Установка сборок в GAC	416
14.3. Установка компонентов.....	419
14.3.1. Программная установка компонентов	421
14.4. Процедура разработки	432
14.5. Лицензирование	433
14.5.1. Провайдер лицензий.....	433
14.5.2. Класс <i>LicFileLicenseProvider</i>	433
14.5.3. Подключение провайдера лицензий к компоненту	434
14.5.4. Класс <i>License</i>	434
14.5.5. Класс <i>LicenseContext</i>	435
14.5.6. Класс <i>LicenseManager</i>	435
14.5.7. Реализация собственного алгоритма лицензирования	436
14.5.8. Получение уникальной информации о компьютере	437
Глава 15. Атрибуты времени разработки	449

Глава 16. FAQ	453
16.1. Общие определения	455
16.2. Режим разработки	455
16.3. Дизайнер формы и компонентов	457
16.4. Отрисовка компонентов	459
16.5. Редактирование свойств и сериализация	460
16.6. Редактирование размеров и положения компонентов	463
Приложение 1. Полезные ссылки	465
Приложение 2. Описание компакт-диска.....	469
Литература	473
Предметный указатель	475

Современные информационные технологии и эффективные решения для вашего предприятия



О компании EPAM Systems

EPAM Systems - крупнейший разработчик проектного (заказного) программного обеспечения и один из ведущих игроков в области консалтинга в Центральной и Восточной Европе. Основана в 1993 г. В штате более 4500 специалистов, выполняющих проекты в более чем 30 странах мира.

Отделения компании расположены в России, Республике Беларусь, Украине, Армении, Казахстане, США, Венгрии, Великобритании, Германии.

Среди клиентов EPAM Systems: SAP, Microsoft, Oracle, BEA Systems, "Ренессанс-Капитал", S7 Airlines, "Газпром нефть", "РосНефть", "Росэнергоатом", Российский фонд федерального имущества, Агентство по Информатизации и Связи Республики Казахстан, Reuters, Samsung America, Colgate-Palmolive, AeroMexico, Coca-Cola, London Stock Exchange и другие.

www.epam-group.ru, www.epam.com

Об Учебном Центре EPAM Systems

Учебный Центр EPAM Systems - внутренние подразделения компании, занимающиеся обучением и подготовкой ИТ-специалистов на территории России, Армении и Казахстана.

С момента основания в 1999 году в Учебном Центре прошли подготовку свыше 500 профессиональных инженеров в области разработки и тестирования программного обеспечения.

www.training.ru

Введение

В начале несколько слов о причинах появления этой книги. Я видел очень много хороших изданий по C#, Visual Studio и множество других, но почему-то не нашел книги, посвященной разработке компонентов в среде Visual Studio. Признаться, я обнаружил несколько книг на английском языке ([1], [2]), но их авторы уделяли слишком большое внимание таким вопросам, как отрисовка компонентов, описание стандартных компонентов или среды, но почти не описывали методы и классы режима разработки. Надеюсь, эта книга поможет тем, кто хочет создавать свои компоненты и пользоваться этим мощным и удобным средством разработки приложений.

Конечно, я не мог включить в эту книгу абсолютно все, но постарался сделать так, чтобы у читателя была вся необходимая информация для создания и распространения своих компонентов без проблем и головной боли. Справочная система MSND — не самый простой источник получения информации, поэтому я надеюсь, что эта книга поможет и тем, кто не хочет тратить время на поиск нужной информации, и тем, кто не очень силен в английском языке.

Эта книга посвящена только разработке компонентов для приложений типа Windows Forms в среде Visual Studio 2005/2008. Компоненты ASP.NET — особая и очень большая тема, требующая отдельной книги, возможно, большей по объему, чем эта. Кроме того, я не описывал здесь расширения, вошедшие в .NET Framework 3.5, такие как Windows Presentation Foundation и т. д. Надеюсь, в ближайшее время получится выпустить еще одну книгу, посвященную только этим вопросам.

Эта книга посвящена именно режиму разработки и не содержит слишком подробной информации о методах отрисовки и графических примитивах. Очень просто найти книгу, в которой написано, как рисовать линию или как залить прямоугольник... Но ведь этого мало! Не менее важен вопрос, как сде-

лать такую отрисовку отдельно для режима разработки и для режима выполнения, или как менять свойства этой заливки в режиме разработки. И именно этому вопросу и посвящена книга.

Для простоты я старался почти все примеры сделать очень небольшими. Каждый из них можно использовать как по отдельности, так и объединять их в единую систему в зависимости от задачи. Чтобы быстро найти нужную информацию, в книге есть две специальные главы: общая сводка атрибутов, используемых во время разработки (*см. главу 15*) и ответы на наиболее часто задаваемые вопросы (*см. главу 16*). Кроме того, в конце книги есть индекс. Надеюсь, это поможет читателю (и мне тоже) быстро найти нужную информацию.

Программные требования

Для компиляции примеров достаточно среды Visual Studio 2005/2008 (C#) и библиотеки .NET 2.0/3.0/3.5. Желательно также иметь справку MSDN, хотя можно пользоваться и сайтом www.microsoft.com.

Аппаратные требования

Достаточно обычного домашнего компьютера, на котором компиляция программы в Visual Studio 2005/2008 занимает приемлемое для вас время.

Установка Visual Studio 2005 (C#) потребует примерно 600 Мбайт на жестком диске, Visual Studio 2008 (C#) — примерно столько же, MSDN — 1,5 Гбайт, Windows DDK — 700 Мбайт, .NET 2.0 — 70 Мбайт.

О программном коде

Книга содержит полные исходные коды всех программ, однако во многих листингах я приводил только изменения кода, относительно предыдущего листинга. Такое сокращение позволяет не только экономить место, но и улучшить понимание кода, делая акцент лишь на новой функциональности. В большинстве листингов я не приводил автоматически генерируемый код, думаю, что набирать такой код "с листа" не очень интересно. Код на компакт-диске содержит полные, компилируемые и работающие тексты всех приложений, без сокращений.

Краткое описание глав

Глава 1 содержит обсуждение общих вопросов разработки приложений с помощью компонентов: в чем достоинства и недостатки такой разработки, какие особенности и подводные камни ожидают программиста на этом пути... Я бы рекомендовал не оставлять эту главу без внимания, хотя, конечно, знаю, что программисты любят сразу переходить к коду, чем читать текст.

Глава 2 содержит небольшое описание среды Visual Studio, необходимое для того, чтобы договориться об именовании окон и общих понятий. Может оказаться, что некоторые мои обозначения покажутся не очень привычными, но в любом случае нам нужно будет договориться, как называть различные части среды, чтобы говорить на одном языке.

Глава 3 содержит минимальное количество теории, необходимой для дальнейшей работы. Если вы знаете, что такое атрибуты, интерфейсы, делегаты и отражение — можете смело пропустить эту главу, хотя некоторые ссылки на нее будут встречаться по тексту книги. Более детальное описание этих вопросов можно найти в других книгах или в MSDN.

Глава 4 содержит описание общей архитектуры и базовых классов времени разработки. Кроме того, эта глава содержит инструкцию к одному из самых важных вопросов — как отлаживать код компонентов во время разработки. Если до этого момента вам не приходилось разрабатывать свои компоненты, то эту главу нужно прочитать обязательно. Впрочем, и более опытные коллеги найдут в ней немало интересного.

Глава 5 содержит пошаговую инструкцию создания собственного компонента. По ходу дела я добавляю сюда описание некоторых базовых атрибутов и другую информацию, которая необходима, независимо от задачи. Это основы создания компонентов.

Глава 6 содержит описание специального объекта, называемого дизайнером. Дизайнеры позволяют управлять набором свойств и поведением компонента в режиме разработки, создавать пользовательские меню и многое другое. В этой главе я постараюсь рассказать про эти возможности.

Глава 7 продолжает наше погружение в мир режима разработки. Она содержит информацию о сервисах, предоставляемых средой для режима разработки. Сервисов очень много, и, как минимум, я рекомендовал бы ознакомиться с их возможностями. Может быть, это значительно упростит разработку компонентов и позволит не изобретать то, что уже реализовано в стандартной библиотеке и среде Visual Studio.

Глава 8 содержит информацию о создании так называемых "сложных свойств", т. е. свойств, содержащих набор других. Кроме того, здесь же я рас-

скажу, как создавать стандартные наборы значений для свойства, а также о динамическом управлении списком свойств.

Глава 9 содержит информацию о создании собственного редактора свойства. Собственные свойства всегда интереснее редактировать как-то по-своему. Кроме того, стандартные редакторы свойств позволяют легко осуществить редактирование коллекций объектов, нарисовать изображение значения свойства и сделать множество других интересных вещей.

Глава 10 содержит информацию о сериализации времени разработки. Среда Visual Studio применяет принципиально новый метод сериализации. Без понимания новых принципов работы не получится построить корректно и эффективно работающий компонент, содержащий, например, свойства-коллекции.

Глава 11 содержит информацию о принципах визуальной отрисовки компонентов и управлении взаимным расположением компонентов. Здесь же я расскажу о некоторых классах и методах, используемых для рисования.

Глава 12 заведет нас в глубину режима разработки. Я расскажу про разработку глобальных свойств компонентов, транзакции дизайнера, многопоточную работу и многое другое. Вполне вероятно, что такие возможности вам не потребуются, но лучше иметь о них хотя бы общее представление.

Глава 13 содержит информацию о компонентах, предназначенных для работы с данными.

Глава 14 рассказывает о вопросах лицензирования и распространения компонентов.

Глава 15 содержит общую сводку атрибутов, используемых в режиме разработки. Кроме того, в ней приводится описание некоторых атрибутов, которым не нашлось места по ходу книги.

Глава 16 содержит ответы на наиболее часто возникающие вопросы. Возможно, с нее стоит начать, если вы уже умеете разрабатывать компоненты, но столкнулись с проблемой, которую не можете решить.

Обозначения

- ❑ Visual Studio — Microsoft Visual Studio, обычно имеется в виду версия 2005/2008, если не указана другая.
- ❑ MSDN — Microsoft Development Network, справочная система, доступная по адресу <http://msdn2.microsoft.com/ru-ru/default.aspx>.
- ❑ WPF — Windows Presentation Foundation.

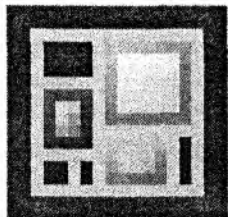
Благодарности

Хочется поблагодарить издательство и всех, кто помогал в разработке и коррекции текста книги. Также спасибо моим коллегам из компании ЕРАМ Systems.

Обратная связь

Обо всех ошибках или пожеланиях сообщайте по адресу **mail@bhv.ru**. На сайте **book.pvasoft.com** можно найти текущий список опечаток и дополнений.

ГЛАВА 1



Что такое компоненты

Компонент (от лат. *componens* — составляющий) — составная часть, элемент чего-либо.

(Выписка из словаря)

Коротко говоря, компонент — это "кирпичик" программы, состоящий из свойств (*properties*), методов (*methods*) и событий (*events*). *Свойства* дают возможность управлять видом и поведением компонента, *методы* — использовать возможности, предоставляемые компонентом, а *события* — реагировать на происходящие внутри компонента события, программировать реакцию компонента на внешние события и т. д.

Разработка программы с помощью компонентов называется *компонентно-ориентированной* разработкой. В этой главе я постараюсь описать, какие преимущества дает такой подход и какие подводные камни ждут нас на этом пути.

1.1. Компоненты: достоинства и недостатки

Компонентно-ориентированная разработка имеет свои сильные и слабые стороны. Несомненными достоинствами является повторная используемость кода, согласованность пользовательского интерфейса, возможность быстрой и продуктивной разработки программ. Именно компоненты позволяют программистам составлять конечный продукт из "кирпичиков", не вдаваясь в детали реализации конкретного компонента. Конечно, наборы классов, используемые при объектно-ориентированном подходе, тоже дают возможность повторного использования кода, но компоненты делают повторное использование кода совершенно естественным.

Если при разработке системы все программисты команды пользуются одним и тем же набором визуальных компонентов, то, само собой, интерфейс у программы будет выполнен в едином стиле. Более того, поменяв, например, вид одного из компонентов, мы изменим его вид везде, где он используется.

Компоненты дают возможность независимой разработки частей интерфейса. Изменения внутри компонента не затрагивают код модулей, в которых он используется. Разработка нескольких независимых классов дает примерно те же результаты, за исключением одной проблемы. Среди программистов средней квалификации наблюдается тенденция перемешивать функциональность классов, путая методы одного класса с другим. Компоненты разделяют код более качественно. Особо "одаренным" можно не отдавать исходный код компонентов, которые не касаются разрабатываемых ими модулей.

Свойства компонентов позволяют наиболее эффективно объяснить другому программисту, как использовать компонент. Специальные редакторы свойств позволяют быстро настроить вид и поведение компонентов.

Наконец, накопив достаточное количество компонентов, можно действительно быстро создать визуальный интерфейс программы, фактически, не написав более ни строчки кода!

Чем же придется заплатить за все это? Как это не удивительно звучит, но платить придется объектно-ориентированным подходом. Возможность гибкого управления поведением компонентов с помощью событий провоцирует написание "событийно-ориентированного" кода. Пусть, например, нам нужен компонент для отображения цветных строк. Объектно-ориентированный подход обязывает нас создать наследника класса `ListBox` и, перекрыв метод `Paint`, реализовать отрисовку цветных строк. Возможность реализовать событие `OnPaint` и не создавать никаких классов подталкивает многих программистов к использованию событий в ущерб объектно-ориентированному подходу. Я специально говорю "подталкивает", т. к. никто не мешает создать новый компонент, умеющий рисовать цветные строки на основе существующего компонента `ListBox`. Такой подход и будет наиболее верным — ведь такие компоненты можно использовать повторно!

Еще одна плата за удобство — необходимость иметь гибкие компоненты. Нет смысла писать компонент, рисующий только строки красного цвета. Такой компонент будет затруднительно использовать где-либо, кроме той программы, для которой он изначально предназначался. Гораздо правильнее написать компонент, рисующий строки заданного цвета, а сам цвет вынести в его свойства (или как-то хранить внутри самой строки). Вот такой компонент можно использовать в любой программе. Такая гибкость требует некоторых дополнительных усилий, необходимость затратить которые не всегда очевидна при разработке одной программы, но вполне окупается при повторном использовании компонента.

Подводя итог этому рассуждению, можно сказать, что цель разработки нового компонента — создание новой функциональности, являющейся независимой, но гибко настраиваемой частью, допускающей повторное использование.

1.2. Особенности разработки компонентов

В отличие от обычного класса, разработка компонента имеет особенности, которые следует учитывать, принимая решение о создании компонента.

Во-первых, пользователем компонента является такой же программист, как и мы с вами. Именно он, а не конечный пользователь программы, использует компонент. Конечному пользователю компонент достанется уже потом, когда программа будет готова.

Во-вторых, компонент имеет три интерфейса: интерфейс времени выполнения, интерфейс времени проектирования и программный интерфейс.

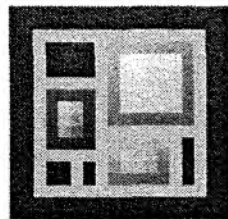
Интерфейс времени выполнения — это поведение компонента во время работы конечной программы. Например, компонент `MaskEdit` имеет поле ввода и умеет проверять правильность вводимых данных. Поведение времени выполнения складывается как из кода, заложенного в сам компонент, так и из кода, созданного пользователем компонента, с помощью обработчиков событий.

Интерфейс времени проектирования позволяет настроить вид и поведение компонента во время разработки. Стандартный дизайнер формы позволяет визуально настроить размер и положение компонента на форме, редактор свойств позволяет легко и быстро менять значения ключевых свойств, влияющих на вид и поведение компонента, а специальные редакторы самого компонента (если они есть) позволяют проводить тонкую настройку свойств компонента еще более быстро и эффективно. Конечно, программирование поведения компонента во время проектирования требует дополнительных усилий, иногда превышающих усилия на программирование самого компонента. Но, поверьте, хороший интерфейс времени проектирования того стоит!

Программный интерфейс — это свойства, методы и события, которые позволяют компонентам общаться между собой и с основной программой.

Основная особенность разработки компонента — сбалансированность гибкости компонента и времени, потраченного на его создание, плюс оценка затрат на разработку компонента с учетом возможности его повторного использования. Слишком гибкий компонент будет иметь очень много свойств, и его будет сложно настраивать. Наоборот, компонент, реализующий слишком частную ситуацию, будет бесполезен в плане повторного использования, и, возможно, затраты на его реализацию не будут иметь смысла.

ГЛАВА 2



Дизайнер MS Visual Studio

Пользователь всегда прав, но не должен об этом знать.

Нет, не пугайтесь, я не собираюсь детально разбирать здесь всю среду разработки MS Visual Studio, тратя бесценное книжное место. Но без некоторых деталей нам все же не обойтись. Как минимум, нам надо договориться об именовании основных окон среды. Итак, создадим новый проект типа **Windows Application** и посмотрим, что в нем есть полезного.

2.1. Среда Visual Studio

При создании нового приложения **Windows Application** среда открывает редактор формы и несколько окон (рис. 2.1), среди которых обычно (в зависимости от настроек среды) окна **Solution Explorer** (Обозреватель проекта), **Toolbox** (Инструменты) и **Properties** (Свойства). Именно эти три окна представляют для нас наибольший интерес.

В окне **Solution Explorer** (Обозреватель проекта) можно выбрать нужный для редактирования элемент: файл кода, форму и т. д. В соответствии с этим меняется редактор в левой части. Для формы отображается редактор формы, называемый еще *дизайнером формы*. Практически аналогичный редактор открывается при редактировании компонентов. При редактировании кода открывается текстовый редактор. Есть и другие редакторы (дизайнеры), но пока нам вполне достаточно знать про эти два.

В следующих разделах я расскажу про каждый из элементов среды более подробно.

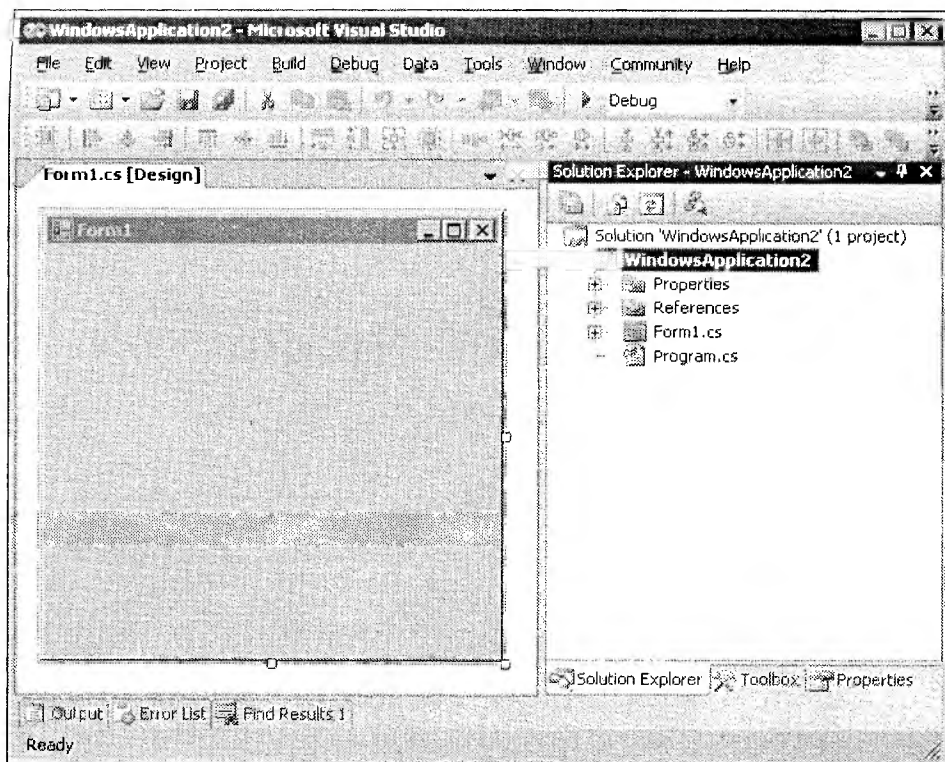


Рис. 2.1. Среда разработки Visual Studio

2.2. Дизайнер формы

В этом разделе я очень кратко опишу основные понятия, которые потребуются нам при работе с редактором формы и компонентов.

Дизайнер формы (form designer) имеет три маркера для изменения размера формы. С их помощью можно увеличить или уменьшить размер формы по высоте и ширине или одновременно по обоим параметрам, пропорционально (рис. 2.2).

Если на форме выбрать какой-нибудь элемент, то редактор показывает *маркеры* изменения размеров и положения этого элемента (glyph). Причем, как видно на рис. 2.3, маркеры меняются в зависимости от реальной возможности изменять размеры. Так, компонент `Button` можно менять по всем параметрам (во всех направлениях), а для компонента `CheckBox` доступно только изменение положения компонента на форме. Соответственно, компонент `Button` имеет восемь маркеров, а компонент `CheckBox` только один.

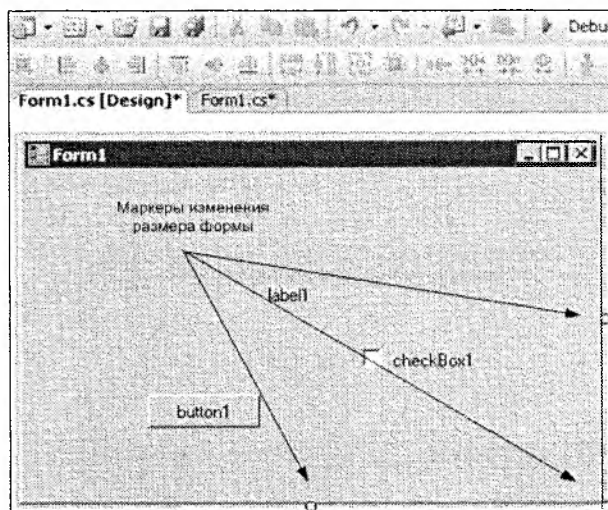


Рис. 2.2. Редактор формы

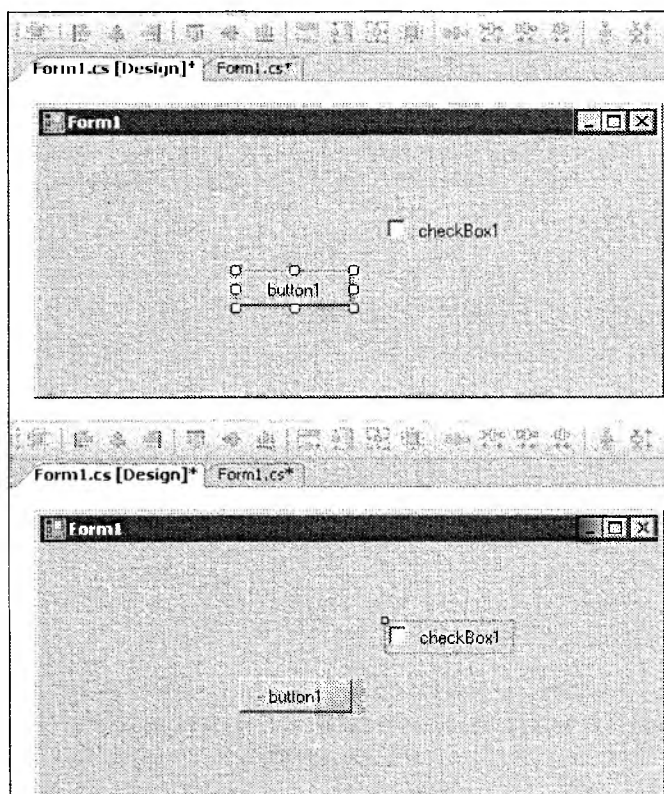


Рис. 2.3. Маркеры изменения размеров компонентов

Если при выборе компонентов на форме удерживать клавишу <Shift>, можно выбрать несколько компонентов одновременно. Первый из них будет иметь белые маркеры и являться *основным компонентом* (primary) в выбранной группе. Остальные выбранные компоненты будут иметь темные маркеры (рис. 2.4).

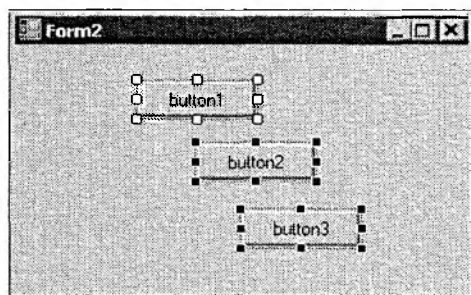


Рис. 2.4. Маркеры первого в группе компонента имеют белый цвет

Интересной особенностью редактора Visual Studio является подсказка о взаимном расположении элементов (snap lines). Например, при перемещении кнопки button1 вправо редактор отображает синюю маркерную линию, ведущую к компоненту Checkbоx1, которая помогает выполнить выравнивание компонента вправо (рис. 2.5).

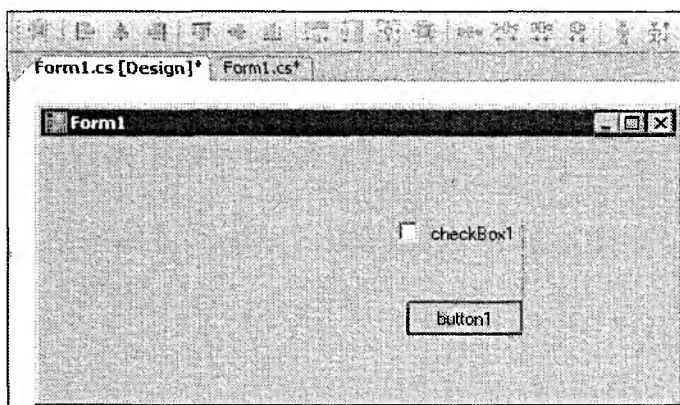


Рис. 2.5. Выравнивание взаимного расположения компонентов формы

2.3. Визуальные и невидимые компоненты в дизайнера формы

Библиотека .NET Framework имеет два типа компонентов: *визуальные* и *невизуальные*. Визуальные компоненты являются элементами пользовательского интерфейса. Это, например, компоненты: кнопка (Button), выпадающий список (ComboBox) или метка (Label). Невизуальные компоненты не имеют пользовательского интерфейса и не могут располагаться на форме. Дизайнер Visual Studio располагает их внизу окна дизайнера¹ (рис. 2.6). Такими компонентами являются, например, компоненты работы с базами данных, таймер (Timer) и компонент работы с последовательным портом (SerialPort).

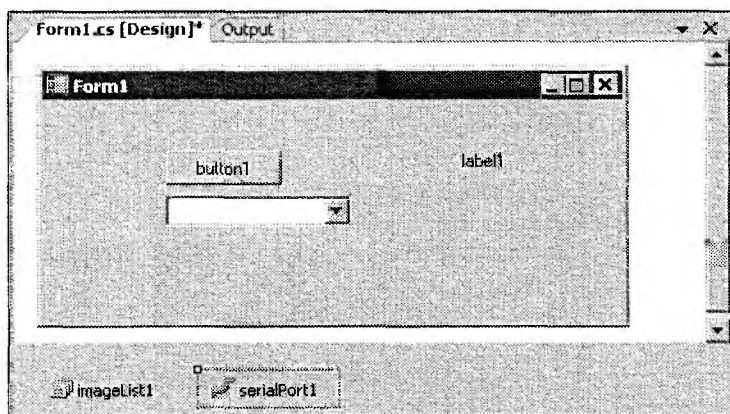


Рис. 2.6. Визуальные и невидимые компоненты

2.4. Палитра компонентов

Окно **Toolbox** (Инструменты) во время редактирования формы превращается в набор компонентов (рис. 2.7). Название окна, которое пришло из Visual Basic, при этом никак не меняется, что, мне кажется, не совсем удачно. Правильнее было бы назвать это окно "палитрой компонентов", как это было в Delphi. С другой стороны, сюда же вынесли содержимое буфера обмена², так что и на чистый набор компонентов это тоже не очень похоже. Давайте

¹ Для дотошного читателя могу сказать, что для таких компонентов создается специальный контейнер, имеющий тип `System.Windows.Forms.Design.ComponentTray`.

² К нашему обсуждению это не относится, но все-таки замечу, что окно **Toolbox** (Инструменты) в Visual Studio можно использовать как многопозиционный буфер обмена. Попробуйте перетащить текст из редактора кода на это окно и обратно.

договоримся, что содержимое окна **Toolbox** (Инструменты), когда мы находимся в редакторе форм и видим набор компонентов, мы будем называть *палитрой компонентов*. Так будет более понятно.

Компоненты в палитре компонентов объединяются в *категории* (categories), которые позволяют быстро найти нужные элементы. Например, категория **Menus&Toolbars** объединяет компоненты для построения меню и панелей инструментов, а категория **Data** — компоненты для работы с данными.

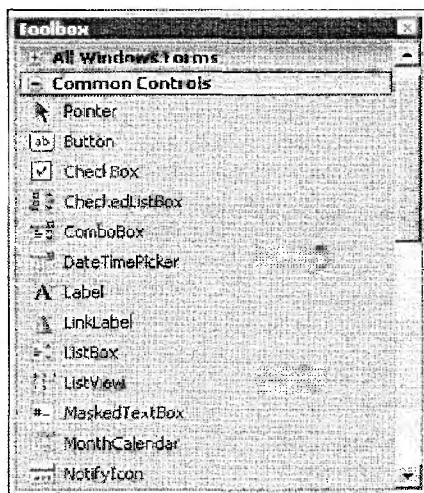


Рис. 2.7. Окно Toolbox

2.5. Окно свойств

Меню **Properties** (Свойства) открывает окно свойств выбранного в текущий момент объекта (рис. 2.8), который можно изменить, выбрав нужный объект в верхнем выпадающем списке. Стандартное окно свойств содержит две закладки¹, переключаемые кнопками  и . Первая кнопка переключает на закладку свойств объекта. Здесь собраны все свойства, которые позволяют посмотреть и изменить выбранный объект. Вторая кнопка переключает на закладку **Events** (События), которая содержит список событий, доступных для выбранного объекта.

Именно свойства и события являются наиболее важными интерфейсами взаимодействия. Чем удачнее подобран набор свойств, тем больше вероят-

¹ В разд. 12.3 я расскажу как добавлять собственные закладки.

ность, что конечные пользователи (такие же программисты, как и мы) останутся довольны. С этой точки зрения лучше сделать больше свойств, чем меньше. Но, конечно, переусердствовать тоже не стоит, иначе пользователь компонента просто запутается. Хорошим подходом также является разработка собственного редактора свойств, позволяющего более удобно (чем обычные редакторы) редактировать все нужные свойства. Еще один вариант — объединение свойств в сложные свойства, состоящие из других свойств. Например, сложное свойство `Font` состоит из нескольких простых свойств. При этом редактировать можно как само свойство `Font`, так и каждую из его составляющих.



Рис. 2.8. Окно Properties

2.6. Редакторы свойств

Каждое свойство, в зависимости от его типа, имеет свой *редактор свойств*. Среда VS предоставляет редакторы для всех стандартных свойств, которые показаны на рис. 2.9—2.15:

- простой редактор текстовых и числовых полей позволяет изменять все свойства с типом `string`, `int` и т. п.;
- редактор выбора из списка позволяет модифицировать перечисления (`enum`), а также свойства типа `bool` (выбор из списка `true/false`);
- редактор цветов позволяет редактировать свойства типа `Color`;

- ❑ редактор шрифтов позволяет изменять шрифт с помощью специального диалога или каждую из составляющих;
- ❑ редакторы свойств TextAlign и Dock показывают специальные диалоговые окна.

В дальнейшем мы научимся создавать собственные редакторы свойств.

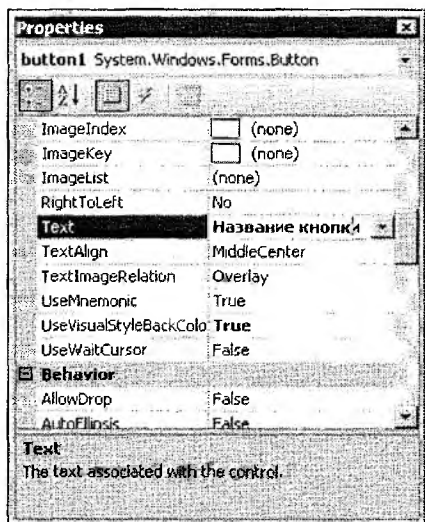


Рис. 2.9. Редактор простого (текстового или числового) поля

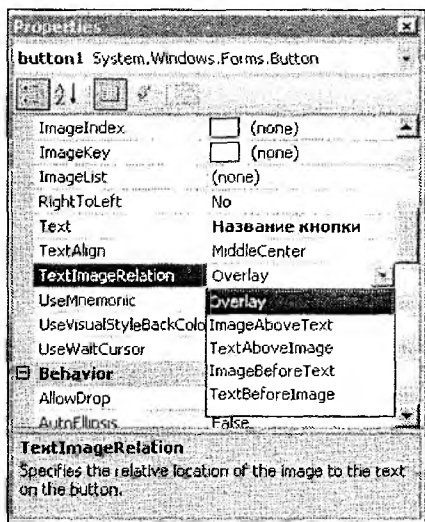


Рис. 2.10. Редактор с помощью выбора из списка



Рис. 2.11. Редактор цветов

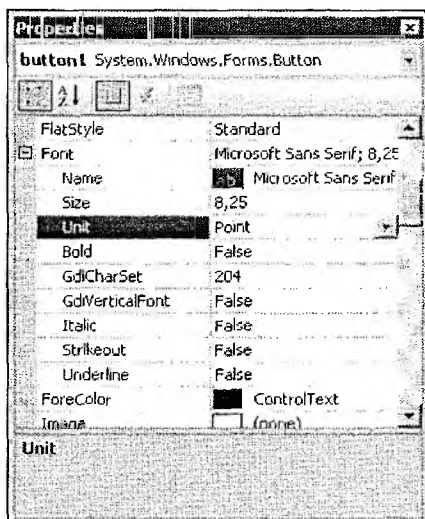


Рис. 2.12. Редактор шрифта (по частям)

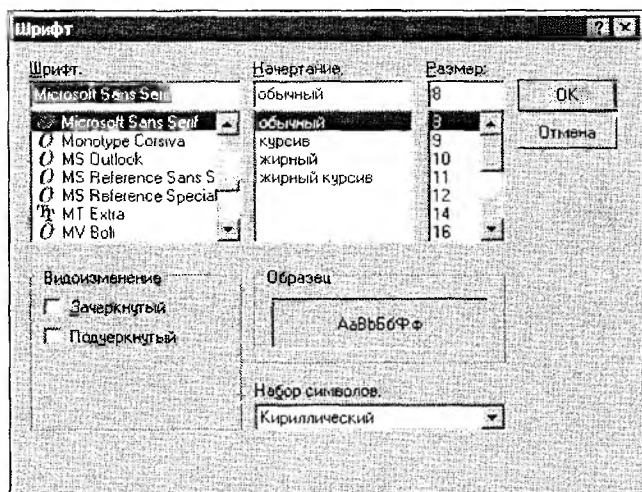
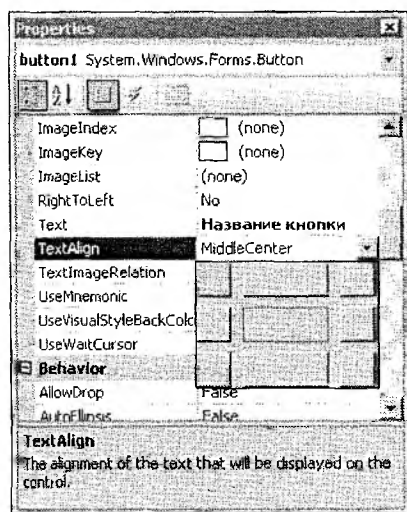
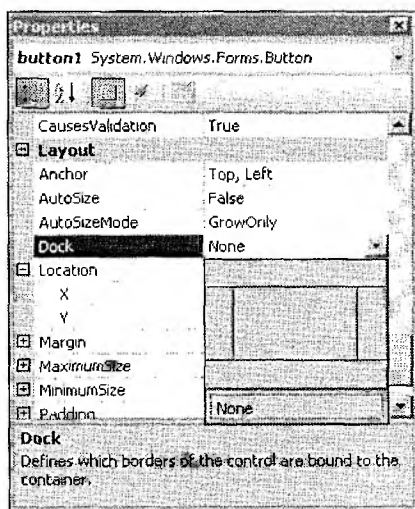
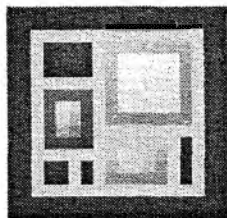


Рис. 2.13. Редактор шрифта

Рис. 2.14. Редактор выравнивания
(свойство TextAlign)Рис. 2.15. Редактор привязки
(свойство Dock)

ГЛАВА 3



Немного теории

Сложные проблемы всегда имеют простые, легкие для понимания неправильные решения.

Эта глава содержит тот минимум теории, который будет необходим для дальнейшего рассказа. Начинаящим я рекомендую обязательно прочитать эту главу. Тем, кто знаком с синтаксисом языка, но не сталкивался с методами отражения и метайнформацией, следует начать с *разд. 3.3*. Если же вы хорошо знакомы с языком C#, можете смело пропустить эту главу и читать дальше.

3.1. Свойства и события

Любой компонент взаимодействует с внешним миром с помощью свойств и событий. *Свойства* (properties) позволяют настраивать и получать параметры компонента, а *события* (events) — реагировать на происходящие внутри компонента события, программировать реакцию компонента на внешние события и т. д.

С точки зрения кода, события — сокращенная запись методов. Рассмотрим, например, такой код:

```
class Test
{
    private int testInt;

    public int TestInt
    {
        get { return testInt; }
```

```
        set { testInt = value; }  
    }  
}
```

Этот фрагмент можно было бы написать так:

```
class Test  
{  
    private int testInt;  
  
    public int GetTestInt()  
    {  
        return testInt;  
    }  
  
    public void SetTestInt(int value)  
    {  
        testInt = value;  
    }  
}
```

Таким образом, запись в виде свойства уменьшает количество кода, оставляя основное преимущество свойств — возможность дополнительных действий при получении или записи значения поля. Действительно, доступ к обычному полю не дает никакой возможности, например, проверить значение на правильность. Сравните два фрагмента кода:

// Доступ к полю

```
class Test  
{  
    public int testInt;  
}  
int v = testInt;
```

// Доступ к свойству

```
class Test  
{  
    private int testInt;  
  
    public int TestInt  
    {  
        get { return testInt; }  
        set {  
            if (testInt < 0)  
                throw new ArgumentOutOfRangeException("TestInt");  
            testInt = value;  
        }  
    }  
}
```

```
        testInt = value;
    }
}
}
```

Дополнительная проверка во время установки значения свойства позволяет отсеять некорректные значения. Аналогичный код для поля нужно было бы писать во всех местах, где производится установка значения этого поля.

Методы `get` и `set`, которые, по сути, являются оболочкой поля, называют *аксессорами* (accessor). Еще одно преимущество свойств — возможность задавать разный уровень доступа аксессоров:

```
class Test
{
    private int testInt;

    public int TestInt
    {
        get { return testInt; }
        protected set { testInt = value; }
    }
}
```

Здесь, аксессор `get` получает уровень видимости `public`, тогда как `set` ограничивается уровнем `protected`. Извне такое свойство будет выглядеть как свойство "только для чтения".

Кажется, информации о свойствах у нас вполне достаточно, перейдем к событиям.

Модель *событий* (event) в .NET основана на *делегатах* (delegate). Делегаты позволяют обращаться к *методам обратного вызова* (callback method). Метод обратного вызова — это механизм, позволяющий объекту получать уведомления, на которые он подписался.

Описание события производится с помощью специального слова `event`:

```
public class TestClass
{
    private event EventHandler OnChange;
    public event EventHandler OnChange
    {
        add { onChange += value; }
        remove { onChange -= value; }
    }
}
```


По виду записи события похожи на свойства. Метод `add` позволяет подписать объект на событие, а метод `remove` — удалить подписку.

Тот же самый код будет сгенерирован, если описать событие с модификатором `public`:

```
public class TestClass
{
    public event EventHandler OnChange;
}
```

Методы `add`, `remove` и закрытое поле делегата будут сгенерированы компилятором автоматически.

Когда компоненту нужно вызвать событие, используется следующий код:

```
if (onChange != null)
    onChange(this, new EventArgs());
```

Проверка на `null` позволяет обработать ситуацию, когда обработчик события не задан.

Класс `EventHandler` описывает тип события. Его определение выглядит так:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Можно сказать, что по умолчанию (т. е. при использовании типа `EventHandler`) у события будут два аргумента. Параметр `sender` определяет "владельца" события, и поэтому при вызове события обычно передается значение `this`. Вторым параметром представляет собой пустой аргумент, описываемый с помощью класса `EventArgs`:

```
public class EventArgs
{
    public static readonly EventArgs Empty;
    public EventArgs();
}
```

Конечно, параметры события не всегда столь примитивны. При необходимости передать больше информации в параметрах нужно создать собственный наследник класса `EventArgs` и описать соответствующий тип параметра:

```
// Описание аргумента с параметром Name
public class ChangeArgs : EventArgs
{
    private readonly string name;
```

```
public ChangeArgs(string name)
{
    this.name = name;
}

public string Name
{
    get { return name; }
}
}

// Описание события
public event EventHandler<ChangeArgs> OnChange;
// Вызов события
if (OnChange != null)
    OnChange(this, new ChangeArgs(this.Name));
```

Сам обработчик события будет выглядеть, например, так:

```
private void gradientLabel1_OnChange(object sender,
                                     MyControl.ChangeArgs e)
{
    // Обработка
}
```

Реализация методов `add` и `remove` самостоятельно нужна, например, если необходимо создать потокобезопасный код (как я уже говорил, по умолчанию можно оставить работу по генерации этих методов компилятору):

```
// Описание объекта синхронизации
private readonly object eventLock = new object();
// Закрытый делегат
private EventHandler<ChangeArgs> onChange;

// Описание события (методы add и remove)
public event EventHandler<ChangeArgs> OnChange
{
    add
    {
        lock (eventLock)
        {
            onChange += value;
        }
    }
}
```

```
remove
{
    lock (eventLock)
    {
        onChange -= value;
    }
}

// Вызов события
if (onChange != null)
    onChange(this, new ChangeArgs(this.Name));
```

Это все, что я хотел рассказать о свойствах и событиях. Конечно, я понимаю, что это очень мало, но, согласитесь, что эту информацию вполне можно найти во многих источниках, например в MSDN или в [5].

3.2. Интерфейсы

Интерфейс — это именованный набор сигнатур методов (см. [5]). Интерфейсы могут определять события и свойства, но не могут определять методов-конструкторов и экземплярных полей.

В С# для определения интерфейса используется ключевое слово `interface`, например:

```
public interface IDisposable
{
    void Dispose();
}

public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Любой класс, наследующий (правильнее сказать, реализующий) интерфейс должен реализовать все сигнатуры, описанные в этом интерфейсе:

```
public class Test : IDisposable
{
    public void Dispose() { Console.WriteLine("Вызов Dispose"); }
}
```

В С# класс может реализовывать несколько интерфейсов, а также интерфейс может наследовать другие интерфейсы.

В этой книге интерфейсы будут интересовать нас только потому, что доступ ко многим методам времени разработки предоставляется только через интерфейсы соответствующих сервисов (см. главу 7). Кроме того, мы будем использовать несколько важных интерфейсов, таких как `IDisposable`, `ICollection`, `IEnumerable` и т. д. Подробнее узнать о них можно в справке MSDN или в [5].

3.3. Метаинформация, отражение и атрибуты

Взаимодействие частей программ между собой подразумевает наличие информации о предоставляемых ими возможностях. Например, к библиотеке DLL прилагается заголовочный файл, описывающий экспортируемые данные, процедуры и структуру типов. Для COM-компонента описание хранится в `idl`-файле или в виде библиотеки типов.

Информация, описывающая компоненты модуля, называется *метаинформацией*. Приставка "мета" подчеркивает, что это "данные о данных", т. е. данные, описывающие другие данные.

Частично технология хранения метаинформации была реализована во многих языках программирования. Например, в Borland Delphi и C++ с помощью методов RTTI (Run-Time Type Identification) можно было получать информацию о типе данных во время выполнения кода. Но в .NET метаинформация играет не дополнительную, а фактически основную роль. Метаинформация в .NET хранит полную и исчерпывающую информацию обо всех типах, методах, параметрах и возвращаемых значениях, независимо от их области видимости. Другими словами, метаинформация содержит описания всех типов и методов — и `public` и `private` и `protected`... Более того, всю эту информацию можно получить программным путем, и она доступна для расширения.

Получение информации о типе во время выполнения называется *отражением* (reflection). Классы и методы, предоставляющие доступ к методам отражения, находятся в пространстве имен `System.Reflection`.

Например, с помощью метода `GetTypes()` можно получить список всех типов, находящихся в сборке:

```
using System;

namespace Reflection2
{
    class Class1
```

```

{
    [STAThread]
    static void Main(string[] args)
    {
        // Загрузить модуль
        System.Reflection.Assembly assembly =
            System.Reflection.Assembly.LoadFrom("Reflection1.exe");
        // Получить список всех классов модуля
        Type[] types = assembly.GetTypes();
        // Отобразить список классов модуля
        foreach (Type t in types)
        {
            Console.WriteLine(t.FullName);
        }
        // Создание объекта по имени типа
        object Obj = Activator.CreateInstance(
            "Reflection1", "Reflection1.TestClass");
    }
}

```

Обратите внимание на последние две строки: метод `CreateInstance()` позволяет создать объект по имени типа!

Каждый управляемый тип в .NET реализует интерфейс `IReflect`, с помощью которого возможен доступ к полям, свойствам, методам и другой информации:

```

// Получение списка свойств
System.Reflection.PropertyInfo[] Props =
    listBox1.GetType().GetProperties();
foreach (System.Reflection.PropertyInfo prop in Props)
{
    listBox1.Items.Add(prop.PropertyType.ToString() + ":" + prop.Name);
}
// Получение скрытых полей структуры
foreach (FieldInfo fi in
    typeof(Account).GetFields(BindingFlags.NonPublic |
        BindingFlags.Instance))
{
    Console.WriteLine(fi.Name);
}

```

Причем, с помощью метаданных можно не только получать, но и устанавливать значения полей и свойств (см. листинг 3.1). И не только открытых (`public`), но и скрытых (`private` и `protected`).

Информацию о конструкторе класса можно получить с помощью метода `GetConstructor()`:

```
ConstructorInfo ci =  
    typeof(LocationClass).GetConstructor(new Type[] { typeof(Point) });
```

Как я уже говорил, метаянформация может быть расширена собственными данными с помощью *атрибутов* (attribute). Атрибут привязывается к объекту с помощью описания в квадратных скобках¹:

```
[CategoryAttribute("Gradient")]  
public Color StartColor  
{  
    get { return startColor; }  
    set { startColor = value; }  
}
```

По соглашению все имена атрибутов заканчиваются словом `Attribute`, но для простоты, язык `C#` позволяет сокращать запись и указывать "чистое" имя:

```
[Category("Gradient")]  
public Color StartColor
```

В круглых скобках указываются через запятую параметры атрибута, которые могут быть позиционными или именованными. Позиционные параметры указываются в определенной последовательности, а именованные описываются в виде равенства:

свойство = значение

Например,

```
[DllImport("kernel32.dll", SetLastError= true, CharSet=CharSet.Auto)]
```

Позиционные параметры указываются в конструкторе, а именованные описываются как свойства, доступные как для чтения, так и для записи. Свойства, доступные только для чтения, не могут быть именованными.

С точки зрения кода, атрибуты — наследники класса `System.Attribute`:

```
public class MyAttribute : Attribute  
{  
}
```

Класс атрибута может иметь атрибут `AttributeUsageAttribute`, который позволяет задать область применения атрибута, возможность наследования и возможность указывать несколько атрибутов одновременно:

¹ Кроме того, существуют атрибуты для сборок (ключевое слово `Assembly`) и для модулей (ключевое слово `Module`), но в этой книге они нам не потребуются.

```
AttributeUsage(AttributeTargets ValidOn,
               bool Inherited,
               bool AllowMultiple)
```

Здесь:

- свойство `ValidOn` имеет тип `AttributeTargets` и задает область применения атрибута. Например, `AttributeTargets.Class`, `AttributeTargets.Method`, `AttributeTargets.Property` и т. д.;
- свойство `Inherited` указывает, будет ли атрибут наследоваться классами, наследниками того, к которому этот атрибут был применен. По умолчанию значение равно `true`;
- параметр `AllowMultiple` указывает, что атрибут может быть привязан к одному объекту несколько раз. Значение по умолчанию равно `false`.

Например,

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple= true)]
public class NewAttribute : Attribute
{
}
```

Аналогично, если класс атрибута имеет собственные свойства, они будут указываться при привязке атрибута к объекту. Пусть, например, описание атрибута выглядит следующим образом:

```
[AttributeUsage(AttributeTargets.All)]
public class TypedCollectionAttribute : Attribute
{
    private Type collectionType;
    private string fieldName;

    public TypedCollectionAttribute(Type collectionType)
    {
        this.collectionType = collectionType;
    }

    public Type CollectionType
    {
        get { return collectionType; }
    }

    public string FieldName
    {
        get { return fieldName; }
    }
}
```

```
    set { fieldName = value; }  
  }  
}
```

В этом случае привязка атрибута может выглядеть так:

```
[TypedCollection(typeof(MyData), FieldName="Data")]  
public class MyCollection : ArrayList, IBindingList, IListedList  
{  
}
```

Добавление новых атрибутов расширяет метаинформацию, позволяя иметь собственные данные в таблицах метаинформации. Разумеется, с помощью методов отражения можно получать список атрибутов и проверять наличие привязки атрибута к объекту.

Получить список атрибутов типа можно с помощью двух методов:

```
object[] GetCustomAttributes(bool inherit);  
object[] GetCustomAttributes(Type attributeType, bool inherit);
```

Первый метод возвращает все атрибуты, а второй — только атрибуты указанного типа. Параметр `inherit` указывает, нужно ли просматривать атрибуты в классах-предках.

Проверить, что к типу привязан некий атрибут, можно с помощью метода:

```
bool IsDefined(Type attributeType, bool inherit);
```

Сигнатура этого метода та же, что и у метода `GetCustomAttributes()`.

Кроме этого, можно воспользоваться одним из статических методов класса `Attribute`.

Например, вот несколько способов перебора атрибутов:

```
foreach (Attribute a in this.GetType().GetCustomAttributes(  
    typeof(TypedCollectionAttribute), false))  
{  
}  
// Получаем атрибут (если он определен один раз)  
DeveloperAttribute MyAttribute = (DeveloperAttribute)  
    Attribute.GetCustomAttribute(t, typeof (DeveloperAttribute));  
// Если допускается несколько одинаковых атрибутов, то  
// нужно использовать другой способ  
DeveloperAttribute[] MyAttribute = (DeveloperAttribute[])  
    Attribute.GetCustomAttributes(t, typeof (DeveloperAttribute));
```

Атрибуты — одна из важнейших составляющих кода времени разработки. С их помощью определяется большинство параметров свойств и классов. Более подробную информацию об атрибутах можно найти в MSDN или в [5].

Листинг 3.1. Установка значения private- и protected-полей

```
using System;
using System.Reflection;

namespace Reflection3
{
    class Class1
    {
        public class Account
        {
            private    long privatevalue = 0;
            protected  long protectedval = 0;

            public long GetPrivatevalue()
            {
                return privatevalue;
            }

            public long GetProtectedval()
            {
                return protectedval;
            }
        }

        [STAThread]
        static void Main(string[] args)
        {
            Account a = new Account();
            Console.WriteLine("a.privatevalue={0}", a.GetPrivatevalue());
            Console.WriteLine("a.protectedval={0}", a.GetProtectedval());

            FieldInfo fil = a.GetType().GetField("privatevalue",
                BindingFlags.NonPublic | BindingFlags.Instance);
            if (fil != null)
            {
                fil.SetValue(a, 1);
            }

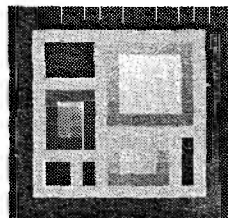
            FieldInfo fi2 = a.GetType().GetField("protectedval",
                BindingFlags.NonPublic | BindingFlags.Instance);
            if (fi2 != null)
            {

```

```
        fi2.SetValue(a, 2);
    }

    Console.WriteLine("a.privatevalue={0}", a.GetPrivatevalue());
    Console.WriteLine("a.protectedval={0}", a.GetProtectedval());
}
}
```

ГЛАВА 4



Начнем с нуля

Тиние едешь, дальше будешь... От того места, куда едешь.

В этой главе я расскажу об общей архитектуре времени разработки. Как компоненты взаимодействуют с формой и друг с другом? Как правильно выбрать базовый класс для своего компонента? Как определить, что компонент находится в режиме разработки? Итак... Поехали!

4.1. Общая архитектура

Компоненты имеют две "стороны жизни" — во время разработки формы в Visual Studio (design time) и во время выполнения готовой программы (run-time). Конечно, в рамках этой книги нас будет интересовать первый режим.

На рис. 4.1 показана архитектура времени разработки. Еще раз повторю несколько основных положений компонентной модели:

- ☐ управление поведением и видом компонентов производится с помощью свойств;
- ☐ каждое свойство компонента имеет определенный тип;
- ☐ для взаимодействия с "внешним миром" компонент имеет события;
- ☐ редактирование свойств и событий производится с помощью редактора свойств.

Понятно, что редактор свойств не может уметь редактировать свойства произвольного типа. Он должен иметь некоторую информацию о типе свойства, о возможных значениях, о значении по умолчанию и т. д. Эта информация привязывается к свойству или типу с помощью атрибутов (см. разд. 3.3).

Кроме того, также с помощью атрибутов к типу или свойству привязывается дизайнер (designer), конвертер типа (type converter) и редактор типа (UI type editor). *Дизайнер* позволяет управлять набором свойств компонента и его поведением в режиме разработки, создавать пользовательские меню, теги и многое другое (см. главу 6). *Конвертер типа* обеспечивает преобразование пользовательского типа в тип, который понятен редактору свойств (см. главу 8). *Редактор типа* позволяет отображать пользовательские диалоги для редактирования свойств компонента (см. главу 9).

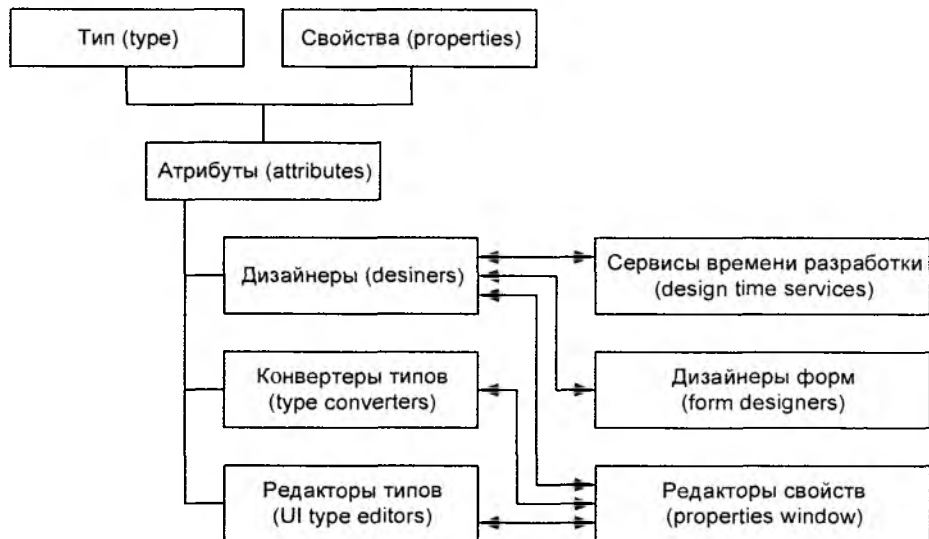


Рис. 4.1. Архитектура времени разработки

Дизайнеры могут использовать *сервисы времени разработки* (design time services), предоставляемые средой Visual Studio (см. главу 7), которые предоставляют компоненту информацию о происходящих событиях, изменениях других свойств и компонентов и т. д. Кроме того, дизайнеры, конвертеры типов и редакторы типов могут использовать *редакторы свойств* (properties window), а дизайнеры также могут использовать *дизайнеры форм* (form designer). Замечу, что конвертеры типов и редакторы типов могут применяться как в режиме разработки, так и во время выполнения, а дизайнеры работают только во время разработки.

Каждую из этих сущностей мы рассмотрим в соответствующих главах книги.

4.2. Класс *Component*

Что же такое компонент? С точки зрения кода, компонент — это класс, реализующий интерфейс `System.ComponentModel.IComponent`. "Минимальный" базовый класс, реализующий этот интерфейс, — это `System.ComponentModel.Component`, т. е. он является базовым классом для всех компонентов, поддерживающих режим разработки. Этот класс реализует три интерфейса:

- `MarshalByRefObject` — реализует поддержку .NET Remoting;
- `IComponent` — реализует функциональность, требуемую режимом разработки;
- `IDisposable` — реализует возможность управления занятыми ресурсами при уничтожении объекта.

Конечно, можно реализовывать интерфейс `IComponent` и без использования класса `Component`, но это может оказаться достаточно затруднительно.

Перечислю основные свойства класса `Component`, которые могут нам понадобиться в дальнейшей работе.

- Свойство `protected virtual bool CanRaiseEvents` возвращает `true`, если компонент будет вызывать обработчики событий. По умолчанию это свойство всегда возвращает `true`. Если компонент не будет вызывать обработчики событий, то свойство `Events` возвращает `null`.
- Свойство `protected bool DesignMode { get; }` — см. следующий раздел. Это очень важное свойство, и мы будем обсуждать его отдельно.
- Свойство `protected EventHandlerList Events { get; }` возвращает список делегатов, присоединенных к компоненту. Если `CanRaiseEvents` вернет `false`, то это свойство вернет `null`.

В режиме разработки у компонента появляются еще три свойства (рис. 4.2)¹:

- свойство `GenerateMember` отвечает за создание переменной, соответствующей компоненту;
- свойство `Locked` управляет возможностью перемещать компонент на форме и изменять его размеры;
- свойство `Modifiers` задает модификатор переменной, которая будет создана, если свойство `GenerateMember` имеет значение `true`.

Если свойство `GenerateMember` равно `true`, дизайнер формы создает переменную, соответствующую компоненту с модификатором, заданным свойством `Modifiers`:

¹ Что означает "появляются в режиме разработки", я расскажу в разд. 6.7.

```

partial class Form1
{
    private System.Windows.Forms.Button button1;

    private void InitializeComponent()
    {
        this.button1 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        this.button1.Location = new System.Drawing.Point(89, 36);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(75, 23);
        this.button1.TabIndex = 0;
        this.button1.Text = "button1";
        this.button1.UseVisualStyleBackColor = true;
        this.button1.Click += new System.EventHandler(this.button1_Click);
    }
}

```

Если же свойство `GenerateMember` имеет значение `false`, то инициализация компонента производится с помощью локальной переменной:

```

partial class Form1
{
    private void InitializeComponent()
    {
        System.Windows.Forms.Button button1;
        button1 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        button1.Location = new System.Drawing.Point(89, 36);
        button1.Name = "button1";
        button1.Size = new System.Drawing.Size(75, 23);
        button1.TabIndex = 0;
        button1.Text = "button1";
        button1.UseVisualStyleBackColor = true;
        button1.Click += new System.EventHandler(this.button1_Click);
    }
}

```

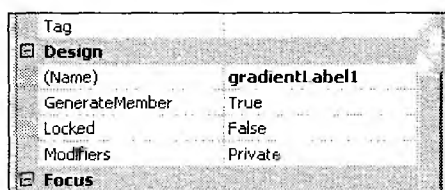


Рис. 4.2. Свойства времени разработки (группа Design)

На рис. 4.3 показана возможность управления редактированием с помощью свойства `Locked`. Верхняя форма соответствует режиму "разрешено" (свойство `Locked` равно `false`), нижняя форма — режиму "запрещено" (свойство `Locked` равно `true`). В первом случае компонент имеет маркеры управления размером и положением компонента, а во втором — нет¹.

С остальными свойствами класса `Component` мы будем разбираться по мере необходимости.

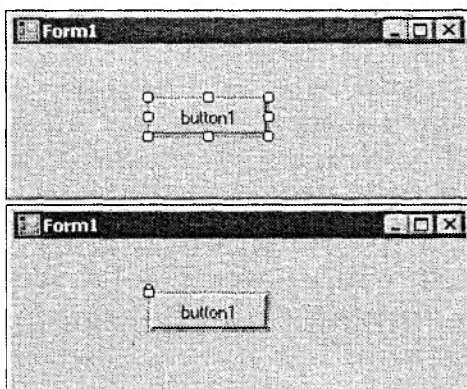


Рис. 4.3. Управление возможностью редактирования с помощью свойства `Locked`

4.3. Контейнеры

Процесс разработки формы заключается в расположении компонентов на форме или на других компонентах. Другими словами, каждый компонент и сама форма (вообще говоря, форма тоже является компонентом) являются *контейнерами* (`container`), которые могут содержать один или несколько компонентов. Для реализации такой возможности предусмотрен интерфейс `System.ComponentModel.IContainer`:

```
public interface IContainer : IDisposable
{
    ComponentCollection Components {get;}
    void Add(IComponent component);
    void Add(IComponent component, string name);
    void Remove(IComponent component);
}
```

¹ Про маркеры и управление ими можно прочитать в *разд. 6.14*.

Свойство `Components` возвращает коллекцию компонентов, которые содержит контейнер. Методы `Add()` позволяют добавить компонент в конец списка компонентов, а метод `Remove()` — удалить указанный компонент из контейнера. Обратите внимание, что методов `Add()` два. Первый добавляет безымянный компонент, а второй — компонент с известным именем¹. К компонентам, содержащимся в коллекции `Components`, можно обращаться как по имени, так и по индексу:

```
Component c1 = container.Components["t1"] as Component;
Component c2 = container.Components[0] as Component;
```

Замечу, что имя компонента должно быть уникально в рамках контейнера. Кроме того, в имени не различаются большие и маленькие буквы, т. е. имена `t1` и `T1` будут считаться одинаковыми. При попытке добавить два компонента с одинаковым именем генерируется исключение `System.ArgumentException`. Для проверки уникальности имени контейнер вызывает виртуальный метод `protected virtual void ValidateName(IComponent component, string name)`

Базовым классом, реализующим интерфейс `IContainer`, является класс `System.ComponentModel.Container`.

Каждый контейнер сам заботится об освобождении всех содержащихся в нем компонентов. Посмотрите на листинг 4.1, результатом его работы будут следующие строки:

```
Метод Dispose: number=2
Метод Dispose: number=1
```

Метод `Dispose()` компонентов, добавленных в контейнер, был вызван автоматически при освобождении самого контейнера.

Понятно, что компонент сам по себе имени не имеет, оно появляется только при добавлении компонента в контейнер, например, при помещении компонента на форму. Для связи контейнера и компонента служит интерфейс `ISite`:

```
public interface ISite : IServiceProvider
{
    // Ссылка на компонент
    IComponent Component { get; }
    // Ссылка на контейнер
    IContainer Container { get; }
    // True, если режим разработки; false, если режим выполнения
    bool DesignMode { get; }
```

¹ На полях замечу, что уникальное имя можно сгенерировать с помощью сервиса `INameCreationService`, с которым мы будем разбираться в разд. 7.8.


```
// Имя компонента или null, если компонент не имеет имени
string Name { get; set; }
}
```

В самом компоненте этот интерфейс можно получить с помощью виртуального свойства `Site`:

```
public class Component : MarshalByRefObject, IComponent, IDisposable
{
    public virtual ISite Site { get; set; }
}
```

Для компонентов, содержащих другие компоненты (не обязательно визуально), имеется класс `NestedContainer`, реализующий интерфейс `INestedContainer`:

```
public interface INestedContainer : IContainer, IDisposable
{
    IComponent Owner { get; }
}

public class NestedContainer :
    Container, INestedContainer, IContainer, IDisposable
{
    // Конструктор
    public NestedContainer(IComponent owner);

    // Возвращает компонент-владелец
    public IComponent Owner { get; }

    // Возвращает имя компонента-владельца
    protected virtual string OwnerName { get; }
}
```

Соответственно, такой контейнер создается "на базе" некоторого существующего компонента:

```
TestComponent owner = new TestComponent(0);
using (NestedContainer container = new NestedContainer(owner))
{
    // Добавляем компоненты в контейнер
}
```

В остальном класс `NestedContainer` совпадает с обычным классом контейнера `Container`.

Листинг 4.1. Контейнер заботится об освобождении ресурсов

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;

namespace Container
{
    class Program
    {
        static void Main(string[] args)
        {
            // Создаем контейнер
            using (Container container = new Container())
            {
                // Добавляем компонент t1
                TestComponent t1 = new TestComponent(1);
                container.Add(t1, "t1");
                // Добавляем компонент t2
                TestComponent t2 = new TestComponent(2);
                container.Add(t2, "t2");
            }
            // Контейнер будет освобожден при выходе из блока using.
            // Дополнительное освобождение t1 и t2 не требуется
        }
    }

    class TestComponent : Component
    {
        private int number;

        public TestComponent(int number)
        {
            this.number = number;
        }

        protected override void Dispose(bool disposing)
        {
            Console.WriteLine("Метод Dispose: number={0}", number);
            base.Dispose(disposing);
        }
    }
}
```

4.4. Хосты

Управление режимом разработки, созданием и уничтожением компонентов формы, загрузкой формы и т.д. осуществляется с помощью интерфейса `System.ComponentModel.Design.IDesignerHost`, который называется *хостом* (host):

```
public interface IDesignerHost : IServiceContainer, IServiceProvider
{
    IContainer Container { get; }
    IComponent RootComponent { get; }
    string RootComponentClassName { get; }
    IComponent CreateComponent(Type componentClass);
    IComponent CreateComponent(Type componentClass, string name);
    void DestroyComponent(IComponent component);
    IDesigner GetDesigner(IComponent component);
    Type GetType(string typeName);

    bool Loading { get; }
    void Activate();
    event EventHandler Activated;
    event EventHandler Deactivated;
    event EventHandler LoadComplete;

    bool InTransaction { get; }
    string TransactionDescription { get; }
    event DesignerTransactionCloseEventHandler TransactionClosed;
    event DesignerTransactionCloseEventHandler TransactionClosing;
    event EventHandler TransactionOpened;
    event EventHandler TransactionOpening;
    DesignerTransaction CreateTransaction();
    DesignerTransaction CreateTransaction(string description);
}
```

Для получения этого интерфейса нужно вызвать метод `GetService()`, передав ему в качестве параметра тип `IDesignerHost` (с методом `GetService()` мы еще будем разбираться при обсуждении сервисов в главе 7).

Я разделил методы и свойства этого интерфейса на три группы. Первая группа относится к управлению компонентами:

- свойство `Container` возвращает контейнер хоста;
- свойство `RootComponent` возвращает "базовый" компонент дизайнера. Обычно это компонент типа `Form` или `UserControl`;

- ❑ свойство `RootComponentClassName` возвращает полное имя базового компонента;
- ❑ методы `CreateComponent()` создают компонент заданного типа и добавляют его в дизайнер;
- ❑ метод `DestroyComponent()` удаляет компонент;
- ❑ метод `GetDesigner()` возвращает дизайнер указанного компонента (см. главу 6);
- ❑ метод `GetType()` возвращает тип по имени.

Вторая группа относится к состоянию дизайнера:

- ❑ свойство `Loading` возвращает `true`, если идет процесс загрузки данных в дизайнер;
- ❑ события `Activated`, `Deactivated` и `LoadComplete` вызываются соответственно при завершении активизации, при завершении деактивизации и после загрузки данных;
- ❑ метод `Activate()` активизирует дизайнер.

Третья группа относится к транзакциям дизайнера, с которыми мы будем разбираться в *разд. 12.2*.

4.5. Дескрипторы свойств

И свойство, и событие являются частями компонента, каждая из которых описывается с помощью класса, называемого *дескриптором* (descriptor). Базовым классом дескриптора является класс `MemberDescriptor`. Два наследника этого класса являются *дескриптором свойств* (класс `PropertyDescriptor`) и *дескриптором событий* (`EventDescriptor`).

Класс `PropertyDescriptor` является абстрактным и имеет следующие абстрактные свойства и методы:

- ❑ свойство `Type ComponentType { get; }` возвращает тип компонента;
- ❑ свойство `bool IsReadOnly { get; }` возвращает `true`, если свойство доступно только для чтения;
- ❑ свойство `Type PropertyType { get; }` возвращает тип свойства;
- ❑ метод `bool CanResetValue(object component);` возвращает `true`, если доступен метод сброса значения свойства в значение по умолчанию (метод `Reset`);
- ❑ метод `object GetValue(object component);` возвращает значение свойства для данного компонента;

- метод `void ResetValue(object component)`; сбрасывает значение свойства в значение по умолчанию;
- метод `void SetValue(object component, object value)`; устанавливает значение свойства;
- метод `bool ShouldSerializeValue(object component)`; возвращает `true`, если значение свойства должно сериализоваться (см. главу 10).

Метод `SetValue()` очень важен в режиме разработки. Дело в том, что среда `Visual Studio` и, в частности, дизайнер формы не могут отследить все присваивания значения свойств всех компонентов. Например, при изменении одного свойства в коде компонента меняется значение другого. Если присвоение делать напрямую, то дизайнер формы не имеет возможности узнать об этом, т. к. будет выполнен только код аксессуара `set` этого свойства. Правильный путь — использовать метод `SetValue()` дескриптора нужного свойства. Именно такое присвоение позволит среде разработки узнать об изменении свойства. Получить дескриптор можно с помощью метода:

```
privatePropertyDescriptor GetPropertyByName(
    Component component, string propertyName)
{
    PropertyDescriptor prop =
        TypeDescriptor.GetProperties(component)[propertyName];
    if (prop == null)
    {
        throw new ArgumentException("Свойство не существует", propertyName);
    }
    return prop;
}
```

Вызов метода будет выглядеть примерно так:

```
GetPropertyByName("StartColor").SetValue(glabel, value);
```

Теперь, когда все нужные службы будут оповещены об изменении значения свойства, присвоение значения свойства будет корректным, например, будет работать команда меню **Undo** (Отмена). Разумеется, в режиме выполнения такие сложности не нужны и вполне достаточно простого присвоения значения свойства.

Класс `PropertyDescriptor` является абстрактным и имеет следующие абстрактные свойства и методы:

- свойство `Type ComponentType { get; }` возвращает тип компонента;
- свойство `Type EventType { get; }` возвращает тип события;

- ❑ свойство `bool IsMulticast { get; }` возвращает `true`, если делегат-обработчик представляет собой цепочку делегатов;
- ❑ метод `void AddEventHandler(object component, Delegate value);` добавляет обработчик к событию;
- ❑ метод `void RemoveEventHandler(object component, Delegate value);` удаляет обработчик события.

Списки событий и свойств описываются с помощью классов `PropertyDescriptorCollection` и `EventDescriptorCollection`. Для их получения используются статические методы `GetProperties()` и `GetEvents()` класса `TypeDescriptor` (листинг 4.2).

Заметьте, что все три класса, которые я описал выше, являются абстрактными. Использовать их напрямую (например, для создания нового свойства) не получится. Вариантов два. Во-первых, в классе `TypeConverter` (см. главу 8) содержится внутренний класс `SimplePropertyDescriptor`, который позволяет частично решить эту проблему. Я буду рассказывать о нем в разд. 8.6. Во-вторых, можно попробовать реализовать собственный класс, являющийся наследником соответствующего базового класса. Например, в листинге 4.3 содержится пример дескриптора для SQL-типов свойств. Самое сложное здесь — методы `GetValue()` и `SetValue()`. Вначале нужно обработать значение `DBNull`, а затем с помощью методов отражения либо получить, либо установить новое значение свойства. Аналогично можно реализовать и другие варианты дескрипторов.

Листинг 4.2. Получение списка свойств и событий компонента

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Test
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

```

private void Form1_Load(object sender, EventArgs e)
{
    // Получаем коллекцию дескрипторов свойств
    PropertyDescriptorCollection pdc =
        TypeDescriptor.GetProperties(this);
    if (pdc != null && pdc.Count > 0)
    {
        Info.Items.Add("Список свойств:");
        // Выводим список свойств
        foreach (PropertyDescriptor pdi in pdc)
        {
            Info.Items.Add(string.Format("{0}", pdi.Name));
        }
    }

    // Получаем коллекцию дескрипторов событий
    EventDescriptorCollection edc = TypeDescriptor.GetEvents(this);
    if (edc != null && edc.Count > 0)
    {
        Info.Items.Add("Список событий:");
        // Выводим список событий
        foreach (EventDescriptor edi in edc)
        {
            Info.Items.Add(string.Format("{0}", edi.Name));
        }
    }
}

```

Листинг 4.3. Пример реализации дескриптора свойств

```

using System;
using System.Data.SqlTypes;
using System.ComponentModel;
using System.Collections;
using System.Reflection;
using System.Diagnostics;

public class SqlPropertyDescriptor : PropertyDescriptor
{
    private Type SqlType;
    private Type BaseType;

```

```
/// Конструктор
public static SqlPropertyDescriptor GetProperty(
    string name, Type sqlType)
{
    // Чтобы вызвать конструктор базового класса, нужен
    // список атрибутов свойства
    Type baseType = sqlType.GetProperty("Value").PropertyType;
    ArrayList attribs =
        new ArrayList(TypeDescriptor.GetAttributes(baseType));
    Attribute[] attrs =
        (Attribute[])attribs.ToArray(typeof(Attribute));
    // Вызываем конструктор базового класса
    return new SqlPropertyDescriptor(name, attrs, sqlType, baseType);
}

// Эти конструкторы нам не нужны
protected SqlPropertyDescriptor(
    MemberDescriptor descr) : base(descr) { }
protected SqlPropertyDescriptor(MemberDescriptor descr,
    Attribute[] attrs) : base(descr, attrs) { }
protected SqlPropertyDescriptor(string name,
    Attribute[] attrs) : base(name, attrs) { }

// Мы используем этот конструктор
protected SqlPropertyDescriptor(string name,
    Attribute[] attrs, Type sqlType, Type baseType)
    : base(name, attrs)
{
    SqlType = sqlType;
    BaseType = baseType;
}

public override bool CanResetValue(object component)
{
    return false;
}

public override void ResetValue(object component)
{
    throw new NotSupportedException();
}
```



```
public override bool ShouldSerializeValue(object component)
{
    return false;
}

public override Type ComponentType
{
    get { return BaseType; }
}

public override bool IsReadOnly
{
    get { return false; }
}

public override Type PropertyType
{
    get { return BaseType; }
}

public override void SetValue(object component, object value)
{
    try
    {
        // Получаем информацию о свойстве
        PropertyInfo pi = component.GetType().GetProperty(this.Name);
        Object o;
        if (value == DBNull.Value)
        {
            // Если это DBNull, возвращаем статическое свойство Null
            o = component.GetType().GetField("Null",
                BindingFlags.Static | BindingFlags.Public |
                BindingFlags.GetField).GetValue(component);
        }
        else
        {
            // Создаем экземпляр значения свойства
            o = pi.PropertyType.GetConstructor(
                new Type[] { BaseType }).Invoke(new Object[] { value });
        }
        // Устанавливаем значение свойства
        pi.SetValue(component, o, null);
    }
}
```

```
        catch (Exception ex)
        {
            // Какая-нибудь обработка исключения
            Debug.WriteLine(ex);
        }
    }

    public override object GetValue(object component)
    {
        try
        {
            object Property = component.GetType().GetProperty(
                this.Name).GetValue(component, null);

            // Если это Null, то возвращаем DBNull
            if ((bool)Property.GetType().GetProperty("IsNull").
                GetValue(Property, null))
                return DBNull.Value;

            // Возвращаем значение свойства
            object Value = Property.GetType().GetProperty("Value").
                GetValue(Property, null);
            return Value;
        }
        catch (Exception ex)
        {
            // Какая-нибудь обработка исключения
            Debug.WriteLine(ex);
        }
        return null;
    }
}
```

4.6. Определение режима разработки

У класса `Component` есть одно важное свойство, про которое я обещал рассказать. Это свойство `DesignMode`. Оно возвращает `true`, если компонент находится в режиме разработки, и `false`, если компонент создан во время выполнения программы. Именно с помощью этого свойства можно отличить режим разработки от режима выполнения.

Такая проверка нужна очень часто. Например, если компонент имеет таймер вызывающий некоторые действия, то в режиме разработки этот таймер не должен работать (листинг 4.4).

Листинг 4.4. Определение режима разработки

```
public partial class MyComponent : Component
{
    private System.Timers.Timer timer;

    public MyComponent()
    {
        // Создаем таймер
        timer = new System.Timers.Timer();
        timer.Elapsed +=
            new System.Timers.ElapsedEventHandler(timer_Elapsed);
        timer.Enabled = true;
    }

    // Метод, вызываемый при срабатывании таймера
    void timer_Elapsed(object sender, ElapsedEventArgs e)
    {
        if (DesignMode) // Если это режим разработки — выходим
            return;

        // Некоторые действия, которые будут выполнены только
        // в режиме выполнения (run-time)
    }
}
```

Обращаю внимание на очень важный момент. Нельзя использовать свойство `DesignMode` в конструкторе компонента или любом другом методе, вызываемом из конструктора, например, так:

```
public partial class MyComponent : Component
{
    private System.Timers.Timer timer;

    public MyComponent()
    {
        if (DesignMode) // Неверно — нельзя использовать
                        // эту проверку в конструкторе
        {
            timer = new System.Timers.Timer();
            timer.Elapsed +=
                new System.Timers.ElapsedEventHandler(timer_Elapsed);
        }
    }
}
```

```

        timer.Enabled = true;
    }
}

```

Это связано с тем, что сначала вызывается конструктор компонента, а уже затем компонент привязывается к контейнеру и создается интерфейс `ISite`. Другими словами, во время выполнения конструктора компонент еще не может определить режим работы и `DesignMode` всегда вернет `false`.

Не откладывая в долгий ящик, сразу расскажу, как быть в таких случаях. Вариантов определения режима разработки довольно много, но у каждого способа есть свои ограничения и недостатки. Можно, например, проверять имя процесса:

```

public new bool DesignMode
{
    get
    {
        return (System.Diagnostics.Process.
            GetCurrentProcess().ProcessName == "devenv");
    }
}

```

Но этот способ не будет работать в среде Mono, Borland Delphi .NET или других редакторах, отличных от Visual Studio. Можно использовать проверку режима работы менеджера лицензий:

```

public new bool DesignMode
{
    get
    {
        return System.ComponentModel.LicenseManager.UsageMode ==
            System.ComponentModel.LicenseUsageMode.DesignTime;
    }
}

```

Это неплохой способ, но код выглядит несколько странно. Можно использовать трассировку стека вызова:

```

private bool IsDesignMode()
{
    if(Site != null )
        return Site.DesignMode;

    StackTrace stackTrace = new StackTrace();
    int frameCount = stackTrace.FrameCount - 1;
}

```

```
for( int frame = 0; frame < frameCount; frame++ )
{
    Type type = stackTrace.GetFrame(frame).GetMethod().DeclaringType;
    if (typeof(IDesignerHost).IsAssignableFrom(type))
        return true;
}
return false;
}
```

Путь тоже неплохой, но получившийся код работает очень медленно. Конечно, можно записать результат в статическую переменную и использовать значение из этого кэша, но, согласитесь, сам по себе этот способ выглядит не очень-то здорово.

Мне кажется, что наиболее корректный выход в такой ситуации — постараться вынести создание объектов из конструктора, используя, например, интерфейс `ISupportInitialize`. Этот интерфейс имеет всего два метода:

```
public interface ISupportInitialize
{
    void BeginInit();
    void EndInit();
}
```

Метод `BeginInit()` вызывается в начале инициализации компонентов, а метод `EndInit()` — при завершении инициализации. В коде формы это выглядит следующим образом:

```
((System.ComponentModel.ISupportInitialize)
    (this. MyComponent)).BeginInit();
... инициализация компонентов ...
((System.ComponentModel.ISupportInitialize)
    (this. MyComponent)).EndInit();
```

Так как метод `EndInit()` вызывается после конструкторов, то в нем проверять флаг `DesignMode` вполне допустимо (листинг 4.5). Способы, перечисленные выше, стоит применять, если вынести создание объектов из конструктора не получается никаким образом.

Кстати, в .NET 3.5, при использовании библиотеки WPF, определить режим разработки можно более простым способом:

```
if (System.ComponentModel.DesignerProperties.GetIsInDesignMode(this))
{
    // Это режим разработки
}
```

Но без WPF придется работать "по старинке".

Листинг 4.5. Создание объектов только в режиме выполнения

```
public class MyComponent : ISupportInitialize
{
    private Timer timer;

    public void BeginInit()
    {
    }

    public void EndInit()
    {
        if (!DesignMode) // Здесь проверять можно
        {
            timer = new Timer();
            timer.Tick += new EventHandler(timer_Tick);
            timer.Interval = 500;
            timer.Enabled = true;
        }
    }
}
```

4.7. Компоненты или элементы управления?

Нетрудно заметить, что класс `Component` имеет очень мало методов и свойств, необходимых самому обычному элементу интерфейса, например, кнопке. Конечно, было бы слишком сложно, если бы пришлось каждый раз реализовывать весь необходимый арсенал заново для каждого элемента, видимого на форме.

Класс `System.Windows.Forms.Control` является базовым классом для реализации визуальных компонентов. Соответственно, он имеет все методы и свойства, необходимые для визуального расположения и отображения.

Свойства и методы, относящиеся к элементу, как части формы и других компонентов:

- ☐ `string Name` — имя элемента;
- ☐ `Control Parent` — элемент, на котором располагается данный компонент.

Свойства и методы, относящиеся к расположению и размеру:

- ☐ `Rectangle Bounds` — границы элемента;
- ☐ `Rectangle ClientRectangle` — клиентская область элемента;
- ☐ `Point Location` — расположение элемента.

Свойства и методы, относящиеся к фокусу (активному компоненту на форме), видимости и активности:

- `bool CanFocus` — `true`, если элемент может принимать фокус;
- `bool CanSelect` — `true`, если элемент может быть выбран;
- `bool ContainsFocus` — `true`, если элемент имеет фокус;
- `bool Enabled` — `true`, если элемент активен;
- `bool Focused` — `true`, если элемент имеет фокус;
- `bool Visible` — `true`, если элемент видим;
- `bool Focus()`; — выбирает элемент;
- `void Hide()`; — скрывает элемент.

Свойства, относящиеся к виду элемента:

- `Color BackColor` — цвет фона;
- `Image BackgroundImage` — фоновая картинка;
- `Font Font` — шрифт для текста;
- `Color ForeColor` — цвет текста.

Свойство, относящееся к меню:

- `ContextMenu ContextMenu` — контекстное меню элемента.

Разумеется, я перечислил не все методы и свойства. Одно описание этого класса занимает несколько тысяч строк — перечислить и описать все свойства просто не реально, да и не нужно. Моя задача сейчас — просто дать общее представление о свойствах и методах визуального компонента.

Итак, если задача заключается в разработке визуального компонента, то в качестве базового компонента следует выбрать класс `Control` (или один из его наследников), а если компонент не должен быть виден, то лучше выбрать класс `Component`. В следующем разделе я расскажу о других правилах выбора базового класса.

4.8. Выбор базового класса

Для того чтобы более четко понимать, как правильно выбрать базовый класс для нового компонента, MSDN формулирует следующие правила.

- Если класс использует некоторые внешние ресурсы (например, файлы, порты и т. п.), но не будет помещаться на форму во время разработки, достаточно реализовать интерфейс `System.IDisposable`. Разумеется, такой класс не будет компонентом, но если вы не собираетесь помещать его на форму, то этого и не требуется.

- Если класс будет помещаться на форму во время разработки, реализуйте интерфейс `System.ComponentModel.IComponent` или наследуйте класс, его реализующий. Следует отметить, что интерфейс `IComponent` является расширением интерфейса `IDisposable`.
- Если класс используется во время разработки формы и должен передаваться по ссылке (*marshaled by reference*), осуществляйте наследование от класса `System.ComponentModel.Component`.
- Если класс используется во время разработки формы и должен передаваться по значению (*marshaled by value*), выполняйте наследование от класса `System.ComponentModel.MarshalByValueComponent`. Таких компонентов в .NET немного. Это, например, классы из пространства имен `System.Data` (`DataSet`, `DataRowView`).
- Если класс используется во время разработки формы, но вы не хотите наследоваться ни от класса `System.ComponentModel.Component`, ни от класса `System.ComponentModel.MarshalByValueComponent`, просто реализуйте интерфейс `IComponent`. Это не очень просто, но возможно.
- Если ваш класс имеет пользовательский интерфейс (другими словами, является визуальным), то он должен быть наследован либо от класса `System.Windows.Forms.Control` (или от класса `System.Web.UI.Control`, если это Web-форма), либо от одного из наследников, наиболее подходящих по функциональности.
- Если класс не применяется во время разработки и не использует внешних ресурсов, то не требуется реализовывать ни интерфейс `IComponent`, ни интерфейс `IDisposable`.

Замечу, что если для невидимых компонентов есть некое подобие свободы выбора базового компонента, то для визуальных компонентов практически наверняка придется явно или не явно организовывать наследование от класса `Control`. Реализовывать всю функциональность этого класса вручную не имеет смысла. Разумеется, при расширении функциональности некоего элемента управления в качестве базового следует использовать именно его, как, например, мы это делали с классом `Label`. На рис. 4.4 показаны некоторые классы, наследованные от класса `Control`:

- класс `System.Windows.Forms.ButtonBase` является базовым классом для кнопок;
- класс `System.Windows.Forms.DataGrid` представляет собой класс для отображения данных в виде таблицы;

¹ Обсуждение .NET Remoting выходит за рамки нашей книги. Достаточно просто знать, что есть два базовых класса.

- ❑ компонент `System.Windows.Forms.GroupBox` позволяет объединять элементы в группу, имеющую имя и рамку;
- ❑ компонент `System.Windows.Forms.Label` отображает метку (этим компонентом я уже пользовался при построении примеров);
- ❑ компонент `System.Windows.Forms.ListControl` является базовым элементом для представления списков (например, элементов `ListBox` и `ComboBox`);
- ❑ компонент `System.Windows.Forms.ListView` предназначен для отображения списков в различных режимах (как, например, список файлов в Проводнике Windows).

Разумеется, это не все компоненты и элементы управления. Их очень и очень много. Я не хочу подробно останавливаться на описании стандартных компонентов и элементов управления. Такое описание можно найти практически в любой книге, посвященной WinForms. Наша же цель — режим разработки, и мы пойдем дальше...

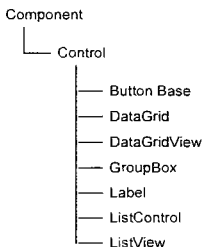


Рис. 4.4. Иерархия классов

4.9. Отладка компонентов в режиме разработки

Прежде чем пойти дальше, мне нужно рассказать об одной из важнейших сторон разработки любого кода — отладке. Работать велепую очень неудобно и непроизводительно, и, конечно, разработчики Visual Studio продумали вопрос отладки компонентов в режиме разработки.

Для отладки компонента в режиме разработки создайте отдельный проект **Windows Forms**, имеющий хотя бы одну форму (впрочем, она добавляется автоматически при создании нового проекта). Назовем его **Test**. Поместите на

форму отлаживаемый компонент. Таким образом, теперь у нас есть проект, содержащий сам компонент (например, MyControl), и проект, содержащий форму с расположенным на ней компонентом (мы назвали этот проект Test).

Теперь нужно настроить свойства проекта MyControl. Откройте вкладку **Debug** в свойствах проекта (рис. 4.5) и установите переключатель **Start Action** в положение **Start external program**. В качестве внешней программы для отладки выберите само приложение Visual Studio. Путь может выглядеть примерно так: D:\Program Files\Microsoft Visual Studio 8\Common7\IDE\devenv.exe (конечно, он зависит от выбранного пути при инсталляции среды).

Остается только установить проект компонента, как запускаемый по умолчанию. (Именно проект MyControl, а не проект Test!) Щелкните правой кнопкой мыши на проекте TestComponent и в контекстном меню выберите команду **Set as StartUp Project** (рис. 4.6). Конфигурирование закончено.

Установите точки останова в нужных местах кода компонента и запустите проект для отладки. Запустится копия Visual Studio, в которой нужно открыть тестовый проект Test и форму, на которую мы поместили отлаживаемый компонент. При вызове методов, в которых присутствуют точки останова, Visual Studio будет переходить в режим отладки.

Теперь действительно все готово, чтобы создать наш первый компонент.

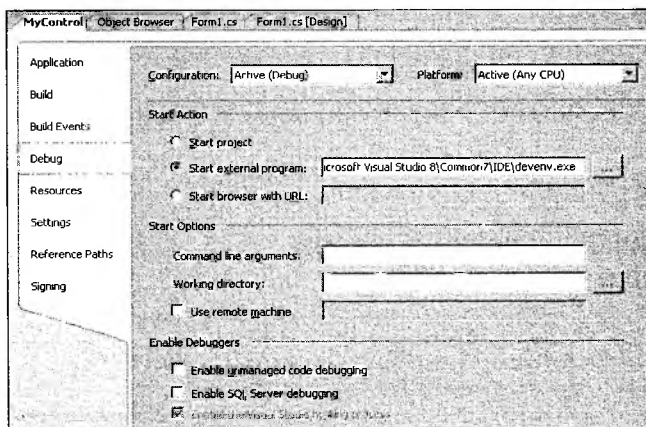


Рис. 4.5. Вкладка Debug в свойствах проекта

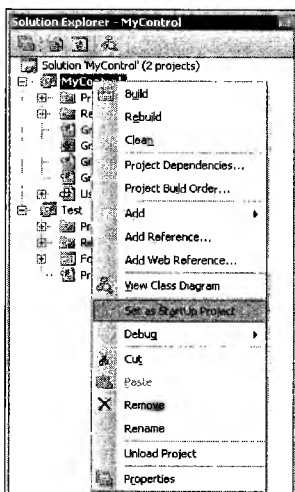
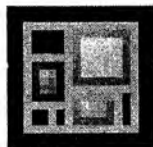


Рис. 4.6. Установка проекта для запуска по умолчанию

ГЛАВА 5



Создаем первый компонент

Чем медленнее ты работаешь — тем меньше ошибок сделаешь.

Теперь у нас достаточно информации, чтобы создать наш первый компонент. Для простоты мы сделаем компонент, расширяющий свойства существующего компонента. Это очень легко, и мне будет проще объяснять на примере, чем описывать последовательность действий. Сделаем, например, компонент, наследованный от класса `Label`, но с градиентной заливкой. В этой главе мы не будем "копать" особенно глубоко. Пока наша задача — пройти минимальный путь создания компонента.

5.1. Создаем проект

Для начала создадим новый проект типа **Windows Control Library** (рис. 5.1). По умолчанию Visual Studio добавляет класс `UserControl1`, который надо переименовать в `GradientLabel` (именно так мы назовем наш класс метки с градиентной заливкой). Можно переименовать и сам файл тоже. Полученный класс мы наследуем от класса `Label` (придется еще удалить присваивание свойства `AutoScaleMode`, т. к. у класса `Label` его нет). Вот, собственно, и все. Компилируем. Наш первый компонент готов!

Те, кто мне не верит (действительно, очень уж все просто), могут добавить к нашему проекту еще один проект — типа **Windows Forms**, а затем открыть редактор любой формы. Наш новый компонент будет блистать в окне **Toolbox** (Инструменты) стандартным значком и отсутствием опознавательных знаков (рис. 5.2). В дальнейшем мы исправим и то, и другое, а пока вспомним, что я обещал градиентную заливку.

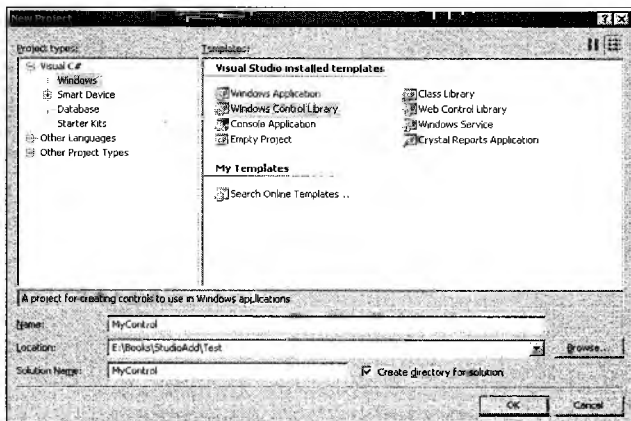


Рис. 5.1. Для создания компонента нужно создать проект **Windows Control Library**

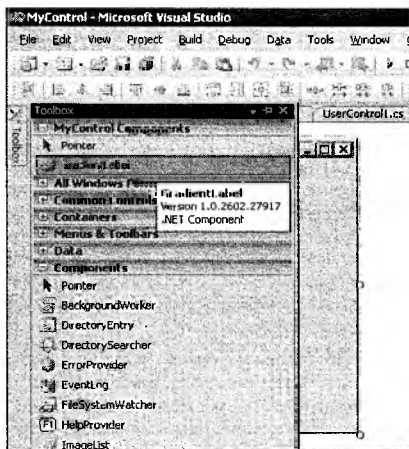


Рис. 5.2. Новый компонент **GradientLabel**

Добавим два свойства — `StartColor` и `EndColor`, которые будут соответствовать цветам начала и завершения градиента, и перекроем метод `OnPaint()`, внутри которого будем рисовать градиентную заливку (листинг 5.1). По умолчанию заливка будет переходить от ярко-зеленого (`Light Green`) к темно-синему (`Dark Blue`) цвету. Результат показан на рис. 5.3 (иллюстрации черно-белые, поэтому придется поверить мне на слово или скомпилировать код, приведенный на компакт-диске).

Класс `GradientLabel` станет нашим базовым проектом, с помощью которого мы будем изучать средства времени разработки.

Листинг 5.1. Класс `GradientLabel`

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace MyControl
{
    public partial class GradientLabel : Label
    {
        private Color startColor = Color.LightGreen;
        private Color endColor = Color.DarkBlue;

        // Цвет начала заливки
        public Color StartColor
        {
            get { return startColor; }
            set { startColor = value; }
        }

        // Цвет завершения заливки
        public Color EndColor
        {
            get { return endColor; }
            set { endColor = value; }
        }

        // Метод OnPaint
        protected override void OnPaint(PaintEventArgs pe)
```

```

{
    // Вызываем базовый OnPaint
    base.OnPaint(pe);

    // Заливаем
    Color c1 = Color.FromArgb(100, startColor);
    Color c2 = Color.FromArgb(100, endColor);
    Brush b = new Drawing2D.LinearGradientBrush(
        ClientRectangle, c1, c2, 10);
    pe.Graphics.FillRectangle(b, ClientRectangle);

    b.Dispose();
}

public GradientLabel()
{
    InitializeComponent();
}
}
}

```



Рис. 5.3. Градиентная заливка

5.2. Добавляем информацию о свойствах

Откроем окно свойств нашего компонента (например, с помощью меню **Property** по щелчку правой кнопки мыши). Как видно из рис. 5.4, два наших новых свойства видны в списке, но, к сожалению, совершенно неинформативны.

Добавить информацию о свойствах позволяют специальные атрибуты `Category` и `Description`:

- ☐ атрибут `Category` определяет название категории, к которой относится свойство. Если он не указан, то свойство считается относящимся к катего-

рии Misc. Для объединения свойств по категориям надо нажать самую левую кнопку в окне редактора свойств;

- атрибут `Description` позволяет задать некоторое строковое описание, которое будет давать представление о свойстве и его назначении.

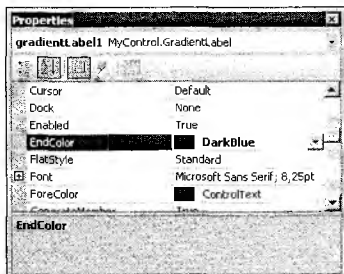


Рис. 5.4. Пока новые свойства неинформативны

Пример описания информации о свойствах показан в листинге 5.2, а результат — окно свойств с информацией — на рис. 5.5.

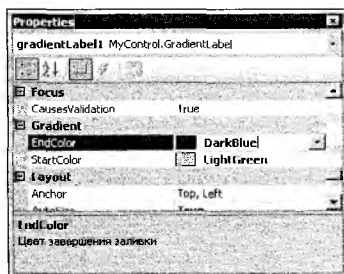


Рис. 5.5. Теперь каждое свойство имеет описание и категорию

Листинг 5.2. Добавляем информацию о свойствах в класс `GradientLabel`

```
[Category("Gradient"), Description("Цвет начала заливки")]
public Color StartColor
```



```
{
    get { return startColor; }
    set { startColor = value; }
}

[Category("Gradient"), Description("Цвет завершения заливки")]
public Color EndColor
{
    get { return endColor; }
    set { endColor = value; }
}
```

5.3. Свойство, событие и значение по умолчанию

Атрибут компонента `DefaultProperty` указывает *свойство по умолчанию*. Это свойство будет активно в редакторе свойств при его открытии первый раз:

```
[DefaultProperty("Text")]
public class Label : Control
```

Обычно в качестве такого свойства указывается наиболее "часто исправляемое" свойство. Для кнопки и текстового поля это свойство `Text`, для списка строк — свойство `Items`.

Событие по умолчанию отмечается с помощью атрибута `DefaultEvent`. Обработчик этого события будет создан при двойном щелчке на компоненте. Обычно таким атрибутом отмечается ключевое событие компонента. Например, при двойном щелчке на кнопке (компонент `Button`) создается обработчик события нажатия кнопки (событие `Click`), а для флажка (компонент `CheckBox`) таким событием будет изменение состояния (событие `CheckedChanged`).

Атрибут `DefaultValue` позволяет указывать *значение свойства по умолчанию*. Из листинга 5.1 видно, что я уже инициализировал свойства некоторыми значениями:

```
private Color startColor = Color.LightGreen;
private Color endColor   = Color.DarkBlue;
```

Зачем же тогда нужен еще и атрибут? Все дело в том, что редактор свойств не имеет информации о коде компонента и не может вычислить эти значения. Конечно, когда мы добавляем компонент на форму, начальные значения свойств вычисляются и попадают в данные формы, и поэтому мы их видим в редакторе свойств. Но только и всего. Оба свойства выделяются жирным

шрифтом, что означает что редактор свойств не видит в них значений по умолчанию (см. рис. 5.5). Именно атрибут `DefaultValue` решает этот вопрос. Атрибут имеет два типа конструкторов. Конструктор с одним параметром позволяет задать значение свойства фактически любого стандартного типа:

```
public DefaultValueAttribute(bool value);
public DefaultValueAttribute(byte value);
public DefaultValueAttribute(float value);
public DefaultValueAttribute(object value);
```

И т. д.

Последний приведенный вариант принимает параметр типа `object`, что позволяет задать значение по умолчанию для любого типа свойства. Второй конструктор используется, если нужно передать в качестве параметра не константу, а, например, статическое свойство:

```
public DefaultValueAttribute(Type type, string value);
```

Например, в нашем случае, самая очевидная конструкция `DefaultValue(Color.LightGreen)` не проходит — `LightGreen` является статическим свойством, а не константой, поэтому приходится написать чуть сложнее:

```
[Category("Gradient"), Description("Цвет начала градиента"),
DefaultValue(typeof(Color), "LightGreen");
```

После компиляции редактор свойств должен признать, что наши свойства имеют значения по умолчанию, и перестать выделять их жирным шрифтом. Кроме того, активизируется команда меню **Reset**, позволяющая установить значение свойства в значение по умолчанию (рис. 5.6).

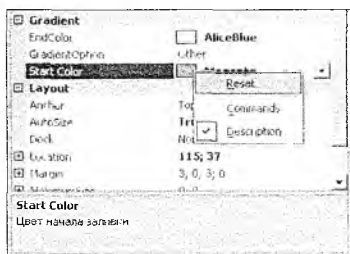


Рис. 5.6. С помощью команды **Reset** можно вернуть свойству значение по умолчанию

5.4. Дополнительные атрибуты свойств

В этом разделе я перечислю несколько атрибутов свойств. Часть из них используется не часто, но, тем не менее, знать о них надо.

По умолчанию все `public`-свойства показываются в редакторе свойств, что не всегда нужно. Отключить показ свойства помогает атрибут `Browsable`. Например, если бы я захотел не показывать свойство `StartColor` в редакторе свойств, то написал бы так:

```
[Browsable(false)]
public Color StartColor
{
    get/set
}
```

Опять же по умолчанию редактор свойств показывает имя так, как оно написано в коде. С помощью атрибута `DisplayName` можно указать любое другое, правда, я не видел, чтобы кто-то этим пользовался. Разве что для вставки пробелов в имя свойства в редакторе свойств или перевода названия на русский язык:

```
[DisplayName("Start Color")]
public Color StartColor
{
    get { return startColor; }
    set { startColor = value; OnChangeProperties(); }
}
[DisplayName("Завершение заливки")]
public Color EndColor
{
    get { return endColor; }
    set { endColor = value; OnChangeProperties(); }
}
```

Атрибут `PasswordPropertyText` позволяет указать, что текстовое поле, представленное таким свойством, будет скрыто в редакторе свойств звездочками, т. е. представляет собой поле ввода пароля и не может быть показано в открытом виде:

```
private string password;
[Description("Пароль"), PasswordPropertyText(true)]
public string Password
{
    get { return password; }
    set { password = value; }
}
```

Атрибут `ReadOnly` позволяет указать, что свойство нельзя редактировать, и редактор свойств показывает такое свойство серым цветом. Важно отметить, что хотя параметром атрибута являются значения `true` или `false`, но реально в метаданные записываются значения `ReadOnlyAttribute.Yes` и `ReadOnlyAttribute.No` соответственно. Это важно, если пытаться читать метаданные через методы отражения.

```
[ReadOnly(true)]
public Color StartColor
{
    get { return startColor; }
    set { startColor = value; }
}
```

При выборе нескольких компонентов в дизайнера редактор свойств отображает все общие свойства компонентов. При изменении значения свойства изменение производится во всех компонентах. Но это не всегда хорошо. Например, имя компонента должно быть уникальным, и, соответственно, желательно запретить одновременную установку этого свойства у выбранных компонентов. Сделать это позволяет атрибут `MergableProperty`, установка значения которого в `false` запрещает "объединение" этого свойства:

```
[MergableProperty(false)]
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Атрибут `ParenthesizePropertyName` с параметром `true` указывает редактору свойств отображать имя свойства в круглых скобках (рис. 5.7):

```
[ParenthesizePropertyName(true)]
public ControlBindingsCollection DataBindings
{
}
```

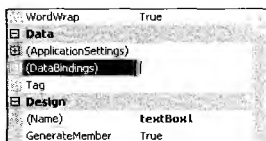


Рис. 5.7. Отображение свойства в круглых скобках

Атрибут `AttributeProvider` позволяет указать набор атрибутов свойства, полученных от другого типа. Например, такое свойство будет иметь все атрибуты класса `IListSource`:

```
[AttributeProvider(typeof(IListSource))]  
public object DataSource  
{  
    get { return this.dataGridView1.DataSource; }  
    set { this.dataGridView1.DataSource = value; }  
}
```

5.5. Реакция на изменение свойств

Дотошный читатель, выполняющий все шаги вместе со мной, давно уже заметил, что изменение значений наших свойств `StartColor` и `EndColor` на виде самого компонента сказывается совсем не сразу (точнее, оно отображается при любой вынужденной перерисовке компонента, например, при смене выделенного компонента). Действительно, ведь никто не сообщает компоненту, что ему нужно немедленно перерисовать себя!

Небольшое изменение кода закрывает этот вопрос (листинг 5.3). Теперь при изменении значения свойств мы сразу видим результат на форме, т. к. метод `Invalidate()` заставляет компонент перерисовать себя.

Кстати, заметьте, я не вызывал метод `Paint()` напрямую! Это некорректно по отношению к Windows, т. к. заставляет компонент перерисовываться независимо от того, нужно ли это операционной системе или нет. Метод `Invalidate()` более "нежен" в этом отношении — компонент будет перерисован, когда у операционной системы дойдет до него время.

Листинг 5.3. Добавляем информацию о свойствах в класс `GradientLabel`

```
[Category("Gradient"), Description("Цвет начала заливки"),  
DefaultValue(typeof(Color), "LightGreen")]  
public Color StartColor  
{  
    get { return startColor; }  
    set { startColor = value; OnChangeProperties(); }  
}  
  
[Category("Gradient"), Description("Цвет завершения заливки"),  
DefaultValue(typeof(Color), "DarkBlue")]  
public Color EndColor
```

```
{
    get { return endColor; }
    set { endColor = value; OnChangeProperties(); }
}

// Перерисовать себя при изменении свойств
private void OnChangeProperties() {
    Invalidate();
}
```

5.6. Описание и значок компонента

Теперь я сдержу обещание, которое давал в предыдущем разделе. Наш компонент должен уметь давать о себе больше информации.

Для добавления строкового описания для компонента мы добавим к классу атрибут `Description`:

```
[Description("Label с градиентной заливкой")]
public partial class GradientLabel : Label
```

Добавить значок для компонента очень просто. Если два способа это сделать: картинка может храниться в отдельном файле или ее можно хранить внутри модуля компонента.

Ну, конечно, сначала ее нужно сделать. Значок должен иметь размер 16×16 пикселей, палитру 16 цветов и формат BMP или ICO.

Для внешнего значка нужно добавить к классу атрибуты в виде:

```
[ToolboxItem(true)]
[ToolboxBitmap("C:\\Test\\GradientLabel.bmp")]
public partial class GradientLabel : Label
```

Первый атрибут говорит, что класс `GradientLabel` будет использоваться как элемент в стандартной панели инструментов **Toolbox**, а второй — где найти файл значка. Вообще говоря, первый атрибут выставлен в значение `true` по умолчанию, т. к. класс является наследником класса `Label`, но хорошей практикой является указать это напрямую.

Если же значок хранится внутри модуля (что мне кажется более естественным и удобным), то соответствующий файл нужно добавить к проекту (меню **Project | Add Existing Item**). В свойствах файла параметр `Build Action` нужно установить в значение `Embedded Resource` (рис. 5.8).

Теперь добавим атрибуты, указывающие, как найти значок:

```
[ToolboxItem(true)]
[ToolboxBitmap(typeof(GradientLabel))]
public partial class GradientLabel : Label
```

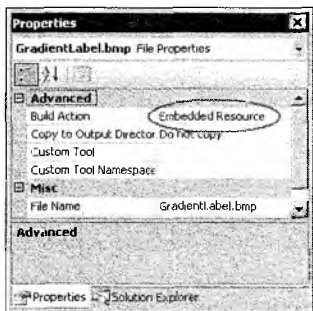


Рис. 5.8. Файл картинки должен иметь свойство Build Action равным Embedded Resource

По умолчанию берется имя файла, совпадающее с именем типа, плюс расширение bmp. Если по какой-то причине нужно указать другое имя файла, то следует использовать второй конструктор атрибута с двумя аргументами:

```
[ToolboxItem(true)]
[ToolboxBitmap(typeof(GradientLabel), "Label.bmp")]
public partial class GradientLabel : Label
```

Не знаю почему, но для компонентов текущего проекта ни значок, ни описание не появляются. Может быть, это ошибка в редакторе Visual Studio, а может быть, так и нужно. В любом случае с появлением значка и описания придется подождать, пока мы не научимся устанавливать наш компонент самостоятельно (см. разд. 5.8).

Кстати, есть и другой способ установки значка. Так как по умолчанию атрибут `ToolboxBitmap` берет файл согласно имени типа, то Visual Studio автоматически загружает значок, если находит такой файл в папке проекта. Конечно, такой способ наиболее простой, но мне кажется, более правильно хранить значок как внутренний ресурс, т. к. иначе значок не будет являться частью сборки, и придется решать вопрос доступности этого файла при распространении компонента.

Хотя установка значка процедура достаточно простая, часто она вызывает проблему. Значок просто не появляется, и понять, почему так происходит и в чем дело, довольно сложно. Наиболее частая причина проблемы — несовпадение реального и ожидаемого имени файла (ресурса). Для начала проверьте, что параметр **Default Namespace** в параметрах проекта соответствует реаль-

ному имени пространства имен проекта. Если это не помогает, можно узнать, какие имена ресурсов присутствуют в сборке, вставив в код такие строки:

```
string[] names =  
    Assembly.GetAssembly(this.GetType()).GetManifestResourceNames();  
foreach (string name in names)  
    MessageBox.Show(name);
```

Проверив, что фактическое имя ресурса совпадает с запрашиваемым, можно легко исправить проблему.

Еще одно замечание касается атрибута `ToolboxItem`. При необходимости решить обратную задачу, т. е. сделать так, чтобы компонент не показывался в палитре компонентов, достаточно задать два атрибута:

```
[DesignTimeVisible(false), ToolboxItem(false)]  
public class TestComponent : Component
```

Атрибут `DesignTimeVisible` с параметром `false` указывает, что компонент не будет виден в режиме разработки, а атрибут `ToolboxItem` с параметром `false` — что компонент не будет появляться в палитре компонентов.

5.7. События

События позволяют компонентам общаться между собой и программой. Например, если форма хочет каким-то образом обрабатывать нажатие на кнопку, то ей необходимо обрабатывать событие `Click` этой кнопки. Каждый компонент имеет множество событий, позволяющих реагировать на внутреннюю жизнь компонента или обрабатывать его реакцию на внешние события: нажатие на клавиши, движения мыши, изменение размера и т. д. Обычно обработчиками событий являются методы формы, на которой расположен компонент.

Для добавления нового события к компоненту нужно не много (листинг 5.4):

1. Добавляем описание переменной `onGradientChange` типа `event`.
2. Описываем событие `OnGradientChange` с типом `EventHandler` и аксессорами `add` и `remove`.
3. Вызываем событие из метода `OnChangeProperties()`.

Заметьте, что перед вызовом обработчика события следует проверить, что обработчик `onGradientChange` не равен `null`. Аксессоры `add` и `remove` можно не указывать, если в этом нет необходимости, но тогда переменную `onGradientChange` нужно описывать с модификатором `public` (см. разд. 3.1).

Новое событие появится в редакторе свойств на вкладке **Events** (События), для добавления обработчика достаточно щелкнуть на нем два раза.

Листинг 5.4. Описание события OnGradientChange

```
public partial class GradientLabel : Label
{
    private event EventHandler onGradientChange;

    [Category("Gradient"),
     Description("Вызывается при изменении свойств заливки")]
    public event EventHandler OnGradientChange
    {
        add
        {
            onGradientChange += value;
        }
        remove
        {
            onGradientChange -= value;
        }
    }
    private void OnChangeProperties()
    {
        Invalidate();
        if (onGradientChange != null)
        {
            onGradientChange(this, new EventArgs());
        }
    }
}
```

5.8. Установка компонентов

Пока наш компонент был виден только в рамках одного проекта. Разумеется, если мы хотим использовать наш компонент "по-настоящему", нужно поместить его в палитру компонентов.

Для установки компонента нужно выбрать команду **Choose Items** в контекстном меню панели **Toolbox** (Инструменты) в нужной группе компонентов (рис. 5.9). В появившемся диалоге (рис. 5.10) нужно нажать кнопку **Browse** и выбрать путь к файлу `MyControl.dll`. После закрытия диалога компонент `GradientLabel` появится в палитре компонентов. Кстати, появились и картинка, и описание компонента (рис. 5.11).

С программной установкой компонентов дело обстоит значительно сложнее. Мы будем разбираться с этим вопросом в *разд. 14.3*.

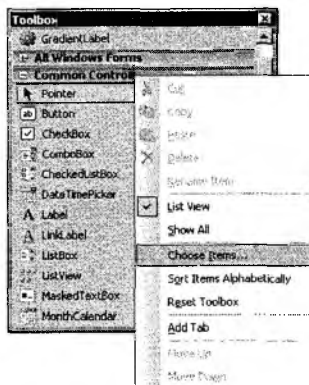


Рис. 5.9. Установка нового компонента

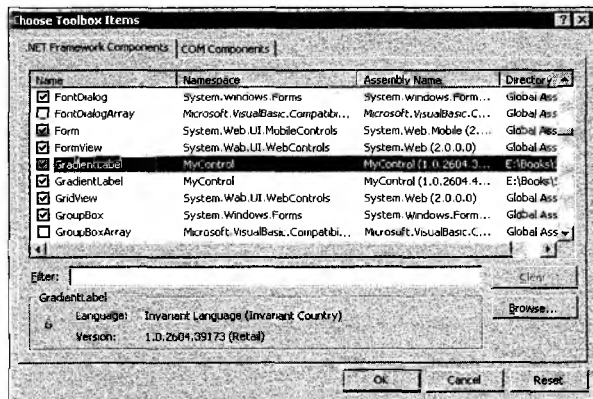


Рис. 5.10. Диалог списка установленных компонентов

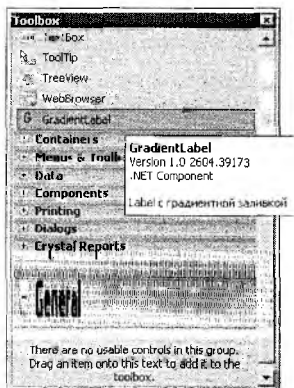


Рис. 5.11. Установленный компонент GradientLabel

5.9. Первые итоги

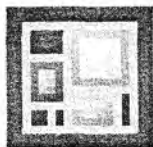
Итак, посмотрим, что у нас получилось. Мы создали компонент, расширяющий функциональность компонента Label. Наш компонент GradientLabel имеет два новых свойства и одно событие, которые видны в редакторе свойств. Конечно, это не много, но достаточно, чтобы создавать компоненты, имеющие новую функциональность.

Пройдя первые шаги, уже можно сформулировать некоторые правила, которых стоит придерживаться при разработке визуальных компонентов:

- ☐ следует помнить, что визуальный компонент существует не только в режиме выполнения, но и в режиме разработки;
- ☐ желательно, чтобы компонент умел отображать различные состояния, такие как выбран (Selected), в фокусе (Focused), активный (Enabled), неактивный (Disabled) и т. п.;
- ☐ желательно позволить пользователю компонента настроить цвета и шрифты, используемые в компоненте;
- ☐ создавайте виртуальные методы в местах, где возможны расширения функциональности и отрисовки;
- ☐ предоставляйте события, информирующие клиента об изменении свойств элемента.

В следующих главах мы будем разбирать каждый шаг построения компонентов более детально и наращивать функциональные возможности компонентов в режиме разработки.

ГЛАВА 6



Дизайнеры

Объявим мебелью кожаный заказчик
(Объявление)

Дизайнер позволяет управлять набором свойств и поведением компонента в режиме разработки, создавать пользовательские меню и многое другое. В этой главе я постараюсь рассказать про эти возможности.

6.1. Что такое дизайнер

Для того чтобы компонент мог общаться с редактором свойств, формой, на которой он расположен, и другими компонентами, используются специальные классы, называемые *дизайнерами* (designers). Классы дизайнеров расположены в пространствах имен `System.Windows.Forms.Design` и `System.ComponentModel.Design`. Специальные классы для Web-компонентов расположены в пространстве имен `System.Web.UI.Design`, но мы их пока трогать не будем.

Вот некоторые классы дизайнеров:

- `System.ComponentModel.Design.ComponentDesigner`;
- `System.Windows.Forms.Design.ControlDesigner`;
- `System.Windows.Forms.Design.ParentControlDesigner`;
- `System.Windows.Forms.Design.ScrollableControlDesigner`.

Базовым классом является класс `ComponentDesigner`. Его наследник `ControlDesigner` реализует дизайнеры для визуальных элементов управления. Следующий в иерархии класс `ParentControlDesigner`, который реализует дизайнер для элементов управления, имеющих дочерние компоненты, позволяя

добавлять, удалять и выделять их. Класс `ScrollableControlDesigner` является наследником класса `ParentControlDesigner` и реализует обработку сообщений `WM_HSCROLL` и `WM_VSCROLL` в режиме разработки.

Базовый класс `ComponentDesigner` позволяет реализовывать следующую функциональность режима разработки:

- `ActionLists` позволяет реализовать смарт-теги (smart tags, см. разд. 6.4);
- `Verbs` позволяет создавать пользовательские контекстные меню (вызываемые по щелчку правой кнопки мыши, см. разд. 6.5);
- `DesignerFile` позволяет управлять списком свойств и событий в редакторе свойств (см. разд. 6.6);
- `DoDefaultAction` позволяет реализовать собственное поведение по двойному щелчку мыши на компоненте. По умолчанию Visual Studio создает фрагмент кода для события по умолчанию (см. разд. 6.9);
- `GetService` позволяет получить доступ к сервисам режима разработки (design mode services, см. главу 7);
- `AssociatedComponents` дает доступ к списку компонентов, связанных с редактируемым;
- `Component` представляет ссылку на редактируемый компонент.

Класс `ControlDesigner` дополняет список функциональности следующими возможностями¹:

- `Control` представляет ссылку на редактируемый элемент управления;
- `SnapLines` возвращает список линий привязок (см. разд. 6.10);
- `SelectionRules` позволяет управлять правилами изменения размера и положения компонента (рис. 2.3, см. разд. 6.16);
- `OnMouseXXX` — набор методов `OnMouse` позволяет задавать реакцию элемента управления на перемещение и нажатие кнопки мыши (см. разд. 6.9);
- `OnDragXXX` — набор методов `OnDrag` позволяет элементу управления реагировать на операции перетаскивания;
- `GetGlyphs` позволяет управлять слоями и маркерами (см. разд. 6.14);
- `GetHitTest` позволяет задать области, в которых элемент управления должен реагировать на события мыши;
- `OnPaintAdornments` позволяет производить дополнительную отрисовку элемента управления (см. разд. 6.8);

¹ Я описываю класс из пространства имен `System.Windows.Forms`.

□ WndProc позволяет напрямую обрабатывать очередь Windows-сообщений в режиме разработки (см. разд. 6.12).

Теперь каждый из этих пунктов мы разберем более подробно.

6.2. Добавление дизайнера в проект

Для того чтобы можно было использовать дизайнеры, расположенные в пространстве имен System.Windows.Forms, необходимо подключить к проекту библиотеку System.Design.dll. При создании нового проекта эта библиотека не подключается. Выберите в контекстном меню проекта пункт **Add Reference** (рис. 6.1), а в появившемся диалоге — модуль System.Design (рис. 6.2). Теперь можно создавать классы дизайнеров и изучать их функциональность.

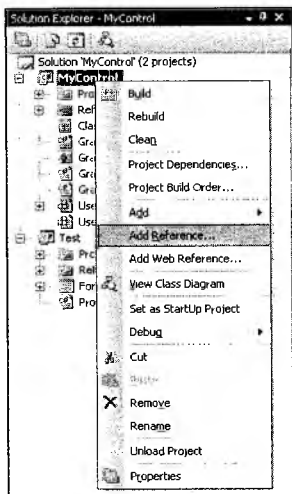


Рис. 6.1. Меню добавления ссылки

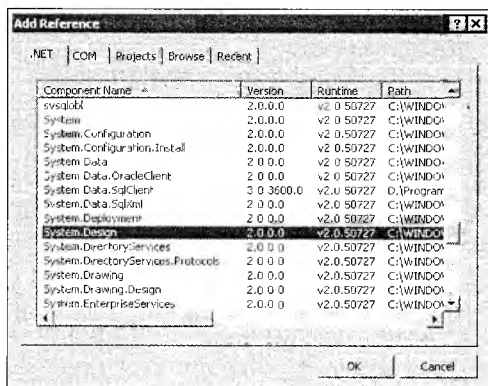


Рис. 6.2. Добавление ссылки на модуль System.Design.dll

6.3. Привязка дизайнера к компоненту

Дизайнер привязывается к компоненту с помощью атрибута `Designer`, которому в качестве параметра передается тип дизайнера. Соответствующий класс должен быть наследником одного из классов дизайнеров.

Пример привязки:

```
[DesignerAttribute(typeof(GradientLabelDesigner))]
public partial class GradientLabel : Label
```

Ничто не мешает привязать к компоненту два дизайнера, но работать будет тот, который привязан последним.

Атрибут `Designer` можно привязывать не только через тип, но и просто по имени типа:

```
[DesignerAttribute("MyControl.GradientLabelDesigner")]
public partial class GradientLabel : Label
```

Такая запись позволяет полностью разделить класс компонента и класс дизайнера, т. е. теперь не используется тип дизайнера в качестве параметра, и соответственно, не требуется наличие его исходного кода в том же проекте. Привязка дизайнера по имени называется *поздним связыванием*. Отдельную сборку с дизайнером необходимо расположить так, чтобы система Visual Studio могла найти и загрузить ее. Этот вопрос мы будем рассматривать при обсуждении установки и распространения компонентов (см. главу 14).

6.4. Смарт-теги

Одним из интересных новшеств Visual Studio являются *смарт-теги*¹ (smart tags), позволяющие вынести в специальный диалог наиболее важные свойства компонента. Например, смарт-тег для компонента `ListBox` открывает диалог редактирования элементов (рис. 6.3). Сам тег выглядит как небольшой значок со стрелочкой.

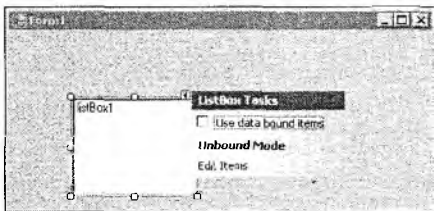


Рис. 6.3. Тег для компонента `ListBox`

В окно смарт-тега можно вынести редактирование свойств и выполнение методов. Редактирование свойств производится с помощью элементов управления, соответствующих типу свойства (например, для типа `bool` используется флажок), а для выполнения методов — ссылки.

Для создания собственного смарт-тега нужно создать список элементов тега, являющийся наследником класса `DesignerActionList`, а в перекрытом свойстве `ActionLists` дизайнера вернуть экземпляр этого списка. Согласно этой инструкции попробуем создать свой тег для нашего компонента `GradientLabel` из главы 5.

Для начала создадим класс `GradientLabelActionList`, являющийся наследником класса `DesignerActionList` (листинг 6.1). В конструкторе этого свойства я сохраняю ссылку на редактируемый компонент (переменная `glabel`). В диалог тега попадают все общедоступные (`public`) свойства и методы, значит, наш диалог будет иметь три элемента: свойство `StartColor`, свойство `EndColor` и метод `InvertColors()`, который, по моей задумке, будет менять местами цвета начала и завершения заливки.

При создании свойств следует обратить внимание на существенное отличие метода, возвращающего значение, и метода установки значения. Если в ак-

¹ Дословный перевод "интеллектуальный тег" или "умный тег" слишком громоздок и не привычен. Давайте использовать название смарт-тег или, для простоты, просто тег.

сессоре `get` нам достаточно вернуть текущее значение свойства компонента, то для аксессора `set` нужно обязательно использовать установку значения свойства через дескриптор свойства (см. разд. 4.5). Для этого используется метод `GetPropertyByName()`, возвращающий дескриптор по имени свойства.

Листинг 6.1. Класс `GradientLabelActionList`

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.Design;
using System.ComponentModel;
using System.Drawing;

namespace MyControl
{
    class GradientLabelActionList : DesignerActionList
    {
        GradientLabel glabel;

        // Конструктор
        public GradientLabelActionList(IComponent component)
            : base(component)
        {
            // Сохраняем ссылку на редактируемый компонент
            glabel = component as GradientLabel;
        }

        // В диалог тега вынесем два свойства и одну команду
        public Color StartColor
        {
            get { return this.glabel.StartColor; }
            set { GetPropertyByName("StartColor").SetValue(glabel, value); }
        }

        public Color EndColor
        {
            get { return this.glabel.EndColor; }
            set { GetPropertyByName("EndColor").SetValue(glabel, value); }
        }

        public void InvertColors()
        {
            Color tmp = glabel.StartColor;
```

```
        StartColor = glabel.EndColor;
        EndColor = tmp;
    }

    // Возвращает дескриптор свойства по имени
    private PropertyDescriptor GetPropertyByName(String propName)
    {
        PropertyDescriptor prop =
            TypeDescriptor.GetProperties(glabel)[propName];
        if (prop == null)
        {
            throw new ArgumentException("Свойство не существует", propName);
        }
        return prop;
    }
}
```

Теперь осталось привязать список элементов тега к компоненту. Для этого нам нужно перекрыть свойство `ActionLists` дизайнера нашего компонента (листинг 6.2).

Листинг 6.2. Привязка `ActionLists` к компоненту

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.ComponentModel.Design;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        private DesignerActionListCollection actionLists;

        public override DesignerActionListCollection ActionLists
        {
            get
```

```

{
    // Если еще не создавали actionList
    if (actionLists == null)
    {
        // Создаем ActionList
        actionLists = new DesignerActionListCollection();
        // Добавляем тег
        actionLists.Add(
            new GradientLabelActionList(this.Component));
    }
    return actionLists;
}
}
}
}

```

Компилируем... и у нашего компонента появился тег, открывающий диалог редактирования цветов градиентной заливки (рис. 6.4).

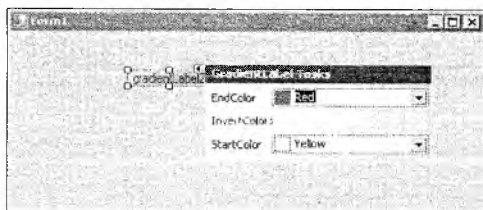


Рис. 6.4. Тег компонента GradientLabel

Остается решить несколько небольших проблем:

- ❑ при вызове пункта меню **InvertColors** меняются цвета компонента, но не обновляется окно самого тега;
- ❑ пункт меню **InvertColors** имеет имя в соответствии с именем метода, и слова написаны слитно, что не очень красиво. Было бы лучше назвать пункт **Invert Colors** или, даже, перевести названия на русский язык. То же самое относится и к именам свойств;
- ❑ хотелось бы расположить пункты в диалоге в соответствии с логикой заливки: сначала редактирование StartColor, затем EndColor и под ними ссылку на метод InvertColors().

Для решения первой проблемы нам нужно получить доступ к сервису тегов (с сервисами нам еще предстоит разбираться в главе 7, пока же примите как должное, что они есть), который и позволит нам обновлять диалоговое окно. Для этого в конструкторе класса `GradientLabelActionList` сохраним ссылку на сервис `DesignerActionUIService`:

```
class GradientLabelActionList : DesignerActionList
{
    DesignerActionUIService designerActionUIService;
    // Конструктор
    public GradientLabelActionList(IComponent component) :
        base(component)
    {
        // Сохраняем ссылку на сервис
        designerActionUIService = GetService(typeof(DesignerActionUIService))
            as DesignerActionUIService;
    }
}
```

Теперь, при вызове метода `InvertColors()`, нужно вызвать обновление диалогового окна тега. Это делается с помощью метода `Refresh()`:

```
public void InvertColors()
{
    // Меняем цвета местами
    Color tmp = glabel.StartColor;
    StartColor = glabel.EndColor;
    EndColor = tmp;
    // Обновляем диалог тега
    designerActionUIService.Refresh(glabel);
}
```

Полный код показан в листинге 6.3.

Листинг 6.3. Класс `GradientLabelActionList` с обновлением диалога при смене свойств

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.Design;
using System.ComponentModel;
using System.Drawing;

namespace MyControl
{
    class GradientLabelActionList : DesignerActionList
```

```
{
    GradientLabel glabel;
    DesignerActionUIService designerActionUIService;

    // Конструктор
    public GradientLabelActionList(IComponent component)
        : base(component)
    {
        // Сохраняем ссылку на редактируемый компонент
        glabel = component as GradientLabel;
        // Сохраняем ссылку на ActionList-сервис
        designerActionUIService =
            GetService(typeof(DesignerActionUIService))
            as DesignerActionUIService;
    }

    // В диалог тега вынесем два свойства и одну команду
    public Color StartColor
    {
        get { return this.glabel.StartColor; }
        set { GetPropertyByName("StartColor").SetValue(glabel, value); }
    }

    public Color EndColor
    {
        get { return this.glabel.EndColor; }
        set { GetPropertyByName("EndColor").SetValue(glabel, value); }
    }

    public void InvertColors()
    {
        // Меняем цвета местами
        Color tmp = glabel.StartColor;
        StartColor = glabel.EndColor;
        EndColor = tmp;
        // Обновляем диалог тега
        designerActionUIService.Refresh(glabel);
    }

    // Возвращаем дескриптор свойства по имени
    private PropertyDescriptor GetPropertyByName(String propName)
    {
        PropertyDescriptor prop =
            TypeDescriptor.GetProperties(glabel)[propName];
    }
}
```

```
if (prop == null)
{
    throw new ArgumentException("Свойство не существует", propName);
}
return prop;
}
}
```

Для решения двух других проблем нам нужно научиться каким-то образом вмешиваться в порядок формирования пунктов меню диалога тега. Сделать это можно, перекрыв метод `GetSortedActionItems()`, возвращающий коллекцию элементов диалогового окна. Более того, этот метод позволяет не только управлять порядком элементов, но формировать более информативный и красивый диалог.

Коллекция элементов представляется классом `DesignerActionItemCollection`. Каждый элемент этой коллекции является одной из реализаций абстрактного класса `DesignerActionItem`:

- `DesignerActionMethodItem` — элемент представления метода (ссылка для вызова метода);
- `DesignerActionPropertyItem` — элемент представления свойства (в зависимости от типа свойства может быть виден как выпадающий список, текстовое поле или переключатель);
- `DesignerActionTextItem` — элемент для вывода статического текста;
- `DesignerActionHeaderItem`¹ — элемент для вывода заголовка группы.

Сначала создадим в нашем диалоговом окне три группы: группу свойств, группу методов и группу для отображения дополнительной информации. Конструктор группы имеет вид:

```
DesignerActionHeaderItem(имя группы для отображения, категория)
```

Код для создания групп выглядит следующим образом:

```
items.Add(new DesignerActionHeaderItem("Свойства" , "Properties"));
items.Add(new DesignerActionHeaderItem("Методы" , "Methods"));
items.Add(new DesignerActionHeaderItem("Информация", "Info"));
```

Конструктор элемента для редактирования свойства, метода и вывода статического текста выглядит так:

¹ Вообще говоря, класс `DesignerActionHeaderItem` является наследником класса `DesignerActionTextItem`.

```

DesignerActionPropertyItem(имя свойства, отображаемое имя,
                           категория, описание)
DesignerActionMethodItem(ссылка на список, имя метода,
                         отображаемое имя, категория, описание)
DesignerActionTextItem(текст, категория)

```

Полный код метода `GetSortedActionItems()` показан в листинге 6.4, а результат его работы — на рис. 6.5.

Кстати, с помощью этого метода можно также формировать список элементов динамически, в зависимости от текущих значений свойств компонента. Например, если мы хотим отображать метод перестановки цветов только в том случае, когда цвета не равны, то можем просто добавить соответствующую проверку в метод `GetSortedActionItems()`, а при каждом изменении свойств выполнять обновление диалога. Соответствующий код показан в листинге 6.5.

Листинг 6.4. Метод `GetSortedActionItems`

```

public override DesignerActionItemCollection GetSortedActionItems()
{
    DesignerActionItemCollection items =
        new DesignerActionItemCollection();

    // Добавляем три группы
    items.Add(new DesignerActionHeaderItem("Свойства", "Properties"));
    items.Add(new DesignerActionHeaderItem("Методы", "Methods"));
    items.Add(new DesignerActionHeaderItem("Информация", "Info"));

    // Добавляем свойства в категорию Properties
    items.Add(new DesignerActionPropertyItem("StartColor",
        "Начальный цвет", "Properties",
        "Цвет начала градиентной заливки"));
    items.Add(new DesignerActionPropertyItem("EndColor",
        "Конечный цвет", "Properties",
        "Цвет завершения градиентной заливки"));

    // Добавляем метод в категорию Methods
    items.Add(new DesignerActionMethodItem(this, "InvertColors",
        "Перевернуть цвета", "Methods",
        "Поменять начальный и конечный цвета местами"));

    // Добавляем статический текст
    string info = string.Format(
        "Размер {0}x{1}", glabel.Width, glabel.Height);
}

```

```
items.Add(new DesignerActionTextItem(info, "Info"));
```

```
return items;
```

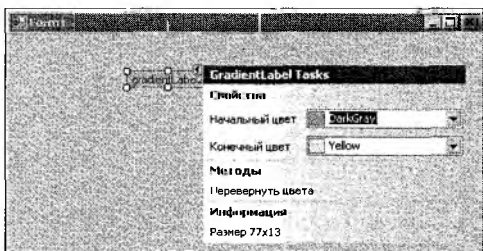


Рис. 6.6. Новый тег компонента `GradientLabel` (с группировкой)

Листинг 6.5. Динамическое формирование списка элементов диалога

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.Design;
using System.ComponentModel;
using System.Drawing;

namespace MyControl
{
    class GradientLabelActionList : DesignerActionList
    {
        ... ..
        // В диалог тега вынесем два свойства и одну команду
        public Color StartColor
        {
            get { return this.glabel.StartColor; }
            set
            {
                GetPropertyByName("StartColor").SetValue(glabel, value);
                // Обновляем диалог тега
                designerActionUIService.Refresh(glabel);
            }
        }
    }
}
```



```
public Color EndColor
{
    get { return this.glabel.EndColor; }
    set
    {
        SetPropertyByName("EndColor").SetValue(glabel, value);
        // Обновляем диалог тега
        designerActionUIService.Refresh(glabel);
    }
}

public void InvertColors()
{
    // Меняем цвета местами
    Color tmp = glabel.StartColor;
    StartColor = glabel.EndColor;
    EndColor = tmp;
}

public override DesignerActionItemCollection GetSortedActionItems()
{
    DesignerActionItemCollection items =
        new DesignerActionItemCollection();

    // Добавляем две группы
    items.Add(new DesignerActionHeaderItem("Свойства", "Properties"));

    // Добавляем свойства в категорию Properties
    items.Add(new DesignerActionPropertyItem("StartColor",
        "Начальный цвет", "Properties",
        "Цвет начала градиентной заливки"));
    items.Add(new DesignerActionPropertyItem("EndColor",
        "Конечный цвет", "Properties",
        "Цвет завершения градиентной заливки"));

    // Добавляем метод InvertColors, только если цвета не равны
    if (StartColor != EndColor)
    {
        items.Add(new DesignerActionHeaderItem("Методы", "Methods"));

        // Добавляем метод в категорию Methods
        items.Add(new DesignerActionMethodItem(
            this, "InvertColors",
```

```
        "Перевернуть цвета", "Methods",  
        "Поменять начальный и конечный цвета местами"));  
    }  
  
    // Добавляем статический текст  
    items.Add(new DesignerActionHeaderItem("Информация", "Info"));  
    string info = string.Format("Размер {0}x{1}",  
                                glabel.Width, glabel.Height);  
    items.Add(new DesignerActionTextItem(info, "Info"));  
  
    return items;  
}  
... ..  
}
```

Теперь еще раз посмотрим на код формирования `ActionList`:

```
// Создаем ActionList  
actionLists = new DesignerActionListCollection();  
// Добавляем тега  
actionLists.Add(new GradientLabelActionList(this.Component));
```

Как видно из этого фрагмента кода, `ActionList` является коллекцией, каждый элемент которой представляет собой объект типа `DesignerActionList`. И я задаю себе неожиданный вопрос: а что будет, если мы добавим еще один элемент в эту коллекцию? Вот так:

```
// Создаем ActionList  
actionLists = new DesignerActionListCollection();  
// Добавляем два тега  
actionLists.Add(new GradientLabelActionList(this.Component));  
actionLists.Add(new GradientLabelAdditionalActionList(this.Component));
```

В этом случае `Visual Studio` не будет показывать два тега, как могло бы показаться. На самом деле оба списка будут объединены в один и показаны последовательно, поэтому особой разницы в методе формирования списка параметров нет. Можно делать один большой список, а можно разбить список параметров на несколько тегов. Пожалуй, единственное различие возникнет при необходимости обновить отображение некоторого набора параметров, а не всех параметров сразу. Сделать это можно, разбив список параметров на наборы `ActionList` и обновляя только те части, которые нужно.

В следующем разделе я расскажу про контекстное меню и еще одну интересную возможность класса смарт-тегов.

6.5. Контекстное меню компонента

В этом разделе я расскажу, как формировать контекстное меню компонента, вызываемое по щелчку правой кнопки мыши. Вообще говоря, контекстное меню формируется с помощью класса `DesignerVerb` и свойства `Verbs` класса `ComponentDesigner`, но, прежде чем начать рассказывать про эти классы, хочу раскрыть небольшой секрет. Класс `DesignerActionMethodItem` имеет еще один конструктор, о котором я не рассказывал в предыдущем разделе. Шестым параметром конструктора можно указать дизайнеру автоматически создать пункт контекстного меню, соответствующий вызываемому методу (рис. 6.6):

```
items.Add(new DesignerActionMethodItem(this, "InvertColors",
    "Перевернуть цвета", "Methods",
    "Поменять начальный и конечный цвета местами",
    true));
```

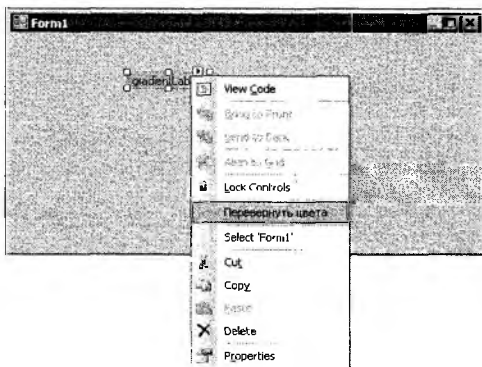


Рис. 6.6. Контекстное меню, соответствующее методам тега

Аналогично созданию тега, для создания контекстного меню нужно перекрыть свойство `Verbs` класса `ComponentDesigner` (листинг 6.6) и вернуть экземпляр класса, который должен быть наследован от `DesignerVerbCollection` (листинг 6.7). Коллекция заполняется объектами класса `DesignerVerb` вида:

```
DesignerVerb(имя_меню, new EventHandler(метод))
```

Например:

```
Add(new DesignerVerb("Перевернуть &цвета",
    new EventHandler(OnInvertColors)));
```

В методе `OnInvertColor()`, в нашем случае, я переставляю местами значения полей `StartColor` и `EndColor`. Снова обращаю внимание, что установка свойств компонента производится с помощью дескрипторов (см. разд. 4.5), получаемых вызовом метода `GetPropertyByName()`, а не напрямую.

И еще одно замечание. Не знаю зачем, но при создании контекстного меню дизайнер автоматически создает окно тега, содержащее пункты контекстного меню. Кроме того, команды, соответствующие пунктам меню, создаются в специальной области редактора свойств (рис. 6.7).

Листинг 6.6. Контекстное меню компонента

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.ComponentModel.Design;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        DesignerVerbCollection verbs;

        // Пункты контекстного меню
        public override DesignerVerbCollection Verbs
        {
            get
            {
                if (verbs == null)
                {
                    verbs = new GradientLabelVerbCollection(this.Component);
                }
                return verbs;
            }
        }
    }
}
```

Листинг 6.7. Коллекция элементов контекстного меню

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.Design;
using System.ComponentModel;
using System.Drawing;

namespace MyControl
{
    class GradientLabelVerbCollection : DesignerVerbCollection
    {
        GradientLabel glabel;

        public GradientLabelVerbCollection(IComponent component)
        {
            // Сохраняем ссылку на редактируемый компонент
            glabel = component as GradientLabel;

            // Создаем пункты меню
            Add(new DesignerVerb("Перевернуть &цвета",
                                new EventHandler(OnInvertColors)));
        }

        private void OnInvertColors(object sender, System.EventArgs e)
        {
            // Меняем цвета местами
            PropertyDescriptor start = GetPropertyByName("StartColor");
            PropertyDescriptor end = GetPropertyByName("EndColor");
            Color tmp = (Color)start.GetValue(glabel);
            start.SetValue(glabel, end.GetValue(glabel));
            end.SetValue(glabel, tmp);
        }

        // Возвращаем дескриптор свойства по имени
        private PropertyDescriptor GetPropertyByName(String propName)
        {
            PropertyDescriptor prop =
                TypeDescriptor.GetProperties(glabel)[propName];
            if (prop == null)
            {
                throw new ArgumentException("Свойство не существует", propName);
            }
        }
    }
}
```

```
return prop;
```

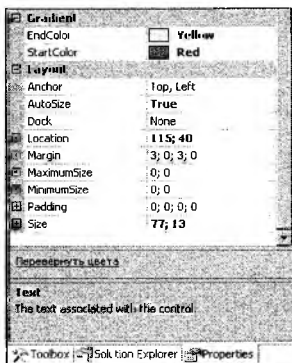


Рис. 6.7. Команды в редакторе свойств, соответствующие пунктам меню

6.6. Скрытие элементов из редактора свойств

В нашем компоненте `GradientLabel` есть одно неудобство: свойство `BackColor`, доставшееся нам по наследству от класса `Label`, противоречит нашим понятиям о градиентной заливке. Было бы хорошо убрать его из редактора свойств, чтобы не вводить пользователей нашего компонента в заблуждение. Добавить этому свойству атрибут `Browsable(false)` не получится, т. к. мы не можем исправить уже существующий класс, поэтому мы создадим специальный класс дизайнера, который будет управлять показом свойств.

Метод `PreFilterProperties()` позволяет реализовать фильтрацию элементов, которые мы хотим скрыть из редактора свойств. Параметром этого метода является словарь (dictionary) всех свойств, которые будут передаваться редактору свойств. Этот словарь представляет собой связь между именем свойства и его дескриптором. Достаточно вызвать метод `Remove()`, и ненужные нам свойства не будут отображаться:

```
properties.Remove("BackColor");
```

Полный код класса `GradientLabelDesigner` показан в листинге 6.8. Теперь, если открыть редактор свойств, то элемента с названием `BackColor` мы уже не найдем.

Аналогично свойствам, с помощью виртуального метода `PreFilterEvents()` можно сделать фильтрацию событий:

```
protected override void PreFilterEvents(IDictionary events)
{
    base.PreFilterEvents(events);
    // здесь удаляем "лишние" события
}
```

Важно понимать, что методы `PreFilterProperties()` и `PreFilterEvents()` вызываются один раз при построении списка свойств для редактора свойств, поэтому динамическое формирование списка свойств и событий таким способом сделать не получится. Как решить этот вопрос, я буду рассказывать в разд. 8.8.

Листинг 6.8. Класс `GradientLabelDesigner`

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        protected override void PreFilterProperties(IDictionary properties)
        {
            base.PreFilterProperties(properties);
            // Удаляем свойство BackColor из списка свойств
            properties.Remove("BackColor");
        }
    }
}
```

6.7. Виртуальные свойства

В предыдущем разделе мы удаляли свойство из списка свойств. Было бы логично разобраться с тем, как добавлять свойства в этот список, т. е. создавать дополнительные свойства компонента, которые существуют только в режиме

разработки. Посмотрите, например, на свойства группы **Design** (рис. 6.8). Свойства `GenerateMember`, `Locked` являются *виртуальными*. Реально у компонента `GradientLabel` таких свойств нет.

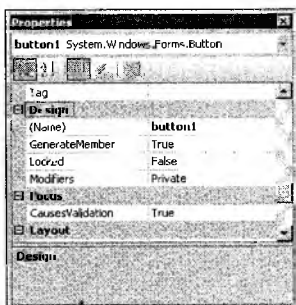


Рис. 6.8. Виртуальные свойства компонента

Для того чтобы добавить свойство, существующее только в режиме разработки, нужно добавить его в словарь `properties` в методе `PreFilterProperties()`, а само свойство будет существовать только в классе дизайнера:

```
class GradientLabelDesigner : ControlDesigner
{
    int virtualProp;
    [DesignerSerializationVisibility(
        DesignerSerializationVisibility.Hidden)]
    [Description("Виртуальное свойство")]
    public int VirtualProp
    {
        get { return virtualProp; }
        set { virtualProp = value; }
    }
}
```

Как я уже говорил, словарь `properties` представляет собой связь между именем свойства и его дескриптором, т. е. для добавления свойства нам потребуется дескриптор. Получить его можно двумя способами. Первый — создать дескриптор с помощью метода `CreateProperty()` класса `TypeDescriptor`:

```
PropertyDescriptor descr = TypeDescriptor.CreateProperty(
    base.GetType(), "VirtualProp", typeof(int), new Attribute[0]);
```


Первый параметр передает тип класса, к которому будет добавлен метод, второй — имя метода, третий — тип и четвертый задает атрибуты. Недостаток этого способа именно в отсутствии атрибутов. Перечислять необходимые атрибуты вручную довольно затруднительно, поэтому лучше пойти другим путем и копировать атрибуты существующего свойства:

```
PropertyDescriptor oldDescr =
    TypeDescriptor.GetProperties(this)["VirtualProp"];
Attribute[] attributes = new Attribute[descr.Attributes.Count];
descr.Attributes.CopyTo(attributes, 0);
PropertyDescriptor descr = TypeDescriptor.CreateProperty(
    base.GetType(), descr, attributes);
```

После этого можно добавить его в свойства компонента:

```
properties["VirtualProp"] = descr;
```

Очень важно, чтобы виртуальное свойство имело атрибут `DesignerSerializationVisibility.Hidden` (см. разд. 10.4), т. к. иначе оно попадет в свойства режима выполнения.

Полный код дизайнера, добавляющего виртуальное свойство, приведен в листинге 6.9.

Еще раз к вопросу о виртуальных свойствах я вернусь в разд. 7.18.

Листинг 6.9. Добавление виртуального свойства `VirtualProp`

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        int virtualProp;
        [DesignerSerializationVisibility(
            DesignerSerializationVisibility.Hidden)]
        [Description("Виртуальное свойство")]
        public int VirtualProp
```

```
{
    get { return virtualProp; }
    set { virtualProp = value; }
}

protected override void PreFilterProperties(IDictionary properties)
{
    base.PreFilterProperties(properties);

    // Берем существующее свойство
   PropertyDescriptor oldDescr =
        TypeDescriptor.GetProperties(this) ["VirtualProp"];
    // Копируем его атрибуты
    Attribute[] attributes =
        new Attribute[oldDescr.Attributes.Count];
    oldDescr.Attributes.CopyTo(attributes, 0);
    // Создаем новое свойство
    PropertyDescriptor descr =
        TypeDescriptor.CreateProperty(
            base.GetType(), oldDescr, attributes);
    // Добавляем его в свойства компонента
    properties["VirtualProp"] = descr;
}
}
```

6.8. Дополнительная отрисовка компонентов

Продолжаем изучать возможности, предоставляемые классом дизайнера. Довольно часто у авторов компонентов возникает желание увековечить свое имя и авторство. Указывать такую информацию во время выполнения программы было бы глупо, такой компонент никому не нужен. А вот компонент, отображающий авторскую информацию во время разработки, вполне можно вытерпеть.

Первый вариант — перекрыть метод `OnPaintBackground()` и производить дополнительную отрисовку, проверяя режим работы:

```
protected override void OnPaintBackground(PaintEventArgs pevent)
{
    base.OnPaintBackground(pevent);
    // Если это режим разработки, выводим дополнительную информацию
    if (DesignMode)
```

```

    {
        string text = "(c) Copyright information here";
        pevent.Graphics.DrawString(text, this.Font,
            new SolidBrush(Color.Black), 0,
            this.Height - (int)pevent.Graphics.MeasureString(
                text, this.Font).Height - 3);
    }
}

```

Это не очень хорошо, т. к. код времени разработки будет смешиваться с кодом времени выполнения. Второй вариант — использовать метод `OnPaintAdornments()` класса `ControlDesigner`, как это показано в листинге 6.10. Никаких дополнительных проверок не требуется, и код, выводящий дополнительную информацию, не мешает коду времени выполнения.

Листинг 6.10. Дополнительная отрисовка компонента

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        protected override void OnPaintAdornments(PaintEventArgs pe)
        {
            base.OnPaintAdornments(pe);
            string text = "(c) Copyright information here";
            pe.Graphics.DrawString(text, this.Control.Font,
                new SolidBrush(Color.Black), 0,
                this.Control.Height - (int)pe.Graphics.MeasureString(text,
                    this.Control.Font).Height - 3);
        }
    }
}

```

6.9. Обработка событий мыши в режиме разработки

Класс `ControlDesigner` имеет специальные методы режима разработки не только для отрисовки, но и для обработки событий мыши. Это довольно удобно, т. к. позволяет отделить код, относящийся к режиму разработки, от кода времени выполнения.

□ Метод `protected virtual void OnMouseEnter();` вызывается, когда мышь попадает на элемент управления.

□ Метод `protected virtual void OnMouseLeave();` вызывается, когда мышь уходит с элемента управления.

В качестве примера я расширил код из предыдущего раздела. В методе `OnMouseEnter()` мы выставляем специальный флаг `mouseOver`, а в методе `OnMouseLeave()` сбрасываем его. Таким образом, проверяя этот флаг при отрисовке элемента управления, мы добиваемся, чтобы информационное сообщение отображалось, только когда курсор мыши находится над компонентом (листинг 6.11).

Здесь же стоит ответить на один вопрос, часто возникающий при обработке событий мыши в режиме разработки. Как обработать двойной щелчок мыши? Проще всего сделать это, перекрыв метод `DoDefaultAction()` (см. разд. 6.11), как это сделано в листинге 6.11. Кстати, с одиночным щелчком мыши дела обстоят несколько сложнее — для обработки одиночного щелчка придется обрабатывать очередь сообщений (см. разд. 6.12).

Листинг 6.11. Пример обработки событий мыши в режиме разработки

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        protected bool mouseOver;
```

```
// Выводим дополнительную информацию во время разработки
protected override void OnPaintAdornments(PaintEventArgs pe)
{
    base.OnPaintAdornments(pe);
    if (mouseOver) // если курсор над компонентом
    {
        string text = "(c) Copyright information here";
        pe.Graphics.DrawString(text,
            this.Control.Font,
            new SolidBrush(Color.Black), 0,
            this.Control.Height - (int)pe.Graphics.MeasureString(
                text, this.Control.Font).Height - 3);
    }
}

// Попадание мыши на область компонента
protected override void OnMouseEnter()
{
    mouseOver = true;
    base.OnMouseEnter();
    Control.Invalidate();
}

// Уход мыши с области компонента
protected override void OnMouseLeave()
{
    mouseOver = false;
    base.OnMouseLeave();
    Control.Invalidate();
}

// Двойной щелчок мыши
public override void DoDefaultAction()
{
    //base.DoDefaultAction();
    MessageBox.Show("Двойной щелчок!");
}
}
```

6.10. Линии взаимного расположения компонентов

Типы линий привязки взаимного расположения компонентов (с.м. разд. 2.2) определяются перечислением `SnapLineType`:

- ☐ `SnapLineType.Top` — горизонтальная линия по верхней границе компонентов;
- ☐ `SnapLineType.Bottom` — горизонтальная линия по нижней границе компонентов;
- ☐ `SnapLineType.Left` — вертикальная линия по левой границе компонентов;
- ☐ `SnapLineType.Right` — вертикальная линия по правой границе компонентов;
- ☐ `SnapLineType.Horizontal` — горизонтальная линия (не по границе);
- ☐ `SnapLineType.Vertical` — вертикальная линия (не по границе);
- ☐ `SnapLineType.Baseline` — базовая линия, зависящая от основной функциональности компонента. Например, для компонента `Label` это линия по середине текста.

Свойство `SnapLines` класса дизайнера `ControlDesigner` возвращает список (`IList`) объектов типа `SnapLine`, определяющих каждую из отображаемых линий.

Создать список линий можно следующим образом:

```
public override IList SnapLines
{
    get
    {
        List<SnapLine> snapLines = new List<SnapLine>();
        snapLines.Add(new SnapLine(SnapLineType.Bottom, 10));
        return snapLines;
    }
}
```

Конструктор класса `SnapLine` имеет два параметра. Первый определяет тип линии, второй — смещение от соответствующей границы. Еще один конструктор этого класса имеет три параметра и позволяет определить приоритеты отображения линий с помощью перечисления `SnapLinePriority`:

- ☐ `Always` — показывать всегда;
- ☐ `High` — наивысший приоритет отображения;

- ☐ Medium — средний приоритет отображения;
- ☐ Low — низший приоритет отображения.

Редактор Visual Studio отображает по одной линии в вертикальном и горизонтальном направлениях, выбирая только линии с максимальным приоритетом.

При необходимости добавить линию в уже существующий список нужно привести список `SnapLines` к типу `ArrayList` и добавить в него нужные элементы:

```
ArrayList snapLines = base.SnapLines as ArrayList;  
snapLines.Add(new SnapLine(SnapLineType.Bottom, 10));
```

Вообще говоря, обычно хватает стандартных линий, и поэтому, я думаю, не стоит задерживаться на этой возможности слишком долго.

6.11. Действие по умолчанию

Действием по умолчанию называется действие, выполняемое по двойному щелчку мышью на компоненте. Стандартное поведение дизайнера в этом случае — создание обработчика события, помеченного как событие по умолчанию (*default event*) с помощью атрибута `DefaultEvent` (см. разд. 5.3).

В разд. 5.7 мы добавляли событие `OnGradientChange`, которое вызывается при изменении свойств градиентной заливки. Для того чтобы обозначить его как событие по умолчанию, достаточно добавить к нему соответствующий атрибут:

```
[DefaultEventAttribute("OnGradientChange")]  
public partial class GradientLabel : Label
```

Компилируем, открываем тестовую форму, щелкаем два раза мышью и получаем обработчик изменения свойств градиентной заливки:

```
private void gradientLabel1_OnGradientChange(object sender, EventArgs e)  
{  
}
```

Кроме того, как я уже описывал в предыдущем разделе, при необходимости обрабатывать двойной щелчок мыши в режиме разработки можно перекрыть метод `DoDefaultAction()` (см. листинг 6.11). А если действие по умолчанию нужно отменить вовсе, достаточно перекрыть этот метод, но базовый метод `base.DoDefaultAction()` не вызывать.

6.12. Прямая обработка очереди сообщений

Если стандартной функциональности режима разработки не хватает, можно напрямую обрабатывать очередь сообщений Windows, также как это делается компонентом в режиме выполнения. Отличие состоит в том, что дизайнер компонента (в отличие от самого компонента) будет получать сообщения только в режиме разработки.

Для обработки очереди сообщений нужно перекрыть метод `WndProc()` класса `ControlDesigner`. Библиотека .NET Framework является мультиплатформенной, поэтому специальных констант, описывающих коды сообщений Windows, она не содержит. Приходится описывать их самостоятельно. В листинге 6.12 показан пример обработки щелчка кнопки мыши. Код сообщения в этом случае равен `0x201`. В случае получения этого кода мы будем выводить небольшое сообщение. Для того чтобы отсечь нажатия вне границ компонента, нужно проверять координаты точки, в которой находится курсор мыши.

Аналогично можно получать не только сообщения о действиях мыши, но и любые другие сообщения.

Листинг 6.12. Обработка щелчка мыши

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        protected const int WM_LBUTTONDOWN = 0x201;

        protected override void WndProc(ref Message m)
        {
            if (m.Msg == WM_LBUTTONDOWN)
            {
                Point pt = Control.PointToClient(Cursor.Position);
            }
        }
    }
}
```



```
if ((pt.X > 0) && (pt.X < Control.Width)
    && (pt.Y > 0) && (pt.Y < Control.Height))
{
    MessageBox.Show("Нажата левая кнопка мыши! " + pt.X);
}
}
base.WndProc(ref m);
}
}
```

6.13. Встроенные компоненты

Компонент может содержать другие компоненты, но возникает одна проблема — во время разработки такие "встроенные" компоненты не получают никаких сообщений от Windows и, соответственно, не реагируют на мышь и клавиатуру.

Попробуйте пролистать значения в компоненте `ListBox` (рис. 6.9), который встроен в компонент `ListBoxControl` (листинг 6.13). Скроллинг попросту не работает, а компонент `ListBox` ведет себя "как нарисованный".

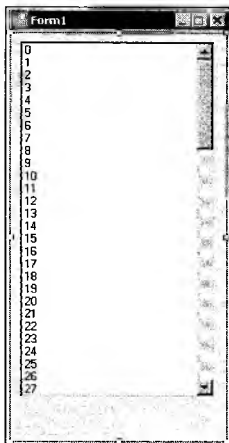


Рис. 6.9. Компонент `ListBox` внутри компонента `ListBoxControl`

Исправить ситуацию помогает специальный дизайнер (точнее сказать, метод `WndProc()` этого дизайнера), который транслирует сообщения Windows в режиме разработки (листинг 6.14). После подключения дизайнера к компоненту внутренний компонент будет работать как настоящий `ListBox`, позволяя прокручивать сообщения. Правда, при этом пропадает возможность перемещать компонент и менять его размеры. Еще раз со встроенными компонентами мы встретимся в *разд. 7.15*, а о возможностях расположения компонентов друг на друге я расскажу в *разд. 6.17*.

Листинг 6.13. Компонент `ListBoxControl`

```
using System;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Drawing.Design;
using System.Data;
using System.Diagnostics;
using System.Windows.Forms;
using System.Security.Permissions;
using System.Windows.Forms.Design;

namespace WindowMessageDesignerExample
{
    public class ListBoxControl : UserControl
    {
        private ListBox listBox1;

        public ListBoxControl()
        {
            InitializeComponent();
        }

        public override ISite Site
        {
            get
            {
                return base.Site;
            }
            set
            {
                base.Site = value;
            }
        }
    }
}
```

```
// Заполняем listBox
for (int i = 0; i < 100; i++)
    listBox1.Items.Add(i.ToString());
}

// Инициализация компонентов
private void InitializeComponent()
{
    this.listBox1 = new ListBox();
    this.SuspendLayout();
    this.listBox1.Location = new System.Drawing.Point(8, 8);
    this.listBox1.Name = "listBox1";
    this.listBox1.Size = new System.Drawing.Size(192, 368);
    this.listBox1.TabIndex = 0;
    this.BackColor = System.Drawing.Color.Beige;
    this.Controls.Add(this.listBox1);
    this.Location = new System.Drawing.Point(500, 400);
    this.Name = "ListBoxControl";
    this.Size = new System.Drawing.Size(442, 422);
    this.ResumeLayout(false);
}
}
```

Листинг 6.14. Дизайнер WindowMessageDesigner, передающий сообщения в режиме разработки

```
public class WindowMessageDesigner : ControlDesigner
{
    public WindowMessageDesigner()
    {
    }

    // Обрабатываем WndProc
    protected override void WndProc(ref Message message)
    {
        if (message.HWnd == this.Control.Handle)
            base.WndProc(ref message);
        else
            this.DefWndProc(ref message);
    }
}
```

6.14. Слои и маркеры

При выборе компонента на форме дизайнер предоставляет возможность изменения размеров элемента управления с помощью *маркеров* (glyph). Количество и положение маркеров может меняться в зависимости от возможностей компонента. Так, например, кнопка имеет восемь маркеров, а компонент CheckBox — один (рис. 6.10).

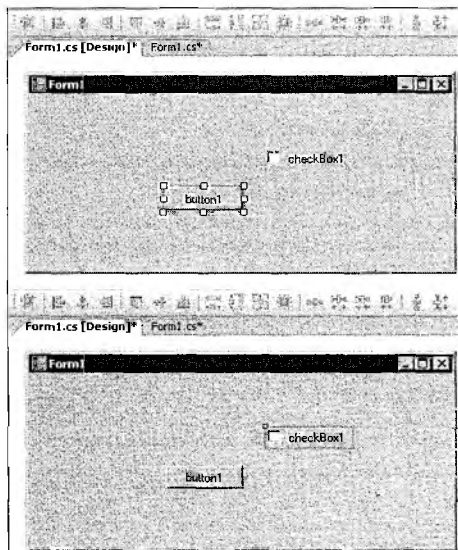


Рис. 6.10. Маркеры изменения размера

Класс `System.Windows.Forms.Design.Behavior.Adorner` представляет собой невидимую прослойку между дизайнером и пользователем. Каждый слой может содержать один или несколько маркеров, представленных классом `System.Windows.Forms.Design.Behavior.Glyph`. Каждый слой может быть активным (видимым) или неактивным (скрытым). Только активные слои могут реагировать на действия пользователя. Элемент управления может иметь несколько слоев, например слой с маркерами изменения размера, невидимый слой для обработки действий мыши и т. д.

Класс `Glyph` является базовым для всех маркеров. Кроме него библиотека .NET включает в себя еще два класса маркеров, являющихся наследниками класса `Glyph`:

- ❑ класс `ComponentGlyph` имеет дополнительное свойство `RelatedComponent`, возвращающее компонент, к которому привязан маркер;
- ❑ класс `ControlBodyGlyph` представляет собой маркер для изменения границ компонента.

Маркер имеет четыре основные "характеристики":

- ❑ свойство `Behavior` возвращает объект типа `System.Windows.Forms.Design.Behavior`, описывающий поведение маркера, т. е. его реакцию на действия пользователя;
- ❑ свойство `Bounds` возвращает границы маркера;
- ❑ метод `GetHitTest()` определяет реальные границы реакции маркера;
- ❑ метод `Paint()` отвечает за отрисовку маркера.

Таким образом, для создания маркера достаточно определить его поведение, границы и вид. Для примера мы реализуем маркер, позволяющий уменьшать и увеличивать размер компонента (`zoom glyph`).

Для добавления маркера к компоненту можно пойти двумя путями. Первый — добавить маркеры напрямую в коллекцию маркеров компонента. Вторым — добавить новый слой и уже в него добавить маркер.

Реализация первого способа показана в листинге 6.15. Дизайнер предоставляет доступ к маркерам с помощью метода `GetGlyphs`, возвращающего коллекцию `System.Windows.Forms.Design.Behavior.GlyphCollection`. Параметр `selectionType` этого метода описывает текущее состояние компонента:

- ❑ значение `GlyphSelectionType.NotSelected` указывает, что компонент не выделен;
- ❑ значение `GlyphSelectionType.SelectedPrimary` означает, что компонент выбран и является первым в списке;
- ❑ `GlyphSelectionType.Selected` означает, что компонент выбран, но первым в списке не является.

Перекрыв метод `GetGlyphs()`, мы добавляем в список маркеров свой маркер `ZoomGlyph`:

```
if (selectionType != GlyphSelectionType.NotSelected)
{
    glyphs.Add(new ZoomGlyph(this.BehaviorService, this.Control));
}
```

Дополнительная проверка состояния компонента нужна для того, чтобы маркер добавлялся в список, только если компонент выбран в дизайнере. Иначе маркер будет присутствовать всегда.

Листинг 6.15. Реализация метода GetGlyphs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.Windows.Forms.Design.Behavior;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        public override GlyphCollection GetGlyphs(
            GlyphSelectionType selectionType)
        {
            // Получаем коллекцию граничных меток
            GlyphCollection glyphs = base.GetGlyphs(selectionType);
            // Добавляем свою метку
            if (selectionType != GlyphSelectionType.NotSelected)
            {
                glyphs.Add(new ZoomGlyph(this.BehaviorService, this.Control));
            }
            // Возвращаем результат
            return glyphs;
        }
    }
}
```

Второй способ показан в листинге 6.16. При инициализации дизайнера мы добавляем слой `zoomAdorner`, а затем добавляем в него маркер типа `zoomGlyph`. В деструкторе дизайнера мы удаляем слой. Управление видимостью также производится в методе `GetGlyphs()`, но в этом случае мы управляем видимостью не конкретного маркера, а всего слоя. Признаться, использовать метод `GetGlyph()` для управления видимостью слоя не совсем правильно.

В разд. 7.5 мы будем рассматривать специальный сервис `ISelectionService`, который позволит решить вопрос более красиво.

Добавление слоя производится с помощью сервиса `BehaviorService`, который предоставляет методы управления маркерами. Более подробно с работой сервисов мы будем разбираться в главе 7, пока же нам достаточно одного свойства и двух методов этого сервиса:

- ❑ свойство `Adorners` возвращает коллекцию слоев;
- ❑ метод `ControlToAdornerWindow()` позволяет вычислить координаты границы (и, соответственно, центра) слоя;
- ❑ метод `SyncSelection()` обновляет маркеры, синхронизируя их положение с текущим положением и размерами компонента.

Листинг 6.16. Добавление слоя

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.Windows.Forms.Design.Behavior;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        private Adorner zoomAdorner;

        public override void Initialize(IComponent component)
        {
            base.Initialize(component);

            // Добавляем слой
            zoomAdorner = new Adorner();
            BehaviorService.Adorners.Add(zoomAdorner);
            // Добавляем метку в новый слой
            zoomAdorner.Glyphs.Add(new ZoomGlyph(BehaviorService, Control));
        }
    }
}
```

```
protected override void Dispose(bool disposing)
{
    // Удаляем слой
    if (disposing && zoomAdorner != null)
    {
        BehaviorService b = BehaviorService;
        if (b != null)
        {
            b.Adorners.Remove(zoomAdorner);
        }
    }
}

public override GlyphCollection GetGlyphs(
    GlyphSelectionType selectionType)
{
    // Управляем видимостью слоя при изменении
    // выделенного компонента. Более правильно
    // использовать сервис ISelectionService.
    BehaviorService b = BehaviorService;
    if (b != null)
    {
        int index = b.Adorners.IndexOf(zoomAdorner);
        if (index > 0)
        {
            b.Adorners[index].Enabled =
                (selectionType != GlyphSelectionType.NotSelected);
        }
    }

    return base.GetGlyphs(selectionType);
}
}
```

Реализация самого класса маркера приведена в листинге 6.17. В конструктор класса `ZoomGlyph` передаются два параметра: сервис `BehaviorService` и сам компонент. Свойство `Bounds` будет возвращать прямоугольник, размером `10x10`, располагающийся в центре компонента (сложная калькуляция внутри аксессора как раз и вычисляет центр компонента). Соответственно, метод `GetHitTest()` будет возвращать курсор в виде руки (`Cursor.Hand`), если указатель мыши попал в этот прямоугольник.


```
        return bounds;
    }
}

// Возвращаем нужный курсор, если указатель мыши
// попал на нашу границу. Иначе возвращаем null.
public override Cursor GetHitTest(Point p)
{
    if (Bounds.Contains(p))
    {
        return Cursors.Hand;
    }

    return null;
}

// Рисуем границу
public override void Paint(PaintEventArgs pe)
{
    // Рисуем кружок с крестиком
    pe.Graphics.FillEllipse(Brushes.Blue, Bounds);
    Pen pen = new Pen(Color.Red);
    pe.Graphics.DrawLine(
        pen, center.X, center.Y-4, center.X, center.Y+4);
    pe.Graphics.DrawLine(
        pen, center.X-4, center.Y, center.X+4, center.Y);
}

// Определяем поведение при попадании курсора на маркер
class ZoomBehavior : Behavior
{
    BehaviorService behaviorService;
    Control control;

    public ZoomBehavior(BehaviorService behaviorService,
        Control control)
    {
        this.behaviorService = behaviorService;
        this.control = control;
    }

    public override bool OnMouseUp(Glyph g, MouseButton button)
    {
        if (button == MouseButton.Left)
```

```

{
    if ((control.Width < (control.Parent.Width - 10)) &&
        (control.Height < (control.Parent.Height - 10)))
    {
        // Пересчитываем размер
        control.Left -= 5;
        control.Top -= 5;
        control.Width += 10;
        control.Height += 10;
        // Обновляем границы
        behaviorService.SyncSelection();
    }
}

if (button == MouseButton.Right)
{
    if ((control.Width > 10) && (control.Height > 10))
    {
        // Пересчитываем размер
        control.Left += 5;
        control.Top += 5;
        control.Width -= 10;
        control.Height -= 10;
        // Обновляем границы
        behaviorService.SyncSelection();
    }
}

// true означает, что мы обработали событие
return true;
}
}

```

Теперь вернемся еще раз к конструктору класса `ZoomGlyph`. Обратите внимание, что в конструктор базового класса мы передали объект типа `ZoomBehavior`, который будет реализовывать реакцию маркера на действия пользователя. Класс `ZoomBehavior` перекрывает метод `OnMouseUp`, который вызывается при щелчке мыши на маркере. Параметр `button` позволяет определить, какая именно кнопка мыши была нажата. При щелчке левой кнопкой мы будем увеличивать размер компонента, а при щелчке правой — уменьшать. Вызов метода `SyncSelection()` необходим для перерисовки маркеров в соответствии с новыми размерами компонента.

Вид получившегося маркера показан на рис. 6.11, а его работу можно посмотреть, запустив код, находящийся на компакт-диске.

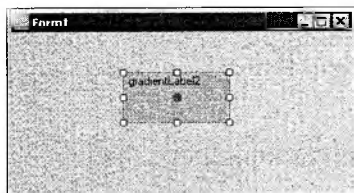


Рис. 6.11. Маркер Zoom

6.15. Реализация дополнительного меню с помощью маркера

Класс `Behavior`, используемый при определении поведения маркера, никак не ограничивает нас, как именно будет реализовано это поведение. В качестве дополнительного примера я хочу показать код, реализующий дополнительное меню, которое открывается при щелчке на маркере. Нечто подобное реализовано в .NET 3.0 для компонентов Windows Workflow Foundation.

Само меню создается в конструкторе класса:

```
public MenuBehavior(BehaviorService behaviorService, Control control)
{
    this.behaviorService = behaviorService;
    this.control = control;

    menu = new ContextMenu();
    menu.MenuItems.Add(new MenuItem("Zoom+",
        new EventHandler(Menu1OnClick)));
    menu.MenuItems.Add(new MenuItem("Zoom-",
        new EventHandler(Menu2OnClick)));
}
```

А в обработчике правой кнопки мыши это меню отображается:

```
public override bool OnMouseUp(Glyph g, MouseButton button)
{
    if (button == MouseButton.Right)
```

```
{  
    menu.Show(control, new Point(control.Width, 0));  
}  
return true;  
}
```

Полный код маркера показан в листинге 6.18, а результат выполнения кода — на рис. 6.12.

Листинг 6.18. Классы MenuGlyph

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.Windows.Forms.Design.Behavior;  
using System.Drawing;  
using System.Windows.Forms;  
  
namespace MyControl  
{  
    class MenuGlyph : Glyph  
    {  
        Control control;  
        BehaviorService behaviorService;  
        Point center;  
  
        public MenuGlyph(BehaviorService behaviorService,  
                        Control control)  
            : base(new MenuBehavior(behaviorService, control))  
        {  
            this.behaviorService = behaviorService;  
            this.control = control;  
        }  
  
        // Границы метки  
        public override Rectangle Bounds  
        {  
            get  
            {  
                // Используем метод сервиса для получения точки  
                // края компонента  
                Point edge = behaviorService.ControlToAdornerWindow(control);
```

```
// Размер компонента
Size size = control.Size;
// Вычисляем центр
center = new Point(edge.X + size.Width, edge.Y);
// Граница — это прямоугольник 10x10
Rectangle bounds = new Rectangle(center.X - 5,
                                   center.Y - 5,
                                   10,
                                   10);

return bounds;
}
}

// Возвращаем нужный курсор, если указатель мыши
// попал на нашу границу. Иначе возвращаем null.
public override Cursor GetHitTest(Point p)
{
    if (Bounds.Contains(p))
    {
        return Cursors.Hand;
    }
    return null;
}

// Рисуем границу
public override void Paint(PaintEventArgs pe)
{
    pe.Graphics.FillRectangle(Brushes.Blue, Bounds);
}

// Определяем поведение при попадании курсора на маркер
class MenuBehavior : Behavior
{
    BehaviorService behaviorService;
    Control control;
    ContextMenu menu;

    public MenuBehavior(BehaviorService behaviorService,
                        Control control)
    {
        this.behaviorService = behaviorService;
        this.control = control;
    }
}
```

```
// Создаем меню
menu = new ContextMenu();
menu.MenuItems.Add(new MenuItem("Zoom+",
                                new EventHandler(Menu1OnClick)));
menu.MenuItems.Add(new MenuItem("Zoom-",
                                new EventHandler(Menu2OnClick)));
}
```

```
void Menu1OnClick(object sender, EventArgs e)
{
    control.Left    -= 5;
    control.Top     -= 5;
    control.Width   += 10;
    control.Height  += 10;
    // Обновляем границы
    behaviorService.SyncSelection();
}
```

```
void Menu2OnClick(object sender, EventArgs e)
{
    // Пересчитываем размер
    control.Left    += 5;
    control.Top     += 5;
    control.Width   -= 10;
    control.Height  -= 10;
    // Обновляем границы
    behaviorService.SyncSelection();
}
```

```
public override bool OnMouseUp(Glyph g, MouseButton button)
{
    if (button == MouseButton.Right)
    {
        // Отображаем меню
        menu.Show(control, new Point(control.Width, 0));
    }
    // True означает, что мы обработали событие
    return true;
}
```

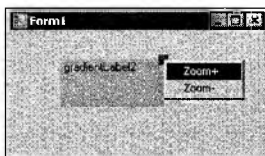


Рис. 6.12. Отображение дополнительного меню с помощью маркера

6.16. Правила изменения размера и положения компонента

Дизайнер компонента позволяет редактировать его положение и размеры, причем каждый компонент сам определяет, какие именно параметры доступны для редактирования. По умолчанию визуальный компонент позволяет изменять размеры как по горизонтали, так и по вертикали, но распространять такое поведение на все компоненты было бы не совсем корректно. Например, для компонента, являющегося разделительной линией (bevel), правильнее было бы запретить менять вертикальный размер, если это горизонтальная разделительная линия. И, наоборот, запретить менять горизонтальный размер для вертикального разделителя.

С точки зрения кода правилами изменения размера и положения управляет свойство `SelectionRules` класса `ControlDesigner`. Для изменения поведения достаточно перекрыть это свойство и возвращать тот набор правил, который согласуется с видом и свойствами компонента:

```
public class BevelLineDesigner : ControlDesigner
{
    public override SelectionRules SelectionRules
    {
        get
        {
            SelectionRules rules = base.SelectionRules;
            // В зависимости от ориентации разделителя
            // формируем правила изменения его размеров
            if (this.Orientation == Orientation.Horizontal)
            {
                rules = SelectionRules.Moveable |
                    SelectionRules.Visible |
                    SelectionRules.LeftSizeable |
                    SelectionRules.RightSizeable;
            }
        }
    }
}
```



```
        else
        {
            rules = SelectionRules.Moveable |
                    SelectionRules.Visible |
                    SelectionRules.TopSizeable |
                    SelectionRules.BottomSizeable;
        }
        return rules;
    }
}
```

Перечисление `SelectionRules` имеет несколько значений:

- ☐ `None` — правила не заданы;
- ☐ `Locked` — запрещено менять размер и положение компонента;
- ☐ `TopSizeable` — можно изменять размер сверху;
- ☐ `BottomSizeable` — можно изменять размер снизу;
- ☐ `LeftSizeable` — можно изменять размер слева;
- ☐ `RightSizeable` — можно изменять размер справа;
- ☐ `AllSizeable` — можно изменять размер с любой стороны;
- ☐ `Moveable` — можно изменять положение компонента;
- ☐ `Visible` — отображать специальную рамку вокруг компонента.

Например, запретить полностью изменение размеров компонента (как, например, у компонента `CheckBox`) можно следующим образом:

```
public override SelectionRules SelectionRules
{
    get
    {
        SelectionRules selectionRules = base.SelectionRules;
        selectionRules &= ~(SelectionRules.AllSizeable);
        return selectionRules;
    }
}
```

При разработке собственного компонента не забывайте, при необходимости настраивать нужные разрешения изменения размеров, чтобы не вводить пользователя в заблуждение.

6.17. Расположение компонентов друг на друге

Класс `ParentControlDesigner` реализует дизайнер, предназначенный для компонентов, которые содержат другие компоненты или сами содержатся в других компонентах. Этот класс имеет множество дополнительных свойств и методов, но наиболее интересны два (и еще про одно свойство я расскажу в следующем разделе):

- метод `CanBeParentedTo()` позволяет определить класс компонентов, на которых может располагаться компонент;
- метод `CanParent()` позволяет определить класс компонентов, которые могут располагаться на компоненте.

Эти два метода управляют расположением компонентов друг на друге. Например, код из листинга 6.19 определяет, что компонент, к которому подключен дизайнер `GradientLabelDesigner`, не может располагаться на компоненте типа `Panel` и не может содержать на себе компоненты типа `Panel`.

Но простого переопределения метода в дизайнере недостаточно, чтобы запретить расположение компонента на определенном классе компонентов — его все еще можно перетащить из палитры компонентов, хотя и нельзя перемещать по форме. Окончательно запретить это действие можно с помощью перекрытия метода `OnParentChanged` самого компонента, как показано в листинге 6.20.

Листинг 6.19. Использование методов `CanParent` и `CanBeParentedTo`

```
class GradientLabelDesigner : ParentControlDesigner
{
    protected override void PreFilterProperties(IDictionary properties)
    {
        base.PreFilterProperties(properties);
        properties.Remove("BackColor");
    }

    public override bool CanParent(Control control)
    {
        return !(control is Panel);
    }

    public override bool CanBeParentedTo(IDesigner parentDesigner)
    {

```

```
        return !(parentDesigner.Component is Panel);
    }
}
```

Листинг 6.20. Запрет расположения компонента на панели

```
public partial class GradientLabel : Label
{
    protected override void OnParentChanged(EventArgs e)
    {
        base.OnParentChanged(e);

        if (Parent != null)
        {
            if (!(Parent is Panel))
                throw new ApplicationException(
                    "Компонент не может располагаться на " + Parent.ToString());
        }
    }
}
```

6.18. Сетка дизайнера

Еще один метод класса `ParentControlDesigner` — метод `DrawGrid`, который позволяет управлять отрисовкой сетки дизайнера. Если он возвращает `false`, сетка не будет рисоваться. Если `true` — сетка будет отрисована независимо от настроек самого дизайнера:

```
class GradientLabelDesigner : ParentControlDesigner
{
    protected override bool DrawGrid
    {
        get
        {
            return false;
        }
    }
}
```

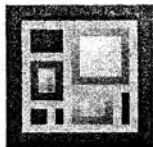
В принципе, перекрытие отрисовки сетки может потребоваться, например, для компонентов, выглядящих как рамка, но не имеющих своего фона. Именно прозрачность компонента будет являться основной проблемой — просту-

пающая сквозь компонент сетка дизайнера не всегда выглядит красиво и соответствует задумке автора. В таких случаях сетку удобно выключить.

6.19. Инициализация компонентов

В заключение главы еще один полезный совет. Конструктор компонента вызывается при создании компонента независимо от того, каким путем создается компонент. Соответственно, он будет вызван и при загрузке формы (загрузке каждого компонента на форме), и при добавлении новых компонентов на форму. Иногда возникает необходимость раздельно инициализировать существующие и новые компоненты, например, при инициализации координат компонента на форме. Решить этот вопрос можно с помощью виртуальных методов класса `ComponentDesigner`: `InitializeExistingComponent` и `InitializeNewComponent`. Первый метод вызывается при инициализации существующих компонентов на форме, а второй — при инициализации нового компонента.

ГЛАВА 7



Сервисы

Плывут две акулы вдоль побережья и видят серфингиста. Одна другой восхищено: "Вот это сервис — на подносе, да еще и с салфеткой!"

Когда компонент помещается в контейнер, он получает доступ к сервисам, предоставляемым этим контейнером с помощью интерфейса `IServiceProvider`:

```
public interface IServiceProvider
{
    object GetService(Type serviceType);
}
```

Из определения интерфейса видно, что доступ к сервисам осуществляется по типу. Нужный сервис можно получить с помощью метода `GetService()` класса `ComponentDesigner`, как, например, мы это делали при создании смарт-тега:

```
designerActionUIService = GetService(typeof(DesignerActionUIService))
                           as DesignerActionUIService;
```

В этой главе мы рассмотрим и другие сервисы. Вообще, в библиотеке .NET имеется множество сервисов, но не все они документированы. Я не уверен, что пользоваться недокументированными сервисами — это хороший тон, но все-таки расскажу про несколько наиболее интересных из них. При разработке приложений, пожалуйста, учитывайте, что недокументированные вещи могут смениться, и код перестанет работать в следующих версиях Framework.

7.1. Сервисы режима разработки

Вот список наиболее важных сервисов времени разработки:

- `System.ComponentModel.Design.IDesignerHost` — возвращает интерфейс хоста (см. разд. 4.4);

- ❑ `System.ComponentModel.Design.IComponentChangeService` — позволяет уведомлять контейнер (и всех, кто подписался на данные события) и получать уведомления об изменении компонентов и их свойств;
- ❑ `System.ComponentModel.Design.ISelectionService` — позволяет получать информацию о выделенных компонентах и управлять выделением;
- ❑ `System.ComponentModel.Design.IMenuCommandService` — позволяет расширять контекстное меню Visual Studio, открываемое при нажатии на форму, или просто создавать команды дизайнера, привязанные, например, к комбинации клавиш;
- ❑ `System.ComponentModel.Design.IReferenceService` — позволяет получить имена компонентов по ссылкам на них и, наоборот, ссылки по именам. Также можно получить список всех компонентов формы;
- ❑ `System.ComponentModel.Design.Serialization.INameCreationService` — позволяет сгенерировать уникальное имя для компонента или проверить имеющееся на уникальность и правильность;
- ❑ `System.ComponentModel.Design.IEventBindingService` — обеспечивает сервис управления событиями компонента. Например, позволяет программно создать обработчик события;
- ❑ `System.ComponentModel.Design.IDesignerOptionService` — предоставляет доступ к параметрам дизайнера, настраиваемым с помощью меню **Tools | Options | Windows Form Designer**;
- ❑ `System.ComponentModel.Design.IExtenderListService` и `IExtenderProviderService` — предоставляют сервисы, позволяющие создавать глобальные свойства, добавляющиеся ко всем компонентам. Эти сервисы мы будем обсуждать отдельно в *разд. 12.1*;
- ❑ `System.ComponentModel.Design.IDesignerEventService` — позволяет получать нотификации о смене дизайнера. В частности, это нотификации о переключении из дизайнера в код и обратно;
- ❑ `System.ComponentModel.Design.IDictionaryService` — предоставляет доступ к таблице пользовательских данных компонента времени разработки;
- ❑ `System.ComponentModel.Design.IHelpService` — обеспечивает доступ к контекстной помощи Visual Studio;
- ❑ `System.ComponentModel.Design.ITypeDescriptorFilterService` — позволяет фильтровать или добавлять описания свойств, событий или атрибутов компонентов;
- ❑ `System.Drawing.Design.IToolboxService` — позволяет взаимодействовать с палитрой компонентов и программно добавлять компоненты на форму;

- `System.ComponentModel.Design.IResourceService` — позволяет получить доступ к ресурсам проекта, например, для компонентов, поддерживающих переключение языков;
- `System.ComponentModel.Design.DesignerActionService` — предоставляет сервис для организации смарт-тегов (см. *разд. 6.4*);
- `System.Windows.Forms.Design.IUIService` — позволяет создавать диалоговые окна и управлять окнами Visual Studio;
- `System.Windows.Forms.Design.IWindowsFormsEditorService` — предоставляет возможность показать собственный диалог при редактировании свойства. Этот сервис я буду рассматривать в *разд. 9.3* при обсуждении редакторов типа;
- `System.ComponentModel.Design.ITypeDiscoveryService` — предоставляет сервис для получения доступных типов. Работу с этим сервисом я покажу на примере в *разд. "Редактирование списка объектов нескольких типов" главы 9*;
- `System.Drawing.Design.IPropertyValueUIService` — позволяет отобразить небольшой значок рядом с именем свойства и обработать двойной щелчок на этом значке. Может использоваться для отображения ошибок значения свойства;
- `System.Windows.Forms.Design.ISplitwindowService` — недокументированный сервис, позволяющий добавить в редактор формы специальное окно, как, например, для невидимых компонентов.

Описывать каждый сервис сам по себе, как мне кажется, смысла не имеет. Мы пойдем другим путем. Я буду описывать некоторые стандартные задачи, а уже от задачи будет зависеть, какой из сервисов надо будет использовать. Кроме того, про некоторые сервисы я буду рассказывать в других главах, там, где необходимость их использования вызвана реальной задачей.

Кроме приведенного выше списка, я расскажу о классе `ProjectItem` и интерфейсе `INestedContainer`. С одной стороны, они не являются сервисами, но с другой, их экземпляры можно получить с помощью вызова метода `GetService()`. Класс `ProjectItem` позволяет получить информацию о Visual Studio, проекте и форме, в которую вставлен компонент. Эта функциональность не документирована в MSDN, но требуется достаточно часто. Интерфейс `INestedContainer` позволяет создавать и использовать компоненты, содержащиеся внутри других компонентов.

Информацию о сервисах `DesignerActionUIService` (см. *разд. 7.2*) и `BehaviorService` (см. *разд. 7.3*) я добавил для полноты картины, хотя мы уже рассматривали эти сервисы в *разд. 6.4* и *6.14*.

7.2. Управление окном смарт-тега

Если выполнение каких-то операций в окне смарт-тега приводит к изменению значения полей компонента, нужно обновить само окно смарт-тега. Для этого надо получить ссылку на сервис `DesignerActionUIService`:

```
designerActionUIService = GetService(typeof(DesignerActionUIService))
```

А затем вызвать метод `Refresh`:

```
designerActionUIService.Refresh(glabel);
```

Мы использовали этот сервис в *разд. 6.4*.

7.3. Управление слоями и маркерами

В *разд. 6.14* мы применяли сервис `BehaviorService` для добавления нового слоя. Кратко опишу основные моменты использования этого сервиса.

Ссылка на этот сервис в классе `ControlDesigner` уже имеется, поэтому получать его с помощью вызова `GetService()` не нужно:

```
Adorner zoomAdorner = new Adorner();  
BehaviorService.Adorners.Add(zoomAdorner);  
zoomAdorner.Glyphs.Add(new ZoomGlyph(BehaviorService, Control));
```

Для удаления слоя мы использовали метод `Remove()`:

```
BehaviorService.Adorners.Remove(zoomAdorner);
```

Для изменения состояния слоя нужно сначала получить его индекс, а затем установить свойство `Enabled` в нужное значение:

```
int index = BehaviorService.Adorners.IndexOf(zoomAdorner);  
if (index > 0)  
{  
    BehaviorService.Adorners[index].Enabled =  
        (selectionType != GlyphSelectionType.NotSelected);  
}
```

После изменения свойств компонента может потребоваться обновить маркеры компонента. Для этого нужно вызвать метод `SyncSelection()`:

```
behaviorService.SyncSelection();
```

При обновлении маркеров (вызванное, например, изменением размеров компонента пользователем) вызывается событие `Synchronize`.

Для преобразования координат экрана и слоя служат методы `AdornerWindowPointToScreen()`, `AdornerWindowToScreen()`, `ScreenToAdornerWindow()` и `ControlToAdornerWindow()`. Так, например, мы вычисляли центр компонента:

```
Point edge = behaviorService.ControlToAdornerWindow(control);
center = new Point(edge.X + (control.Size.Width / 2),
                  edge.Y + (control.Size.Height / 2));
```

Соответствующий код можно найти в *разд. 6.14*.

7.4. Обработка изменений компонентов и их свойств

Рассмотрим ситуацию, когда к компоненту `Button` привязан компонент `ImageList` (список картинок), и, соответственно, кнопка отображает одну из картинок, содержащихся в `ImageList`. Все работает замечательно до тех пор, пока кто-нибудь не удалит с формы компонент `ImageList`. Если кнопка не узнает об этом событии и попытается получить свою картинку, в приложении возникнет исключительная ситуация и, вероятнее всего, система `Visual Studio` будет аварийно закрыта.

Для того чтобы одни компоненты могли узнавать об изменении других, служит сервис `System.ComponentModel.Design.IComponentChangeService`, имеющий следующий набор событий:

- `event ComponentEventHandler ComponentAdding` вызывается перед добавлением компонента;
- `event ComponentEventHandler ComponentAdded` вызывается после добавления компонента;
- `event ComponentChangingEventHandler ComponentChanging` вызывается перед изменением свойства компонента;
- `event ComponentChangedEventHandler ComponentChanged` вызывается после изменения свойства компонента;
- `event ComponentEventHandler ComponentRemoving` вызывается перед удалением компонента;
- `event ComponentEventHandler ComponentRemoved` вызывается после удаления компонента;
- `event ComponentRenameEventHandler ComponentRename` вызывается после переименования компонента.

В листинге 7.1 показан пример компонента `ComponentClass`, который обрабатывает все события сервиса `IComponentChangeService` и отображает соответ-

вующие сообщения в специальном списке сообщений `logListBox`. При создании компонента (правильнее сказать, при привязке компонента к контейнеру в свойстве `Site`) нужно зарегистрировать соответствующие обработчики событий (метод `RegisterChangeNotifications()`), а при уничтожении компонента — освободить их (метод `ClearChangeNotifications()`). Как это выглядит в Visual Studio, показано на рис. 7.1.

При необходимости уведомить сервис об изменениях в свойствах компонента используются методы `OnComponentChanging()` и `OnComponentChanged()`, например, так:

```
if (DesignMode)
{
    IComponentChangeService srv =
        GetService(typeof(IComponentChangeService))
            as IComponentChangeService;
    if (srv != null)
    {
        srv.OnComponentChanged(this, null, null, null);
    }
}
```

Событие `ComponentRename` может оказаться важным, если в некотором списке хранится не только ссылка на компонент, но и его имя.

Лисинг 7.1. Пример обработки событий сервиса `IComponentChangeService`

```
using System;
using System.Data;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;

namespace IComponentChangeServiceExample
{
    public class ComponentClass : UserControl
    {
        // Буфер для отображения сообщений
        private ListBox logListBox;

        // Конструктор
        public ComponentClass()
```

```
{
    InitializeComponent();
}

private void InitializeComponent()
{
    this.logListBox = new System.Windows.Forms.ListBox();
    this.SuspendLayout();
    //
    // logListBox
    //
    this.logListBox.Dock = System.Windows.Forms.DockStyle.Fill;
    this.logListBox.Location = new System.Drawing.Point(0, 0);
    this.logListBox.Name = "logListBox";
    this.logListBox.Size = new System.Drawing.Size(200, 95);
    this.logListBox.TabIndex = 0;
    //
    // ComponentClass
    //
    this.Controls.Add(this.logListBox);
    this.Name = "ComponentClass";
    this.Size = new System.Drawing.Size(200, 100);
    this.ResumeLayout(false);
}

// Свойство Site устанавливается дизайнером только
// в режиме разработки в момент установки связи между
// дизайнером и компонентом. Именно в этот момент мы
// будем регистрировать обработчики событий сервиса
public override ISite Site
{
    get
    {
        return base.Site;
    }
    set
    {
        // Очистить предыдущие обработчики,
        // если они были установлены
        ClearChangeNotifications();

        // Сохраняем значение Site
        base.Site = value;
    }
}
```

```

        // Регистрируем свои обработчики уведомлений
        // об изменении состояния компонентов
        RegisterChangeNotifications();
    }
}

// Удаление обработчиков событий
private void ClearChangeNotifications()
{
    // Получаем ссылку на сервис
    IComponentChangeService changeService =
        (IComponentChangeService)GetService(
            typeof(IComponentChangeService));

    // Если сервис не null, значит, это режим разработки
    if (changeService != null)
    {
        changeService.ComponentChanged -=
            new ComponentChangedEventHandler(OnComponentChanged);
        changeService.ComponentChanging -=
            new ComponentChangingEventHandler(OnComponentChanging);
        changeService.ComponentAdded -=
            new ComponentEventHandler(OnComponentAdded);
        changeService.ComponentAdding -=
            new ComponentEventHandler(OnComponentAdding);
        changeService.ComponentRemoved -=
            new ComponentEventHandler(OnComponentRemoved);
        changeService.ComponentRemoving -=
            new ComponentEventHandler(OnComponentRemoving);
        changeService.ComponentRename -=
            new ComponentRenameEventHandler(OnComponentRename);
    }
}

// Регистрируем обработчики событий
private void RegisterChangeNotifications()
{
    // Получаем ссылку на сервис
    IComponentChangeService changeService =
        (IComponentChangeService)GetService(
            typeof(IComponentChangeService));

    if (changeService != null)
    {

```

```
changeService.ComponentChanged +=
    new ComponentChangedEventHandler(OnComponentChanged);
changeService.ComponentChanging +=
    new ComponentChangingEventHandler(OnComponentChanging);
changeService.ComponentAdded +=
    new ComponentEventHandler(OnComponentAdded);
changeService.ComponentAdding +=
    new ComponentEventHandler(OnComponentAdding);
changeService.ComponentRemoved +=
    new ComponentEventHandler(OnComponentRemoved);
changeService.ComponentRemoving +=
    new ComponentEventHandler(OnComponentRemoving);
changeService.ComponentRename +=
    new ComponentRenameEventHandler(OnComponentRename);
}
}

// Вызывается после изменения свойств компонента
private void OnComponentChanged(
    object sender, ComponentChangedEventArgs ce)
{
    if (ce.Component != null &&
        ((IComponent)ce.Component).Site != null && ce.Member != null)
        DoLog(string.Format("Поле {0} компонента {1} изменено",
            ce.Member.Name, ((IComponent)ce.Component).Site.Name));
}

// Вызывается до изменения свойств компонента
private void OnComponentChanging(object sender,
    ComponentChangingEventArgs ce)
{
    if (ce.Component != null &&
        ((IComponent)ce.Component).Site != null && ce.Member != null)
        DoLog(string.Format("Поле {0} компонента {1} будет изменено",
            ce.Member.Name,
            ((IComponent)ce.Component).Site.Name));
}

// Вызывается до добавления компонента
private void OnComponentAdded(object sender, ComponentEventArgs ce)
{
    DoLog(string.Format("Добавлен компонент {0}",
        ce.Component.Site.Name));
}
```

```
// Вызывается после добавления компонента
private void OnComponentAdding(object sender, ComponentEventArgs ce)
{
    DoLog(string.Format("Будет добавлен компонент типа {0}",
        ce.Component.GetType().FullName));
}

// Вызывается после удаления компонента
private void OnComponentRemoved(object sender, ComponentEventArgs ce)
{
    DoLog(string.Format("Удален компонент {0}",
        ce.Component.Site.Name));
}

// Вызывается до удаления компонента
private void OnComponentRemoving(object sender,
    ComponentEventArgs ce)
{
    DoLog(string.Format("Компонент {0} будет удален",
        ce.Component.Site.Name));
}

// Вызывается при переименовании компонента
private void OnComponentRename(object sender,
    ComponentRenameEventArgs ce)
{
    DoLog(string.Format("Компонент {0} переименован в {1}",
        ce.OldName, ce.NewName));
}

// Добавляем строку в журнал
private void DoLog(string text)
{
    logListBox.Items.Add(text);
    logListBox.SelectedIndex = logListBox.Items.Count - 1;
}

// Сброс обработчиков при освобождении компонента
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        ClearChangeNotifications();
    }
}
```

```
base.Dispose(disposing);
```



Рис. 7.1. Вид компонента ComponentClass

7.5. Управление выделенными компонентами

В разд. 6.14 нам надо было управлять видимостью слоя с нашим маркером в зависимости от выбранного компонента. Если компонент выбран, мы показывали слой, если выделение снималось или перемещалось на другой компонент, слой надо было скрыть. Тогда я использовал не самый лучший способ определения изменения выбора элемента, и пришла пора разобраться, как сделать это наиболее корректно.

Для получения нотификаций об изменении выделенного компонента и управления выделением служит сервис `IService`.

Вот список наиболее интересных свойств, методов и событий этого сервиса:

- свойство `object PrimarySelection` возвращает "основной" (т. е. выбранный первым) компонент;
- свойство `int SelectionCount` возвращает количество выбранных компонентов;
- событие `event EventHandler SelectionChanging` вызывается перед изменением выбранного компонента;

- ❑ событие `event EventHandler SelectionChanged` вызывается после изменения выбранного компонента;
- ❑ метод `bool GetComponentSelected(object component)` позволяет проверить, является ли компонент выбранным;
- ❑ метод `ICollection GetSelectedComponents()` возвращает коллекцию выбранных компонентов;
- ❑ методы `void SetSelectedComponents(ICollection components)` и `void SetSelectedComponents(ICollection components, SelectionTypes selectionType)` позволяют установить выбранные компоненты.

В листинге 7.2 показано, как правильно обработать событие изменения выбранного компонента с помощью сервиса `IService`. Все, что нужно сделать, — это получить интерфейс сервиса с помощью метода `GetService()` и зарегистрировать обработчик события `SelectionChanged`.

Листинг 7.2. Пример обработки событий сервиса `IService`

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.Windows.Forms.Design.Behavior;
using System.ComponentModel.Design;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        private Adorner zoomAdorner;

        public override void Initialize(IComponent component)
        {
            base.Initialize(component);

            // Добавляем слой
            zoomAdorner = new Adorner();
            zoomAdorner.Enabled = false;
            BehaviorService.Adorners.Add(zoomAdorner);
        }
    }
}
```



```
// Добавляем метку в новый слой
zoomAdorner.Glyphs.Add(new ZoomGlyph(BehaviorService, Control));

// Регистрируем свой обработчик изменения выделения
// с помощью сервиса ISelectionService
ISelectionService s = (ISelectionService)
    GetService(typeof(ISelectionService));
s.SelectionChanged += new EventHandler(OnSelectionChanged);
)

protected override void Dispose(bool disposing)
{
    // Удаляем слой
    if (disposing && zoomAdorner != null)
    {
        BehaviorService b = BehaviorService;
        if (b != null)
        {
            b.Adorners.Remove(zoomAdorner);
        }
    }
}

// Вызывается при изменении выделения компонента
private void OnSelectionChanged(object sender, System.EventArgs e)
{
    // Управляем видимостью слоя при изменении
    // выделенного компонента
    BehaviorService b = BehaviorService;
    if (b != null)
    {
        int index = b.Adorners.IndexOf(zoomAdorner);
        if (index > 0)
        {
            ISelectionService s = (ISelectionService)
                GetService(typeof(ISelectionService));
            b.Adorners[index].Enabled =
                s.PrimarySelection == this.Component;
        }
    }
}
```

Метод `SetSelectedComponent()` полезен при организации меню **Выделить все компоненты** такого же типа или аналогичного. Кроме того, иногда возникает необходимость повторного создания компонента на форме. Например, при изменении каких-то глобальных настроек проще заново создать компонент, чем заставить его пересчитать изменения. Для этого можно использовать такую последовательность действий:

1. Сохранить положение компонента на форме.
2. Выделить компонент с помощью метода `SetSelectedComponent()`.
3. С помощью сервиса `IMenuCommandService` (см. следующий раздел) вырезать компонент в буфер и снова вставить его.
4. Восстановить положение компонента.
5. С помощью вызова `SetSelectedComponent(null)` убрать выделение с компонента.

В листинге 7.3 показана реализация этого алгоритма.

Листинг 7.3. Пересоздание компонента

```
protected void ReLoadComponent(Component component)
{
    Point p = new Point();
    // Сохраняем положение на форме
    if (component is Control)
    {
        p = (component as Control).Location;
    }
    // Получаем интерфейсы сервисов
    IMenuCommandService imcs =
        (IMenuCommandService)this.GetService(typeof(IMenuCommandService));
    ISelectionService iss =
        (ISelectionService)this.GetService(typeof(ISelectionService));
    // Выделяем компонент
    iss.SetSelectedComponents(new IComponent[] { component },
        SelectionTypes.Auto);
    // Вызываем удаление (Cut) и вставку (Paste)
    imcs.GlobalInvoke(StandardCommands.Cut);
    imcs.GlobalInvoke(StandardCommands.Paste);
    // Запоминаем ссылку на компонент
    Component cmp = (Component)iss.PrimarySelection;
    // Сбрасываем выделение
    iss.SetSelectedComponents(null);
}
```

```
// Обновляем свойства
PropertyDescriptor.Refresh(cmp);
// Восстанавливаем положение
if (cmp is Control)
{
    (cmp as Control).Location = p;
    (cmp as Control).Focus();
}
}
```

7.6. Расширение контекстного меню формы

В разд. 6.5 с помощью свойства `Verbs` класса `ComponentDesigner` мы расширяли контекстное меню компонента. Сервис `IMenuService` позволяет аналогично расширить контекстное меню дизайнера, т. е. формы, на которую помещаются компоненты. Следует помнить, что сервис добавляет "глобальное" меню, поэтому перед добавлением я сначала удаляю это меню. Если оно еще не было добавлено — ничего страшного не произойдет, а если было, то это поможет избежать дублирования. В листинге 7.4 показано добавление глобального меню, при вызове которого выводится небольшое сообщение.

Листинг 7.4. Создание глобального меню с помощью сервиса `IMenuService`

```
using System;
using System.Windows.Forms.Design;
using System.ComponentModel;
using System.ComponentModel.Design;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        DesignerVerb dverb;
        MenuCommand mc;

        public override void Initialize(IComponent component)
        {
            base.Initialize(component);

            // Получаем интерфейс сервиса
            IMenuCommandService mcs = (IMenuCommandService)component.Site.
                GetService(typeof(IMenuCommandService));
```

```

// Добавляем меню
dverb = new DesignerVerb("Глобальное меню",
                        new EventHandler(OnGlobal));

dverb.Visible = true;
dverb.Enabled = true;
dverb.Supported = true;
mcs.RemoveVerb(dverb);
mcs.AddVerb(dverb);
}

private void OnGlobal(object sender, System.EventArgs e)
{
    System.Windows.Forms.MessageBox.Show("Вызвано меню Global");
}
}
}

```

7.7. "Горячие" команды формы

Метод `AddCommand` сервиса `IMenuCommandService` позволяет создавать обработчики "горячих" команд, как показано в листинге 7.5. Обратите внимание, что перед тем как добавить пункт меню, его надо удалить, чтобы избежать дублирования. Конкретная команда назначается с помощью класса `MenuCommands` или `StandardCommands`. Эти команды не являются напрямую отражением клавиатурных комбинаций, это именно набор команд (листинг 7.6). Например, `LockCommand` или `BringForward` не являются кодами клавиш.

Листинг 7.5. Создание "горячей" команды формы в режиме разработки

```

using System;
using System.Windows.Forms.Design;
using System.ComponentModel;
using System.ComponentModel.Design;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        DesignerVerb dverb;
        MenuCommand mc;

        public override void Initialize(IComponent component)
        {
            base.Initialize(component);

```

```
// Получаем интерфейс сервиса
IMenuCommandService mcs = (IMenuCommandService)component.Site.
    GetService(typeof(IMenuCommandService));

// Добавляем обработчик клавиши <Home>
mc = new MenuCommand(new EventHandler(OnKeyHome),
    MenuCommands.KeyHome);

mc.Enabled = true;
mc.Visible = true;
mc.Supported = true;
mcs.RemoveCommand(mc);
mcs.AddCommand(mc);
}

// Обработчик клавиши <Home>
private void OnKeyHome(object sender, EventArgs e)
{
    System.Windows.Forms.MessageBox.Show("Нажата клавиша <Home>.");
}
}
```

Листинг 7.6. Классы StandardCommands и MenuCommands

```
using System;

namespace System.ComponentModel.Design
{
    public class StandardCommands
    {
        public static readonly CommandID AlignBottom;
        public static readonly CommandID AlignHorizontalCenters;
        public static readonly CommandID AlignLeft;
        public static readonly CommandID AlignRight;
        public static readonly CommandID AlignToGrid;
        public static readonly CommandID AlignTop;
        public static readonly CommandID AlignVerticalCenters;
        public static readonly CommandID ArrangeBottom;
        public static readonly CommandID ArrangeIcons;
        public static readonly CommandID ArrangeRight;
        public static readonly CommandID BringForward;
        public static readonly CommandID BringToFront;
        public static readonly CommandID CenterHorizontally;
        public static readonly CommandID CenterVertically;
```

```
public static readonly CommandID Copy;
public static readonly CommandID Cut;
public static readonly CommandID Delete;
public static readonly CommandID DocumentOutline;
public static readonly CommandID F1Help;
public static readonly CommandID Group;
public static readonly CommandID HorizSpaceConcatenate;
public static readonly CommandID HorizSpaceDecrease;
public static readonly CommandID HorizSpaceIncrease;
public static readonly CommandID HorizSpaceMakeEqual;
public static readonly CommandID LineupIcons;
public static readonly CommandID LockControls;
public static readonly CommandID MultiLevelRedo;
public static readonly CommandID MultiLevelUndo;
public static readonly CommandID Paste;
public static readonly CommandID Properties;
public static readonly CommandID PropertiesWindow;
public static readonly CommandID Redo;
public static readonly CommandID Replace;
public static readonly CommandID SelectAll;
public static readonly CommandID SendBackward;
public static readonly CommandID SendToBack;
public static readonly CommandID ShowGrid;
public static readonly CommandID ShowLargeIcons;
public static readonly CommandID SizeToControl;
public static readonly CommandID SizeToControlHeight;
public static readonly CommandID SizeToControlWidth;
public static readonly CommandID SizeToFit;
public static readonly CommandID SizeToGrid;
public static readonly CommandID SnapToGrid;
public static readonly CommandID TabOrder;
public static readonly CommandID Undo;
public static readonly CommandID Ungroup;
public static readonly CommandID VerbFirst;
public static readonly CommandID Verblast;
public static readonly CommandID VertSpaceConcatenate;
public static readonly CommandID VertSpaceDecrease;
public static readonly CommandID VertSpaceIncrease;
public static readonly CommandID VertSpaceMakeEqual;
public static readonly CommandID ViewCode;
public static readonly CommandID ViewGrid;
public StandardCommands();
}
}
```

```
using System.ComponentModel.Design;

namespace System.Windows.Forms.Design
{
    public sealed class MenuCommands : StandardCommands
    {
        public static readonly CommandID ComponentTrayMenu;
        public static readonly CommandID ContainerMenu;
        public static readonly CommandID DesignerProperties;
        public static readonly CommandID EditLabel;
        public static readonly CommandID KeyCancel;
        public static readonly CommandID KeyDefaultAction;
        public static readonly CommandID KeyEnd;
        public static readonly CommandID KeyHome;
        public static readonly CommandID KeyInvokeSmartTag;
        public static readonly CommandID KeyMoveDown;
        public static readonly CommandID KeyMoveLeft;
        public static readonly CommandID KeyMoveRight;
        public static readonly CommandID KeyMoveUp;
        public static readonly CommandID KeyNudgeDown;
        public static readonly CommandID KeyNudgeHeightDecrease;
        public static readonly CommandID KeyNudgeHeightIncrease;
        public static readonly CommandID KeyNudgeLeft;
        public static readonly CommandID KeyNudgeRight;
        public static readonly CommandID KeyNudgeUp;
        public static readonly CommandID KeyNudgeWidthDecrease;
        public static readonly CommandID KeyNudgeWidthIncrease;
        public static readonly CommandID KeyReverseCancel;
        public static readonly CommandID KeySelectNext;
        public static readonly CommandID KeySelectPrevious;
        public static readonly CommandID KeyShiftEnd;
        public static readonly CommandID KeyShiftHome;
        public static readonly CommandID KeySizeHeightDecrease;
        public static readonly CommandID KeySizeHeightIncrease;
        public static readonly CommandID KeySizeWidthDecrease;
        public static readonly CommandID KeySizeWidthIncrease;
        public static readonly CommandID KeyTabOrderSelect;
        public static readonly CommandID SelectionMenu;
        public static readonly CommandID SetStatusRectangle;
        public static readonly CommandID SetStatusText;
        public static readonly CommandID TraySelectionMenu;
        public MenuCommands();
    }
}
```

7.8. Работа с именами и типами компонентов

При необходимости получить компонент по имени или типу используется сервис `IServiceReference`. Вот его основные методы:

- ❑ метод `string GetName(object reference)` возвращает имя компонента;
- ❑ метод `object GetReference(string name)` возвращает компонент по имени. Если компонент с таким именем не найден, метод вернет `null`;
- ❑ метод `object[] GetReferences(Type baseType)` возвращает все компоненты указанного типа;
- ❑ метод `object[] GetReferences()` возвращает все объекты, доступные в режиме разработки. При этом следует помнить, что не все они являются компонентами — в этот список входят и объекты дизайнеров.

В листинге 7.7 показан пример использования сервиса `IServiceReference`. Результатом его работы (рис. 7.2) является сообщение о наличии компонентов того же типа, что и текущий выбранный. Для этого я сначала регистрирую обработчик смены выбранного компонента с помощью сервиса `IServiceSelection`, а уже в этом обработчике собирают необходимую информацию.

Листинг 7.7. Использование сервиса `IServiceReference`

```
using System;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Data;
using System.Windows.Forms;
using System.Text;
using System.Collections.Generic;

namespace ReferenceServiceExample
{
    public class ReferenceServiceControl :
        System.Windows.Forms.UserControl
    {
        // Хранит отображаемые строки
        private List<string> drawLines;
```



```
// Ссылка на интерфейс сервиса ISelectionService
private ISelectionService selectionService;

public ReferenceServiceControl()
{
    // Свойства компонента
    this.BackColor = Color.White;
    this.SetStyle(ControlStyles.ResizeRedraw, true);
    this.Name = "ReferenceServiceControl";
    this.Size = new System.Drawing.Size(500, 250);

    // Инициализируем свойства
    drawLines = new List<string>();
    selectionService = null;
}

// Перекрываем свойство Site, чтобы иметь возможность
// обработать событие OnSelectionChanged
public override System.ComponentModel.ISite Site
{
    get
    {
        return base.Site;
    }
    set
    {
        // Если обработчик уже зарегистрирован, удаляем его
        if (selectionService != null)
            selectionService.SelectionChanged -=
                new EventHandler(this.OnSelectionChanged);

        // Сохраняем значение свойства Site. Оно может быть null,
        // если компонент находится не в режиме разработки
        base.Site = value;

        // Получаем интерфейс сервиса и устанавливаем обработчик
        selectionService = (ISelectionService)
            this.GetService(typeof(ISelectionService));
        if (selectionService != null)
        {
            selectionService.SelectionChanged +=
                new EventHandler(this.OnSelectionChanged);
        }
    }
}
```

```
// ОБНОВИТЬ ВИД КОМПОНЕНТА
RefreshSelection();
}
}
)

private void RefreshSelection()
{
    drawLines = new List<string>();

    if (selectionService.PrimarySelection == null)
    {
        drawLines.Add("Ни один компонент не выбран");
        this.Refresh();
        return;
    }

    drawLines.Add(string.Format("Выбран компонент {0} тип {1}",
        ((IComponent)selectionService.PrimarySelection).Site.Name,
        selectionService.PrimarySelection.GetType().FullName));

    // Получаем интерфейс сервиса
    IReferenceService rs = (IReferenceService)
        this.GetService(typeof(IReferenceService));
    if (rs != null)
    {
        // Метод GetName возвращает имя компонента
        string name = rs.GetName(selectionService.PrimarySelection);
        drawLines.Add(string.Format("GetName()={0}", name));

        // Ищем все ссылки на компоненты того же типа,
        // как у текущего выбранного
        drawLines.Add(string.Format("Компоненты такого же типа:"));
        object[] components = rs.GetReferences(
            selectionService.PrimarySelection.GetType());
        foreach (IComponent component in components)
        {
            drawLines.Add(string.Format(" {0}", component.Site.Name));
        }
    }

    this.Refresh();
}
```

```
private void OnSelectionChanged(object sender, EventArgs e)
{
    // При изменении выбранного компонента обновляем информацию
    RefreshSelection();
}

protected override void OnPaint(PaintEventArgs e)
{
    // Просто рисуем созданные строки
    int line = 5;
    foreach (string s in drawLines)
    {
        e.Graphics.DrawString(s,
            new Font(FontFamily.GenericMonospace, 9),
            new SolidBrush(Color.Blue), 5, line);
        line += 10;
    }
}
```

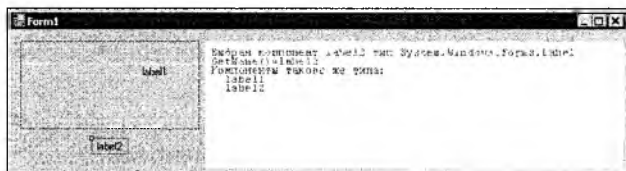


Рис. 7.2. Использование сервиса IReferenceService

Еще одно замечание. Для получения списка всех компонентов (наиболее часто возникающая задача для сервиса IReferenceService) правильно получать список с помощью вызова `GetReferences(typeof(IComponent))`, т. к. вызов этого метода без параметров возвращает не только все компоненты, но и все объекты дизайнеров.

Сервис `INameCreationService` позволяет создать имя компонента и проверить имя на правильность. Сервис имеет всего три метода:

- метод `string CreateName(IContainer container, Type dataType)` создает уникальное в рамках контейнера имя компонента;
- методы `bool IsValidName(string name)` и `void ValidateName(string name)` позволяют проверить корректность имени (листинг 7.8).

Отличие двух последних методов в том, что первый просто возвращает `false`, если имя некорректно, а второй генерирует исключение, позволяющее получить дополнительную информацию об ошибке.

Листинг 7.8. Пример реализации метода `ValidateName`

```
public void ValidateName(string name)
{
    // Проверяем каждый символ строки
    for (int i = 0; i < name.Length; i++)
    {
        UnicodeCategory uc = Char.GetUnicodeCategory(name[i]);
        switch (uc)
        {
            case UnicodeCategory.UppercaseLetter:
            case UnicodeCategory.LowercaseLetter:
            case UnicodeCategory.TitlecaseLetter:
            case UnicodeCategory.DecimalDigitNumber:
                break;
            default:
                throw new Exception(string.Format(
                    "Символ {0} в имени '{1}' не корректен.", i, name));
        }
    }
}
```

7.9. Управление событиями компонента

Управление событиями компонента производится с помощью сервиса `IEventBindingService`. Вот его основные методы:

- ❑ метод `string CreateUniqueMethodName(IComponent component, EventDescriptor e)` генерирует уникальное имя события;
- ❑ метод `ICollection GetCompatibleMethods(EventDescriptor e)` возвращает коллекцию обработчиков событий, по сигнатуре совместимых с указанным;
- ❑ метод `EventDescriptor GetEvent(PropertyDescriptor property)` возвращает дескриптор события по дескриптору свойства;
- ❑ метод `PropertyDescriptorCollection GetEventProperties(EventDescriptorCollection events)` конвертирует коллекцию дескрипторов событий в дескрипторы свойств;

- ❑ метод `PropertyDescriptor GetEventProperty(EventDescriptor e)` возвращает дескриптор свойства по дескриптору события;
- ❑ методы `bool ShowCode(int lineNumber)`, `bool ShowCode(IComponent component, EventDescriptor e)` и `bool ShowCode()` открывают редактор кода либо по указанному номеру строки, либо на описании указанного события, либо в текущем месте кода. Методы возвращают `true`, если удалось открыть редактор кода.

В качестве примера я приведу компонент `EventBindingServiceControl`, имеющий в контекстном меню специальную команду **Создать testEvent**, при выполнении которой создается обработчик события `testEvent` (листинг 7.9). В самом компоненте это событие не вызывается, но для данного примера это не важно. Для создания обработчика мы сначала получаем список всех событий компонента (переменная `edc`), затем находим дескриптор нужного нам обработчика и сохраняем его в переменной `ed`. Метод `GetEventProperty()` позволяет найти дескриптор свойства, соответствующего обработчику. После этого остается только получить уникальное имя для нового обработчика (метод `CreateUniqueMethodName()`) и создать сам обработчик в коде, т. е. установить имя обработчика метода (метод `SetValue()`). В конце метод `ShowCode()` переключает нас в редактор кода.

И еще несколько слов об уникальном имени обработчика события. При создании уникального имени используется следующий шаблон:

`<имя компонента>_<имя обработчика>[_<уникальный номер>]`

В нашем примере, если обработчик еще не создан, будет сгенерировано имя `eventControll_testEvent`. При повторном вызове имя обработчика будет иметь вид `eventControll_testEvent_1`. И т. д.

Листинг 7.9. Программное создание обработчика события

```
using System;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Data;
using System.Windows.Forms;
using System.Windows.Forms.Design;

namespace EventBindingServiceTest
{
    [Designer(typeof(EventDesigner))]
}
```

```
public class EventBindingServiceControl : UserControl
{
    // Событие, для которого будем создавать обработчик.
    // В нашем компоненте оно никогда не вызывается,
    // но в данном случае это не существенно.
    public event System.EventHandler testEvent;

    // Конструктор компонента
    public EventBindingServiceControl()
    {
        this.BackColor = Color.White;
        this.Size = new Size(100, 100);
    }
}

// Дизайнер компонента EventControl
public class EventDesigner : ControlDesigner
{
    public EventDesigner()
    {
    }

    // Обработчик команды меню "Создать testEvent"
    private void ConnectEvent(object sender, EventArgs e)
    {
        IEventBindingService eventservice =
            (IEventBindingService)this.Component.Site.GetService(
                typeof(IEventBindingService));
        if (eventservice != null)
        {
            // Получаем коллекцию дескрипторов событий
            EventDescriptorCollection edc =
                TypeDescriptor.GetEvents(this.Component);
            if (edc == null || edc.Count == 0)
                return;

            EventDescriptor ed = null;
            // Ищем событие с именем "testEvent"
            foreach (EventDescriptor edi in edc)
            {
                if (edi.Name == "testEvent")
                {
                    ed = edi;
                }
            }
        }
    }
}
```

```
        break;
    }
}

// Не нашли -- ошибка. Выходим.
if (ed == null)
    return;

// Получаем дескриптор свойства
PropertyDescriptor pd = eventservice.GetEventProperty(ed);
if (pd == null)
    return;

// Создаем уникальное имя для обработчика события
string eventName =
    eventservice.CreateUniqueMethodName(this.Component, ed);

// Создаем обработчик события с указанным именем
pd.SetValue(this.Component, eventName);

// Переключаемся в редактор кода на описание метода
eventservice.ShowCode(this.Component, ed);
}
}

// Добавляем пункт в контекстное меню компонента
public override DesignerVerbCollection Verbs
{
    get
    {
        DesignerVerbCollection dvc = new DesignerVerbCollection();
        dvc.Add(new DesignerVerb("Создать testEvent",
            new EventHandler{ConnectEvent}));
        return dvc;
    }
}
}
```

7.10. Доступ к параметрам дизайнера

Сервис `IDesignerOptionService` предоставляет доступ к параметрам дизайнера. В ручном режиме эти параметры доступны через меню **Tools | Options |**

Windows Forms Designer (рис. 7.3), а сервис позволяет читать и устанавливать их программным путем.

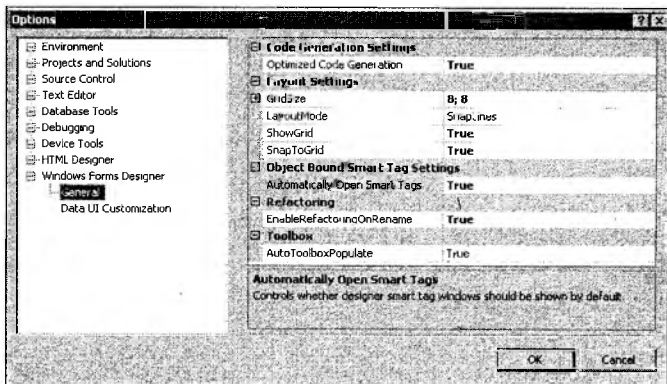


Рис. 7.3. Диалог параметров дизайнера

Сервис имеет всего два метода:

- ❑ метод `object GetOptionValue(string pageName, string valueName)` позволяет прочесть значение выбранного параметра;
- ❑ метод `void SetOptionValue(string pageName, string valueName, object value)` позволяет установить значение выбранного параметра.

Параметр `pageName` задает имя страницы диалога параметров в виде строки `WindowsFormsDesigner\\General`, а параметр `valueName` — имя самого параметра. Возвращаемый тип зависит от параметра (табл. 7.1). Вот как, например, выглядит чтение параметров `GridSize` и `ShowGrid`:

```
IDesignerOptionService designerOptionService =
    (IDesignerOptionService)this.GetService(
        typeof(IDesignerOptionService));
Size size = (Size)designerOptionService.GetOptionValue(
    "WindowsFormsDesigner\\General", "GridSize");
bool showGrid = (bool)designerOptionService.GetOptionValue(
    "WindowsFormsDesigner\\General", "ShowGrid");
```

Аналогично выполняется и установка параметров.

Таблица 7.1. Имена и типы параметров дизайнера

Имя параметра	Тип	Описание
GridSize	Size	Размер сетки дизайнера
GridSize.Width	Int32	Высота сетки дизайнера (только чтение)
GridSize.Height	Int32	Ширина сетки дизайнера (только чтение)
ShowGrid	Boolean	True, если сетка отображается
SnapToGrid	Boolean	True, если компоненты выравниваются по сетке
LayoutMode	Microsoft.VisualStudio.Windows.Forms.LayoutMode	Метод выравнивания компонентов
ObjectBoundSmartTagAutoShow	Boolean	True, если разрешается показывать смарт-теги автоматически
AutoToolboxPopulate	Boolean	True, если компоненты добавляются в палитру компонентов автоматически
UseOptimizedCodeGeneration	Boolean	True, если кэширование компонентов включено

Класс `DesignerOptionService` реализует этот интерфейс и предоставляет методы для обращения к параметрам по имени:

```
DesignerOptionService designerOptionService =
    (DesignerOptionService)this.GetService(
        typeof(IDesignerOptionService));

Size size = (Size)
    designerOptionService.Options["WindowsFormsDesigner"].
        Properties["GridSize"].GetValue(Component);
```

Класс `WindowsFormsDesignerOptionService` предоставляет типизированные пол-
л тех же параметров:

```
WindowsFormsDesignerOptionService ws =
    new WindowsFormsDesignerOptionService();

Size size = ws.CompatibilityOptions.GridSize;
```

Этот сервис используется довольно редко, но знать про него полезно.

7.11. Нотификации о смене дизайнера

Каждый раз при открытии дизайнера формы Visual Studio создает соответствующий класс дизайнера. При закрытии формы этот объект уничтожается. Кроме того, при переключении между страницами редактора текущий активный дизайнер (active designer) переключается в соответствии с выбранной страницей. Получать уведомления о смене дизайнера позволяет сервис `IDesignerEventService`, который имеет следующий список свойств и событий:

- ❑ свойство `IDesignerHost.ActiveDesigner` возвращает активного дизайнера или `null`, если активного дизайнера нет (например, фокус находится в редакторе кода);
- ❑ свойство `DesignerCollection.Designers` возвращает коллекцию дизайнеров, созданных к текущему моменту и еще не уничтоженных;
- ❑ событие `event ActiveDesignerEventHandler ActiveDesignerChanged` вызывается при смене активного дизайнера (например, при переключении из формы в код или из формы в другую форму);
- ❑ событие `event DesignerEventHandler DesignerCreated` вызывается при создании нового дизайнера (например, при открытии формы);
- ❑ событие `event EventHandler SelectionChanged` вызывается при изменении текущего выбранного компонента (см. также разд. 7.5).

При смене дизайнера в событие `ActiveDesignerChanged` передается параметр типа `ActiveDesignerEventArgs`, содержащий поля `OldDesigner` и `NewDesigner`, соответствующие предыдущему и новому дизайнеру. Стоит отдельно отметить, что при переключении из дизайнера формы в код поле `NewDesigner` будет равно `null`, а при переключении обратно пустым будет параметр `OldDesigner`.

В листинге 7.10 показан пример компонента, принимающего уведомления о смене дизайнера, а на рис. 7.4 — результат работы.

Листинг 7.10. Получение нотификаций о смене дизайнера

```
using System;
using System.Data;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;
```

```
namespace DesignerEventServiceExample
{
    public class DesignerEventComponent : UserControl
    {
        // Буфер для отображения сообщений
        private ListBox logListBox;

        // Конструктор
        public DesignerEventComponent()
        {
            InitializeComponent();
        }

        private void InitializeComponent()
        {
            this.logListBox = new System.Windows.Forms.ListBox();
            this.SuspendLayout();
            //
            // logListBox
            //
            this.logListBox.Dock = System.Windows.Forms.DockStyle.Fill;
            this.logListBox.Location = new System.Drawing.Point(0, 0);
            this.logListBox.Name = "logListBox";
            this.logListBox.Size = new System.Drawing.Size(200, 95);
            this.logListBox.TabIndex = 0;
            //
            // DesignerEventComponent
            //
            this.Controls.Add(this.logListBox);
            this.Name = "DesignerEventComponent";
            this.Size = new System.Drawing.Size(200, 100);
            this.ResumeLayout(false);
        }

        // Свойство Site устанавливается дизайнером только в режиме
        // разработки в момент установки связи между дизайнером
        // и компонентом. Именно в этот момент мы будем
        // регистрировать обработчики событий сервиса
        public override ISite Site
        {
            get
            {
                return base.Site;
            }
        }
    }
}
```

```
set
{
    // Очистить предыдущие обработчики, если они были установлены
    ClearChangeNotifications();

    // Сохраняем значение Site
    base.Site = value;

    // Регистрируем свои обработчики уведомлений
    RegisterChangeNotifications();
}
}

// Удаление обработчиков события
private void ClearChangeNotifications()
{
    // Получаем ссылку на сервис
    IDesignerEventService designerEventService =
        (IDesignerEventService)GetService(
            typeof(IDesignerEventService));

    // Если сервис не null, значит, это режим разработки
    if (designerEventService != null)
    {
        designerEventService.ActiveDesignerChanged -=
            new ActiveDesignerEventHandler(
                designerEventService_ActiveDesignerChanged);
        designerEventService.DesignerCreated -=
            new DesignerEventHandler(
                designerEventService_DesignerCreated);
        designerEventService.DesignerDisposed -=
            new DesignerEventHandler(
                designerEventService_DesignerDisposed);
    }
}

// Регистрируем обработчики событий
private void RegisterChangeNotifications()
{
    // Получаем ссылку на сервис
    IDesignerEventService designerEventService =
        (IDesignerEventService)GetService(
            typeof(IDesignerEventService));
```

```
if (designerEventService != null)
{
    designerEventService.ActiveDesignerChanged +=
        new ActiveDesignerEventHandler(
            designerEventService_ActiveDesignerChanged);
    designerEventService.DesignerCreated +=
        new DesignerEventHandler(
            designerEventService_DesignerCreated);
    designerEventService.DesignerDisposed +=
        new DesignerEventHandler(
            designerEventService_DesignerDisposed);
}
}

// Дизайнер закрыт
void designerEventService_DesignerDisposed(
    object sender, DesignerEventArgs e)
{
    DoLog("Дизайнер закрыт");
}

// Дизайнер создан
void designerEventService_DesignerCreated(
    object sender, DesignerEventArgs e)
{
    DoLog("Дизайнер создан");
}

// Дизайнер изменен
void designerEventService_ActiveDesignerChanged(
    object sender, ActiveDesignerEventArgs e)
{
    string newType = "<none>";
    if (e.NewDesigner != null)
        if (e.NewDesigner.GetType() != null)
            newType = e.NewDesigner.GetType().ToString();

    string oldType = "<none>";
    if (e.OldDesigner != null)
        if (e.OldDesigner.GetType() != null)
            oldType = e.OldDesigner.GetType().ToString();

    DoLog(string.Format("Смена с {0} на {1}", oldType, newType));
}
```

```

// Добавляем строку в журнал
private void DoLog(string text)
{
    logListBox.Items.Add(text);
    logListBox.SelectedIndex = logListBox.Items.Count - 1;
}

// Сброс обработчиков при освобождении компонента
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        ClearChangeNotifications();
    }
    base.Dispose(disposing);
}
}
}

```

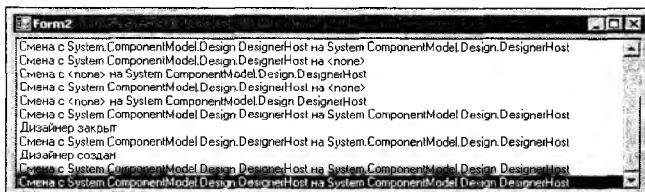


Рис. 7.4. Уведомления об изменениях дизайнера

7.12. Пользовательские данные времени разработки

Очень часто возникает задача сохранить некоторые данные, нужные только для режима разработки. Если бы такие данные являлись свойствами компонента, можно было бы использовать способ, описанный в *разд. 6.7*. Но добавление свойств времени разработки — не всегда подходящий способ. В некоторых случаях подойдет использование сервиса `IDictionaryServiceDesigner`, который предоставляет доступ к хэш-таблице данных компонента, доступных только в режиме разработки. Эта таблица хранит соответствие ключа (key) и значения (value).

Сервис имеет два основных метода:

- метод `object GetValue(object key)` возвращает значение ключа;
- метод `void SetValue(object key, object value)` устанавливает значение ключа. Если такого ключа еще нет, то он добавляется.

В листинге 7.11 показан пример компонента, который при инициализации дизайнера сохраняет некоторый набор данных в таблице данных, а при отображении считывает их. В качестве данных я добавляю типизированный массив `List<string>`, в который записано несколько строк информации.

Листинг 7.11. Хранение пользовательских данных времени разработки

```
using System;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Data;
using System.Windows.Forms;
using System.Collections.Generic;
using System.Windows.Forms.Design;

namespace DictionaryServiceControlExample
{
    [DesignerAttribute(typeof(IDictionaryServiceDesigner),
                      typeof(IDesigner))]
    public class DictionaryServiceControl : UserControl
    {
        // Хранит отображаемые строки
        private List<string> drawLines;

        public DictionaryServiceControl()
        {
            // Инициализация компонента
            drawLines = new List<string>();
            this.Size = new Size(344, 88);
            this.BackColor = Color.White;
        }

        // Рисуем данные, полученные с помощью
        // сервиса IDictionaryService
        protected override void OnPaint(PaintEventArgs e)
```

```
{
    // Получаем интерфейс сервиса IDictionaryService
    IDictionaryService ds =
        (IDictionaryService)GetService(
            typeof(IDictionaryService));
    if (ds != null)
    {
        // Получаем значение ключа "DesignerData"
        List<string> designerData =
            (List<string>)ds.GetValue("DesignerData");
        if (designerData != null)
        {
            // Просто рисуем созданные строки
            int line = 5;
            foreach (string s in designerData)
            {
                e.Graphics.DrawString(s,
                    new Font(FontFamily.GenericMonospace, 9),
                    new SolidBrush(Color.Blue), 5, line);
                line += 10;
            }
        }
        else
        {
            base.OnPaint(e);
        }
    }
}

// Дизайнер компонента
public class IDictionaryServiceDesigner : ControlDesigner
{
    public IDictionaryServiceDesigner()
    {
    }

    public override void Initialize(IComponent component)
    {
        base.Initialize(component);
        IDictionaryService ds =
            (IDictionaryService)component.Site.GetService(
                typeof(IDictionaryService));
    }
}
```



```
if (ds != null)
{
    // Создаем ключ DesignerData и привязываем
    // к нему List<string>
    if (ds.GetValue("DesignerData") == null)
    {
        ds.SetValue("DesignerData", new List<string>());
    }
    // Получаем значение DesignerData и заполняем некоторые строки
    List<string> list = (List<string>)ds.GetValue("DesignerData");
    if (list != null)
    {
        list.Clear();
        list.Add("Тип дизайнера : " + this.GetType().Name);
        list.Add("Тип компонента: " + this.Control.GetType().Name);
        list.Add("Имя компонента: " + this.Control.Name);
    }
}
```

7.13. Получение информации о проекте

Расположенный на форме компонент обладает полной информацией обо всех принадлежащих ему объектах, но иногда возникает необходимость посмотреть вверх и получить информацию о проекте, которому принадлежит форма, содержащая вставленный компонент. В этом случае можно использовать класс `ProjectItem`. Получить его экземпляр можно недокументированным вызовом метода `Site.GetService(typeof(ProjectItem))`. Например, в листинге 7.12 показан код компонента, получающего имя формы, имя документа и даже число точек останова в текущем проекте (рис. 7.5). Код здесь достаточно простой и, я думаю, в дополнительных комментариях не нуждается. Единственное замечание — для работы этого компонента требуется подключить библиотеки `EnvDTE` и `EnvDTE80`, содержащие описание класса `ProjectItem` (рис. 7.6).

Листинг 7.12. Получение информации о проекте

```
using System;
using System.Data;
using System.Drawing;
```

```
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;

using EnvDTE;
using EnvDTE80;

namespace ProjectItemControlExample
{
    public class ProjectItemControl : UserControl
    {
        // Буфер для отображения сообщений
        private ListBox logListBox;

        // Конструктор
        public ProjectItemControl()
        {
            InitializeComponent();
        }

        private void InitializeComponent()
        {
            this.logListBox = new System.Windows.Forms.ListBox();
            this.SuspendLayout();
            //
            // logListBox
            //
            this.logListBox.Dock = System.Windows.Forms.DockStyle.Fill;
            this.logListBox.Location = new System.Drawing.Point(0, 0);
            this.logListBox.Name = "logListBox";
            this.logListBox.Size = new System.Drawing.Size(200, 95);
            this.logListBox.TabIndex = 0;
            //
            // ProjectItemControl
            //
            this.Controls.Add(this.logListBox);
            this.Name = "ProjectItemControl";
            this.Size = new System.Drawing.Size(200, 100);
            this.ResumeLayout(false);
        }
    }
}
```

```
// Свойство Site устанавливается дизайнером только в режиме
// разработки в момент установки связи между дизайнером и компонентом.
public override ISite Site
{
    get
    {
        return base.Site;
    }
    set
    {
        // Сохраняем значение Site
        base.Site = value;

        // Получаем информацию
        PrepareInformation();
    }
}

// Получаем информацию
private void PrepareInformation()
{
    ClearLog();

    if (base.Site == null)
    {
        return;
    }

    // Получаем интерфейс ProjectItem
    ProjectItem pi =
        (ProjectItem) Site.GetService(typeof(ProjectItem));
    if (pi == null)
    {
        return;
    }

    DoLog(string.Format("Форма {0}", pi.Name));
    if (pi.Document != null)
        DoLog(string.Format("Имя документа {0}", pi.Document.FullName));
    if (pi.Properties.DTE.Debugger != null)
        DoLog(string.Format("Число точек останова {0}",
            pi.Properties.DTE.Debugger.Breakpoints.Count));
    DoLog(string.Format("Путь к VS {0}", pi.Properties.DTE.FileName));
}
```

```
// Добавляем строку в журнал
private void DoLog(string text)
{
    logListBox.Items.Add(text);
    logListBox.SelectedIndex = logListBox.Items.Count - 1;
}

private void ClearLog()
{
    logListBox.Items.Clear();
}
```

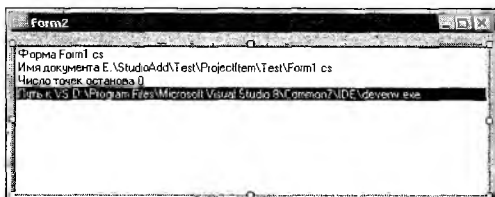


Рис. 7.5. Получение информации о проекте

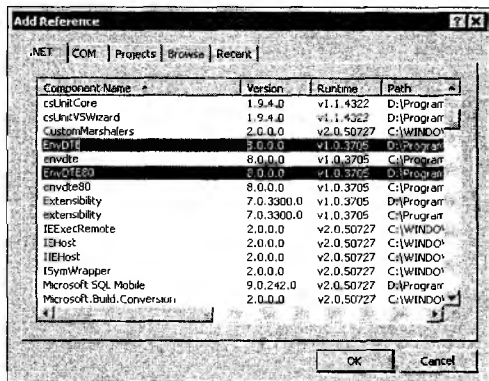


Рис. 7.6. Добавление ссылок на библиотеки EnvDTE и EnvDTE80

7.14. Вывод отладочной информации

В предыдущем разделе я выводил строки прямо в содержимое компонента *ListBox*. Для отладочных целей это неплохо, но для реального компонента такой путь не очень удачен. Гораздо лучше выводить отладочные сообщения туда, куда их положено выводить — в окно **Output** среды. Получить ссылку на это окно и вывести сообщение можно с помощью кода, показанного в листинге 7.13.

По умолчанию в Visual Studio существуют два источника отладочной информации: компилятор (*Build*) и основной (*General*). В моем коде добавляется еще один с названием *Test*. Именно в него выводится отладочная информация.

Листинг 7.13. Вывод отладочной информации в окно *Output*

```
// Получаем интерфейс ProjectItem
ProjectItem pi = (ProjectItem) Site.GetService(typeof(ProjectItem));
if (pi == null)
    return;

DTE2 dte = (DTE2) pi.DTE;

if (dte == null)
    return;

// Ссылка на окно отладочной информации
OutputWindow outputWin = dte.ToolWindows.OutputWindow;
OutputWindowPane pane;

// Получаем ссылку на источник отладочной информации.
// Если он еще не существует, создаем.
try
{
    pane = outputWin.OutputWindowPanes.Item("Test");
}
catch
{
    pane = outputWin.OutputWindowPanes.Add("Test");
}

// Активируем окно
outputWin.Parent.Activate();
pane.Activate();
```

```
// Выводим строку  
pane.OutputString(text);
```

7.15. Встроенные компоненты

В разд. 6.13 я уже рассказывал о компонентах, содержащих другие компоненты. Тогда для передачи Windows-сообщений мы использовали специальный обработчик `WndProc`. Это позволяло "выключать" режим дизайнера для внутренних компонентов, т. е. компоненты начинали работать так, как будто они работают в режиме выполнения. В этом разделе я хочу рассказать об интерфейсе `INestedContainer`, позволяющем делать внутренние компоненты фактически реальными компонентами формы.

Добавим в компонент, содержащий внутри себя `ListBox`, вызов метода `Add` класса `INestedContainer`, регистрируя, таким образом, внутренний компонент (листинг 7.14).

Полученный компонент содержит внутри себя компонент `ListBox`, который полностью поддерживается дизайнером, т. е. позволяет менять размеры, положение, запускать смарт-теги и т. п. Более того, он виден в инспекторе объектов с именем `nestedContainerControl.logListBox`, которое составляется из имени контейнера и имени компонента (рис. 7.7).

Стоит сразу отметить, что изменения, сделанные для внутренних компонентов (например, изменение компонента `ListBox`) не сохраняются. При переоткрытии формы все эти изменения пропадут. Чтобы это не случилось, нужно позаботиться о корректной сериализации свойств внутренних компонентов (см. главу 10).

Листинг 7.14. Регистрация внутреннего компонента с помощью интерфейса `INestedContainer`

```
using System;  
using System.Data;  
using System.Drawing;  
using System.Collections;  
using System.ComponentModel;  
using System.ComponentModel.Design;  
using System.Windows.Forms;  
using System.Windows.Forms.Design;  
  
namespace NestedContainerExample  
{  
    public class NestedContainerControl : UserControl
```

```
{
    // Буфер для отображения сообщений
    private ListBox logListBox;

    // Конструктор
    public NestedContainerControl()
    {
        InitializeComponent();
    }

    private void InitializeComponent()
    {
        this.logListBox = new ListBox();
        this.SuspendLayout();
        this.logListBox.Location = new System.Drawing.Point(5, 5);
        this.logListBox.Name = "logListBox";
        this.logListBox.Size = new System.Drawing.Size(345, 268);
        this.logListBox.TabIndex = 0;
        this.BackColor = System.Drawing.Color.Beige;
        this.Controls.Add(this.logListBox);
        this.Location = new System.Drawing.Point(500, 400);
        this.Name = "ComponentClass1";
        this.Size = new System.Drawing.Size(442, 422);
        this.ResumeLayout(false);
    }

    // Свойство Site устанавливается дизайнером только
    // в режиме разработки в момент установки связи между
    // дизайнером и компонентом. Именно в этот момент мы
    // будем регистрировать обработчики событий сервиса
    public override ISite Site
    {
        get
        {
            return base.Site;
        }
        set
        {
            // Сохраняем значение Site
            base.Site = value;

            // регистрация внутреннего компонента
            if (value != null)
```

```

    {
        INestedContainer nc =
            (INestedContainer)this.Site.GetService(
                typeof(INestedContainer));

        if (nc != null)
        {
            nc.Add(this.logListBox, "logListBox");
        }
    }
}
// Заполняем ListBox
for (int i = 0; i < 100; i++)
    logListBox.Items.Add(i.ToString());
}
}
}

```

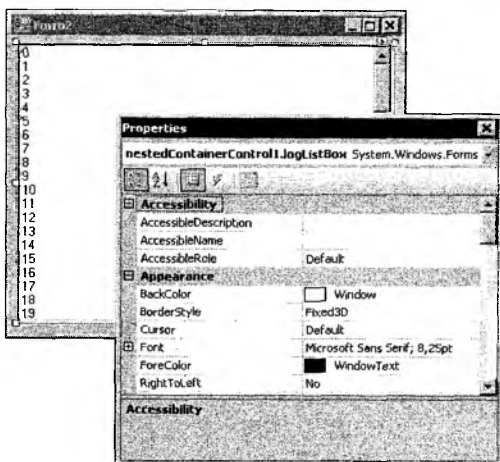


Рис. 7.7. Внутренний компонент ListBox имеет имя nestedContainerControl1.logListBox

7.16. Взаимодействие с палитрой компонентов

Само по себе взаимодействие с палитрой компонентов (**Toolbox**) обычному компоненту не требуется. Скорее всего, такая возможность нужна дизайнеру формы. Но, тем не менее, это может оказаться полезным и в других случаях. В этом разделе я расскажу о сервисе `ToolboxService`, который позволяет не только получать информацию о зарегистрированных компонентах и управлять палитрой компонентов, но и программно добавлять компоненты на форму. Сервис `ToolboxService` имеет следующие свойства и методы (приведены только самые интересные):

- свойство `CategoryNameCollection CategoryNames` возвращает коллекцию имен категорий в палитре компонентов;
- свойство `string SelectedCategory` позволяет получать и устанавливать выбранную категорию;
- метод `ToolboxItemCollection GetToolboxItems(string category)` возвращает коллекцию компонентов указанной категории;
- метод `void SetSelectedToolboxItem(ToolboxItem toolboxItem)` выбирает категорию;
- метод `ToolboxItem GetSelectedToolboxItem()` возвращает выбранный компонент.

Таким образом, категории палитры компонентов описываются именами, а каждый компонент — классом `ToolboxItem`. Этот класс содержит следующие поля:

- `AssemblyName AssemblyName` — возвращает описание сборки, содержащей компонент;
- `Bitmap Bitmap` — возвращает картинку компонента;
- `string Company` — возвращает авторскую информацию о компоненте (если она указана);
- `AssemblyName[] DependentAssemblies` — возвращает список сборок, необходимых для работы компонента;
- `string Description` — возвращает описание компонента (если оно указано);
- `string DisplayName` — возвращает имя компонента, как его видно в палитре компонентов;
- `string TypeName` — возвращает имя типа;
- `virtual string Version` — возвращает версию компонента (если она указана).

Компонент, приведенный в листинге 7.15, содержит два списка `ListBox`. В первый будет заноситься список категорий, а во второй — список компонентов категории, выбранной в первом списке. Кроме того, при выборе категории из списка категорий она будет автоматически выбираться в палитре компонентов. По двойному щелчку на строке в списке компонентов выбранный компонент будет добавляться на форму так, как будто его перетаскили с палитры компонентов. Вся описанная функциональность достаточно проста и понятна из кода (это простые вызовы методов, которые я описал ранее), кроме программного создания компонента, о котором я расскажу чуть подробнее.

Итак, выбранный компонент дает нам объект типа `ToolboxItem`. Теперь нам нужна сама форма. Получить ее можно с помощью метода `FindForm()`:

```
Form parentForm = this.FindForm();
```

Хост формы получается с помощью метода `GetService()` с параметром `IDesignerHost`:

```
IDesignerHost parentHost = (IDesignerHost)parentForm.Site.GetService(
    typeof(IDesignerHost));
```

Метод `CreateComponents()` класса `ToolboxItem` создает нужные нам компоненты:

```
components = item.CreateComponents(parentHost);
```

Для компонентов этого достаточно, а вот для визуальных элементов — нет. Для компонентов типа `Control` нужно задать их местоположение и добавить в список элементов формы. Чтобы все компоненты не создавались в одном и том же месте формы, переменная `controlSpacingMultiplier` содержит сдвиг координат, увеличивающийся при каждом добавлении визуального компонента:

```
for (int i = 0; i < components.Length; i++)
{
    // Если это видимый компонент
    if (components[i].GetType().IsSubclassOf(typeof(Control)))
    {
        // Меняем его расположение
        ((Control)components[i]).Location =
            new Point(20 * controlSpacingMultiplier,
                    20 * controlSpacingMultiplier);
        if (controlSpacingMultiplier > 10)
            controlSpacingMultiplier = 0;
        else
            controlSpacingMultiplier++;
    }
}
```

```
// Регистрируем его в списке компонентов формы
parentForm.Controls.Add((Control)components[1]);
}
}
```

Вид получившегося компонента показан на рис. 7.8. Слева видны четыре последовательно созданных компонента типа CheckBox.

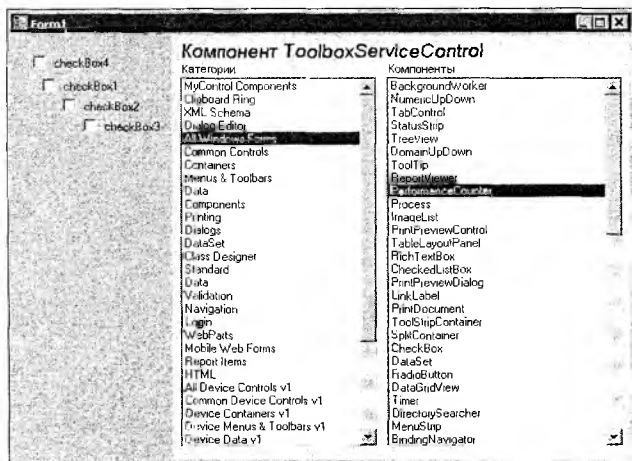


Рис. 7.8. Компонент ToolboxServiceControl

Листинг 7.15. Взаимодействие с палитрой компонентов

```
using System;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Drawing.Design;
using System.Data;
using System.Diagnostics;
using System.Windows.Forms;
using System.Windows.Forms.Design;
```

```
namespace ToolboxServiceExample
{
    [DesignerAttribute(typeof(WindowMessageDesigner),
                      typeof(IDesigner))]
    public class ToolboxServiceControl : UserControl
    {
        // Для хранения списка категорий
        private ListBox categoryNames;
        // Для хранения списка компонентов
        private ListBox componentNames;
        // Интерфейс сервиса IToolboxService
        private IToolboxService toolboxService = null;
        // Сдвиг координат. Используется для
        // добавления компонентов на форму.
        private int controlSpacingMultiplier;

        // Инициализация
        public ToolboxServiceControl()
        {
            InitializeComponent();
            controlSpacingMultiplier = 0;
        }

        // Свойство Site устанавливается не равным null,
        // если мы в режиме разработки
        public override ISite Site
        {
            get
            {
                return base.Site;
            }
            set
            {
                base.Site = value;

                // В режиме разработки получаем интерфейс сервиса
                if (base.Site != null)
                {
                    toolboxService = (IToolboxService)this.GetService(
                        typeof(IToolboxService));
                    // Если удалось получить сервис, получаем список категорий
                    // и компонентов в категории
                    if (toolboxService != null)

```

```
{
    UpdateCategoryLists();
}
else
{
    toolboxService = null;
}
}
}

// Обновляем список категорий
private void UpdateCategoryLists()
{
    try
    {
        categoryNames.BeginUpdate();

        if (toolboxService != null)
        {
            // Повторно создаем список категорий
            categoryNames.Items.Clear();
            for (int i = 0; i < toolboxService.CategoryNames.Count; i++)
            {
                categoryNames.Items.Add(
                    toolboxService.CategoryNames[i]);

                // Если это текущая выбранная категория
                if (toolboxService.CategoryNames[i] ==
                    toolboxService.SelectedCategory)
                {
                    // Сохраняем ее и выбираем ее в listbox
                    categoryNames.SelectedIndex = i;
                    // Создаем список компонентов в ней
                    UpdateComponentList();
                }
            }
        }
    }
    finally
    {
        categoryNames.EndUpdate();
    }
}
```

```
// Обновляем список компонентов в выбранной категории
private void UpdateComponentList()
{
    try
    {
        componentNames.BeginUpdate();

        componentNames.Items.Clear();

        ToolboxItemCollection tools = toolboxService.GetToolboxItems(
            toolboxService.SelectedCategory);
        if (tools != null)
        {
            for (int j = 0; j < tools.Count; j++)
            {
                componentNames.Items.Add(tools[j].DisplayName);
            }
        }
    }
    finally
    {
        componentNames.EndUpdate();
    }
}

// Если выбрали новую категорию,
// обновляем список компонентов в ней
private void UpdateSelectedCategory(
    object sender, System.EventArgs e)
{
    if (toolboxService != null)
    {
        // Меняем текущую выбранную категорию
        // в палитре компонентов
        toolboxService.SelectedCategory =
            (string)categoryNames.SelectedItem;
        UpdateComponentList();
    }
}

// Выбрали другой компонент,
// обновляем текущий компонент в палитре компонентов
private void UpdateSelectedItem(object sender,
    System.EventArgs e)
```

```
(
    if (toolboxService != null)
    {
        // Если есть выбранная категория
        if (categoryNames.SelectedIndex != -1)
        {
            // Компонент в текущей категории
            if ((string)categoryNames.SelectedItem ==
                toolboxService.SelectedCategory)
            {
                ToolboxItemCollection tools =
                    toolboxService.GetToolboxItems(
                        toolboxService.SelectedCategory);
                toolboxService.SetSelectedToolboxItem(
                    tools[componentNames.SelectedIndex]);
            }
            //else
            // Обновляем список категорий
            // UpdateCategoryLists();
        }
    }
}

// Двойной щелчок на имени компонента создает компонент
private void CreateComponent(object sender, EventArgs e)
{
    // Текущий компонент в палитре компонентов
    ToolboxItem item = toolboxService.GetSelectedToolboxItem();

    // Ничего не выбрано -- выход
    if (item == null)
        return;

    // Получаем форму
    Form parentForm = this.FindForm();
    if (parentForm == null)
        return;

    // Получаем хост формы
    IDesignerHost parentHost =
        (IDesignerHost)parentForm.Site.GetService(
            typeof(IDesignerHost));

    if (parentHost == null)
        return;
}
```

```
// Создаем компонент
IComponent[] components = null;
try
{
    components = item.CreateComponents(parentHost);
}
catch (Exception ex)
{
    MessageBox.Show(ex.ToString(), "Ошибка");
}

// Не получилось создать - выходим
if (components == null)
    return;

// Располагаем видимые компоненты на форме
for (int i = 0; i < components.Length; i++)
{
    // Если это видимый компонент
    if (components[i].GetType().IsSubclassOf(typeof(Control)))
    {
        // Меняем его расположение
        ((Control)components[i]).Location =
            new Point(20 * controlSpacingMultiplier,
                20 * controlSpacingMultiplier);
        if (controlSpacingMultiplier > 10)
            controlSpacingMultiplier = 0;
        else
            controlSpacingMultiplier++;
        // Регистрируем его в списке компонентов формы
        parentForm.Controls.Add((Control)components[i]);
    }
}

// Показать строки
protected override void OnPaint(PaintEventArgs e)
{
    SolidBrush blackBrush = new SolidBrush(Color.Black);
    e.Graphics.DrawString("Компонент ToolboxServiceControl",
        new Font("Arial", 14), blackBrush, 6, 4);
    e.Graphics.DrawString("Категории",
        new Font("Arial", 8), blackBrush, 8, 26);
}
```



```
e.Graphics.DrawString("Компоненты",
    new Font("Arial", 8), blackBrush, 208, 26);
}

private void InitializeComponent()
{
    this.categoryNames = new List<string>();
    this.componentNames = new List<string>();
    this.SuspendLayout();
    this.categoryNames.Location = new System.Drawing.Point(8, 41);
    this.categoryNames.Name = "categoryNames";
    this.categoryNames.Size = new System.Drawing.Size(192, 368);
    this.categoryNames.TabIndex = 0;
    this.categoryNames.SelectedIndexChanged +=
        new System.EventHandler(this.UpdateSelectedCategory);
    this.componentNames.Location = new System.Drawing.Point(208, 41);
    this.componentNames.Name = "componentNames";
    this.componentNames.Size = new System.Drawing.Size(228, 368);
    this.componentNames.TabIndex = 3;
    this.componentNames.SelectedIndexChanged +=
        new System.EventHandler(this.UpdateSelectedItem);
    this.componentNames.DoubleClick +=
        new EventHandler(this.CreateComponent);
    this.BackColor = System.Drawing.Color.Beige;
    this.Controls.AddRange(new Control[] {
        this.componentNames,
        this.categoryNames});
    this.Location = new System.Drawing.Point(500, 400);
    this.Name = "ToolboxServiceControl";
    this.Size = new System.Drawing.Size(442, 422);
    this.ResumeLayout(false);
}

// Специальный дизайнер позволяет работать с двумя List<string>
// внутри компонента как в режиме выполнения,
// например, прокручивать элементы
public class WindowMessageDesigner : ControlDesigner
{
    public WindowMessageDesigner()
    {
    }
}
```

```
// Передаем сообщения Windows в дизайнер
protected override void WndProc(ref Message message)
{
    if (message.HWnd == this.Control.Handle)
        base.WndProc(ref message);
    else
        this.DefWndProc(ref message);
}
}
```

7.17. Контекстная подсказка компонента

Конечно, не все программисты, использующие ваш компонент, смогут догадаться о назначении его свойств и методов. Было бы хорошо добавить возможность запуска подсказки по нажатию клавиши <F1>. Для этих целей используется сервис `IHelpService`:

- ❑ метод `void AddContextAttribute(string name, string value, HelpKeywordType keywordType)` добавляет ключевое слово контекстной подсказки;
- ❑ метод `void RemoveContextAttribute(string name, string value)` удаляет ключевое слово контекстной подсказки;
- ❑ метод `void ClearContextAttributes()` удаляет все добавленные ключевые слова контекстной подсказки;
- ❑ метод `void ShowHelpFromKeyword(string helpKeyword)` отображает контекстную подсказку для указанного ключевого слова;
- ❑ метод `void ShowHelpFromUrl(string helpUrl)` отображает контекстную подсказку, расположенную по указанному адресу.

Тип контекстной подсказки `HelpKeywordType` может принимать несколько значений, но нас пока интересует только одно — добавление ключевого слова для подсказки, запускаемой по клавише <F1>. Это значение равно `HelpKeywordType.F1Keyword`.

В листинге 7.16 показан небольшой пример использования этого сервиса. Компонент `HelpServiceControl` позволяет выполнить три команды: добавить ключевое слово, удалить ключевое слово и показать подсказку. В данном случае в качестве ключевого слова я использую имя сервиса, соответственно, при нажатии на клавишу <F1> откроется справка MSND, рассказывающая про используемый нами сервис. Ну, конечно, если справка доступна и ключевое слово добавлено.

Листинг 7.16. Управление контекстной подсказкой

```
using System;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.Windows.Forms.Design;

namespace HelpServiceSample
{
    [Designer(typeof(HelpDesigner))]
    public class HelpServiceControl : UserControl
    {
        public HelpServiceControl()
        {
            this.Size = new Size(320, 100);
            this.BackColor = Color.White;
        }
    }

    // Дизайнер для управления подсказкой
    public class HelpDesigner : ControlDesigner
    {
        public HelpDesigner()
        {
        }

        // Добавляем два меню в контекстное меню компонента
        public override DesignerVerbCollection Verbs
        {
            get
            {
                return new DesignerVerbCollection(new DesignerVerb[] {
                    new DesignerVerb("Добавить слово IHelpService",
                        new EventHandler(this.AddKeyword)),
                    new DesignerVerb("Удалить слово IHelpService",
                        new EventHandler(this.RemoveKeyword)),
                    new DesignerVerb("Показать подсказку",
                        new EventHandler(this.ShowKeyword))
                });
            }
        }
    }
}
```

```
// Добавить ключевое слово
private void AddKeyword(object sender, EventArgs e)
{
    // Получаем интерфейс сервиса IHelpService
    IHelpService hs =
        (IHelpService)this.Control.Site.GetService(
            typeof(IHelpService));

    // Добавляем ключевое слово
    hs.AddContextAttribute("keyword", "IHelpService",
        HelpKeywordType.FlKeyword);
}

// Удалить ключевое слово
private void RemoveKeyword(object sender, EventArgs e)
{
    // Получаем интерфейс сервиса IHelpService
    IHelpService hs =
        (IHelpService)this.Control.Site.GetService(
            typeof(IHelpService));

    // Удаляем ключевое слово
    hs.RemoveContextAttribute("keyword", "IHelpService");
}

// Показать подсказку к слову IHelpService
private void ShowKeyword(object sender, EventArgs e)
{
    // Получаем интерфейс сервиса IHelpService
    IHelpService hs =
        (IHelpService)this.Control.Site.GetService(
            typeof(IHelpService));

    // Показать подсказку
    hs.ShowHelpFromKeyword("IHelpService");
}
```

7.18. Добавление и удаление свойств, событий и атрибутов компонентов

В разд. 6.6 я показывал, как делать фильтрацию свойств и событий с помощью методов дизайнера. Те же действия можно выполнить, зарегистрировав дополнительный сервис фильтра `ITypeDescriptorFilterService`. В отли

чие от предыдущих сервисов, сервисы фильтров организуют цепочку, последовательно вызывая друг друга. Соответственно, при регистрации сервиса мы должны позаботиться, чтобы новый фильтр вызывал предыдущий. Для этого в переменную `oldService` мы сохраним ссылку на предыдущий сервис (листинг 7.17) и будем использовать ее при вызове предыдущего фильтра:

```
public bool FilterProperties(IComponent component,
                           IDictionary properties)
{
    if (oldService != null) // Вызываем предыдущий фильтр
        oldService.FilterProperties(component, properties);
}
```

Основное отличие этого кода от использования методов дизайнера заключается в том, что фильтр является "глобальным" фильтром в рамках дизайнера. Другими словами, его методы вызываются для всех компонентов формы. Обратите внимание на проверку, которую приходится делать внутри метода `FilterProperties()`:

```
if (component is GradientLabel)
{
    properties.Remove("BackColor");
}
```

Таким образом, мы удаляем свойство `BackColor` только для компонента `GradientLabel` (если помните, мы использовали этот компонент для тестов в главе 6). Если эту проверку убрать, то свойство исчезнет у всех компонентов формы.

С помощью фильтра можно добавлять свойства компонента (аналогично как мы это делали в разд. 6.7). Для этого нужно создать дескриптор свойства и вызвать метод `Add()` (листинг 7.18). Конечно, сначала следует убедиться, что такое свойство еще не было добавлено. Кстати, заметьте, что я добавляю свойство с именем `TestProp`, которое отражается на свойство `Test` в дизайнера. Само свойство `Test` ничего не делает, но с его помощью можно менять свойства компонента, используя метод `GetPropertyByName()` (см. листинг 7.1).

Вообще говоря, сервис позволяет не только добавлять и удалять свойства, но и подменять базовые свойства. Для этого достаточно удалить оригинальное свойство компонента, а затем добавить его, как я описывал ранее.

Листинг 7.17. Добавление фильтра `ITypedDescriptorFilterService`

```
using System;
using System.Collections;
using System.Collections.Generic;
```

```
using System.Text;
using System.Windows.Forms.Design;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.ComponentModel.Design;

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        // Хранит ссылку на предыдущий объект сервиса
        private ITypeDescriptorFilterService oldservice = null;

        public override void Initialize(IComponent component)
        {
            base.Initialize(component);
            LoadTypeDescriptorFilterService();
        }

        protected override void PostFilterProperties(IDictionary properties)
        {
            base.PostFilterProperties(properties);
        }

        // Удаляем сервис при освобождении дизайнера
        protected override void Dispose(bool disposing)
        {
            RemoveTypeDescriptorFilterService();
        }

        // Загрузить MyDesignerFilterService
        private void LoadTypeDescriptorFilterService()
        {
            // Сохраняем текущий фильтр
            ITypeDescriptorFilterService tdfs =
                (ITypeDescriptorFilterService)
                    this.Component.Site.GetService(
                        typeof(ITypeDescriptorFilterService));
            oldservice = tdfs;
            // Получаем IDesignerHost для управления сервисами
            IDesignerHost dh =
                (IDesignerHost) this.Component.Site.GetService(
                    typeof(IDesignerHost));
        }
    }
}
```

```
// Удаляем текущий ITypeDescriptorFilterService
dh.RemoveService(typeof(ITypeDescriptorFilterService));
// Добавляем свой ITypeDescriptorFilterService
dh.AddService(typeof(ITypeDescriptorFilterService),
               new MyDesignerFilterService(oldservice));
}

// Выгрузить MyDesignerFilterService
private void RemoveTypeDescriptorFilterService()
{
    // Получаем IDesignerHost для управления сервисами
    IDesignerHost dh = (IDesignerHost)this.GetService(
        typeof(IDesignerHost));

    // Удаляем свой сервис
    dh.RemoveService(typeof(ITypeDescriptorFilterService));
    // Если предыдущий существовал, восстанавливаем его
    if (oldservice != null)
        dh.AddService(typeof(ITypeDescriptorFilterService),
                       oldservice);
}
}

public class MyDesignerFilterService : ITypeDescriptorFilterService
{
    ITypeDescriptorFilterService oldService = null;

    public MyDesignerFilterService(
        ITypeDescriptorFilterService oldService)
    {
        this.oldService = oldService;
    }

    public bool FilterAttributes(IComponent component,
                                IDictionary attributes)
    {
        if (oldService != null)
            oldService.FilterAttributes(component, attributes);
        return true;
    }

    public bool FilterEvents(IComponent component,
                             IDictionary events)
```

```

{
    if (oldService != null)
        oldService.FilterEvents(component, events);

    // Фильтруем обработчики событий
    if (component is GradientLabel)
    {
        events.Remove("Click");
    }

    return true;
}

public bool FilterProperties(IComponent component,
                           IDictionary properties)
{
    if (oldService != null)
        oldService.FilterProperties(component, properties);

    // Фильтруем свойства
    if (component is GradientLabel)
    {
        properties.Remove("BackColor");
    }

    return true;
}
}
}

```

Листинг 7.18. Добавление свойства

```

namespace MyControl
{
    class GradientLabelDesigner : ControlDesigner
    {
        ... ..
        private int test;
        public int Test
        {
            get { return test; }
            set { test = value; }
        }
    }
}

```



```
public class MyDesignerFilterService : ITypeDescriptorFilterService
{
    ... ..
    public bool FilterProperties(IComponent component,
                               IDictionary properties)
    {
        if (oldService != null)
            oldService.FilterProperties(component, properties);

        // Фильтруем свойства
        if (component is GradientLabel)
        {
            properties.Remove("BackColor");

            if (properties["TestProp"] == null)
            {
               PropertyDescriptor pd =
                    TypeDescriptor.CreateProperty(
                        typeof(GradientLabelDesigner), "Test", typeof(int));
                properties.Add("TestProp", pd);
            }
        }
        return true;
    }
}
```

7.19. Работа с файлами ресурсов

Для компонентов, предназначенных работать с несколькими языками, удобно использовать сервис `IResourceService`, предоставляющий доступ к ресурсным файлам. Сервис имеет два метода:

- метод `GetResourceReader(CultureInfo info)` возвращает интерфейс `IResourceReader`, позволяющий читать файл ресурсов для указанной культуры;
- метод `GetResourceWriter(CultureInfo info)` возвращает интерфейс `IResourceWriter`, позволяющий записывать файлы ресурсов для указанной культуры.

Основная идея переключения языка компонента заключается в хранении всех строк в ресурсных файлах. При выборе языка выбирается соответствующий ресурсный файл, и все строки "переводятся" на этот язык. Единственное,

о чем стоит позаботиться, — это чтобы строки корректно вписывались в отведенное для них место, т. к. длины строк на различных языках будут разными.

Файл ресурса можно создать следующим образом:

```
// Получаем интерфейс сервиса
IResourceService rs =
    (IResourceService) this.Component.Site.GetService(
        typeof(IResourceService));
// Получаем интерфейс IResourceWriter для текущего языка
IResourceWriter rw = rs.GetResourceWriter(CultureInfo.CurrentUICulture);
// Записываем строки в ресурс
rw.AddResource("string1", "строка-1");
rw.AddResource("string2", "строка-2");
rw.AddResource("string3", "строка-3");
// Генерируем ресурсный файл
rw.Generate();
// Освобождаем объекты
rw.Close();
```

Чтение производится с помощью интерфейса `IDictionaryEnumerator`, который возвращает метод `GetEnumerator()` интерфейса `IResourceReader`. К вопросу локализации мы еще вернемся в *разд. 10.6* и *12.7*.

7.20. Создание диалоговых окон в режиме разработки

Сервис `IUIService` позволяет создавать диалоговые окна в режиме разработки:

- ❑ методы `void ShowMessage(string message)`, `void ShowMessage(string message, string caption)` и `DialogResult ShowMessage(string message, string caption, MessageBoxButtons buttons)` отображают диалоговое сообщение, как на рис. 7.9. Последний метод позволяет управлять количеством кнопок диалога;
- ❑ методы `void ShowError(Exception ex)`, `void ShowError(string message)` и `void ShowError(Exception ex, string message)` отображают сообщение об ошибке, как на рис. 7.10. Сообщение об ошибке может задаваться либо просто строкой, либо с помощью передачи исключения;
- ❑ метод `DialogResult ShowDialog(Form form)` отображает форму в режиме дизайнера.

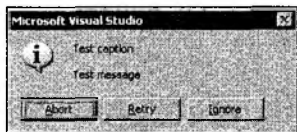


Рис. 7.9. Диалоговое окно с тремя кнопками

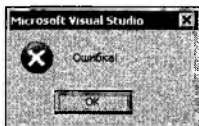


Рис. 7.10. Сообщение об ошибке

Перечисление `MessageBoxButtons` позволяет задавать следующие конфигурации кнопок, состав которых легко угадать из названий: `OK`, `OKCancel`, `AbortRetryIgnore`, `YesNoCancel`, `YesNo`, `RetryCancel`.

Кстати, параметр `caption` вовсе не задает заголовок окна, как можно было подумать. Он задает значение первой строки диалога (см. рис. 7.9).

Метод `ShowDialog()` выводит пользовательскую форму в режиме дизайнера. Стандартный код, отображающий форму в обоих режимах, выглядит примерно так:

```
// Пытаемся получить интерфейс сервиса
IUIService uiService =
    this.GetService(typeof(IUIService)) as IUIService;
using (MyForm dlg = new MyForm())
{
    // Будет хранить результат диалога
    DialogResult dlgResult = DialogResult.OK;
    // Если сервис существует, значит, мы в режиме разработки.
    // Используем сервис для отображения диалога.
    if (uiService != null)
        dlgResult = uiService.ShowDialog(dlg);
    else
        // Мы в режиме выполнения.
        // Просто показываем диалог.
        dlgResult = dlg.ShowDialog();

    if (dlgResult == DialogResult.OK)
    {
    }
}
```

В листинге 7.19 представлен небольшой пример, демонстрирующий использование сервиса `IUIService`. Контекстное меню компонента `IUIServiceExampleControl` содержит три пункта, позволяющие показать сообщение с тремя кнопками (см. рис. 7.9), сообщение об ошибке (см. рис. 7.10) и

форму ExampleForm (рис. 7.11). Код самой формы я не привожу, т. к. это просто произвольная форма. Интересующиеся могут найти полный код на компакт-диске.

Листинг 7.19. Диалоговые окна в режиме разработки

```
using System;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.Windows.Forms.Design;

namespace MyControl
{
    // Компонент, использующий дизайнер IUIServiceTestDesigner
    [DesignerAttribute(typeof(IUIServiceTestDesigner),
                      typeof(IDesigner))]
    public class IUIServiceExampleControl : UserControl
    {
        public IUIServiceExampleControl()
        {
            this.BackColor = Color.Beige;
            this.Width = 255;
            this.Height = 60;
        }
    }

    // Дизайнер показывает работу сервиса IUIService
    // с помощью команд контекстного меню
    public class IUIServiceTestDesigner : ControlDesigner
    {
        public IUIServiceTestDesigner()
        {
        }

        // Контекстное меню для демонстрации
        // возможностей сервиса IUIService
        public override DesignerVerbCollection Verbs
        {
            get
            {
                DesignerVerbCollection coll = new DesignerVerbCollection();
```

```
coll.Add(new DesignerVerb(
    "Тестовое сообщение в окне с тремя кнопками",
    new EventHandler(this.ShowTestMessage)));
coll.Add(new DesignerVerb(
    "Сообщение об ошибке",
    new EventHandler(this.ShowErrorMessage)));
coll.Add(new DesignerVerb(
    "Отображение формы",
    new EventHandler(this.ShowDialog)));
return coll;
}
}

// Показывает сообщение в окне с тремя кнопками
// Abort, Retry, Ignore
private void ShowTestMessage(object sender, EventArgs e)
{
    IUIService service = (IUIService)this.GetService(
        typeof(IUIService));
    if (service != null)
        service.ShowMessage("Test message", "Test caption",
            MessageBoxButtons.AbortRetryIgnore);
}

// Показывает окно сообщения об ошибке
private void ShowErrorMessage(object sender, EventArgs e)
{
    IUIService service = (IUIService)this.GetService(
        typeof(IUIService));
    if (service != null)
        service.ShowError(new Exception("Ошибка!"));
}

// Показывает форму ExampleForm
private void ShowDialog(object sender, EventArgs e)
{
    IUIService service = (IUIService)this.GetService(
        typeof(IUIService));
    if (service != null)
        service.ShowDialog(new ExampleForm());
}
}
```

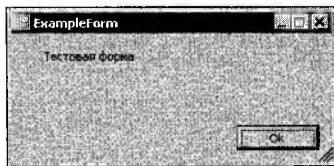


Рис. 7.11. Вывод формы

7.21. Управление окнами Visual Studio

Еще одно применение сервиса `IUIServiceTestDesigner` — управление окнами Visual Studio. Метод

```
bool ShowToolWindow(Guid toolWindow)
```

позволяет отобразить одно из окон, описанных в перечислении `StandardToolWindows`: `ObjectBrowser`, `OutputWindow`, `ProjectExplorer`, `PropertyBrowser`, `RelatedLinks`, `ServerExplorer`, `TaskList`, `Toolbox`.

В листинге 7.20 показан пример, открывающий окно списка задач (task list).

Листинг 7.20. Управление окнами Visual Studio

```
using System;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.Windows.Forms.Design;

namespace MyControl
{
    // Компонент, использующий дизайнер IUIServiceTestDesigner
    [DesignerAttribute(typeof(IUIServiceTestDesigner),
        typeof(IDesigner))]
    public class IUIServiceExampleControl : UserControl
    {
        public IUIServiceExampleControl()
        {
            this.BackColor = Color.Beige;
            this.Width = 255;
        }
    }
}
```

```
        this.Height = 60;
    }
}

// Дизайнер показывает работу сервиса IUIService
// с помощью команд контекстного меню
public class IUIServiceTestDesigner : ControlDesigner
{
    public IUIServiceTestDesigner()
    {
    }

    // Контекстное меню для демонстрации
    // возможностей сервиса IUIService
    public override DesignerVerbCollection Verbs
    {
        get
        {
            DesignerVerbCollection coll = new DesignerVerbCollection();
            coll.Add(new DesignerVerb(
                "Показать окно Task List",
                new EventHandler(this.ShowToolWindow)));
            return coll;
        }
    }

    // Показывает окно Task List
    private void ShowToolWindow(object sender, EventArgs e)
    {
        IUIService service = (IUIService)this.GetService(
            typeof(IUIService));

        if (service != null)
            UIService.ShowToolWindow(StandardToolWindows.TaskList);
    }
}
```

7.22. Сервис *IPropertyValueUIService*

Сервис *IPropertyValueUIService* предоставляет возможность отобразить рядом с именем свойства специальный значок (и даже не один), задать обработчик двойного щелчка на этом значке и установить всплывающую подсказку.

Это дает дополнительные возможности редактору свойств. Например, я покажу, как использовать этот сервис для отображения ошибочных значений.

Пусть свойство `Test` компонента `PropertyUIComponent` (листинг 7.21) должно принимать только положительные значения. В более сложных случаях можно добавить условие, что отрицательные значения считаются ошибочными, в зависимости от других свойств компонента. Но для примера нам достаточно простого условия.

Как обычно, интерфейс сервиса мы получаем в аксессоре `set` свойства `Site`. Этот метод вызывается только в режиме разработки, что позволяет избегать некорректных действий в режиме выполнения. Если значение свойства `Site` не равно `null`, мы устанавливаем делегат сервиса, вызывая метод `AddPropertyValueUIHandler()`. Если же `Site` равно `null`, то делегат удаляется с помощью вызова метода `RemovePropertyValueUIHandler()`.

Делегат, зарегистрированный в сервисе, вызывается для каждого свойства компонента всякий раз, когда нужно отобразить свойство (первый раз или при изменении значения). Разумеется, все самые интересные вещи происходят в самом обработчике, который имеет три параметра:

```
private void testPropertyValueUIHandler(  
    ITypeDescriptorContext context,  
    PropertyDescriptor propDesc,  
    ArrayList itemList)
```

С помощью первого параметра можно получить ссылку на сам компонент и, соответственно, к значениям всех свойствам компонента:

```
PropertyUIComponent component =  
    (PropertyUIComponent) context.Instance;
```

Второй параметр передает дескриптор свойства, для которого вызывается делегат. Третий параметр передает список значков, добавляемых к свойству. Каждый значок описывается классом `PropertyValueUIItem`.

Конструктор класса `PropertyValueUIItem` имеет три параметра: картинка, делегат, вызываемый при двойном щелчке на картинке, и подсказка, отображаемая при подведении на картинку курсора мыши:

```
new PropertyValueUIItem(img,  
    new PropertyValueUIItemInvokeHandler(this.testInvoke),  
    "Необходимо значение большее нуля")
```

Картинку мы загрузим из ресурсного файла:

```
Image img = new Bitmap(typeof(Resource), "Error.bmp");
```


Делегат, вызываемый по двойному щелчку, в нашем примере содержит очень простой код, инвертирующий знак значения свойства:

```
int value = (int)propDesc.GetValue(this);  
propDesc.SetValue(this, -value);
```

Сам делегат имеет три параметра: контекст `context`, дескриптор свойства `propDesc` и ссылку на картинку `item`. С помощью ссылки на компонент `context.Instance` можно проводить более сложный анализ свойств или изменять любые другие свойства, отличные от того, на котором был произведен щелчок.

Получившийся компонент имеет свойство `Test`. Если установить значение этого свойства меньше или равное нулю, то рядом с именем свойства будет показан красный восклицательный знак, означающий, что свойство имеет некорректное значение (рис. 7.12). Эту функциональность можно использовать не только при отображении ошибочных значений, но и для отображения любой другой информации.

Листинг 7.21. Использование сервиса `IPropertyValueUIService` для отображения ошибочных значений

```
using System;  
using System.Collections;  
using System.ComponentModel;  
using System.ComponentModel.Design;  
using System.Drawing;  
using System.Drawing.Design;  
using System.IO;  
using System.Runtime.Serialization;  
using System.Runtime.Serialization.Formatters.Binary;  
using System.Windows.Forms;  
using System.Windows.Forms.Design;  
using System.Resources;  
using MyControl;  
  
namespace PropertyValueUIServiceExample  
{  
    // Тестовый компонент  
    public class PropertyUIComponent : UserControl  
    {  
        // Тестовое свойство. С помощью сервиса IPropertyValueUIService  
        // подключается изображение, подсказка и реакция на двойной  
        // щелчок на картинке. Но только если значение  
        // свойства меньше нуля.
```

```

private int test = 10;
[Description("Значения >0 верны. Значения <= 0 ошибочны!")]
public int Test
{
    get
    {
        return test;
    }
    set
    {
        test = value;
    }
}

// Делегат, который вызывает сервис.
// Делегат вызывается для каждого свойства компонента.
private void testPropertyValueUIHandler(
    ITypeDescriptorContext context,
    PropertyDescriptor propDesc, ArrayList itemList)
{
    // Ссылка на компонент
    PropertyUIComponent component =
        (PropertyUIComponent)context.Instance;

    // Если это свойство Test
    if (propDesc.DisplayName.Equals("Test"))
    {
        if ((int)propDesc.GetValue(this) <= 0)
        {
            // Загружаем картинку из ресурсов
            Image img = new Bitmap(typeof(Resource), "Error.bmp");
            // Устанавливаем картинку и обработчик двойного щелчка
            itemList.Add(new PropertyValueUIItem(img,
                new PropertyValueUIItemInvokeHandler(
                    this.testInvoke),
                    "Необходимо значение большее нуля"));
        }
    }
}

// Вызывается при двойном щелчке на картинке
private void testInvoke(ITypeDescriptorContext context,
    PropertyDescriptor propDesc,
    PropertyValueUIItem item)

```

```
(
    if (MessageBox.Show("Скорректировать знак числа?",
        "Ошибка", MessageBoxButtons.YesNo) ==
        DialogResult.Yes)
    {
        // Меняем знак значения свойства
        int value = (int)propDesc.GetValue(this);
        // Устанавливаем новое значение
        propDesc.SetValue(this, -value);
    }
)

// Установка сервиса. Вызывается только в режиме разработки.
public override ISite Site
{
    get
    {
        return base.Site;
    }

    set
    {
        // Если Site != null, устанавливаем обработчик
        if (value != null)
        {
            base.Site = value;
            // Получить ссылку на сервис
            IPropertyValueUIService uiService =
                (IPropertyValueUIService)
                this.GetService(typeof(IPropertyValueUIService));
            if (uiService != null)
            {
                // Установить обработчик
                uiService.AddPropertyValueUIHandler(
                    new PropertyValueUIHandler(
                        this.testPropertyValueUIHandler));
            }
        }
        // Если Site равно null, удаляем обработчик
        else
        {
            // Получить ссылку на сервис
            IPropertyValueUIService uiService =
                (IPropertyValueUIService)
                this.GetService(typeof(IPropertyValueUIService));
```


Код добавления специальной области очень простой:

```
private object splitControl;

public override ISite Site
{
    get
    {
        return base.Site;
    }
    set
    {
        base.Site = value;

        Type splitType = Type.GetType("System.Windows.Forms.
            Design.ISplitWindowService,System.Design");
        object split = GetService(splitType);

        if (value != null)
        {
            Label label = new Label();
            label.Text = "Метка в split-окне";
            splitControl = label;

            MethodInfo mi = splitType.GetMethod("AddSplitWindow",
                BindingFlags.Instance | BindingFlags.Public);
            mi.Invoke(split, new object[] { splitControl });
        }
        else
        {
            MethodInfo mi = splitType.GetMethod("RemoveSplitWindow",
                BindingFlags.Instance | BindingFlags.Public);
            mi.Invoke(split, new object[] { splitControl });
        }
    }
}
```

Как я уже сказал, сервис является внутренним, поэтому приходится обращаться к нему с помощью методов отражения. В моем коде в дополнительную область добавляется компонент типа `Label` с некоторым текстом (рис. 7.13). Вообще говоря, можно добавить в нее любой визуальный компонент.

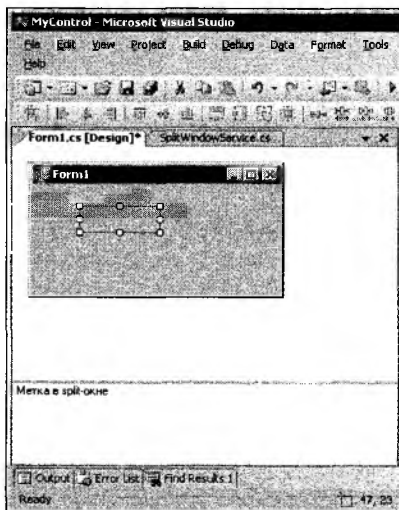
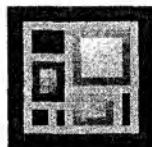


Рис. 7.13. Дополнительное окно с информацией

Еще раз напоминаю, что сервис недокументирован, хотя и используется самой средой Visual Studio.

ГЛАВА 8



Конвертеры типов

Не все так сложно, как кажется, но и не так просто, как хочется!

Посмотрите на свойство `Location` в редакторе свойств (рис. 8.1). В отличие от обычных свойств оно отображается в виде сложного свойства, т. е. свойства, состоящего из нескольких частей. С одной стороны, доступно редактирование каждой из частей в отдельности. С другой, можно редактировать сложное свойство в виде строки. В этой главе я расскажу, как создавать такие свойства. И не только...

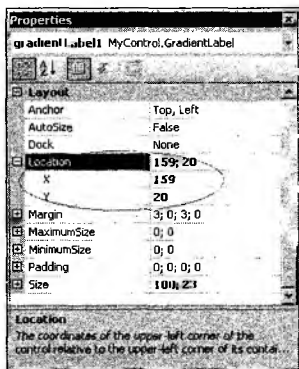


Рис. 8.1. Сложное свойство в редакторе свойств

8.1. Сложное свойство

Сложное свойство является классом, объединяющим несколько свойств. Для нашего класса градиентной заливки `GradientLabel` сложное свойство может объединять свойства `StartColor` и `EndColor`. Пример такого класса я привел в листинге 8.1, а листинг 8.2 показывает обновленный класс `GradientLabel`, использующий новое свойство. Фактически я сохранил всю функциональность. Свойства `StartColor` и `EndColor`, как и раньше, сохраняют начальный и конечный цвета градиентной заливки.

Листинг 8.1. Класс сложного свойства `GradientParameters`

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using System.ComponentModel;

namespace MyControl
{
    [TypeConverter(typeof(GradientConverter))]
    public class GradientParameters
    {
        Color startColor;
        Color endColor;

        // Делегат, вызываемый при изменении свойств
        public delegate void PropertyChangedEventHandler(
            string propertyName);
        public event PropertyChangedEventHandler PropertyChanged;

        // Конструктор
        public GradientParameters(Color startColor, Color endColor)
        {
            this.startColor = startColor;
            this.endColor = endColor;
        }

        // Цвет начала заливки
        [Category("Gradient"), Description("Цвет начала заливки"),
        DefaultValue(typeof(Color), "LightGreen")]
```



```
public Color StartColor
{
    get { return startColor; }
    set {startColor = value;
        DoPropertyChanged("StartColor"); }
}

// Цвет завершения Заливки
[Category("Gradient"),
 Description("Цвет завершения заливки"),
 DefaultValue(typeof(Color), "DarkBlue")]
public Color EndColor
{
    get { return endColor; }
    set { endColor = value;
        DoPropertyChanged("EndColor"); }
}

// Вызывается при изменении свойств заливки
private void DoPropertyChanged(string name)
{
    if (PropertyChanged != null)
        PropertyChanged(name);
}

// Преобразование строкового представления параметров
public static GradientParameters Parse(string value)
{
    // Делим строку на две части
    string[] v = ((string)value).Split(new char[] { ',' });
    return new GradientParameters(
        Color.FromName(v[0]), Color.FromName(v[1]));
}

// Преобразование параметров в строку
public override string ToString()
{
    return string.Format("{0},{1}",
        StartColor.ToKnownColor(), EndColor.ToKnownColor());
}
```

Листинг 8.2. Класс GradientLabel, использующий свойство типа GradientParameters

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace MyControl
{
    [ToolboxItem(true)]
    [ToolboxBitmap(typeof(GradientLabel), "GradientLabel.bmp")]
    [DesignerAttribute(typeof(GradientLabelDesigner))]
    public partial class GradientLabel : Label
    {
        GradientParameters gradient = new GradientParameters(
            Color.LightGreen, Color.DarkBlue);

        public GradientLabel()
        {
            InitializeComponent();
            // Подписываемся на изменение свойств заливки
            Gradient.PropertyChanged +=
                new GradientParameters.PropertyChangedEventHandler(
                    Gradient_PropertyChanged);
        }

        [Category("Gradient")]
        [Description("Свойства градиентной заливки")]
        public GradientParameters Gradient
        {
            get { return gradient; }
            set { gradient = value; }
        }

        protected override void OnPaint(PaintEventArgs pe)
        {
            // Вызываем базовый OnPaint
            base.OnPaint(pe);
        }
    }
}
```

```

// Заливаем
Color c1 = Color.FromArgb(100, Gradient.StartColor);
Color c2 = Color.FromArgb(100, Gradient.EndColor);
Brush b = new Drawing2D.LinearGradientBrush(
    ClientRectangle, c1, c2, 10);
pe.Graphics.FillRectangle(b, ClientRectangle);

b.Dispose();
}

// Обработчик изменения свойств градиентной заливки
void Gradient_PropertyChanged(string propertyname)
{
    // Перерисовать компонент
    Invalidate();
}
}

```

Научить компонент реагировать на изменение "внутренних" свойств (т. е. свойств, которые находятся внутри сложного свойства) можно двумя способами. Вариант первый — добавить специальный делегат `PropertyChanged`, который позволит подписываться на изменение внутренних свойств класса `GradientProperties` (кстати, обратите внимание, что я использовал стандартный тип делегата `PropertyChangedEventHandler`, применяемый при уведомлениях об изменении свойств, а в качестве параметра передается имя изменившегося свойства):

```

// Цвет начала заливки
public Color StartColor
{
    get { return startColor; }
    set { startColor = value;
        DoPropertyChanged("StartColor"); }
}

// Вызывается при изменении свойств заливки
private void DoPropertyChanged(string name)
{
    if (PropertyChanged != null)
        PropertyChanged(name);
}

```

Второй вариант реализации — добавить таким свойствам атрибут `NotifyParentProperty`, означающий, что при изменении значения свойства необходимо уведомить об этом "внешнее" свойство:

```
[NotifyParentProperty(true)]  
public GradientParameters Gradient  
{  
    get { return gradient; }  
    set { gradient = value; }  
}
```

Наиболее интересными в последнем листинге являются методы `ToString()` и `Parse()`. Первый позволяет получать строковое представление для нашего класса (как видно из кода, это значения полей `StartColor` и `EndColor`, перечисленных через запятую), а второй — производить обратное преобразование, т. е. на основе строкового представления заполнять поля класса `GradientParameters`. Именно эти методы позволяют редактору свойств работать со строковым представлением. Теперь осталось только научить его понимать это представление.

8.2. Конвертеры типов

Для того чтобы редактор свойств понял, что наше сложное свойство можно редактировать с помощью редактора строк, необходимо создать специальный класс, называемый *конвертером типа* (type converter). Конвертер типа является наследником класса `System.ComponentModel.TypeConverter`. Для реализации полнофункционального наследника необходимо, как минимум, переопределить следующие методы:

- ❑ метод `bool CanConvertFrom(ITypeDescriptorContext context, Type sourceType)` возвращает `true`, если тип `sourceType` можно преобразовать к нужному нам классу сложного свойства. В нашем случае мы будем возвращать `true`, если `sourceType` является типом `string`;
- ❑ метод `object ConvertFrom(ITypeDescriptorContext context, CultureInfo culture, object value)` возвращает объект сложного свойства, созданный из объекта `value`. В нашем случае `value` будет являться строкой, а для создания экземпляра класса `GradientParameters` мы будем вызывать метод `Parse`;
- ❑ метод `object ConvertTo(ITypeDescriptorContext context, CultureInfo culture, object value, Type destinationType)` производит обратное преобразование, а именно возвращает представление значения объекта `value` к типу `destinationType`. В нашем случае мы будем возвращать строковое представление, если `destinationType` является типом `string`.

В следующем разделе я скажу несколько слов о стандартных конвертерах типов, а уже затем мы перейдем к созданию конвертера типа для нашего класса.

8.3. Конвертеры типов .NET Framework 2.0

Библиотека .NET Framework содержит множество конвертеров. Я приведу лишь несколько из них:

- `System.ComponentModel.BaseNumberConverter` — базовый класс для преобразования числовых значений. От него наследованы многие конвертеры, такие как `ByteConverter`, `DecimalConverter`, `Int16Converter` и др.;
- `System.ComponentModel.BooleanConverter` — конвертер для типа `bool`;
- `System.ComponentModel.CharConverter` — конвертер для `Unicode`-символов;
- `System.ComponentModel.CollectionConverter` — конвертер для коллекций;
- `System.ComponentModel.DateTimeConverter` — конвертер для даты и времени;
- `System.ComponentModel.EnumConverter` — конвертер для перечислений;
- `System.ComponentModel.StringConverter` — конвертер для типа `string`;
- `System.Drawing.FontConverter` — конвертер для шрифтов;
- `System.Drawing.ColorConverter` — конвертер для типа `Color`;
- `System.Drawing.PointConverter` — конвертер для типа `Point`;
- `System.Drawing.RectangleConverter` — конвертер для типа `Rectangle`;
- `System.Drawing.SizeConverter` — конвертер для типа `Size`.

Небольшой пример показывает работу конвертера `BooleanConverter` (листинг 8.3). При подключении такого конвертера к свойству типа `bool` вместо обычных значений `true` и `false` будут показываться значения `да` и `нет` (рис. 8.2). Вообще говоря, ничто не мешает нам использовать конвертеры и в режиме выполнения (листинг 8.4).

Листинг 8.3. Использование `BooleanConverter`

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Globalization;

namespace MyControl
{
    public class BoolTestControl : Component
    {
        private bool test;
```

```
// Тестовое свойство с подключенным конвертером
[TypeConverter(typeof(BooleanTypeConverter))]
public bool Test
{
    get { return test; }
    set { test = value; }
}

// Конвертер
class BooleanTypeConverter : BooleanConverter
{
    const string Yes = "Да";
    const string No = "Нет";

    // Преобразуем bool в Да/Нет
    public override object ConvertTo(ITypeDescriptorContext context,
                                     CultureInfo culture,
                                     object value,
                                     Type destType)
    {
        return (bool)value ? Yes : No;
    }

    // Преобразуем Да/Нет в bool
    public override object ConvertFrom(ITypeDescriptorContext context,
                                       CultureInfo culture,
                                       object value)
    {
        return (string)value == Yes;
    }
}
}
```

Листинг 8.4. Использование BooleanConverter в режиме выполнения

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;

namespace ConverterTest
{
```

```
class Program
{
    static void Main(string[] args)
    {
        // Создаем конвертер
        BooleanConverter bc = new BooleanConverter();
        // Конвертируем строку в bool
        bool t = (bool)bc.ConvertFromString("true");
        Console.WriteLine(t);
        // Конвертируем bool в строку
        string s = (string)bc.ConvertTo(false, typeof(string));
        Console.WriteLine(s);
    }
}
```

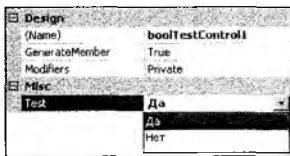


Рис. 8.2. Работа BooleanConverter

Еще одна часто возникающая задача — отображение описаний значений перечисления вместо стандартных значений. Для решения этой задачи используется конвертер `EnumConverter`. Хранить сами описания можно либо в коде (что не очень красиво), либо в ресурсах, либо в метаданных. Последний вариант представляется мне наиболее наглядным. Чтобы не изобретать собственный атрибут, воспользуемся уже существующим атрибутом `Description`. Теперь в методе `ConvertTo()` нужно получить значение этого атрибута, а в методе `ConvertFrom()` — найти элемент перечисления, имеющий такой атрибут. Реализация этого алгоритма показана в листинге 8.5, а результат работы — на рис. 8.3.

Листинг 8.5. Использование EnumConverter

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
```

```
using System.Globalization;
using System.Reflection;

namespace MyControl
{
    // Перечисление с атрибутами описаний
    public enum Operation
    {
        [Description("Не выбрано")]
        None = 0,
        [Description("Сложение")]
        Add,
        [Description("Вычитание")]
        Sub,
        [Description("Умножение")]
        Mul,
        [Description("Деление")]
        Div
    }

    // Тестовый компонент со свойством типа Enum
    public class EnumTestComponent : Component
    {
        private Operation operation;

        [TypeConverter(typeof(EnumTypeConverter))]
        public Operation Operation
        {
            get { return operation; }
            set { operation = value; }
        }
    }

    // Конвертер значений в описания и обратно
    class EnumTypeConverter : EnumConverter
    {
        private Type enumType;

        // Конструктор запоминает тип перечисления
        public EnumTypeConverter(Type enumType) : base(enumType)
        {
            this.enumType = enumType;
        }
    }
}
```



```
// Разрешаем преобразование перечисления в строку
public override bool CanConvertTo(
    ITypeDescriptorContext context, Type destType)
{
    return destType == typeof(string);
}

// Конвертируем перечисление в строку
public override object ConvertTo(
    ITypeDescriptorContext context,
    CultureInfo culture,
    object value, Type destType)
{
    // Получаем значение атрибута Description
    // для значения перечисления
    FieldInfo fi =
        enumType.GetField(Enum.GetName(enumType, value));
    DescriptionAttribute da =
        (DescriptionAttribute)Attribute.GetCustomAttribute(
            fi, typeof(DescriptionAttribute));

    // Если атрибут не задан, возвращаем значение как есть
    if (da != null)
        return da.Description;
    else
        return value.ToString();
}

// Разрешаем конвертировать перечисление из строки
public override bool CanConvertFrom(
    ITypeDescriptorContext context, Type srcType)
{
    return srcType == typeof(string);
}

// Конвертируем строку в значение перечисления
public override object ConvertFrom(
    ITypeDescriptorContext context,
    CultureInfo culture,
    object value)
{
    // Надо найти тот элемент перечисления,
    // у которого атрибут Description равен искомому значению
}
```

```

foreach (FieldInfo fi in enumType.GetFields())
{
    DescriptionAttribute da = (DescriptionAttribute)
        Attribute.GetCustomAttribute(
            fi, typeof(DescriptionAttribute));

    if ((da != null) && ((string)value == da.Description))
        return Enum.Parse(enumType, fi.Name);
}
// Если не нашли значение в атрибутах,
// пытаемся преобразовать строку в перечисление
return Enum.Parse(enumType, (string)value);
}
}
}

```

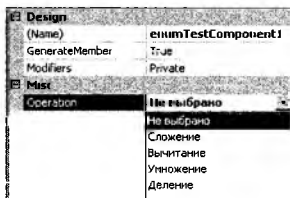


Рис. 8.3. Работа EnumConverter

8.4. Реализация собственного конвертера типа

Класс `GradientConverter`, являющийся реализацией конвертера типа для `GradientParameters`, показан в листинге 8.6.

Теперь остается только привязать его к нашему свойству, для того чтобы редактор свойств знал, что для данного свойства используется созданный нами конвертер типа. Для привязки конвертера типа к свойству нужно задать атрибут `TypeConverter`. Сделать привязку можно для конкретного свойства:

```

[TypeConverter(typeof(GradientConverter))]
public GradientParameters Gradient
{
    get { return gradient; }
    set { gradient = value; }
}

```

Или можно привязать конвертер типа к классу `GradientParameters`:

```
[TypeConverter(typeof(GradientConverter))]  
public class GradientParameters
```

В этом случае он будет использоваться во всех экземплярах этого класса.

Результат работы конвертера типа показан на рис. 8.4. Все выглядит хорошо, если бы не одна проблема. Фактически мы преобразовали класс `GradientParameters` в строку, и, соответственно, все свойства, составляющие этот класс, стали не видны.

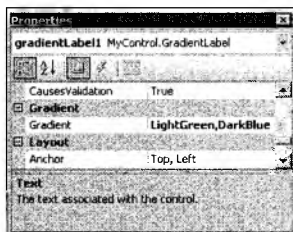


Рис. 8.4. Отображение свойства `GradientParameters` в виде строки в редакторе свойств

Листинг 8.6. Класс конвертера типа `GradientParameters`

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.ComponentModel;  
using System.Globalization;  
using System.Drawing;  
  
namespace MyControl  
{  
    // Этот класс разрешает отображать тип GradientParameters  
    // в строку, что позволяет показывать его в диалоге свойств.  
    class GradientConverter : TypeConverter  
    {  
        // Перекрываем метод CanConvertFrom.  
        // Интерфейс ITypeDescriptorContext передает контекст преобразования.  
        public override bool CanConvertFrom(  
            ITypeDescriptorContext context, Type sourceType)  
        {  
            return true;  
        }  
    }  
}
```

```
{
    // Разрешаем преобразовывать string
    if (sourceType == typeof(string))
    {
        return true;
    }
    // Вызываем базовый метод
    return base.CanConvertFrom(context, sourceType);
}

// Перекрываем метод ConvertFrom. Производим преобразование
// в пользовательский тип.
public override object ConvertFrom(
    ITypeDescriptorContext context,
    CultureInfo culture, object value)
{
    // Преобразовываем строку в GradientParameters
    if (value is string)
    {
        return GradientParameters.Parse(value as string);
    }
    return base.ConvertFrom(context, culture, value);
}

// Перекрываем метод ConvertTo. Производим преобразование
// из пользовательского типа.
public override object ConvertTo(
    ITypeDescriptorContext context,
    CultureInfo culture, object value,
    Type destinationType)
{
    // Преобразовываем GradientParameters в строку
    if (destinationType == typeof(string))
    {
        if (value != null)
        {
            if (value is GradientParameters)
            {
                return {(GradientParameters)value}.ToString();
            }
            else
            {
                return value;
            }
        }
    }
}
```

```
    }  
    }  
    return base.ConvertTo(context, culture, value, destinationType);  
}  
}
```

Для того чтобы напомнить редактору свойств, что наш класс имеет два свойства, нужно перекрыть еще два метода базового класса `TypeConverter`:

- метод `bool GetPropertiesSupported(ITypeDescriptorContext context)` должен возвращать `true`, если объект, полученный после работы конвертера типа, имеет свойства, которые должен обрабатывать редактор свойств;
- метод `PropertyDescriptorCollection GetProperties(ITypeDescriptorContext context, object value, Attribute[] attributes)` возвращает коллекцию описаний свойств, которые нужно отображать в редакторе свойств. В нашем случае создавать свои дескрипторы необходимости нет, вполне достаточно вызвать метод `GetProperties()` класса `TypeDescriptor`, возвращающий нужную нам коллекцию. Параметр `attributes` задает фильтр свойств, которые просит редактор свойств (о фильтре см. разд. 8.8).

Получившийся класс приведен в листинге 8.7, а рис. 8.5 показывает, что теперь наш класс не только представлен строкой, но и имеет два свойства.

Для полноты картины замечу, что можно перекрыть еще метод `bool CanConvertTo(ITypeDescriptorContext context, Type destinationType)`, возвращающий `true`, если значение нашего класса можно преобразовать к типу `destinationType`. Правда, по умолчанию этот метод всегда возвращает `true`, если `destinationType` является типом `string`.

Листинг 8.7. Класс конвертера типа `GradientParameters` с двумя свойствами

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.ComponentModel;  
using System.Globalization;  
using System.Drawing;  
  
namespace MyControl  
{  
    // Этот класс позволяет отображать тип GradientParameters  
    // в строку.  
    class GradientConverter : TypeConverter
```

```

{
    .....
    // Возвращает true, если нужно отображать свойства класса
    public override bool GetPropertiesSupported(
        ITypeDescriptorContext context)
    {
        return true;
    }

    // Возвращает коллекцию свойств, которые нужно отображать
    public override PropertyDescriptorCollection GetProperties(
        ITypeDescriptorContext context,
        object value,
        Attribute[] attributes)
    {
        return TypeDescriptor.GetProperties(
            typeof(GradientParameters));
    }
}

```

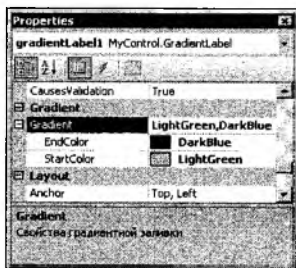


Рис. 8.5. Свойство GradientParameters состоит из двух полей

8.5. Стандартный набор значений

Для enum-типов в редакторе свойств отображается выпадающий список с перечислением всех возможных значений этого типа. Это происходит автоматически, т. к. редактор типа знает, что других значений быть не может. Для других типов, не имеющих определенного набора значений, можно реализовать *стандартный набор значений* (standard values).

Сделаем, например, для нашего свойства `Gradient` несколько заранее определенных значений. Это могут быть комбинации цветов наиболее часто используемых заливок. Для этого нужно перекрыть два метода в конвертере типа:

- метод `bool GetStandardValuesSupported(ITypeDescriptorContext context)` возвращает `true`, если конвертер типа имеет стандартный набор значений;
- метод `StandardValuesCollection GetStandardValues(ITypeDescriptorContext context)` возвращает сам набор значений, представленный классом коллекции `StandardValuesCollection`.

Итак, первый метод у нас будет просто возвращать `true`, не обращая ни на что внимания:

```
public override bool GetStandardValuesSupported(
    ITypeDescriptorContext context)
{
    return true;
}
```

Для реализации второго метода потребуется сам набор значений. Пусть он будет состоять из трех элементов:

```
private ArrayList values;
values = new ArrayList(new string[] {
    "LightGreen,DarkBlue", "White,Red", "White,Blue" });
```

Конструктор класса `StandardValuesCollection` позволяет сконвертировать `ICollection` в объект этого класса:

```
TypeConverter.StandardValuesCollection svc =
    new TypeConverter.StandardValuesCollection(values);
```

Получившийся код конвертера показан в листинге 8.8.

Листинг 8.8. Класс конвертера типа `GradientParameters` с двумя свойствами

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Globalization;
using System.Drawing;
using System.Collections;

namespace MyControl
{
```

```
// Этот класс позволяет отображать тип GradientParameters в строку
class GradientConverter : TypeConverter
{
    private ArrayList values;

    public GradientConverter()
    {
        // Инициализируем набор значений
        values = new ArrayList(new string[] {
            "LightGreen,DarkBlue", "White,Red", "White,Blue" });
    }

    // Перекрываем метод CanConvertFrom.
    // Интерфейс ITypeDescriptorContext передает контекст преобразования.
    public override bool CanConvertFrom(
        ITypeDescriptorContext context, Type sourceType)
    {
        // Разрешаем преобразовывать string
        if (sourceType == typeof(string))
        {
            return true;
        }
        // Вызываем базовый метод
        return base.CanConvertFrom(context, sourceType);
    }

    // Перекрываем метод ConvertFrom. Производим преобразование
    // в пользовательский тип.
    public override object ConvertFrom(
        ITypeDescriptorContext context,
        CultureInfo culture, object value)
    {
        // Преобразовываем строку в GradientParameters
        if (value is string)
        {
            string svalue = value as string;
            return GradientParameters.Parse(svalue);
        }

        return base.ConvertFrom(context, culture, value);
    }

    // Перекрываем метод ConvertTo. Производим преобразование
    // из пользовательского типа.
```



```
public override object ConvertTo(ITypeDescriptorContext context,
                                CultureInfo culture, object value,
                                Type destinationType)
{
    // Преобразовываем GradientParameters в строку
    if (destinationType == typeof(string))
    {
        if (value != null)
        {
            if (value is GradientParameters)
            {
                return ((GradientParameters)value).ToString();
            }
            else
            {
                return value;
            }
        }
    }
    return base.ConvertTo(
        context, culture, value, destinationType);
}

// Возвращаем true, если нужно отображать свойства класса
public override bool GetPropertiesSupported(
    ITypeDescriptorContext context)
{
    return true;
}

// Возвращаем коллекцию свойств, которые нужно отображать
public override PropertyDescriptorCollection GetProperties(
    ITypeDescriptorContext context,
    object value, Attribute[] attributes)
{
    return TypeDescriptor.GetProperties(
        typeof(GradientParameters));
}

// Возвращаем true, означающее, что тип поддерживает
// стандартный набор значений
public override bool GetStandardValuesSupported(
    ITypeDescriptorContext context)
```

```

    {
        return true;
    }

    // Возвращаем набор значений как StandardValuesCollection
    public override TypeConverter.StandardValuesCollection
        GetStandardValues(
            ITypeDescriptorContext context)
    {
        TypeConverter.StandardValuesCollection svc =
            new TypeConverter.StandardValuesCollection(values);
        return svc;
    }
}

```

Алгоритм получения стандартных значений примерно такой:

1. Редактор свойств вызывает метод `GetPropertiesSupported()` и узнает, что свойство должно показываться с набором стандартных значений.
2. Вызывается метод `GetStandardValues()`, который возвращает коллекцию стандартных значений.
3. Для каждого значения вызывается метод `ConvertFrom()`, возвращающий объект нужного типа.
4. Для каждого объекта вызывается метод `ConvertTo()` (или `ToString()`), возвращающий строковое представление объекта.
5. Набор строк показывается в редакторе свойств.

Таким образом, независимо от того, какие значения записаны в исходной коллекции, в редактор свойств попадут значения, прошедшие "цикл конвертирования". Кстати, если в методах `ConvertFrom()` или `ConvertTo()` возникнет исключение, то Visual Studio просто "падает" и закрывается.

Обратите внимание, что метод `ConvertFrom()` вызывает метод `Parse`, который создает новый объект параметров:

```

public static GradientParameters Parse(string value)
{
    string[] v = ((string)value).Split(new char[] { ',', ' ' });
    return new GradientParameters(
        Color.FromName(v[0]), Color.FromName(v[1]));
}

```

Именно этот созданный объект будет присваиваться полю `Gradient` нашего компонента при выборе одного из стандартных значений, а значит, делегат,

привязанный к объекту для уведомления об изменении полей `Gradient`, потерится. Самый простой выход из положения — просто отказаться от делегата и вызывать метод `GradientPropertyChanged` прямо в блоке `set` свойства `Gradient` (листинг 8.9).

На этом создание конвертера типа закончено. Результат работы нового конвертера показан на рис. 8.6.

Листинг 8.9. Свойство `Gradient` класса `GradientLabel`

```
public partial class GradientLabel : Label
{
    GradientParameters gradient =
        new GradientParameters(Color.LightGreen, Color.DarkBlue);

    [Category("Gradient")]
    [Description("Свойства градиентной заливки")]
    [TypeConverter(typeof(GradientConverter))]
    public GradientParameters Gradient
    {
        get { return gradient; }
        set {
            gradient = value;
            GradientPropertyChanged();
        }
    }

    // Обработчик изменения свойств градиентной заливки
    void GradientPropertyChanged()
    {
        // Перерисовать компонент
        Invalidate();
    }
}
```

Для полноты картины замечу, что кроме описанных двух методов, у конвертера есть еще метод `GetStandardValuesExclusive()`. Если он возвращает `true`, то ручное редактирование значения свойства не доступно, соответственно, значения можно выбирать только из предложенного списка:

```
public override bool GetStandardValuesExclusive(
    ITypeDescriptorContext context)
{
    return true;
}
```

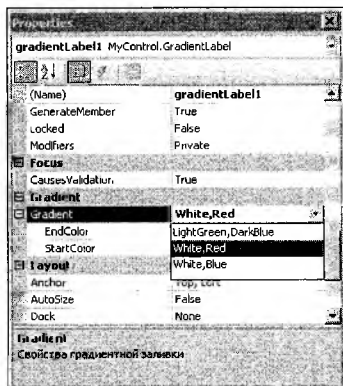


Рис. 8.6. Стандартный набор для свойства GradientParameters

К сожалению, история на этом не заканчивается. В Visual Studio есть небольшая, но неприятная проблема. Если после выбора стандартного значения сделать двойной щелчок на поле редактирования, Visual Studio выводит сообщение об ошибке (рис. 8.7).

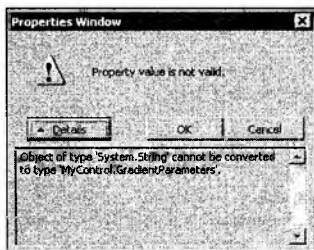


Рис. 8.7. Сообщение об ошибке при двойном щелчке

Это происходит из-за того, что при двойном щелчке среда пытается записать в поле Gradient строковое значение, что, конечно же, не удастся. Побороть эту проблему можно, хотя и не очень красиво. Нужно создать "теневое" свойство, "отражающее" существующее, но имеющее тип object, и именно его

показывать в редакторе свойств. А правильное свойство скрыть от редактора, но использовать в коде. Сделать два свойства с одинаковым именем, разумеется, не получится, но здесь поможет атрибут `DisplayName`. Задание атрибута `DesignerSerializationVisibility` (см. разд. 10.4) позволяет не сохранять одно и то же значение два раза. Код получившегося компонента показан в листинге 8.10. Проблема с двойным щелчком исправлена.

Листинг 8.10. "Теневое" свойство для исправления проблемы с двойным щелчком

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;
namespace MyControl
{
    [ToolboxItem(true)]
    [ToolboxBitmap(typeof(GradientLabel), "GradientLabel.bmp")]
    public partial class GradientLabel : Label
    {
        GradientParameters gradient =
            new GradientParameters(Color.LightGreen, Color.DarkBlue);
        public GradientLabel()
        {
            InitializeComponent();
        }
        // Это свойство видно редактору свойств
        [Category("Gradient")]
        [Description("Свойства градиентной заливки")]
        [TypeConverter(typeof(GradientConverter))]
        [DisplayName("Gradient")]
        [DesignerSerializationVisibility(
            DesignerSerializationVisibility.Hidden)]
        public object GradientProp
        {
            get { return Gradient; }
            set
            {
                // Если это реальное значение,
                // а не ошибочный string — сохраняем
```

```
        if (value.GetType() == typeof(GradientParameters))
        {
            Gradient = value as GradientParameters;
        }
    }
}
// Это свойство не видно редактору свойств,
// но его удобно использовать в коде
[Browsable(false)]
public GradientParameters Gradient
{
    get { return gradient; }
    set
    {
        gradient = value;
        GradientPropertyChanged();
    }
}
protected override void OnPaint(PaintEventArgs pe)
{
    // Вызываем базовый OnPaint
    base.OnPaint(pe);
    // Заливаем
    Color c1 = Color.FromArgb(100, Gradient.StartColor);
    Color c2 = Color.FromArgb(100, Gradient.EndColor);
    Brush b = new Drawing2D.LinearGradientBrush(
        ClientRectangle, c1, c2, 10);
    pe.Graphics.FillRectangle(b, ClientRectangle);
    b.Dispose();
}
// Обработчик изменения свойств градиентной заливки
void GradientPropertyChanged()
{
    // Перерисовать компонент
    Invalidate();
}
}
```

8.6. Класс *SimplePropertyDescriptor*

Как я уже говорил в *разд. 4.5*, все дескрипторы являются или внутренними, или абстрактными классами, поэтому использовать их "напрямую" не получится. Реализовывать свой дескриптор с нуля — занятие достаточно сложное. Класс `TypeConverter` содержит внутри себя интересный класс `SimplePropertyDescriptor`, реализующий большинство необходимых методов дескриптора. Это тоже абстрактный класс, но его наследнику достаточно реализовать всего два метода:

- `object GetValue(object component)` — возвращает значение свойства;
- `void SetValue(object component, object value)` — устанавливает значение свойства.

К сожалению, реализовывать наследника этого класса можно только в рамках конвертера типа, т. к. этот класс объявлен как `protected`, но часто этого вполне достаточно. Пример реализации показан в листинге 8.11.

Листинг 8.11. Реализация наследника класса `SimplePropertyDescriptor`

```
public class ExampleTypeConverter : TypeConverter
{
    sealed class CustomPropertyDescriptor : SimplePropertyDescriptor
    {
        private readonly PropertyDescriptor property;

        public CustomPropertyDescriptor(PropertyDescriptor property,
            Attribute[] attributes) : base(property.ComponentType,
            property.Name, property.PropertyType, attributes)
        {
            this.property = property;
        }

        // Реализация GetValue (возвращает значение свойства)
        public override object GetValue(object component)
        {
            return property.GetValue(component);
        }

        // Реализация SetValue (устанавливает значение свойства)
        public override void SetValue(object component, object value)
        {
            property.SetValue(component, value);
        }
    }
}
```

8.7. Редактирование флагов

В разд. 8.3 я показал, как использовать конвертер `EnumConverter` при редактировании перечисления. Теперь я хочу рассказать, как редактировать перечисление, если оно отмечено флагом `Flags`, т. е. возможно выбрать несколько значений перечисления. Разумеется, значения флагов должны иметь значения степеней двойки:

```
[Flags]
public enum Operation
{
    [Description("Не выбрано")]
    None = 0,
    [Description("Сложение")]
    Add = 1,
    [Description("Вычитание")]
    Sub = 2,
    [Description("Умножение")]
    Mul = 4,
    [Description("Деление")]
    Div = 8
}
```

Основная идея — сделать конвертер, который будет создавать набор виртуальных свойств, в соответствии со значениями перечисления. Каждое из таких свойств должно иметь тип `bool`. Если оно имеет значение `true`, значит, соответствующий флаг выставлен. Значение `None` (числовое значение 0) говорит о том, что ни один из флагов не выставлен, поэтому в список виртуальных свойств оно попадать не должно.

Как мы уже обсуждали, класс `PropertyDescriptor` является абстрактным, поэтому нам потребуется свой класс дескриптора. Методы `GetValue()` и `SetValue()` должны отражать работу с битовыми флагами.

Алгоритм создания списка свойств выглядит следующим образом:

1. Получаем все значения перечисления.
2. Просматриваем все значения.
3. Если это значение 0 (т. е. `None`), мы его пропускаем, т. к. оно не нужно в списке свойств.
4. Пытаемся получить значение атрибута `Description`. Если атрибут задан, мы будем использовать это значение как имя свойства, а если не задан, то будем использовать само значение.

5. Создаем новый дескриптор и добавляем его в выходную коллекцию.
6. На основе полученной коллекции создаем коллекцию `PropertyDescriptorCollection`.

Полный код конвертера и дескриптора (он является внутренним классом конвертера) представлен в листинге 8.12. Как это выглядит в редакторе свойств, показано на рис. 8.8.

Работа с флагами заключается в трех операциях (переменная `val` представляет текущее значение, а переменная `value` — значение флага):

- `(val & value) > 0` — проверка, что флаг `value` установлен в значении `val`;
- `val | value` — установка флага;
- `val & ~value` — сброс флага.

В остальном код ничего нового не содержит. Все эти операции мы уже разбирали.

Листинг 8.12. Конвертер флагов

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Reflection;

namespace MyControl
{
    public class FlagsEnumConverter : TypeConverter
    {
        // Это составное свойство
        public override bool GetPropertiesSupported(
            ITypeDescriptorContext context)
        {
            return true;
        }

        // Возвращаем коллекцию дескрипторов свойства
        public override PropertyDescriptorCollection GetProperties(
            ITypeDescriptorContext context,
            object value, Attribute[] attributes)
        {
            // Получаем все значения перечисления
            Array values = Enum.GetValues(
                context.PropertyDescriptor.PropertyType);
```

```
// Тип перечисления
Type propType = context.PropertyDescriptor.PropertyType;

// Выходная коллекция дескрипторов
List<PropertyDescriptor> outputCollection =
    new List<PropertyDescriptor>();

// Переписываем значения как свойства
for (int i = 0; i < values.Length; i++)
{
    object propValue = values.GetValue(i);

    // Значение 0 (none) пропускаем
    if (Convert.ToInt32(propValue) == 0)
        continue;

    // Получаем имя значения
    string name = Enum.GetName(propType, propValue);

    // Получаем список свойств
    FieldInfo fi = propType.GetField(name);

    // Пытаемся получить атрибут Description
    DescriptionAttribute da = (DescriptionAttribute)
        Attribute.GetCustomAttribute(
            fi, typeof(DescriptionAttribute));

    // Если атрибут Description есть, то берем
    // его значение как имя свойства
    if (da != null)
        name = da.Description;

    // Добавляем дескриптор в коллекцию
    outputCollection.Add(new EnumFieldDescriptor(name,
        propValue, propType,
        context.PropertyDescriptor, context.Instance));
}

// Получаем коллекцию дескрипторов
return new PropertyDescriptorCollection(
    outputCollection.ToArray());
}
```

```
// Специальный класс дескриптора значения перечисления
private class EnumFieldDescriptor : PropertyDescriptor
{
    private string name;
    private long value;
    private Type enumType;
    private PropertyDescriptor parentProp;
    private object parentComp;

    // Конструктор
    internal EnumFieldDescriptor(
        string name, object value, Type enumType,
        PropertyDescriptor parentProp, object parentComp)
        : base(name, null)
    {
        this.name = name;
        this.value = Convert.ToInt64(value);
        this.enumType = enumType;
        this.parentProp = parentProp;
        this.parentComp = parentComp;
    }

    // Возможность сброса значения
    public override bool CanResetValue(object component)
    {
        return true;
    }

    // Тип компонента
    public override Type ComponentType
    {
        get { return enumType; }
    }

    // Метод GET
    public override object GetValue(object component)
    {
        long val = Convert.ToInt64(component);
        return (val & value) > 0;
    }

    // Метод SET
    public override void SetValue(object component, object value)
```

```

    {
        long val = Convert.ToInt32(component);
        if ((bool)value)
            parentProp.SetValue(parentComp,
                Enum.ToObject(enumType, val | this.value));
        else
            parentProp.SetValue(parentComp,
                Enum.ToObject(enumType, val & ~this.value));
        OnValueChanged(this, EventArgs.Empty);
    }

    // Возможность редактирования
    public override bool IsReadOnly
    {
        get { return false; }
    }

    // Тип свойства
    public override Type PropertyType
    {
        get { return typeof(bool); }
    }

    // Метод сброса
    public override void ResetValue(object component)
    {
        SetValue(component, false);
    }

    // Выключить сериализацию
    public override bool ShouldSerializeValue(object component)
    {
        return false;
    }
}

```

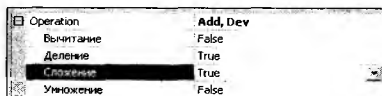


Рис. 8.8. Вид редактора флагов

8.8. Динамическое управление набором свойств

В разд. 6.6 я показывал, как скрыть элементы из редактора свойств, а в разд. 5.4 рассказывал об атрибуте `Browsable`. Но это было скрытие "насовсем", т. е. свойство просто не видно в режиме разработки. Сейчас я хочу показать, как сделать динамическое управление свойствами, когда некоторые свойства будут видны или скрыты в зависимости от значений других свойств.

Добавим в наш компонент градиентной заливки свойство `GradientActive`. Если оно имеет значение `true`, компонент показывает градиентную заливку, а если `false`, то работает как обычный компонент `Label`. Конечно, практического смысла это свойство не имеет, но в качестве примера вполне подходит. Соответствующий код показан в листинге 8.13.

Листинг 8.13. Компонент `GradientLabel` со свойством `GradientActive`

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace MyControl
{
    [ToolboxItem(true)]
    [ToolboxBitmap(typeof(GradientLabel), "GradientLabel.bmp")]
    [DesignerAttribute(typeof(GradientLabelDesigner))]
    [TypeConverter(typeof(GradientLabelTypeConverter))]
    public partial class GradientLabel : Label
    {
        private Color startColor = Color.LightGreen;
        private Color endColor = Color.DarkBlue;
        private bool gradientActive = true;

        /// <summary>
        /// true, если заливка включена
        /// </summary>
        [Category("Gradient"), Description("Градиент включен"),
        DefaultValue(typeof(bool), "true")]
        [RefreshProperties(RefreshProperties.All)]
    }
}
```

```
public bool GradientActive
{
    get { return gradientActive; }
    set
    {
        gradientActive = value;
        OnPropertyChanged();
    }
}

/// <summary>
/// Цвет начала заливки
/// </summary>
[Category("Gradient"), Description("Цвет начала заливки"),
DefaultValue(typeof(Color), "LightGreen"),
AmbientValue(typeof(Color), "Empty")]
public Color StartColor
{
    get
    {
        if (startColor == Color.Empty)
            return Parent.BackColor;
        return startColor;
    }
    set
    {
        startColor = value;
        OnPropertyChanged();
    }
}

/// <summary>
/// Цвет завершения заливки
/// </summary>
[Category("Gradient"), Description("Цвет завершения заливки"),
DefaultValue(typeof(Color), "DarkBlue")]
public Color EndColor
{
    get { return endColor; }
    set
    {
        endColor = value;
        OnPropertyChanged();
    }
}
```

```
/// <summary>
/// Вызывается при изменении свойства заливки
/// </summary>
private void OnChangeProperties()
{
    Invalidate();
}

/// <summary>
/// Отрисовка компонента
/// </summary>
/// <param name="pe"></param>
protected override void OnPaint(PaintEventArgs pe)
{
    // Вызываем базовый OnPaint
    base.OnPaint(pe);

    // Заливаем, если заливка включена
    if (gradientActive)
    {
        Color c1 = Color.FromArgb(100, startColor);
        Color c2 = Color.FromArgb(100, endColor);
        Brush b = new Drawing2D.LinearGradientBrush(
            ClientRectangle, c1, c2, 10);
        pe.Graphics.FillRectangle(b, ClientRectangle);
        b.Dispose();
    }
}

/// <summary>
/// Конструктор
/// </summary>
public GradientLabel()
{
    InitializeComponent();
}
}
```

Итак, наша задача формулируется следующим образом: не показывать свойства `StartColor` и `EndColor` в редакторе свойств, если значение `GradientActive` равно `false`.

В предыдущем разделе мы подключали конвертер типа к конкретному свойству, а сейчас мы подключим его ко всему компоненту и, таким образом, бу-

дем работать со свойствами всего компонента (или, правильнее сказать, конвертер подключается к типу `GradientLabel` и применяется ко всем его экземплярам):

```
[TypeConverter(typeof(GradientLabelTypeConverter))]  
public partial class GradientLabel : Label
```

Про метод `GetPropertiesSupported()` я уже рассказывал (см. разд. 8.4). В нашем случае он всегда будет возвращать `true`.

Вся основная работа будет идти в методе `GetProperties()`, который возвращает коллекцию дескрипторов свойств компонента. Еще раз напомним его описание:

```
public override PropertyDescriptorCollection GetProperties(  
    ITypeDescriptorContext context,  
    object value, Attribute[] attributes)  
{  
}
```

Параметр `attributes` задает фильтр, согласно которому выбираются свойства компонента. Обычно редактор свойств запрашивает свойства с атрибутом `Browsable.Yes`, т. е. только те, которые были предназначены для отображения. Получить нужную коллекцию можно с помощью метода `TypeDescriptor.GetProperties()`:

```
PropertyDescriptorCollection coll = // без учета фильтра  
    TypeDescriptor.GetProperties(value.GetType());  
PropertyDescriptorCollection coll = // с учетом фильтра  
    TypeDescriptor.GetProperties(value.GetType(), attributes);
```

Для того чтобы метод `GetProperties()` вызывался при изменении значения свойства `GradientActive`, это свойство должно быть отмечено атрибутом `RefreshProperties(RefreshProperties.All)`. Он говорит редактору свойств, что при изменении значения свойства нужно обновить все другие.

Итак, нам надо таким образом модифицировать коллекцию `coll`, чтобы свойства `StartColor` и `EndColor` не были видны в редакторе свойств в случае, когда `GradientActive` равно `false`. Все было бы просто, если бы не одна проблема: коллекция `coll`, которую мы получили, является коллекцией "только для чтения". Из нее нельзя удалять значения или менять их. Придется искать другой выход — переписывать эту коллекцию в другую, фильтруя значения самостоятельно. Конструктор класса `PropertyDescriptorCollection` имеет следующий вид:

```
public PropertyDescriptorCollection(  
    PropertyDescriptor[] properties, bool readOnly)
```


Последний параметр задает режим "только для чтения". Его можно просто скопировать из существующей коллекции:

```
(coll as IList).IsReadOnly
```

Теперь займемся созданием массива дескрипторов свойств (первый параметр конструктора). Общий алгоритм такой:

1. Получить коллекцию свойств без учета фильтра, т. к. фильтровать свойства придется вручную.
2. Создать массив для хранения свойств.
3. Просмотреть каждое свойство исходной коллекции.
4. Если надо, корректировать атрибуты свойства.
5. Проверить фильтр (т. к. атрибут мог измениться после нашей корректировки).
6. Если фильтр удовлетворяет условию, добавить свойство в выходной массив.
7. Создать `PropertyDescriptorCollection` на основе полученного массива.

Корректировка атрибутов будет довольно простой. Если свойство `GradientActive` равно `false`, то свойствам `StartColor` и `EndColor` добавляем атрибут `Browsable.No`, а иначе добавляем атрибут `Browsable.Yes`:

```
if ((property.Name == "StartColor") || (property.Name == "EndColor"))
{
    if ((value as GradientLabel).GradientActive)
    {
        attributeList.Add(BrowsableAttribute.Yes);
    }
    else
    {
        attributeList.Add(BrowsableAttribute.No);
    }
}
```

Для проверки фильтра используется метод `Contains` класса `AttributeCollection`:

```
AttributeCollection newCollection =
    AttributeCollection.FromExisting(
        property.Attributes, attributeArray);
if (newCollection.Contains(attributes))
{
}
```

Код получившегося конвертера показан в листинге 8.14. Теперь, если выставить значение `GradientActive` в `false`, свойства `StartColor` и `EndColor` будут скрыты.

Листинг 8.14. Конвертер типа, скрывающий свойства `StartColor` и `EndColor` в зависимости от свойства `GradientActive`

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Collections;

namespace MyControl
{
    class GradientLabelTypeConverter : TypeConverter
    {
        public override bool GetPropertiesSupported(
            ITypeDescriptorContext context)
        {
            return true;
        }

        public override PropertyDescriptorCollection GetProperties(
            ITypeDescriptorContext context,
            object value, Attribute[] attributes)
        {
            // Получаем исходный список свойств, игнорируя фильтр
            PropertyDescriptorCollection coll =
                TypeDescriptor.GetProperties(value.GetType());

            // Новый список свойств
            List<PropertyDescriptor> outputCollection =
                new List<PropertyDescriptor>();

            // Просматриваем все свойства
            foreach (PropertyDescriptor property in coll)
            {
                // Копируем исходные атрибуты в новый массив
                List<Attribute> attributeList = new List<Attribute>();
                foreach (Attribute attribute in property.Attributes)
                {
                    attributeList.Add(attribute);
                }
            }
        }
    }
}
```

```
// Если это нужные нам свойства, модифицируем атрибуты
if ((property.Name == "StartColor") ||
    (property.Name == "EndColor"))
{
    if ((value as GradientLabel).GradientActive)
    {
        attributeList.Add(BrowsableAttribute.Yes);
    }
    else
    {
        attributeList.Add(BrowsableAttribute.No);
    }
}

// Новый массив атрибутов
Attribute[] attributeArray = attributeList.ToArray();

// Создаем AttributeCollection для проверки фильтра
AttributeCollection newCollection =
    AttributeCollection.FromExisting(
        property.Attributes, attributeArray);

// Проверяем фильтр
if (newCollection.Contains(attributes))
{
    // Если фильтр пройден — добавляем свойство
    // в выходную коллекцию
    outputCollection.Add(new CustomPropertyDescriptor(
        property, attributeArray));
}

// Возвращаем PropertyDescriptorCollection
return new PropertyDescriptorCollection(
    outputCollection.ToArray(),
    (coll as IList).IsReadOnly);
}

protected sealed class CustomPropertyDescriptor :
    SimplePropertyDescriptor
{
    private readonly PropertyDescriptor property;
```

```

public CustomPropertyDescriptor(PropertyDescriptor property,
                                Attribute[] attributes)
    : base(property.ComponentType, property.Name,
           property.PropertyType, attributes)
{
    this.property = property;
}

private PropertyDescriptor PropertyDescriptor
{
    get { return property; }
}

public override object GetValue(object component)
{
    return PropertyDescriptor.GetValue(component);
}

public override void SetValue(object component, object value)
{
    PropertyDescriptor.SetValue(component, value);
}
}

```

8.9. Стандартное сложное свойство

В предыдущих разделах я показывал, как с помощью конвертера типа сделать сложное свойство, которое видно в редакторе свойств как свойство, состоящее из нескольких полей, каждое из которых редактируется отдельно. Очень часто этого вполне достаточно и больше никакие возможности конвертера типа не нужны. В этом случае можно использовать стандартный класс `ExpandableObjectConverter`, позволяющий сделать сложное свойство редактируемым:

```

[TypeConverter(typeof(ExpandableObjectConverter))]
public class GradientParameters
{
    ...
}

```

Теперь поля сложного свойства будут доступны для редактирования.

8.10. Стандартный набор значений-классов

С помощью класса `ExpandableObjectConverter` можно реализовать очень интересное поведение редактора свойства. Рассмотрим некий компонент, который будет описывать запись об интервью (конечно, я мог бы придумать пример на основе класса `GradientLabel`, но, кажется, он уже порядком всем наделен):

```
public class InterviewControl : Component
{
    private InterviewRecord interviewRecord;

    [Description("Запись об интервью")]
    public InterviewRecord InterviewRecord
    {
        get
        {
            return interviewRecord;
        }
        set
        {
            interviewRecord = value;
        }
    }
}
```

Единственным свойством этого компонента будет запись об интервью (разумеется, я оставил только одно свойство в целях упрощения кода, в реальной задаче свойств у этого компонента будет очень много), представляемая абстрактным классом `InterviewRecord`:

```
public abstract class InterviewRecord
{
    private string interviewer;

    [Description("Кто проводит интервью")]
    public string Interviewer
    {
        get { return interviewer; }
        set { interviewer = value; }
    }
}
```

Конкретные типы интервью будут описываться наследниками этого класса. Пока нам будет достаточно двух таких классов — телефонного интервью (класс `PhoneInterview`) и личной встречи (класс `OnlineInterview`):

```
public class PhoneInterview : InterviewRecord
{
    private DateTime interviewDateTime;
    private string phoneNumber;

    [Description("Номер телефона")]
    public string PhoneNumber
    {
        get { return phoneNumber; }
        set { phoneNumber = value; }
    }

    [Description("Дата и время интервью")]
    public DateTime InterviewDateTime
    {
        get { return interviewDateTime; }
        set { interviewDateTime = value; }
    }
}

public class OnlineInterview : InterviewRecord
{
    private DateTime interviewDate;
    private string address;

    [Description("Адрес офиса")]
    public string Address
    {
        get { return address; }
        set { address = value; }
    }

    [Description("Дата интервью")]
    public DateTime InterviewDate
    {
        get { return interviewDate; }
        set { interviewDate = value; }
    }
}
```

Разумеется, можно добавлять и другие типы, например, интервью через Интернет или по почте...

Добавив компонент на форму, мы увидим, что свойство недоступно для редактирования (рис. 8.9), т. к. редактор свойств просто не понимает, что делать

с нашим классом. Прямое использование конвертера `ExpandableObjectConverter` в этом случае не спасает — ведь к свойству пока не привязан конкретный класс, поэтому нужно реализовать специальный класс конвертера.



Рис. 8.9. Пока свойство `InterviewRecord` не редактируемое

Класс `InterviewRecordConverter`, показанный в листинге 8.15, позволяет редактировать свойство `InterviewRecord`. В принципе, ничего нового в этом классе нет. Обо всех методах я уже рассказывал. Самый интересный код заключается в формировании списка стандартных значений. Если в предыдущих примерах мы формировали его из строк, то в этом конвертере формирование стандартного набора значений производится из набора классов:

```
private readonly object[] _standardValues = new object[] {  
    null,  
    new PhoneInterview(),  
    new OnlineInterview()  
};  
  
/// <summary>  
/// Возвращаем стандартный набор значений  
/// </summary>  
public override TypeConverter.StandardValuesCollection GetStandardValues(  
    ITypeDescriptorContext context)  
{  
    return new TypeConverter.StandardValuesCollection(_standardValues);  
}
```

Посмотрите, что это дает. Во-первых, наше свойство стало редактируемым и имеет три стандартных значения (рис. 8.10). Но самое интересное поведение нашего свойства проявляется при выборе значения, отличного от (None): набор вложенных свойств меняется в соответствии с типом выбранной записи. Для телефонного интервью появляется поле номера телефона и время (рис. 8.11), а для личной встречи — адрес и дата (рис. 8.12).

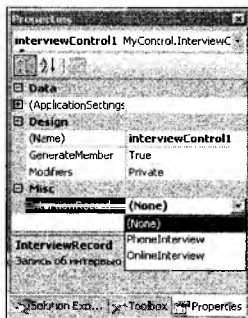


Рис. 8.10. Свойство InterviewRecord стало редактируемым и имеет три стандартных значения



Рис. 8.11. Набор свойств для типа PhoneInterview

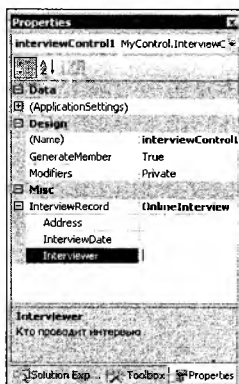


Рис. 8.12. Набор свойств для типа OnlineInterview

Такие редакторы дают очень мощный и удобный механизм редактирования свойств компонентов.

Еще один способ реализации такого поведения я опишу в *разд. 9.3.4*.

Листинг 8.15. Конвертер типа со стандартными значениями-классами

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;

namespace MyControl
{
    public class InterviewRecordConverter : ExpandableObjectConverter
    {
        /// <summary>
        /// Разрешаем конвертировать из строки
        /// </summary>
        public override bool CanConvertFrom(
            ITypeDescriptorContext context,
            Type sourceType)
        {
            if (sourceType == typeof(string))
            {
                return true;
            }
            return base.CanConvertFrom(context, sourceType);
        }

        /// <summary>
        /// Разрешаем конвертировать в строку
        /// </summary>
        public override bool CanConvertTo(
            ITypeDescriptorContext context,
            Type destinationType)
        {
            if (destinationType == typeof(string))
            {
                return true;
            }
            return base.CanConvertTo(context, destinationType);
        }
    }
}
```

```
/// <summary>
/// Конвертируем из строки в класс
/// </summary>
public override object ConvertFrom(
    ITypeDescriptorContext context,
    System.Globalization.CultureInfo culture,
    object value)
{
    if (value is string)
    {
        InterviewRecord interviewRecord = null;
        switch ((string)value)
        {
            case "PhoneInterview":
            {
                interviewRecord = new PhoneInterview();
                break;
            }
            case "OnlineInterview":
            {
                interviewRecord = new OnlineInterview();
                break;
            }
        }
        return interviewRecord;
    }

    return base.ConvertFrom(context, culture, value);
}

/// <summary>
/// Конвертируем в строку
/// </summary>
public override object ConvertTo(
    ITypeDescriptorContext context,
    System.Globalization.CultureInfo culture,
    object value, Type destinationType)
{
    if (destinationType == typeof(string))
    {
        string convertedValue = "[None]";
        if (value is PhoneInterview)
        {

```

```
        convertedValue = "PhoneInterview";
    }
    else
        if (value is OnlineInterview)
        {
            convertedValue = "OnlineInterview";
        }
    return convertedValue;
}

return base.ConvertTo(
    context, culture, value, destinationType);
}

/// <summary>
/// Есть набор стандартных значений
/// </summary>
public override bool GetStandardValuesSupported(
    ITypeDescriptorContext context)
{
    return true;
}

/// <summary>
/// Стандартный набор значений
/// </summary>
private readonly object[] _standardValues = new object[]{
    null,
    new PhoneInterview(),
    new OnlineInterview()
};

/// <summary>
/// Возвращаем стандартный набор значений
/// </summary>
public override TypeConverter.StandardValuesCollection
    GetStandardValues(ITypeDescriptorContext context)
{
    return
        new TypeConverter.StandardValuesCollection(_standardValues);
}

/// <summary>
/// Запретить ввод других значений, кроме стандартных
/// </summary>
```

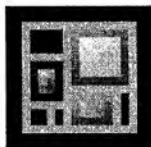
```
public override bool GetStandardValuesExclusive(  
    ITypeDescriptorContext context)  
{  
    return true;  
}  
}
```

8.11. Определение порядка отображения свойств

Иногда возникает желание поменять порядок отображения свойств в редакторе свойств. Так, например, по умолчанию свойства класса `GradientParameters` отображаются в порядке `EndColor`, `StartColor`. Мне кажется, более логичным был бы обратный порядок. Поменять его можно в момент получения списка свойств в методе `GetProperties()`:

```
public override PropertyDescriptorCollection GetProperties(  
    ITypeDescriptorContext context,  
    object value, Attribute[] attributes)  
{  
    return TypeDescriptor.GetProperties(  
        typeof(GradientParameters)).Sort(  
            new string[] { "StartColor", "EndColor" });  
}
```

ГЛАВА 9



Редактор типа

Мышь — это не роскошь, а средство передвижения курсора.

Редактор свойств имеет довольно много различных возможностей редактирования значений (см. разд. 2.6), но даже этого иногда не хватает. В таких случаях нужно создавать собственный *редактор типа* (UI type editor).

9.1. Класс *UITypeEditor*

Создание нового редактора типа заключается в создании наследника класса `System.Drawing.Design.UITypeEditor`, который привязывается к редактируемому свойству с помощью атрибута `EditorAttribute`:

```
[Editor(typeof(MyPropertyEditor), typeof(UITypeEditor))]
public byte MyProperty
{
    get/set
}
```

Либо можно привязать редактор непосредственно к типу данных:

```
[Editor(typeof(MyEnumEditor), typeof(UITypeEditor))]
public enum MyEnum
{
    Value1 = 1,
    .....
}
```

Сам класс `UITypeEditor` предоставляет свойства и методы для создания собственного редактора:

- ❑ виртуальный метод `object EditValue(ITypeDescriptorContext context, IServiceProvider provider, object value)` отображает диалог редактирования значения свойства и возвращает новое значение свойства (если значение свойства не изменилось, надо вернуть старое значение);
- ❑ виртуальный метод `UITypeEditorEditStyle GetEditStyle(ITypeDescriptorContext context)` определяет стиль редактора: `UITypeEditorEditStyle.DropDown` показывает справа треугольничек, как в выпадающем списке, а `UITypeEditorEditStyle.Modal` показывает кнопку с тремя точками;
- ❑ виртуальный метод `bool GetPaintValueSupported(ITypeDescriptorContext context)` определяет, нужно ли отрисовывать изображение значения (рис. 9.1);
- ❑ виртуальный метод `void PaintValue(PaintValueEventArgs e)` позволяет показать изображение значения (если, конечно, метод `GetPaintValueSupported()` вернул `true`);
- ❑ виртуальное свойство `bool IsDropDownResizable` определяет, можно ли изменять размеры выпадающего диалога.

Как обычно, сначала я расскажу о некоторых стандартных редакторах типов, а затем мы перейдем к созданию собственного редактора.

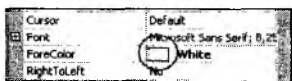


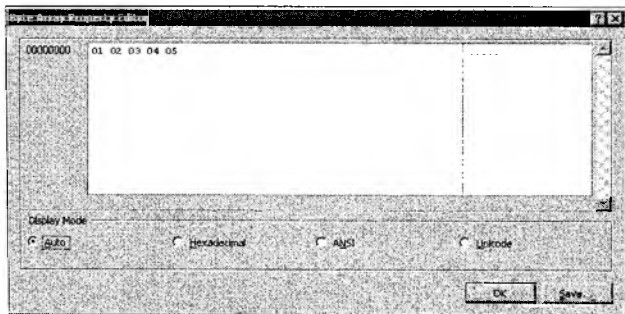
Рис. 9.1. Квадрат изображения

9.2. Классы стандартных редакторов типов

В этом разделе я хочу просто привести список некоторых стандартных редакторов типов. Про некоторые из них я еще расскажу.

В пространстве имен `System.ComponentModel.Design` определены такие редакторы типов:

- ❑ `BinaryEditor` — редактор бинарных данных (рис. 9.2);
- ❑ `CollectionEditor` — редактор коллекций (я расскажу о нем дальше);
- ❑ `DateTimeEditor` — редактирование даты и времени (отображает календарь);
- ❑ `MultilineStringEditor` — редактор многострочного текста (рис. 9.3).



ByteArray	Byte[] Array
[0]	1
[1]	2
[2]	3
[3]	4
[4]	5

Рис. 9.2. Редактор BinaryEditor

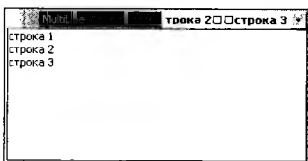


Рис. 9.3. Редактор MultilineStringEditor

Пространство имен `System.Drawing.Design` содержит редакторы, предназначенные для свойств визуального представления:

- `ContentAlignmentEditor` — редактирование выравнивания (alignment);
- `CursorEditor` — редактор для выбора файлов курсоров (cur);
- `FontEditor` — редактор для шрифтов;
- `ColorEditor` — редактор для цветов;
- `FontNameEditor` — редактор для имени шрифта;
- `IconEditor` — редактор для выбора файла значка;
- `ImageEditor` — редактор для выбора картинки.

Редакторы для форм представлены в пространстве имен `System.Windows.Forms.Design`:

- ☐ `AnchorEditor` — редактор для свойства "якорь" (тип `AnchorStyles`) (рис. 9.4);
- ☐ `BorderSidesEditor` — редактор для свойства границы компонента `ToolStripStatusLabel` (тип `ToolStripStatusLabelBorderSides`) (рис. 9.5);
- ☐ `DockEditor` — редактор для свойства взаимного расположения (свойство `Dock`);
- ☐ `FileNameEditor` — редактор для выбора имени файла;
- ☐ `FolderNameEditor` — редактор для выбора имени каталога;
- ☐ `ShortcutKeysEditor` — редактор для выбора "горячей" клавиши.

Пространство имен `System.Web.UI.Design` предоставляет редакторы для Web-приложений, но в этой книге я про них рассказывать не буду.

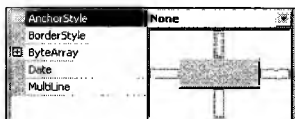


Рис. 9.4. Редактор `AnchorEditor`

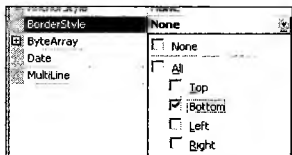


Рис. 9.5. Редактор `BorderSidesEditor`

9.3. Реализация собственного редактора типа

Теперь разберемся с методами класса `UITypeEditor` более подробно.

С методом `GetEditStyle()` все просто. Если этот метод возвращает `UITypeEditorEditStyle.DropDown`, то отображается небольшой треугольник, как у компонента `DropDown`. Значение `UITypeEditorEditStyle.Modal` отображает кнопку с тремя точками, как у компонента выбора файла.

Прежде чем рассматривать основной метод редактора `EditValue()`, мне надо рассказать о сервисе `IWindowsFormsEditorService`, который используется при создании редактора типа. Сервис имеет всего три метода:

- ☐ метод `void DropDownControl(Control control)` создает диалог редактора в виде выпадающего диалога наподобие выпадающего списка;

- метод `void CloseDropDown()` закрывает диалог, созданный с помощью метода `DropDownControl`;
- метод `DialogResult ShowDialog(Form dialog)` отображает диалог редактирования свойства и возвращает результат редактирования.

Стандартный прототип метода `EditValue()` имеет вид:

```
public override object EditValue(ITypeDescriptorContext context,
                                IServiceProvider provider,
                                object value)
{
    if (context != null && context.Instance != null && provider != null)
    {
        // Получаем интерфейс сервиса
        IWindowsFormsEditorService edSvc =
            (IWindowsFormsEditorService)provider.GetService(
                typeof(IWindowsFormsEditorService));
        if (edSvc != null)
        {
            // Устанавливаем текущее значение для редактирования
            ... = value;
            // Вызываем DropDownControl или ShowDialog
            edSvc.DropDownControl(...);
            // Получаем новое значение
            value = ...
        }
    }
    // Возвращаем либо старое, либо новое значение
    return value;
}
```

9.3.1. Выпадающий диалог

Как я уже говорил, выпадающий диалог оформляется в виде визуального компонента (наследника класса `Control`) и регистрируется с помощью вызова метода сервиса `DropDownControl(myEditorControl)`. Диалог обычно закрывается, когда пользователь выберет новое значение или откажется от выбора (например, нажмет клавишу <Esc>). Программное закрытие окна можно выполнить, вызвав метод `CloseDropDown()`.

Стандартный прототип компонента редактирования можно описать так:

- метод `SetValue()` устанавливает текущее значение свойства;
- метод `GetValue()` возвращает значение свойства после редактирования. Если значение не изменилось, метод должен вернуть старое значение;

- событие `OnValueChanged()`. В обработчике этого события, редактор типа может вызвать метод `CloseDropDown()` и закрыть редактор сразу после установки нового значения свойства.

Соответствующий код выглядит примерно так:

```
MyEditorControl editor = new MyEditorControl();
editor.ValueChanged += new EventHandler(this.ValueChanged);
editor.SetValue(value);
edSvc.DropDownControl(editor);
value = editor.GetValue();
```

Обработчик `ValueChanged` очень простой:

```
private void ValueChanged(object sender, EventArgs e)
{
    if (edSvc != null)
    {
        edSvc.CloseDropDown();
    }
}
```

Для сложных свойств, значение или отображение которых зависит от других свойств компонента, простого значения `value` мало и можно использовать сам редактируемый компонент, возвращаемый значением `Instance` параметра `context` метода `EditValue()`:

```
public override object EditValue(ITypeDescriptorContext context,
                                IServiceProvider provider,
                                object value)
{
    if (context != null && context.Instance != null && provider != null)
    {
        // Редактируемый компонент
        MyComponent propValue = (MyComponent) context.Instance;
    }
}
```

В качестве примера я сделал небольшой компонент "прогресс" (листинг 9.1). Он имеет всего три свойства: `Min`, `Max` и `Position`. Первые два задают границы некоторого значения, а `Position` — текущее значение. Компонент отображает это значение с помощью заполнения прямоугольника цветом (рис. 9.6).

Без дополнительного редактора типа все три свойства этого компонента будут видны как обычные числовые свойства. Задача состоит в том, чтобы свойство `Position` можно было редактировать более удобно и просто. Или.

другими словами, нужно сделать редактор типа `PositionUIEditor` для свойства `Position`:

```
[Editor(typeof(PositionUIEditor), typeof(UITypeEditor))]
public int Position
{
    get/set
}
```

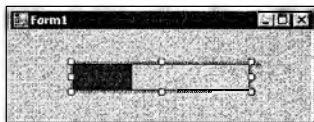


Рис. 9.6. Вид компонента `MinMaxControl`

Класс `PositionUIEditor` реализован по приведенному выше шаблону (листинг 9.2). Метод `GetEditStyle()` возвращает значение `UITypeEditorEditStyle.DropDown`, означающее, что редактор будет выглядеть как выпадающий список. В методе `EditValue()` нам потребуется компонент, который будет являться редактором свойства. Вид самого редактируемого компонента нам вполне подходит, поэтому было бы логично сделать наследника компонента `MinMaxControl` и реализовать в нем необходимую функциональность (листинг 9.3):

- конструктор должен обязательно устанавливать размеры компонента. Ширина редактора будет определяться шириной редактора свойств, а вот высота будет определена самим компонентом (если не включено разрешение "изменять размеры диалога");
- метод `OnMouseDown()` позволит устанавливать новое значение, отмечая позицию на компоненте (метод `SetNewvalue()`);
- событие `valueChanged` будет вызываться при выборе значения и позволит закрыть окно диалога редактирования сразу после выбора значения (обработчик `editorControl_valueChanged` в классе `PositionUIEditor`).

Вид получившегося редактора показан на рис. 9.7. Если перекрыть свойство `IsDropDownResizable` и возвращать `true`, то редактор будет иметь возможность изменять размер (рис. 9.8):

```
public override bool IsDropDownResizable
{
    get
```

```
(
    return true;
)
}
```

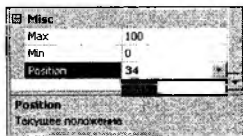


Рис. 9.7. Вид редактора свойства Position

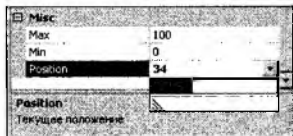


Рис. 9.8. Вид редактора свойства Position, если IsDropDownResizable возвращает true

Листинг 9.1. Компонент MinMaxControl

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using System.Drawing;
using System.ComponentModel;
using System.Drawing.Design;

namespace MyControl
{
    public class MinMaxControl : Control
    {
        private int min = 0;
        private int max = 100;
        private int position = 50;

        [Description("Минимум")]
        [DefaultValue(0)]
        public int Min
        {
            get { return min; }
            set { min = value; Invalidate(); }
        }

        [Description("Максимум")]
        [DefaultValue(100)]
```

```
public int Max
{
    get { return max; }
    set { max = value; Invalidate(); }
}

[Description("Текущее положение")]
[Editor(typeof(PositionUIEditor), typeof(UITypeEditor))]
public int Position
{
    get { return position; }
    set { position = value; Invalidate(); }
}

// Отрисовка
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);

    if (Min != Max)
    {
        SolidBrush fillBrush = new SolidBrush(Color.Blue);
        SolidBrush backBrush = new SolidBrush(this.BackColor);

        // Рисуем прямоугольник
        Rectangle nonFillPart = ClientRectangle;
        float percentValue =
            ((float)Position / ((float)Max - (float)Min));
        int nonDimLength =
            (int)(percentValue * (float)nonFillPart.Width);
        nonFillPart.X += nonDimLength;
        nonFillPart.Width -= nonDimLength;
        // Заливаем весь компонент
        e.Graphics.FillRectangle(fillBrush, ClientRectangle);
        // Восстанавливаем цвет в незаполненной части
        e.Graphics.FillRectangle(backBrush, nonFillPart);

        fillBrush.Dispose();
        backBrush.Dispose();
    }

    // Памка
    Rectangle rect = ClientRectangle;
```

```

        e.Graphics.DrawRectangle(SystemPens.WindowText,
                                rect.X, rect.Y, rect.Width - 1, rect.Height - 1);
    }
}

```

Листинг 9.2 Редактор типа для свойства Position

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing.Design;
using System.ComponentModel;
using System.Windows.Forms.Design;

namespace MyControl
{
    class PositionUIEditor : UITypeEditor
    {
        private IWindowsFormsEditorService edSvc = null;

        // Стил ь редактора – выпадающий диалог
        public override UITypeEditorEditStyle GetEditStyle(
            System.ComponentModel.ITypeDescriptorContext context)
        {
            return UITypeEditorEditStyle.DropDown;
        }

        public override object EditValue(ITypeDescriptorContext context,
            IServiceProvider provider, object value)
        {
            if (context != null &&
                context.Instance != null && provider != null)
            {
                // Получаем интерфейс сервиса
                edSvc = (IWindowsFormsEditorService)
                    provider.GetService(typeof(IWindowsFormsEditorService));
                if (edSvc != null)
                {
                    // Создаем компонент для редактирования
                    PositionEditorControl editorControl =
                        new PositionEditorControl();

```

```

// Подписываемся на получение уведомлений
// об изменении значения
editorControl.ValueChanged +=
    new EventHandler(editorControl_ValueChanged);
// Текущий редактируемый компонент
MinMaxControl currentControl =
    context.Instance as MinMaxControl;

// Устанавливаем свойства и текущее
// значение для редактирования
editorControl.Min = currentControl.Min;
editorControl.Max = currentControl.Max;
editorControl.Position = (int)value;

// Вызываем DropDownControl
edSvc.DropDownControl(editorControl);

// Получаем новое значение
value = editorControl.Position;
    }
}
// Возвращаем либо старое, либо новое значение
return value;
}

// Вызывается при изменении значения свойства
// в редакторе типа
void editorControl_ValueChanged(object sender, EventArgs e)
{
    if (edSvc != null)
    {
        // Закрыть редактор
        edSvc.CloseDropDown();
    }
}
}
}
}

```

Листинг 9.3. Компонент для редактора типа для свойства `Position`

```

using System;
using System.Collections.Generic;
using System.Text;

```

```
using System.Windows.Forms;
using System.Drawing;

namespace MyControl
{
    class PositionEditorControl : MinMaxControl
    {
        // Событие, которое будет вызываться при изменении значения.
        // Означает, что пользователь выбрал новое значение.
        private EventHandler onValueChanged;
        public event EventHandler ValueChanged
        {
            add
            {
                onValueChanged += value;
            }
            remove
            {
                onValueChanged -= value;
            }
        }

        // В конструкторе надо задать размеры
        public PositionEditorControl()
        {
            this.Size = new System.Drawing.Size(100, 15);
        }

        // Пользователь выбирает новое значение щелчком мыши
        protected override void OnMouseDown(MouseEventArgs e)
        {
            base.OnMouseDown(e);
            SetNewValue(new Point(e.X, e.Y));
        }

        // Вычислить и установить новое значение свойства
        private void SetNewValue(Point pointClick)
        {
            if (ClientRectangle.Contains(pointClick))
            {
                float percentage =
                    (float)pointClick.X / (float)ClientRectangle.Width;
```



```
        Position = (int)(percentage * (float)(Max - Min));
        OnValueChanged(EventArgs.Empty);
    }
}

// Вызывается при изменении значения свойства
// Позволяет редактору типа закрыть редактирование
protected void OnValueChanged(EventArgs e)
{
    if (onValueChanged != null)
    {
        onValueChanged.Invoke(this, e);
    }
}
}
```

В заключение раздела замечу, что параметр `context` имеет еще несколько полезных вещей:

- методы `OnComponentChanging` и `OnComponentChanged` позволяют уведомить дизайнера о начале и завершении изменения свойств компонента;
- свойство `PropertyDescriptor` предоставляет доступ к дескриптору редактируемого свойства.

В следующем разделе я расскажу о редакторе в виде модального диалога.

9.3.2. Модальный диалог

Отображение модального диалога подразумевает отображение некой формы, в которой производится редактирование значения свойства. Для свойства `Position` компонента `MinMaxControl` такая форма может выглядеть как на рис. 9.9. Щелчок на прямоугольнике (на самом деле это компонент `PositionEditorControl`) устанавливает новое значение, кнопка **Сохранить** сохраняет изменения, а кнопка **Отмена** оставляет старое значение свойства. Код формы очень простой (листинг 9.4). В конструкторе производится инициализация свойств формы, а свойство `Position` позволяет получить новое значение.

Для отображения формы используется метод `ShowModal()`, как показано в листинге 9.5. Код достаточно простой и почти полностью совпадает с тем, который мы уже обсуждали, поэтому я не буду останавливаться на нем дополнительно.

Теперь редактирование свойства `Position` будет производиться с помощью модального диалога.

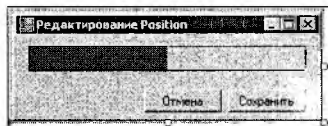


Рис. 9.9. Форма для редактирования свойства `Position`

Листинг 9.4. Форма редактирования свойства `Position`

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MyControl
{
    public partial class PositionEditForm : Form
    {
        public int Position
        {
            get
            {
                return positionEditorControl.Position;
            }
            set
            {
                positionEditorControl.Position = value;
            }
        }

        public PositionEditForm(MinMaxControl control)
        {
            InitializeComponent();
            positionEditorControl.Min = control.Min;
            positionEditorControl.Max = control.Max;
        }
    }
}
```

```
        positionEditorControl.Position = control.Position;
    }
}
}
```

Листинг 9.5. Редактор для редактирования свойства Position

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing.Design;
using System.ComponentModel;
using System.Windows.Forms.Design;
using System.Windows.Forms;

namespace MyControl
{
    class PositionUIEditor : UITypeEditor
    {
        private IWindowsFormsEditorService edSvc = null;

        // Стиль редактора – модальный диалог
        public override UITypeEditorEditStyle GetEditStyle(
            System.ComponentModel.ITypeDescriptorContext context)
        {
            return UITypeEditorEditStyle.Modal;
        }

        public override object EditValue(ITypeDescriptorContext context,
            IServiceProvider provider,
            object value)
        {
            if (context != null &&
                context.Instance != null && provider != null)
            {
                // Получаем интерфейс сервиса
                edSvc = (IWindowsFormsEditorService)
                    provider.GetService(typeof(IWindowsFormsEditorService));

                if (edSvc != null)
                {
                    // Текущий редактируемый компонент
                    MinMaxControl currentControl =
                        context.Instance as MinMaxControl;
                }
            }
        }
    }
}
```

```

        // Создаем форму для редактирования
        using (PositionEditForm form =
            new PositionEditForm(currentControl))
        {
            // Вызываем модальный диалог
            if (edSvc.ShowDialog(form) == DialogResult.OK)
            {
                // Получаем новое значение
                value = form.Position;
            }
        }
    }
    // Возвращаем либо старое, либо новое значение
    return value;
}
}
}

```

9.3.3. Изображение значений

Изображение значений используется в стандартных компонентах довольно часто. Для свойства `ImageIndex` отображается картинка, для цвета — цветной квадратик и т. д. Для свойства `Position` компонента `MinMaxControl` можно отображать само значение, так же как это делает сам компонент.

Включить вывод изображения можно, перекрыв метод `GetPaintValueSupported()`:

```

public override bool GetPaintValueSupported(
    ITypeDescriptorContext context)
{
    // Редактируемый компонент
    MinMaxControl control = (MinMaxControl)context.Instance;
    // Если есть возможность показать значение
    return control.Min != control.Max;
}

```

Сама отрисовка производится в методе `PaintValue()`, код которого полностью идентичен коду рисования самого компонента. Единственный параметр этого метода `e` передает всю необходимую для рисования информацию. Границу заливки можно получить с помощью свойства `e.Bounds`, ссылку на сам компонент возвращает свойство `e.Context.Instance`. Полный код показан в листинге 9.6, а получившийся результат — на рис. 9.10.

Листинг 9.6. Изображение значения свойства Position с помощью метода PaintValue

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing.Design;
using System.ComponentModel;
using System.Windows.Forms.Design;
using System.Drawing;

namespace MyControl
{
    class PositionUIEditor : UITypeEditor
    {
        // Рисовать изображение значений
        public override bool GetPaintValueSupported(
            ITypeDescriptorContext context)
        {
            // Редактируемый компонент
            MinMaxControl control = (MinMaxControl)context.Instance;
            // Если есть возможность показать значение
            return control.Min != control.Max;
        }

        public override void PaintValue(PaintValueEventArgs e)
        {
            base.PaintValue(e);

            // Редактируемый компонент
            MinMaxControl control = (MinMaxControl) e.Context.Instance;
            // Значение свойства
            int value = (int)e.Value;

            if (control.Max != control.Min)
            {
                SolidBrush fillBrush = new SolidBrush(Color.Blue);
                SolidBrush backBrush = new SolidBrush(Color.White);

                // Рисуем прямоугольник
                Rectangle nonFillPart = e.Bounds;
                float percentValue = ((float)value /
                    ((float)control.Max - (float)control.Min));
```

```

int nonDimLength =
    (int)(percentValue * (float)nonFillPart.Width);
nonFillPart.X += nonDimLength;
nonFillPart.Width -= nonDimLength;
// Заливаем весь компонент
e.Graphics.FillRectangle(fillBrush, e.Bounds);
// Восстанавливаем цвет в незаполненной части
e.Graphics.FillRectangle(backBrush, nonFillPart);

fillBrush.Dispose();
backBrush.Dispose();
}
}
}
}

```

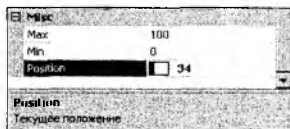


Рис. 9.10. Изображение значения свойства Position

9.3.4. Использование стандартных редакторов типов

Конечно же, не всегда нужно писать собственные редакторы типов. Часто вполне достаточно стандартных. В этом разделе я хочу рассказать о некоторых из них и возможных проблемах, возникающих при их использовании.

Редактирование коллекций

Далее я расскажу о редакторе типа `CollectionEditor` и связанных с ним редакторах.

Будьте внимательны: класс `CollectionEditor` находится в сборке `System.Design.dll`. Некоторые другие классы, о которых я буду рассказывать далее, находятся в сборке `System.Drawing.dll`. Часто ссылки на эти сборки нужно подключать вручную.

Редактирования *StringCollection*

Давайте посмотрим на простое свойство типа *StringCollection*:

```
private StringCollection stringCollection = new StringCollection();  
public StringCollection StringCollection  
{  
    get { return stringCollection; }  
    set { stringCollection = value; }  
}
```

При добавлении компонента с таким свойством на форму редактор свойств отображает кнопку с тремя точками, показывая, что он готов его редактировать. Более того, при нажатии на эту кнопку появляется редактор строковой коллекции, но вот добавить в него новую строку не получится — редактор выдает ошибку "Constructor on type 'System.String' not found" (рис. 9.11).

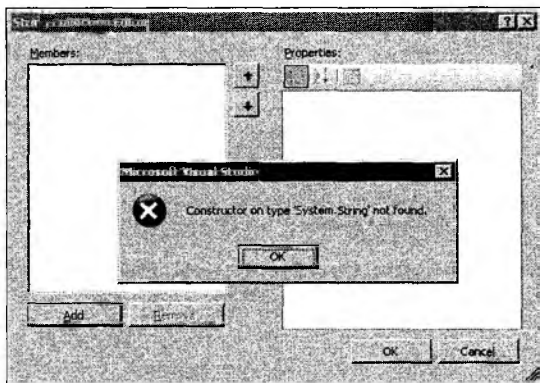


Рис. 9.11. Ошибка при попытке редактирования *StringCollection*

Исправить ситуацию можно добавлением атрибута, определяющего редактор типа *StringCollectionEditor*:

```
[DesignerSerializationVisibility(  
    DesignerSerializationVisibility.Content)]  
[Editor( "System.Windows.Forms.Design.StringCollectionEditor,  
        System.Design",  
    "System.Drawing.Design.UITypeEditor, System.Drawing")]}
```

```
public StringCollection StringCollection
{
    get { return stringCollection; }
    set { stringCollection = value; }
}
```

Вид редактора при этом изменяется на обычный редактор многострочного текста (рис. 9.12). Про атрибут `DesignerSerializationVisibility` я буду рассказывать в главе 10.

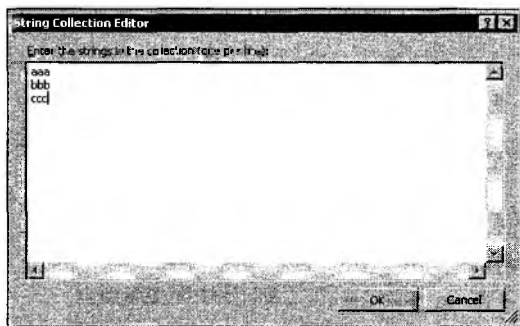


Рис. 9.12. Вид стандартного редактора String Collection Editor

Кстати, в некоторых случаях указание строковых имен типов редакторов не работает, и лучше либо указывать редакторы с помощью прямого указания типа `Editor(typeof(StringCollectionEditor), typeof(UITypeEditor))`, либо указывать конкретную версию сборки, в которой надо искать указанные типы¹:

```
[Editor("System.Windows.Forms.Design.StringCollectionEditor,
        System.Design,
        Version=2.0.0.0, Culture=neutral,
```

¹ Я привожу версии сборок для Framework 2.0. Для Framework 1.1 атрибут будет выглядеть так: `[Editor("System.Windows.Forms.Design.StringCollectionEditor, System.Design, Version=1.0.5000.0, culture=neutral, PublicKeyToken=b03f5f7f11d50a3a", "System.Drawing.Design.UITypeEditor, System.Drawing, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")]`.


```
PublicKeyToken=b03f5f7f11d50a3a",  
"System.Drawing.Design.UITypeEditor, System.Drawing,  
Version=2.0.0.0, Culture=neutral,  
PublicKeyToken=b03f5f7f11d50a3a"
```

```
}}
```

Теперь коллекцию строк можно редактировать без проблем.

Редактирование списка объектов

Теперь рассмотрим свойство, имеющее тип списка (в нашем случае `ArrayList`), который будет состоять из объектов типа `Person` (листинг 9.7).

```
private PersonArray personArray = new PersonArray();  
public PersonArray PersonArray  
{  
    get { return personArray; }  
    set { personArray = value; }  
}
```

Редактор свойств понимает, что это список неких объектов, но, конечно же, понять, что это список именно объектов `Person`, не может. Соответственно, редактор этого свойства выглядит не очень красиво и, прямо скажем, бесполезным (рис. 9.13).

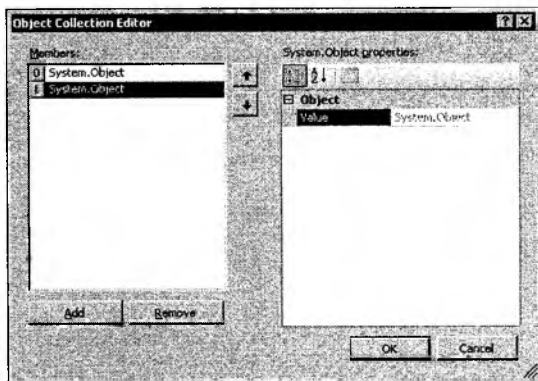


Рис. 9.13. Стандартный редактор коллекции объектов

Листинг 9.7. Классы `Person` и `PersonArray`

```
public class PersonArray : ArrayList
{
}

public class Person
{
    private string firstName = string.Empty;
    private string lastName = string.Empty;

    [Category("Персональные данные")]
    [Description("Имя")]
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    [Category("Персональные данные")]
    [Description("Фамилия")]
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }
}
```

Для добавления возможности редактирования списка объектов нужно создать специальный класс редактора (класс `PersonArrayEditor`). Базовым классом этого редактора является класс `CollectionEditor`, а для того чтобы редактор мог работать с объектами типа `Person`, требуется выполнить несколько действий:

- ☐ класс `PersonArrayEditor` должен перекрывать метод `CreateInstance()`. Этот метод должен возвращать объект `Person` и добавлять его в коллекцию;
- ☐ класс `PersonArray` должен реализовывать индексатор `this[index]`. Он позволяет редактору узнать, объекты какого типа будут добавляться в коллекцию;
- ☐ и конечно, редактор `PersonArrayEditor` должен быть привязан к свойству `PersonArray` (или типу `PersonArray`) с помощью атрибута `Editor`.

Код трех этих классов приведен в листингах 9.8—9.10. Результат работы этого кода показан на рис. 9.14. Обратите внимание на заголовок окна. Вместо

Object Collection Editor редактор подставляет название редактируемого типа объектов, и теперь окно называется **Person Collection Editor**.

Еще одна деталь на рис. 9.14: по умолчанию каждый добавляемый объект имеет имя типа, т. е. `MyControl.Person` (второй объект на рис. 9.14). Выводить дополнительную информацию позволяет метод `GetDisplayText()` класса `PersonArrayEditor`:

```
protected override string GetDisplayText(object value)
{
    Person person = value as Person;
    string result = base.GetDisplayText(value);

    if ((person != null) && (person.LastName.Length > 0 ||
        person.FirstName.Length > 0))
    {
        result = string.Format("{0} {1}", person.FirstName, person.LastName);
    }
    return result;
}
```

Теперь вместо стандартного `MyControl.Person` мы увидим имя и фамилию (первый объект на рис. 9.14).

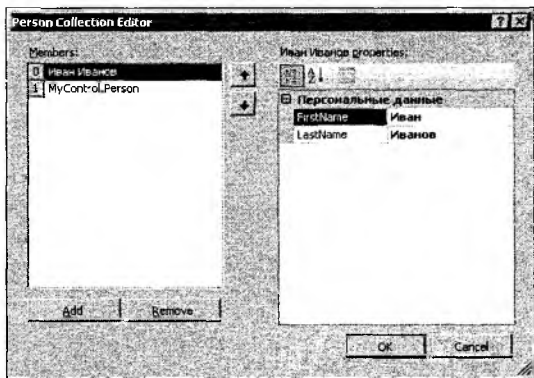


Рис. 9.14. Редактор коллекции объектов `Person`

Кроме атрибута `Category`, который мы использовали при описании полей класса `Person` (и, соответственно, все свойства на рис. 9.14 принадлежат указанной категории), можно использовать атрибут `DisplayName`, позволяющий изменить имя отображаемого свойства:

```
{DisplayName("Фамилия")}  
public string LastName  
{  
    get/set  
}
```

А вот с описанием свойств (атрибут `Description`) придется подождать до разд. "Изменение формы редактора коллекций".

Для добавления возможности сериализации свойства `PersonArray` требуется добавить к свойству атрибут `DesignerSerializationVisibility` (см. главу 10). Таким образом, полный список атрибутов свойства `personArray` будет выглядеть так:

```
[Editor(typeof(PersonArrayEditor), typeof(UITypeEditor))]  
[TypeConverter(typeof(PersonArrayTypeConverter))]  
[DesignerSerializationVisibility(  
    DesignerSerializationVisibility.Content)]  
public PersonArray PersonArray
```

Теперь наш редактор стал полнофункциональным.

Листинг 9.8. Класс `PersonArray`

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.Collections;  
using System.ComponentModel;  
using System.Drawing.Design;  
  
namespace MyControl  
{  
    public class PersonArray : ArrayList  
    {  
        public new Person this[int index]  
        {  
            set { base[index] = value; }  
            get { return (Person)base[index]; }  
        }  
    }  
}
```

ЛИСТИНГ 9.9. Класс PersonArrayEditor

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.Design;

namespace MyControl
{
    class PersonArrayEditor : CollectionEditor
    {
        public PersonArrayEditor(Type type) : base(type)
        {
        }

        protected override string GetDisplayText(object value)
        {
            Person person = value as Person;
            string result = base.GetDisplayText(value);

            if ((person != null) &&
                (person.LastName.Length > 0 || person.FirstName.Length > 0))
            {
                result = string.Format("{0} {1}",
                    person.FirstName, person.LastName);
            }
            return result;
        }

        protected override object CreateInstance(Type itemType)
        {
            object obj = null;

            if (itemType.FullName == "MyControl.Person")
            {
                PersonArray array = ((MyComponent) Context.Instance).PersonArray;
                obj = new Person();
                array.Add(obj);
            }
            return obj;
        }
    }
}
```

Листинг 9.10. Привязка редактора к свойству

```
[Editor(typeof(PersonArrayEditor), typeof(UITypeEditor))]
public PersonArray PersonArray
{
    get { return personArray; }
    set { personArray = value; }
}
```

Редактирование списка объектов нескольких типов

Коллекция объектов не обязательно должна состоять из объектов одного типа. В таких случаях редактору коллекции необходимо сообщить, объекты каких типов предполагается добавлять в коллекцию. Сделать это можно, переопределив метод `CreateNewItemTypes()`:

```
protected override Type[] CreateNewItemTypes()
{
    return new Type[] { typeof(Button), typeof(ListBox), typeof(Label) }
```

Но теперь в методе `CreateInstance()` мы уже не можем создать экземпляр нужного типа напрямую, т. к. типов может быть несколько. Конечно, можно использовать оператор `switch`, но это решение не очень гибкое, и мы воспользуемся классом `Activator`:

```
protected override object CreateInstance(Type itemType)
{
    // Здесь надо получить сам массив array.
    // Создаем экземпляр класса через Activator.
    object obj = Activator.CreateInstance(itemType);
    // Добавляем в массив
    array.Add(obj);
    // Возвращаем сам объект
    return obj;
}
```

Правда, если создаваемые объекты имеют различные конструкторы, все-таки придется использовать `switch` или `if`.

Вернемся к нашему примеру. Пусть у нас есть два наследника класса `Person` (листинг 9.11). Не обращайте внимания, что классы пустые, сейчас это не главное. Соответственно, реализация метода `CreateNewItemTypes()` будет выглядеть примерно так:

```
protected override Type[] CreateNewItemTypes()
{
    return new Type[] {
        typeof(Person), typeof(Person1), typeof(Person2) };
}
```

Такое решение не всегда является удобным, поэтому здесь я хочу рассказать про специальный сервис, позволяющий получать типы, наследованные от указанного. Сервис называется `ITypeDiscoveryService`. Как обычно, получение ссылки на сервис производится с помощью метода `GetService()`, но... в методе `CreateNewItemTypes()` нет ссылки на провайдер, позволяющий получить сервис. Зато такой параметр есть в методе `EditValue()`, и в нем нам придется заполнять специальный массив типов, который потом мы вернем в методе `CreateNewItemTypes()`.

Получив интерфейс сервиса `ITypeDiscoveryService`, можно вызывать метод `GetTypes()`, возвращающий все типы, которые являются наследниками типа указанного в качестве параметра. Полученный список мы сохраняем в специальный список типов `result`:

```
foreach (Type type in tds.GetTypes(typeof(Person), false))
{
    if (!result.Contains(type))
    {
        result.Add(type);
    }
}
```

Полный код получившегося редактора представлен в листинге 9.12, а его вид показан на рис. 9.15. Как видно из рисунка, вместо обычной кнопки **Add** редактор отображает кнопку с выпадающим списком, который позволяет выбрать тип добавляемого объекта.

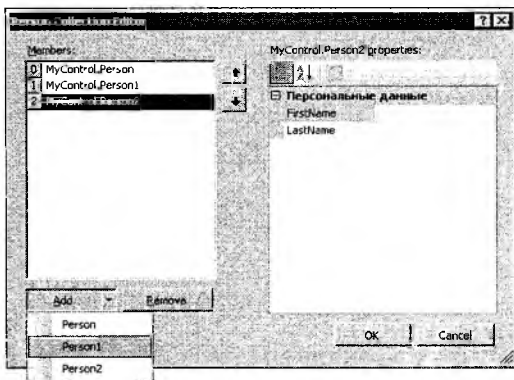


Рис. 9.15. Редактор коллекции объектов нескольких типов

Листинг 9.11. Классы `Person`, `Person1` и `Person2`

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;

namespace MyControl
{
    public class Person
    {
        private string firstName;
        private string lastName;

        [Category("Персональные данные")]
        [Description("Имя")]
        public string FirstName
        {
            get { return firstName; }
            set { firstName = value; }
        }

        [Category("Персональные данные")]
        [Description("Фамилия")]
        public string LastName
        {
            get { return lastName; }
            set { lastName = value; }
        }

        public override string ToString()
        {
            if (!String.IsNullOrEmpty(LastName) &&
                !String.IsNullOrEmpty(FirstName))
            {
                return string.Format("{0} {1}", FirstName, LastName);
            }

            return base.ToString();
        }
    }

    public class Person1 : Person
    {
    }
}
```



```
public class Person2 : Person
{
}
}
```

Листинг 9.12. Редактор коллекции, состоящей из нескольких типов

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.Design;
using System.ComponentModel;
using System.Windows.Forms;

namespace MyControl
{
    class PersonArrayEditor : CollectionEditor
    {
        private Type[] returnedTypes;

        public PersonArrayEditor(Type type) : base(type)
        {
        }

        // Сообщаем редактору возможные типы объектов
        protected override Type[] CreateNewItemTypes()
        {
            return returnedTypes;
        }

        public override object EditValue(
            ITypeDescriptorContext context,
            IServiceProvider provider, object value)
        {
            if (returnedTypes == null)
            {
                // Если мы хотим использовать сервис, то придется
                // инициализировать доступные типы здесь,
                // т. к. нам нужен объект provider
                returnedTypes = GetReturnedTypes(provider);
            }
            return base.EditValue(context, provider, value);
        }
    }
}
```

```
// Получаем доступные типы
private Type[] GetReturnedTypes(IServiceProvider provider)
{
    List<Type> result = new List<Type>();

    // Получаем интерфейс сервиса
    ITypeDiscoveryService tds = (ITypeDiscoveryService)
        provider.GetService(typeof(ITypeDiscoveryService));

    if (tds != null)
    {
        // Нам нужны все "родственники" типа Person
        foreach (Type type in tds.GetTypes(typeof(Person), false))
        {
            // Если в нашем списке такого типа еще нет,
            // то добавляем его
            if (!result.Contains(type))
            {
                result.Add(type);
            }
        }
    }

    return result.ToArray();
}

// Представление объекта в редакторе
protected override string GetDisplayText(object value)
{
    if (value != null)
        return value.ToString();
    else
        return base.GetDisplayText(value);
}

// Объекты теперь имеют разный тип, поэтому
// для создания экземпляра объекта используется класс Activator
protected override object CreateInstance(Type itemType)
{
    PersonArray array = ((MyComponent)Context.Instance).PersonArray;
    object obj = Activator.CreateInstance(itemType);
    array.Add(obj);
}
```

```
        return obj;
    }
}
```

Изменение формы редактора коллекций

Класс `CollectionEditor` содержит внутренний класс `CollectionForm`, с помощью которого реализуется редактор коллекций. Получить ссылку на эту форму можно посредством метода `CreateCollectionForm()` класса `CollectionEditor`. Например, следующий код заменяет заголовок окна:

```
protected override CollectionForm CreateCollectionForm()
{
    CollectionForm form = base.CreateCollectionForm();
    form.Text = "Редактор списка людей";
    return form;
}
```

При большом желании можно реализовать собственный редактор и вернуть его в качестве результата этого метода.

Класс `CollectionForm` является наследником класса `Form`, соответственно, с редактором коллекций можно работать как с обычной формой: перебирать элементы, добавлять свои и т. д.:

```
// Перебираем все элементы формы
foreach (Control control in form.Controls)
{
    foreach (Control controll in control.Controls)
    {
    }
}
```

Как видно из кода, перебор всех элементов формы нужно проводить во вложенном цикле. В качестве примера я покажу, как отобразить значение атрибута `Description` в редакторе коллекции (рис. 9.16), так же как он отображается в обычном редакторе свойств. Сделать это очень просто — нужно найти компонент с типом `VsPropertyGrid` и включить свойство `HelpVisible` (листинг 9.13).

Листинг 9.13. Включение отображения описания свойств

```
protected override CollectionForm CreateCollectionForm()
{
    CollectionForm form = base.CreateCollectionForm();
    form.Text = "Редактор списка людей";
```

```
// Перебираем все компоненты формы
foreach (Control control in form.Controls)
{
    foreach (Control controll in control.Controls)
    {
        // Нашли редактор свойств
        if (controll.GetType().ToString() ==
            "System.Windows.Forms.Design.VsPropertyGrid")
        {
            // Включаем отображение описаний свойств
            ((PropertyGrid)controll).HelpVisible = true;
            ((PropertyGrid)controll).HelpBackColor = SystemColors.Info;
        }
    }
}
return form;
}
```

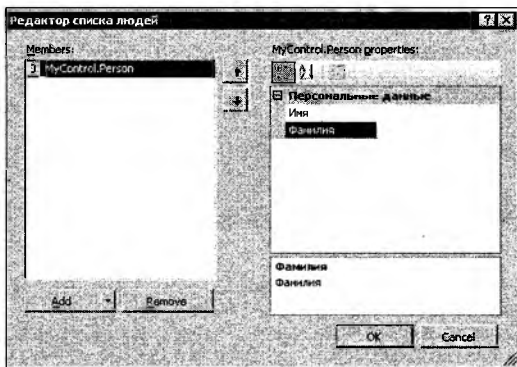


Рис. 9.16. Отображение описания свойства

Замена строки (Collection)

По умолчанию редактор свойств коллекций отображает строку (Collection). Заменить эту строку на более информативную (или локализованную) позволяет добавление в код специального конвертера типа (см. главу 8), код которого показан в листинге 9.14:

```
[Editor(typeof(PersonArrayEditor), typeof(UITypeEditor))]
[TypeConverter(typeof(PersonArrayTypeConverter))]
public PersonArray PersonArray
```

Теперь вместо стандартной строки мы увидим строку **Список...** (рис. 9.17).

Листинг 9.14. Конвертер типа для замены (Collection) на свою строку

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Globalization;

namespace MyControl
{
    class PersonArrayTypeConverter : TypeConverter
    {
        public override bool CanConvertTo(ITypeDescriptorContext context,
                                           Type destinationType)
        {
            if (destinationType == typeof(string))
                return true;
            return base.CanConvertTo(context, destinationType);
        }

        public override object ConvertTo(
            ITypeDescriptorContext context,
            CultureInfo culture, object value, Type destinationType)
        {
            if (destinationType == typeof(string))
            {
                return "Список...";
            }
            return base.ConvertTo(context, culture,
                                 value, destinationType);
        }
    }
}
```

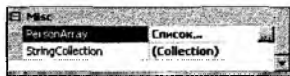


Рис. 9.17. Замена стандартной строки коллекции

Редактирование массива объектов

Сейчас я хочу рассказать еще об одном варианте набора объектов, когда поле описывается не в виде коллекции, а в виде массива:

```
private Person[] personArray1;

public Person[] PersonArray1
{
    get { return personArray1; }
    set { personArray1 = value; }
}
```

Для редактирования такого поля можно использовать стандартный редактор `ArrayEditor`, который, как и все другие, подключается к полю с помощью атрибута `Editor`:

```
[Editor(typeof(ArrayEditor), typeof(UITypeEditor))]
public Person[] PersonArray1
{
    get/set
}
```

Класс `ArrayEditor` является наследником класса `CollectionEditor`, и к нему применимо все, что я рассказывал о классе `CollectionEditor`. Вид окна редактора также не отличается, а вот отображение поля в редакторе свойств выглядит по-другому (рис. 9.18).

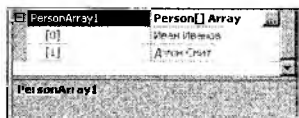


Рис. 9.18. Отображение массива объектов в редакторе свойств

Кроме того, такие свойства сериализуются в код именно как массив объектов (см. главу 10):

```
MyControl.Person person1 = new MyControl.Person();
MyControl.Person person2 = new MyControl.Person();
person1.FirstName = "Иван";
person1.LastName = "Иванов";
person2.FirstName = "Джон";
person2.LastName = "Смит";
```

```
this.myComponent1.PersonArray1 = new MyControl.Person[] {  
    person1, person2};
```

На этом мы закончим рассмотрение редактирования коллекций и массивов и перейдем к более простым редакторам.

Редактор выбора имени файла и каталога

Для выбора имени файла используется редактор `FileNameEditor`, но не напрямую, а с помощью наследника. Использовать редактор напрямую не всегда удобно, т. к. в этом случае редактор предлагает выбор всех файлов, а чаще всего требуется выбор файлов с определенным расширением. Пример такого редактора показан в листинге 9.15, а его внешний вид — на рис. 9.19. Задание фильтра расширений и других свойств диалога выбора файла производится в методе `InitializeDialog()`. Единственным параметром этого метода является сам диалог выбора файла. Описание его свойств можно найти в MSDN.

Листинг 9.15. Редактор выбора имени файла

```
using System;  
using System.Text;  
using System.ComponentModel;  
using System.Windows.Forms.Design;  
using System.Windows.Forms;  
using System.Drawing.Design;  
  
namespace MyControl  
{  
    public class FileNameTestComponent : Component  
    {  
        private string fileName;  
  
        // Свойство имени файла  
        [Editor(typeof(DocFileTypeEditor), typeof(UITypeEditor))]  
        public string FileName  
        {  
            get { return fileName; }  
            set { fileName = value; }  
        }  
    }  
  
    // Редактор имени файла  
    class DocFileTypeEditor : FileNameEditor
```

```

{
    // Инициализируем свойства диалога
    protected override void InitializeDialog(
        OpenFileDialog openFileDialog)
    {
        openFileDialog.CheckFileExists = false;
        openFileDialog.CheckPathExists = true;
        openFileDialog.DefaultExt = "doc";
        openFileDialog.RestoreDirectory = true;
        openFileDialog.Filter =
            "Документы (*.doc)|*.doc|Все файлы (*.*)|*.*";
    }
}
}

```

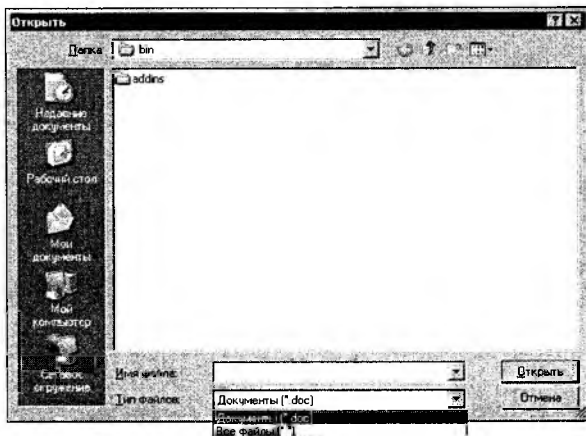


Рис. 9.19. Редактор имени файла, позволяющий выбрать расширение файла

Аналогично, редактор `FolderNameEditor` предоставляет возможность редактирования имени каталога и настройки диалога `FolderBrowser` (рис. 9.20):

```

class DocFolderTypeEditor : FolderNameEditor
{
    protected override void InitializeDialog(
        FolderNameEditor.FolderBrowser folderBrowser)

```



```
{  
    base.InitializeDialog(folderBrowser);  
    // Устанавливаем свойства folderBrowser  
}  
}
```

Описание класса FolderBrowser можно найти в MSDN.

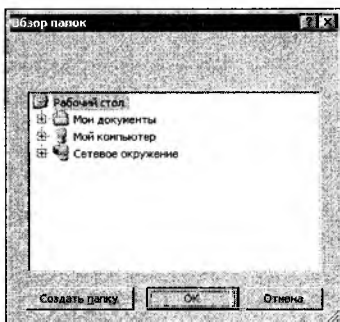


Рис. 9.20. Редактор имени каталога

Редактор-дерево

Еще один интересный редактор типа, о котором я хочу рассказать, позволяет отображать редактируемое значение в виде дерева. Соответствующий класс редактора называется `ObjectSelectorEditor` и является наследником обычного `UITypeEditor`. В дополнение ко всем базовым методам этот класс добавляет виртуальный метод:

```
void FillTreeWithData(  
    ObjectSelectorEditor.Selector selector,  
    ITypeDescriptorContext context,  
    IServiceProvider provider);
```

Перекрыв этот метод, можно заполнить дерево, представленное специальным классом `Selector`, конкретными значениями.

Класс `Selector` является наследником класса `TreeView` и фактически добавляет только один метод, позволяющий добавить новый узел в дерево:

```
public ObjectSelectorEditor.SelectorNode AddNode(
    string label,
    object value,
    ObjectSelectorEditor.SelectorNode parent);
```

Класс `SelectorNode` — это обычный `TreeNode`, но имеющий специальное поле `public object value;`

и соответствующий конструктор:

```
public SelectorNode(string label, object value);
```

Собственно, код редактора выглядит следующим образом:

```
public class SelectorEditor : ObjectSelectorEditor
{
    protected override void FillTreeWithData(Selector selector,
        ITypeDescriptorContext context, IServiceProvider provider)
    {
        base.FillTreeWithData(selector, context, provider);

        SelectorNode node1 = selector.AddNode("L1", "V1", null);
        SelectorNode node2 = selector.AddNode("L2", "V2", node1);
        SelectorNode node3 = selector.AddNode("L3", "V3", node1);

        selector.ShowLines = true;
        selector.ExpandAll();
    }
}
```

В результате редактор будет выглядеть так, как на рис. 9.21. Обратите внимание на последние две строки. Первая из них включает отображение линий в дереве, а вторая — раскрывает дерево. Без этих строк вы увидите только значение `L1`, а остальное будет скрыто.

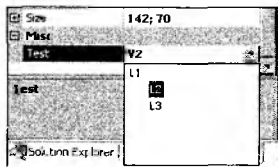


Рис. 9.21. Редактор в виде дерева

В примере, приведенном ранее, в дереве отображаются строки L1, L2, L3, но реальными значениями свойства будут являться значения v1, v2 и v3:

```
this.objectSelectorEditorControll.Test = "v2";
```

Разумеется, значениями свойства могут быть не только строки. Точнее, тип значений должен совпадать с типом свойства, к которому прикреплен редактор. Можно, например, организовать набор классов:

```
SelectorNode node1 = selector.AddNode(  
    "Button", new Button(), null);  
SelectorNode node2 = selector.AddNode(  
    "CheckBox", new CheckBox(), null);  
SelectorNode node3 = selector.AddNode(  
    "RadioButton", new RadioButton(), null);
```

Что-то подобное я показывал в *разд. 8.10*, но там список был линейным, а с помощью этого редактора можно создать дерево. Да и код получается более простым. Но самое существенное отличие заключается в том, что в этот список можно добавлять уже существующие объекты, не создавая их заново. Например, можно создать редактор выбора компонентов, находящихся на форме. Сделать это с помощью конвертера типа не получится.

Для того чтобы можно было редактировать свойства классов-значений, нужно добавить атрибут `ExpandableObjectConverter`:

```
private object test1;  
[Editor(typeof(SelectorEditor1), typeof(UITypeEditor))]  
[TypeConverterAttribute(typeof(ExpandableObjectConverter))]  
public object Test1  
{  
    get { return test1; }  
    set { test1 = value; }  
}
```

Как это выглядит в редакторе свойств, показано на рис. 9.22.

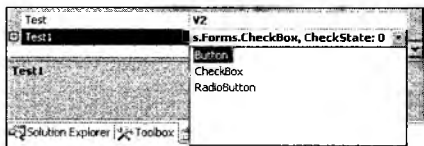


Рис. 9.22. Редактор в виде дерева

9.3.5. Редактор с параметрами

Подключенные редактора свойства производятся посредством атрибута, поэтому очевидно, что класс редактора будет создан с помощью конструктора по умолчанию, т. е. конструктора без параметров. Но что делать, если нужно вызвать конструктор с параметрами? Например, я хочу передавать параметр `resizable`, означающий возможность изменения размеров окна (разумеется, это только пример, особого практического смысла этот параметр не несет, здесь важен сам принцип передачи параметров):

```
class PositionUIEditor : UITypeEditor
{
    bool resizable = true;

    public PositionUIEditor(bool resizable)
    {
        this.resizable = resizable;
    }

    public override bool IsDropDownResizable
    {
        get
        {
            return resizable;
        }
    }
}
```

Для решения этого вопроса придется реализовать новый дескриптор свойства, который содержит вызов редактора, но уже с параметрами. Как я уже говорил, класс `PropertyDescriptor` является абстрактным, и использовать его напрямую нельзя. Поэтому мы сделаем наследника этого класса, но т. к. нам нужно только перекрыть метод вызова редактора, то все другие методы мы переадресуем к исходному дескриптору (листинг 9.16). Подмена редактора выглядит так:

```
if (editorBaseType == typeof(UITypeEditor))
    // Вызываем редактор с параметром!
    return new PositionUIEditor(false);
```

Листинг 9.16. Дескриптор свойства, вызывающий редактор с параметрами

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
using System.ComponentModel;
using System.Drawing.Design;

namespace MyControl
{
    public class PositionPropertyDescriptor : PropertyDescriptor
    {
        private PropertyDescriptor baseProperty;

        // Сохраняем исходный дескриптор
        public PositionPropertyDescriptor(PropertyDescriptor baseProperty)
            : base(baseProperty)
        {
            this.baseProperty = baseProperty;
        }

        // Подменяем редактор
        public override object GetEditor(Type editorBaseType)
        {
            if (editorBaseType == typeof(UITypeEditor))
            {
                // Вызываем редактор с параметром!
                return new PositionUIEditor(false);
            }
            return baseProperty.GetEditor(editorBaseType);
        }

        // Все остальные методы просто переадресуем к исходным
        public override bool CanResetValue(object component)
        {
            return baseProperty.CanResetValue(component);
        }

        public override Type ComponentType
        {
            get { return baseProperty.ComponentType; }
        }

        public override object GetValue(object component)
        {
            return baseProperty.GetValue(component);
        }

        public override bool IsReadOnly
        {
            get { return baseProperty.IsReadOnly; }
        }
    }
}
```

```
public override Type PropertyType
{
    get { return baseProperty.PropertyType; }
}

public override void ResetValue(object component)
{
    baseProperty.ResetValue(component);
}

public override void SetValue(object component, object value)
{
    baseProperty.SetValue(component, value);
}

public override bool ShouldSerializeValue(object component)
{
    return baseProperty.ShouldSerializeValue(component);
}
}
```

Теперь остается только подменить сам дескриптор. Сделать это можно с помощью добавления к компоненту дизайнера (см. главу 6):

```
[Designer(typeof(PositionDesigner))]
public class MinMaxControl : Control
{
}
```

Класс `PositionDesigner` перекрывает метод `PostFilterProperties()`, в котором и происходит подмена дескриптора (листинг 9.17).

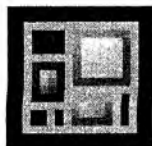
Разумеется, при необходимости можно реализовать более сложную логику, согласно нужным параметрам редактора. В этом примере вполне достаточно понимания того, что был вызван конструктор с параметрами, а не конструктор по умолчанию.

Листинг 9.17. Дизайнер, подменяющий дескриптор свойства `Position`

```
public class PositionDesigner : ControlDesigner
{
    protected override void PostFilterProperties(IDictionary properties)
```

```
{  
    // Подменяем дескриптор свойства  
   PropertyDescriptor baseProperty =  
        (PropertyDescriptor)properties["Position"];  
    properties["Position"] =  
        new PositionPropertyDescriptor(baseProperty);  
}
```

ГЛАВА 10



Сериализация времени разработки

Хуже всего на производительность влияют попытки ее увеличить.

В этой главе я постараюсь рассказать, каким образом среда Visual Studio производит сохранение информации о компонентах на форме.

10.1. Провайдеры

Для начала сделаем простое приложение, состоящее из одной формы, и разместим на ней компонент Button (рис. 10.1). Посмотрите на код, появившийся в файле Form1.Designer.cs:

```
this.button1 = new System.Windows.Forms.Button();  
this.button1.Location = new System.Drawing.Point(72, 35);  
this.button1.Name = "button1";  
this.button1.Size = new System.Drawing.Size(75, 23);  
this.button1.TabIndex = 0;  
this.button1.Text = "button1";  
this.button1.UseVisualStyleBackColor = true;
```

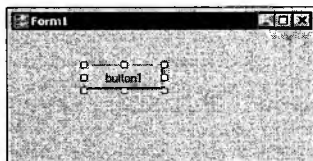


Рис. 10.1. Форма с компонентом Button

Теперь добавьте кнопке картинку (свойство `Image`) и посмотрите на получившийся код:

```
System.ComponentModel.ComponentResourceManager resources =
    new System.ComponentModel.ComponentResourceManager(typeof(Form1));
this.button1 = new System.Windows.Forms.Button();
this.button1.Image = ((System.Drawing.Image)
    (resources.GetObject("button1.Image")));
this.button1.Location = new System.Drawing.Point(72, 35);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "button1";
this.button1.UseVisualStyleBackColor = true;
```

Понятно, что среде не удалось сохранить картинку в виде кода и пришлось сохранить ее в ресурсном файле. Этот файл называется `Form1.resx`, а данные в нем выглядят примерно так:

```
<data name="button1.Image" type="System.Drawing.Bitmap,
    System.Drawing" mimetype=
    "application/x-microsoft.net.object.bytearray.base64">
    <value>
        Qk26AQEAAAAAAAAAAScWAAEGsAAGEAAABhAAAAaxAQ
    </value>
</data>
```

Генерация кода и данных, или, другими словами, *сериализация времени разработки* (*design time serialisation*) производится с помощью специальных провайдеров, которые называются *CodeDom-провайдеры*. Провайдером для C# является класс `Microsoft.CSharp.CSharpCodeProvider`, а провайдером для Visual Basic — класс `Microsoft.VisualBasic.VBCodeProvider`. Детально рассматривать работу провайдеров в рамках этой книги я не буду и ограничусь чисто практическими вопросами.

10.2. Преимущества модели провайдеров

За время развития языков программирования были сделаны разные попытки хранить данные о расположенных элементах. В первых версиях Delphi это были `dfm`-файлы бинарного вида, хранящие информацию времени выполнения. В более поздних версиях появилась возможность хранить такие файлы в текстовом виде. В Visual Basic такая информация хранилась в специальных `frm`-файлах и т. д.

Основных характеристик, которые должен иметь файл описания, две: файл должен иметь простой и понятный формат, доступный для редактирования в текстовом редакторе, а загрузка описания должна производиться достаточно быстро (особенно в режиме выполнения). По первой причине не подходят бинарные файлы. Их невозможно редактировать и неудобно сравнивать между собой. По этой же причине их неудобно хранить в репозитории кода. По второй причине не подходят текстовые файлы описаний, интерпретируемые во время выполнения. Конечно, их удобно читать и редактировать, но скорость их загрузки может оказаться критичной, особенно если форма содержит много компонентов.

Среда Visual Studio .NET использует очень интересный способ хранения информации времени выполнения: данные о компонентах сериализуются в код, что мы и видели в примере, который я показывал в начале этого раздела. Таким образом, информация хранится в текстовом виде, ее просто редактировать, и, т. к. в конечном итоге получается скомпилированный код, не тратится время на интерпретацию информации во время выполнения. Конечно, некоторое время тратится на преобразование кода в форму при открытии формы в режиме дизайнера, но это время невелико и не столь критично, как затраты времени выполнения.

10.3. Принцип работы провайдеров

Создание кода, описывающего форму, производится в два этапа. Первый этап заключается в создании "дерева кода", представляющего собой описание кода в виде объектов. Такое описание не зависит от конкретного языка программирования. После этого, на втором этапе, провайдер конкретного языка генерирует код на нужном языке программирования. Понятно, что если мы не собираемся писать собственный язык программирования, то вмешиваться в кодогенерацию не стоит, поэтому дальше мы будем обсуждать только первую часть процесса.

Итак, создание "дерева" происходит следующим образом:

1. Создается CodeDom-генератор нужного языка программирования (в нашем случае это C#).
2. Сериализатор перебирает все публичные (public) свойства компонента.
3. Определяет, подлежит ли свойство сериализации.
4. Если да, то сериализует содержимое свойства в дерево и передает его CodeDom-генератору.

Если свойство является простым типом, то проблем с сериализацией нет. Если свойство является компонентом (точнее сказать реализует интерфейс

IComponent), то процесс может повторяться рекурсивно. В случае, когда свойство не является ни простым типом, ни компонентом, дело обстоит сложнее. В этом случае среде нужно каким-то образом суметь сериализовать свойство, о котором она не имеет фактически никакой информации. Для этого используется конвертер типа (см. главу 8), позволяющий преобразовать исходный тип к типу InstanceDescriptor.

Для демонстрации последней проблемы сделаем компонент TestComponent, который содержит свойство Location типа LocationType, которое, в свою очередь, содержит поле типа Point (листинг 10.1). Для того чтобы поле можно было редактировать, мы подключим к нему стандартный конвертер ExpandableObjectConverter (см. разд. 8.9). Добавьте компонент на форму и загляните в получившийся код. Никакой информации о свойстве Location не сохранилось. Как я уже говорил, это происходит потому, что среда разработки не знает, каким образом нужно работать с классом LocationType.

Теперь подключите к классу LocationType конвертер, приведенный в листинге 10.2. Свойство будет сохранено в коде:

```
this.testComponent1.Location =  
    new SerializeTest.LocationType(new System.Drawing.Point(5, 5));
```

Основная работа конвертера происходит в методе ConvertTo(), который производит преобразование исходного типа в тип InstanceDescriptor:

```
ConstructorInfo ci = typeof(LocationType).GetConstructor(  
    new Type[] { typeof(Point) });  
LocationType t = (LocationType)value;  
return new InstanceDescriptor(ci, new object[] { t.Location });
```

Сначала мы получаем конструктор, который позволит нам создать объект нужного типа. В данном случае мы выбираем конструктор с одним параметром, но в общем случае может потребоваться выбирать один из нескольких конструкторов, в зависимости от значений полей объекта или, например, текущей культуры (параметр culture). Затем, на основе полей исходного объекта value мы создаем экземпляр класса InstanceDescriptor, который и позволяет Visual Studio сериализовать поле Location в код.

Теперь посмотрим, как можно управлять сериализацией полей классов.

Листинг 10.1. Класс с несериализуемым полем

```
using System;  
using System.ComponentModel;  
using System.ComponentModel.Design.Serialization;
```

```
using System.Reflection;
using System.Drawing;
using System.Globalization;

namespace SerializeTest
{
    // Тестовый компонент со свойством типа LocationType,
    // внутри которого есть свойство типа Point
    public class TestComponent : Component
    {
        LocationType location;

        public TestComponent()
        {
            location = new LocationType(new Point(5, 5));
        }

        public LocationType Location
        {
            get
            {
                return location;
            }
            set
            {
                location = value;
            }
        }
    }

    // Без подключения конвертера свойства этого типа не будут
    // сериализованы
    [TypeConverter(typeof(LocationClassConverter))]
    public class LocationType
    {
        Point location;

        public Point Location
        {
            get
            {
                return location;
            }
        }
    }
}
```

```
        set
        {
            location = value;
        }
    }

    public LocationType(Point location)
    {
        this.location = location;
    }
}
```

Листинг 10.2. Класс конвертера типа LocationType

```
class LocationClassConverter : TypeConverter
{
    public override bool GetPropertiesSupported(
        ITypeDescriptorContext context)
    {
        return true;
    }

    public override PropertyDescriptorCollection GetProperties(
        ITypeDescriptorContext context,
        object value, Attribute[] attributes)
    {
        return TypeDescriptor.GetProperties(typeof(LocationType));
    }

    // Возвращает возможность конвертирования
    public override bool CanConvertTo(
        ITypeDescriptorContext context, Type destinationType)
    {
        // Для InstanceDescriptor разрешаем конвертирование
        if (destinationType == typeof(InstanceDescriptor))
        {
            return true;
        }

        return base.CanConvertTo(context, destinationType);
    }
}
```

```
// Производит конвертирование
public override object ConvertTo(ITypeDescriptorContext context,
    CultureInfo culture, object value, Type destinationType)
{
    if (destinationType == typeof(InstanceDescriptor))
    {
        // Ищем конструктор с параметром Point
        ConstructorInfo ci =
            typeof(LocationType).GetConstructor(
                new Type[] { typeof(Point) });
        // Объект, который будет сериализовываться
        LocationType t = (LocationType)value;
        // Создаем InstanceDescriptor
        return new InstanceDescriptor(ci, new object[] { t.Location });
    }

    return base.ConvertTo(context, culture, value, destinationType);
}
```

10.4. Управление сериализацией

В предыдущем разделе я рассказывал, как среда осуществляет процесс сериализации. В частности о том, что при сериализации просматриваются все публичные свойства. Каждое из них анализируется на необходимость сериализации.

Указать необходимость сериализации можно с помощью атрибута `DesignerSerializationVisibility`:

- ☐ `DesignerSerializationVisibility.Hidden` — запретить сериализацию свойства;
- ☐ `DesignerSerializationVisibility.Visible` — разрешить сериализацию свойства;
- ☐ `DesignerSerializationVisibility.Content` — сериализовать свойство "по содержанию".

Зачем нужно запрещать сериализацию свойства? С несколькими случаями мы уже сталкивались. В *разд. 6.7* мы скрывали от сериализации виртуальные свойства, т. е. свойства, которые существуют только в режиме разработки. При исправлении проблемы в стандартном наборе значений (*см. разд. 8.5*) пришлось дублировать свойство `GradientProp`, а чтобы одно и то же свойство не сохранялось два раза, одно из них мы скрывали. Таким образом, можно

сформулировать две причины, по которым нужно скрывать свойство от сериализации:

- свойство не существует в режиме выполнения;
- значение свойства сохраняется каким-то другим образом.

Листинг 10.3 показывает пример компонента, имеющего два свойства, дублирующие свойства самого компонента. Понятно, что непосредственное значение этого свойства сохранится компонентом, и поэтому нет смысла сохранять их еще раз.

Еще один случай — дублирование одного из свойств "внутреннего" компонента:

```
[DesignerSerializationVisibility(
    DesignerSerializationVisibility.Hidden)]
public DateTime DateValue
{
    get{ return pTextBox.DateValue; }
    set{ pTextBox.DateValue = value; }
}
```

Возможен вариант, когда в сохранении свойства нет смысла, т. к. оно изменяется только в режиме выполнения:

```
[Browsable(false)]
[DesignerSerializationVisibility(
    DesignerSerializationVisibility.Hidden)]
public bool IsModified
{
    get{ return pTextBox.Modified; }
    set{ pTextBox.Modified = value; }
}
```

Но такие свойства нет смысла выносить в редактор свойств, и поэтому у него присутствует еще атрибут `Browsable(false)`.

Листинг 10.3. Компонент, имеющий дублированные свойства

```
using System;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.ComponentModel.Design.Serialization;
```

```
namespace ContentSerializationExample
{
    // Класс для демонстрации использования
    // атрибута DesignerSerializationVisibility
    public class ContentSerializationControl : UserControl
    {
        [DesignerSerializationVisibility(
            DesignerSerializationVisibility.Hidden)]
        public DimensionData Dimensions
        {
            get
            {
                return new DimensionData(this);
            }
        }
    }

    // Этот атрибут говорит, что свойства этого класса
    // будут показываться в Редакторе свойств как отдельные поля
    [TypeConverterAttribute(typeof(ExpandableObjectConverter))]
    public class DimensionData
    {
        // Сохраняем владельца свойств
        private Control owner;

        // Конструктор
        internal DimensionData(Control owner)
        {
            this.owner = owner;
        }

        // Свойство OwnerLocation берет значение у объекта owner
        public Point OwnerLocation
        {
            get
            {
                return owner.Location;
            }
            set
            {
                owner.Location = value;
            }
        }
    }
}
```



```
// Свойство OwnerSize берет значение у объекта owner
public Size OwnerSize
{
    get
    {
        return owner.Size;
    }
    set
    {
        owner.Size = value;
    }
}
}
```

Атрибут `DesignerSerializationVisibility.Content` указывает, что сериализация будет производиться с помощью сохранения содержимого объекта. Например, если это набор строк, как, например, в листинге 10.4, то соответствующий код будет представлять собой инициализацию строкового массива:

```
this.serializationDemoControll.Strings =
    new string[] { "aa", "bb", "cc"};
```

Обратите внимание на инициализацию свойства в конструкторе:

```
this.Strings = new string[0];
```

Если этого не сделать или инициализировать не свойство, а только поле `stringsValue`, то при пустом наборе строк иногда возникает ошибка (рис. 10.2).

Для типа `string[]` разницы между атрибутами `Visible` и `Content` не видно, а вот тип `StringCollection` показывает ее более наглядно. Сравните листинги 10.5 и 10.6. Я специально выделил полужирным шрифтом отличающийся код. Первый листинг сохраняет коллекцию в "режиме" `Visible`. Результат будет таким:

```
ComponentResourceManager resources =
    new ComponentResourceManager(typeof(Form1));
this.serializationDemoControll.StringCollection =
    ((StringCollection) (resources.GetObject(
        "serializationDemoControll.StringCollection")));
```

Во втором случае используется атрибут `Content` и результат будет таким:

```
this.serializationDemoControll.StringCollection.AddRange(
    new string[] { "aaa", "bbb"});
```

Как видно из результата сериализации, атрибут `visible` создает новый объект (вызывая метод `GetObject()`), а атрибут `Content` подразумевает, что объект уже существует. При попытке сериализовать неинициализированный объект Visual Studio выдаст ошибку (рис. 10.3).

Листинг 10.4. Сериализация "по содержимому"

```
using System;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.ComponentModel.Design.Serialization;
using System.Text;

namespace ContentSerializationExample
{
    public class SerializationDemoControl : UserControl
    {
        private String[] stringsValue;

        public SerializationDemoControl()
        {
            this.Strings = new string[0];
        }

        [DesignerSerializationVisibility(
            DesignerSerializationVisibility.Content)]
        [DefaultValue(new String[0])]
        public String[] Strings
        {
            get
            {
                return this.stringsValue;
            }
            set
            {
                if (value != null)
                {
                    this.stringsValue = value;
                }
            }
        }
    }
}
```

```

        else
        {
            this.stringsValue = new string[0];
        }
    }
}
}
}

```

Листинг 10.5. Сериализация `StringCollection` с атрибутом `Visible`

```

using System;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.ComponentModel.Design.Serialization;
using System.Text;
using System.Collections.Specialized;

namespace ContentSerializationExample
{
    public class SerializationDemoControl : UserControl
    {
        private StringCollection stringCollection = null;

        [DesignerSerializationVisibility(
            DesignerSerializationVisibility.Visible)]
        [Editor("System.Windows.Forms.Design.StringCollectionEditor,
            System.Design",
            "System.Drawing.Design.UITypeEditor, System.Drawing")]
        public StringCollection StringCollection
        {
            get { return stringCollection; }
            set { stringCollection = value; }
        }
    }
}

```

Листинг 10.6. Сериализация `StringCollection` с атрибутом `Content`

```

using System;
using System.ComponentModel;

```

```
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.ComponentModel.Design.Serialization;
using System.Text;
using System.Collections.Specialized;

namespace ContentSerializationExample
{
    public class SerializationDemoControl : UserControl
    {
        private StringCollection stringCollection = new StringCollection();

        [DesignerSerializationVisibility(
            DesignerSerializationVisibility.Visible)]
        [Editor("System.Windows.Forms.Design.StringCollectionEditor,
            System.Design",
            "System.Drawing.Design.UITypeEditor, System.Drawing")]
        public StringCollection StringCollection
        {
            get { return stringCollection; }
            set { stringCollection = value; }
        }
    }
}
```



Рис. 10.2. Ошибка при сериализации неинициализированного свойства

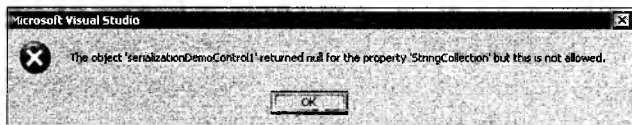


Рис. 10.3. Ошибка при попытке сериализации неинициализированного свойства в режиме Content

10.5. Динамическое управление сериализацией

Атрибуты `DesignerSerializationVisibility` и `DefaultValue` определяют необходимость сериализации статически, т. е. в конструкторы этих методов передаются константы. Если же необходимо управлять сериализацией программно, то нужно использовать другой способ: вместо статических атрибутов нужно сделать метод, имя которого состоит из строки `ShouldSerialize` и имени свойства. Например, для управления сериализацией свойства `BackColor` будет использоваться метод `ShouldSerializeBackColor`. В случае свойства `Strings` метод управления должен называться `ShouldSerializeStrings` (листинг 10.7). Использовать одновременно и метод управления, и атрибуты нельзя.

Атрибут `DefaultValue` определяет значение свойства по умолчанию (см. разд. 5.3). Аналогично методу, определяющему необходимость сериализации, для меню **Reset** можно создать метод `ResetИмяСвойства`, который будет вызываться при выборе этого меню. Например, для свойства `Strings` метод будет выглядеть так:

```
private void ResetStrings()
{
    stringsValue = new string[0];
}
```

При этом следует учитывать, что активность меню **Reset** зависит от значения метода `ShouldSerialize`. Если этот метод возвращает `true`, меню будет активным, а если `false`, то меню недоступно.

Листинг 10.7. Динамическое управление сериализацией

```
using System;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Windows.Forms;
using System.ComponentModel.Design.Serialization;
using System.Text;

namespace ContentSerializationExample
{
    public class SerializationDemoControl : UserControl
```

```
{
    private String[] stringsValue;

    public SerializationDemoControl()
    {
        // Инициализация обязательна
        this.Strings = new string[0];
    }

    private bool ShouldSerializeStrings()
    {
        return (stringsValue != null) && (stringsValue.Length > 0);
    }

    // Нельзя использовать и ShouldSerialize-метод,
    // и атрибут DefaultValue
    // [DefaultValue(new String[0])]
    public String[] Strings
    {
        get
        {
            return this.stringsValue;
        }
        set
        {
            if (value != null)
            {
                this.stringsValue = value;
            }
            else
            {
                this.stringsValue = new string[0];
            }
        }
    }
}
```

10.6. Локализация

Следующий шаг Microsoft в развитии модели сериализации в код состоял в реализации возможности локализации приложений. Посмотрите, как сериализуется текст кнопки:

```
this.button1 = new System.Windows.Forms.Button();  
this.button1.Name = "button1";  
this.button1.Text = "Execute";
```

При необходимости локализовать приложение (например, сменить название **Execute** на **Запустить**) придется где-то хранить новые названия всех локализуемых элементов, загружать такие названия и присваивать их. Редактор Visual Studio позволяет сделать это автоматически, без единой строки кода.

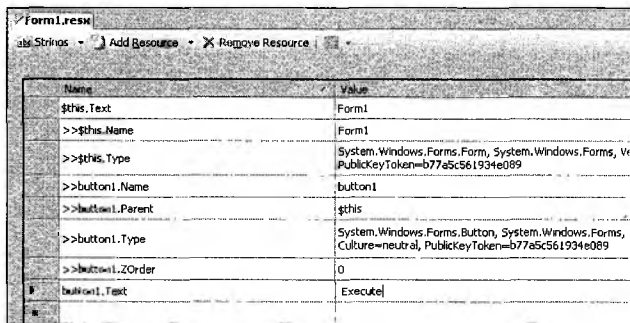
Выставив свойство `Localizable` главной формы в значение `true`, мы сообщаем сериализатору о необходимости сохранять значения всех названий не в код, а в ресурсный файл. В результате код будет выглядеть следующим образом:

```
private void InitializeComponent()  
{  
    ComponentResourceManager resources =  
        new ComponentResourceManager(typeof(Form1));  
    this.button1 = new System.Windows.Forms.Button();  
    this.SuspendLayout();  
    //  
    // button1  
    //  
    resources.ApplyResources(this.button1, "button1");  
    this.button1.Name = "button1";  
    this.button1.UseVisualStyleBackColor = true;  
    //  
    // Form1  
    //  
    resources.ApplyResources(this, "$this");  
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;  
    this.Controls.Add(this.button1);  
    this.Name = "Form1";  
    this.WaitCursor = true;  
    this.Load += new System.EventHandler(this.Form1_Load);  
    this.ResumeLayout(false);  
}
```

Ресурсный файл формы будет содержать всю информацию, доступную для локализации (рис. 10.4). Для того чтобы ресурсный файл был доступен для изменения в режиме выполнения, нужно изменить его свойства: значение свойства **Build Action** сменить с **Embedded Resource** на **Content**, а значение свойства **Copy to Output Directory** сменить с **Do not copy** на **Copy always** (рис. 10.5). Теперь ресурсный файл будет храниться отдельно (не будет

вкомпилирован в выполняемый модуль) и будет скопирован в выходной каталог.

Хочу обратить внимание, что при разработке компонента нужно стараться таким образом реализовывать свойства компонента, чтобы сделать возможным его дальнейшую локализацию. Будет не очень хорошо, если часть видимых пользователю названий будет доступна для локализации, а часть останется внутри кода компонента. Старайтесь максимально облегчить жизнь своему коллеге-программисту.



Name	Value
\$this.Text	Form1
>>\$this.Name	Form1
>>\$this.Type	System.Windows.Forms.Form, System.Windows.Forms, PublicKeyToken=b77a5c561934e089
>>button1.Name	button1
>>button1.Parent	\$this
>>button1.Type	System.Windows.Forms.Button, System.Windows.Forms, Culture=neutral, PublicKeyToken=b77a5c561934e089
>>button1.ZOrder	0
button1.Text	Execute

Рис. 10.4. Ресурсный файл хранит информацию, доступную для локализации



Рис. 10.5. Свойства ресурсного файла

10.7. Настраиваемые свойства

Очень часто свойства компонента одновременно оказываются и настройками программы или системы. Например, свойства `PortName` и `BaudRate` компонента `SerialPort`, очевидно, являются настройками системы. Было бы очень неудобно хранить эти настройки в коде программы и перекомпилировать ее при изменении номера порта или скорости обмена.

Один из вариантов — реализовать собственный интерфейс для редактирования таких настроек. Это несложно, но не всегда нужно. Часто вполне достаточно простого хранения настроек в конфигурационном файле, без возможности его редактирования из программы. Вполне подойдет даже обычный конфигурационный файл `app.config`.

Среда Visual Studio предоставляет очень удобную возможность автоматического сохранения значения свойств в конфигурационный файл. Каждый компонент имеет специальное виртуальное (т. е. доступное только в режиме разработки) свойство (**ApplicationSettings**), которое находится в секции **Data** (рис. 10.6). Редактор вложенного свойства (**PropertyBinding**) позволяет задать соответствие свойств компонента и секций в конфигурационном файле (рис. 10.7).

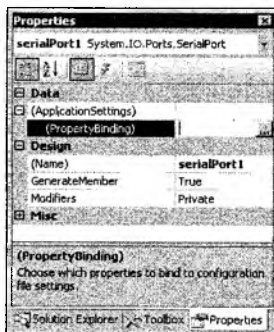


Рис. 10.6. Свойство (ApplicationSettings)

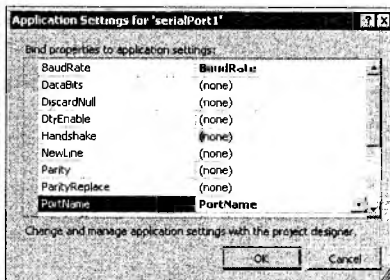


Рис. 10.7. Редактор свойства (PropertyBinding)

После привязки свойств `PortName` и `BaudRate` сериализованный код выглядит следующим образом:

```
this.serialPort1 = new System.IO.Ports.SerialPort(this.components);
this.serialPort1.BaudRate =
    global::Test.Properties.Settings.Default.BaudRate;
this.serialPort1.PortName =
    global::Test.Properties.Settings.Default.PortName;
```

А файл конфигурации выглядит так:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    * * * * *
  </configSections>
  <userSettings>
    <Test.Properties.Settings>
      <setting name="BaudRate" serializeAs="String">
        <value>9600</value>
      </setting>
      <setting name="PortName" serializeAs="String">
        <value>COM1</value>
      </setting>
    </Test.Properties.Settings>
  </userSettings>
</configuration>
```

Конечно, такой простой интерфейс редактирования настроек программы подходит не всегда, но часто он позволяет существенно сэкономить время разработки, если простого файла конфигурации достаточно и нет необходимости создавать специальные диалоги редактирования.

Несколько слов про еще один полезный атрибут. Если разработчик считает, что некоторое свойство было бы хорошо вынести в настройки приложения, он может сообщить об этом, установив атрибут `SettingsBindable(true)`¹:

```
[SettingsBindable(true)]
public TestProperty TestProperty
{
    get { return testProperty; }
    set { testProperty = value; }
}
```

Теперь такое свойство будет сразу вынесено в секцию **Data** рядом со свойством `PropertyBinding` (рис. 10.8).

¹В .NET 1.0/1.1 этот атрибут назывался `RecommendedAsConfigurableAttribute`.

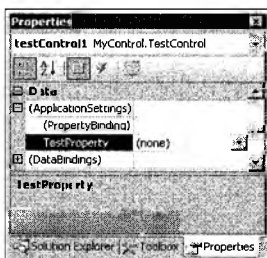


Рис. 10.8. Свойство, рекомендованное для настройки

10.8. Управление кодом сериализации

В заключение главы я хочу сказать несколько слов о формировании кода, генерируемого сериализатором. Должен сразу сказать, что эта тема огромная и, возможно, заслуживает отдельной книги. В этом разделе я могу лишь в общих чертах описать основные классы, используемые при сериализации, и показать, что можно сделать с их помощью.

Для тестов сделаем специальный компонент с одним-единственным свойством:

```
public class TestControl : Control
{
    private string testProperty = "test-value";
    public string TestProperty
    {
        get { return testProperty; }
        set { testProperty = value; }
    }
}
```

Добавим такой компонент на форму и посмотрим код, который генерирует провайдер сериализации (несущественную часть кода я удалил):

```
this.testControl1 = new MyControl.TestControl();
//
// testControl1
//
this.testControl1.Location = new System.Drawing.Point(154, 112);
this.testControl1.Name = "testControl1";
```

```
this.testControl1.Size = new System.Drawing.Size(75, 23);  
this.testControl1.TabIndex = 0;  
this.testControl1.TestProperty = "test-value";  
this.testControl1.Text = "testControl1";
```

Код, относящийся к нашему компоненту, состоит из трех блоков: конструктора, комментария и заполнения свойств. Каким образом можно повлиять на этот код?

Для управления кодом сериализации нужно добавить собственный сериализатор. Сделать это можно с помощью атрибута `DesignerSerializer`:

```
[DesignerSerializer(typeof(TestSerializer),  
                    typeof(CodeDomSerializer))]  
public class TestControl : Control
```

Первый параметр задает тип сериализатора, а второй — тип базового класса. Соответственно, описание нашего сериализатора должно иметь вид:

```
public class TestSerializer : CodeDomSerializer  
{  
}
```

Класс `CodeDomSerializer` имеет два виртуальных метода, выполняющих базовую функциональность сериализатора. Перекрыв их, мы сможем влиять на процессы сериализации и десериализации:

```
public class TestSerializer : CodeDomSerializer  
{  
    public override object Deserialize(  
        IDesignerSerializationManager manager,  
        object codeObject)  
    {  
        return baseClassSerializer.Deserialize(manager, codeObject);  
    }  
  
    public override object Serialize(  
        IDesignerSerializationManager manager,  
        object value)  
    {  
        object codeObject = baseClassSerializer.Serialize(manager, value);  
        return codeObject;  
    }  
}
```

С помощью интерфейса менеджера сериализации `IDesignerSerializationManager` можно получить объект самого сериализатора:

```
CodeDomSerializer baseClassSerializer =
    (CodeDomSerializer)manager.GetSerializer(
        typeof(TestControl).BaseType, typeof(CodeDomSerializer));
```

С помощью этого объекта можно управлять содержимым кода, генерируемого сериализатором. Но перед этим я должен рассказать, с помощью каких классов описывается код.

Базовым объектом описания кода (звучит занятно — код классов описывается с помощью классов, но это действительно так) является класс `System.CodeDom.CodeObject`. От него наследуются классы, описывающие элементы кода, например:

- `CodeComment` — комментарий;
- `CodeDirective` — директива кода;
- `CodeExpression` — выражение;
- `CodeNamespace` — пространство имен;
- `CodeStatement` — абстрактный класс для всех кодовых конструкций.

Соответствующие наборы элементов описываются коллекциями:

- `CodeCommentStatementCollection` — коллекция комментариев;
- `CodeDirectiveCollection` — коллекция директив кода;
- `CodeExpressionCollection` — коллекция выражений;
- `CodeStatementCollection` — коллекция кодовых конструкций.

С помощью этих классов можно вмешиваться в код, генерируемый сериализатором, например, можно добавить строку комментария в начало кода:

```
string commentText = "Комментарий, добавленный из сериализатора";
CodeCommentStatement comment =
    new CodeCommentStatement(commentText);
statements.Insert(0, comment);
```

Полный код сериализатора представлен в листинге 10.8. В результате работы этого сериализатора будет сгенерирован код, содержащий дополнительную строку комментария:

```
// Комментарий, добавленный из сериализатора
//
// testControll
//
this.testControll.Location = new System.Drawing.Point(109, 74);
this.testControll.Name = "testControll";
this.testControll.Size = new System.Drawing.Size(75, 23);
```

```
this.testControl1.TabIndex = 0;  
this.testControl1.TestProperty = "test-value";  
this.testControl1.Text = "testControl1";
```

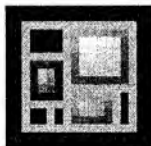
Аналогично, можно вмешиваться и в другие конструкции автоматически генерируемого кода.

Листинг 10.8. Класс, изменяющий код сериализации компонента

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.ComponentModel.Design.Serialization;  
using System.CodeDom;  
  
namespace MyControl  
{  
    public class TestSerializer : CodeDomSerializer  
    {  
        public override object Deserialize(  
            IDesignerSerializationManager manager,  
            object codeObject)  
        {  
            // Получаем сериализатор  
            CodeDomSerializer baseClassSerializer =  
                (CodeDomSerializer) manager.GetSerializer(  
                    typeof(TestControl).BaseType,  
                    typeof(CodeDomSerializer));  
  
            // Пока просто вызываем базовый десериализатор  
            return baseClassSerializer.Deserialize(manager, codeObject);  
        }  
  
        public override object Serialize(  
            IDesignerSerializationManager manager,  
            object value)  
        {  
            // Получаем сериализатор  
            CodeDomSerializer baseClassSerializer =  
                (CodeDomSerializer) manager.GetSerializer(  
                    typeof(TestControl).BaseType,  
                    typeof(CodeDomSerializer));  
  
            // Получаем сериализуемый объект  
            object codeObject = baseClassSerializer.Serialize(manager, value);
```

```
// В принципе, объект может представляться разными способами.  
// Сейчас мы обрабатываем только случай представления  
// объекта набором строк кода.  
if (codeObject is CodeStatementCollection)  
{  
    // Последовательность строк кода  
    CodeStatementCollection statements =  
        (CodeStatementCollection)codeObject;  
  
    // Добавляем комментарий в самое начало кода  
    string commentText =  
        "Комментарий, добавленный из сериализатора";  
    CodeCommentStatement comment =  
        new CodeCommentStatement(commentText);  
    statements.Insert(0, comment);  
}  
return codeObject;
```

ГЛАВА 11



Рендеринг, расположение и поведение компонентов

Можно ли программу, написанную под Windows, называть подоконником?

Чтобы не путаться в терминах, процесс построения изображения визуального представления компонента я буду называть *рендерингом*¹ (rendering). В этой главе я расскажу про основные принципы рендеринга.

11.1. Рендеринг компонентов

В предыдущих главах мы уже сталкивались с отрисовкой визуальных компонентов, а именно с методами `OnPaint()`, `OnPaintBackground()`. Также мы рассматривали метод `OnPaintAdornments()` класса `ControlDesigner`, позволяющий проводить отрисовку компонента только в режиме дизайнера. Теперь пришла пора поговорить об этом более подробно.

Базовый класс визуальных компонентов `Control` имеет событие `Paint`, которое вызывается при необходимости обновить вид компонента. Виртуальный метод `OnPaint()` позволяет переопределить отрисовку компонента:

```
protected virtual void OnPaint(PaintEventArgs pe);
```

Единственный аргумент этого метода передает всю необходимую информацию, требуемую для отрисовки компонента. Класс `PaintEventArgs` имеет всего два свойства:

¹ Такой термин существует в теории построения графических сцен. При описании построения компонентов в англоязычной литературе используется то же самое слово `rendering`, поэтому я не вижу причин, почему не воспользоваться удобным русским переводом, тем более что он давно прижился в устной речи.


```
public class PaintEventArgs : EventArgs, IDisposable
{
    public Rectangle ClipRectangle { get; }
    public Graphics Graphics { get; }
}
```

Свойство `ClipRectangle` возвращает прямоугольную область, в рамках которой компонент может рисовать. Свойство `Graphics` предоставляет доступ к графическим методам, например, рисования линии, текста или круга.

При реализации своего обработчика `OnPaint` можно вызывать базовый обработчик `base.OnPaint`, если нужно, чтобы компонент принимал сообщения о необходимости перерисовки. Или, базовый метод можно не вызывать, если компонент постоянно перерисовывается. В последнем случае игнорирование базового метода позволит избежать мерцания.

Как я уже говорил, при необходимости вызывать обновление вида компонента, не следует вызывать метод `paint()` напрямую. Более правильный путь — вызвать метод `Invalidate()`, который позволит Windows обновить компонент, когда у операционной системы дойдет до него очередь. Кроме того, метод `Invalidate()` позволяет указать конкретную область компонента, которую нужно обновить, что может значительно повысить скорость отрисовки (например, если компонент рисует цифры на фоне, то обновлять следует только тот участок компонента, на котором рисуются цифры).

Еще один метод, относящийся к рендерингу, — это метод `OnPaintBackground()`:

```
protected virtual void OnPaintBackground(PaintEventArgs pevent);
```

Этот метод позволяет переопределить отрисовку фона компонента. Метод можно использовать, например, для рисования градиентной заливки.

Замечу, что некоторые компоненты, такие как `TextBox`, рисуются напрямую операционной системой, и метод `Paint()` (и, соответственно обработчик `OnPaint`) никогда не вызовутся.

11.2. Стандартный рендеринг

Наверное, можно придумать ситуацию, когда компонент должен выглядеть как стандартный (например, кнопка), но нет возможности наследовать его от соответствующего базового класса. Рисовать вид компонента вручную слишком сложно.

Класс `System.Windows.Forms.ControlPaint` предоставляет набор методов для рисования стандартных компонентов Windows и их элементов, например:

- метод `void DrawBorder(Graphics graphics, Rectangle bounds, Color color, ButtonBorderStyle style)` рисует рамку компонента;

- метод `void DrawButton(Graphics graphics, Rectangle rectangle, ButtonState state)` рисует кнопку;
- метод `void DrawCheckBox(Graphics graphics, Rectangle rectangle, ButtonState state)` рисует переключатель;
- метод `void DrawComboButton(Graphics graphics, Rectangle rectangle, ButtonState state)` рисует выпадающий список.

Это только несколько примеров. В классе `ControlPaint` множество методов, и приводить их все в рамках этой книги не имеет смысла.

Кроме того, библиотека .NET 2.0 предоставляет несколько классов, позволяющих рисовать стандартные компоненты с учетом стилей Windows:

- класс `ButtonRenderer` рисует кнопку;
- класс `CheckBoxRenderer` рисует переключатель;
- класс `GroupBoxRenderer` рисует групповую панель;
- класс `RadioButtonRenderer` рисует переключатель выбора.

Проверить, доступна ли отрисовка с использованием стилей, можно с помощью свойства `Application.RenderWithVisualStyles`, которое возвращает `true`, если стили доступны. Включить использование стилей можно с помощью метода `EnableVisualStyles()`¹:

```
[STAThread] static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new Form1());
}
```

Получить информацию о стилях операционной системы можно с помощью класса `System.Windows.Forms.VisualStyles`.

Еще несколько классов позволяют рисовать компоненты, только если стили доступны, иначе вызов методов этих классов вызовет исключение:

- `ComboBoxRenderer` — рисует выпадающий список;
- `ProgressBarRenderer` — рисует индикатор выполнения;
- `ScrollBarRenderer` — рисует ползунок;
- `TabRenderer` — рисует закладку;

¹При вызове этого метода иногда возникают проблемы с компонентом `ImageList`. Устранить их помогает вызов метода `Application.DoEvents()` сразу после включения поддержки стилей.

- ❑ `TextBoxRenderer` — рисует элемент ввода текста;
- ❑ `TrackBarRenderer` — рисует индикатор для выбора значения.

Эти классы имеют метод `IsSupported()`, позволяющий проверить возможность их использования:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);

    // Стили поддерживаются?
    if (!ComboBoxRenderer.IsSupported)
    {
        // Рисуем без стилей
        ControlPaint.DrawComboButton(e.Graphics,
            this.ClientRectangle, ButtonState.Normal);
    }
    else
    {
        // Рисуем со стилями
        ComboBoxRenderer.DrawDropDownButton(e.Graphics,
            this.ClientRectangle, ComboBoxState.Normal);
    }
}
```

Кроме методов, рисующих сразу элемент управления, в .NET Framework есть множество методов, рисующих графические примитивы. Это, например, `FillRectangle()`, `DrawString()`, `DrawLine()`, `DrawImage()` и множество других. Описывать их все я не буду. Полное и подробное описание этих методов можно найти почти в любой книге, посвященной приложениям **Windows Forms**.

11.3. Управление отрисовкой и поведением компонентов

Кроме собственно самой отрисовки, библиотека .NET предоставляет дополнительный метод `SetStyle()`¹ управления отрисовкой и поведением компонента:

```
protected void SetStyle(ControlStyles flag, bool value)
```

¹ Не путайте стили, которые я описывал в предыдущем разделе, и стили поведения.

Класс `ControlStyles` представляет собой флаги, каждый из которых соответствует некоторой характеристике компонента. Параметр `value` либо устанавливает соответствующий флаг (если он равен `true`), либо сбрасывает его.

Вот список наиболее полезных флагов.

- ☐ `AllPaintingInWmPaint`. Если этот флаг выставлен, то компонент игнорирует сообщение `WM_ERASEBKGD`. Метод `OnPaintBackground` в этом случае вызывается прямо из обработчика сообщения `WM_PAINT`. Этот флаг может использоваться для уменьшения мерцания. Флаг можно выставять только, если установлен флаг `UserPaint`.
- ☐ `ContainerControl`. Если этот флаг выставлен, то компонент ведет себя как контейнер для других компонентов.
- ☐ `DoubleBuffer`. При установке этого флага отрисовка производится сначала в специальный буфер, а затем, после полного завершения отрисовки, результат копируется на экран. Такое поведение уменьшает мерцание во время перерисовки компонентов. Если этот флаг установлен, то нужно также установить флаги `UserPaint` и `AllPaintingInWmPaint`.
- ☐ `FixedHeight`. Если этот флаг установлен, компонент блокирует все попытки изменить высоту во время автоматического выравнивания. Например, если при изменении размера шрифта система попытается изменить высоту компонента, то она не изменится.
- ☐ `FixedWidth`. Блокировка изменения ширины.
- ☐ `Opaque`. Если этот флаг выставлен, то компонент рисуется непрозрачным и без фона.
- ☐ `OptimizedDoubleBuffer`. Если этот флаг выставлен, то отрисовка сначала производится в специальный буфер и затем на экран. Это помогает предотвратить мерцание. Если этот флаг выставлен, то необходимо также выставить флаг `AllPaintingInWmPaint`.
- ☐ `ResizeRedraw`. Если этот флаг выставлен, то компонент перерисовывается во время изменения размера.
- ☐ `Selectable`. Если этот флаг выставлен, то компонент может получать фокус.
- ☐ `StandardClick`. Если этот флаг выставлен, то компонент реализует стандартное поведение по щелчку.
- ☐ `StandardDoubleClick`. Если флаг выставлен, то компонент реализует стандартное поведение по двойному щелчку. Этот флаг игнорируется, если не выставлен флаг `StandardClick`.
- ☐ `SupportsTransparentBackColor`. Если этот флаг выставлен, то компонент может иметь значение поля `BackColor` с составляющей `alpha` меньшей, чем

255, т. е. прозрачный цвет. При этом должен быть установлен флаг `UserPaint`, а сам компонент должен быть наследником класса `Control`.

- ☐ `UserMouse`. Если этот флаг выставлен, то события мыши не обрабатываются операционной системой. Считается, что всю необходимую обработку производит сам компонент.
- ☐ `UserPaint`. Если этот флаг выставлен, то компонент рисует себя до того, как это сделает операционная система. Если флаг не выставлен, то метод `Paint` не вызывается.

Установка нужного стиля производится либо из конструктора компонента, либо из нужного метода. В последнем случае может потребоваться вызов метода `UpdateStyles()` для немедленного применения нового значения флага. При изменении стиля вызывается событие `StyleChanged`. Метод `GetStyle()` позволяет получить текущее состояние флагов.

11.4. Компонент, невидимый в режиме выполнения

При установке свойства `Visible` в значение `false` компонент перестает быть видим в режиме выполнения:

```
textBox1.Visible = false;
```

Но при этом он также перестает принимать сообщения мыши и пользовательский ввод. Если компонент должен быть невидим, но продолжать функционировать, его следует сделать прозрачным:

```
SetStyle(ControlStyles.SupportsTransparentBackColor, true);  
this.BackColor = Color.Transparent;
```

Правда, это, скорее, "эмуляция" прозрачности, чем реальная реализация, т. к. прозрачность достигается вызовом метода `OnPaintBackground()` компонента-владельца, что срабатывает далеко не всегда.

11.5. Управление расположением элементов

Каждый визуальный элемент управления, на котором расположены другие компоненты, может самостоятельно управлять их расположением, как это делает, например, компонент `FlowLayoutPanel`. В режиме выполнения можно обойтись обработкой события `Resize`, но в режиме разработки такой способ не подходит.

Свойство `LayoutEngine` класса `Control` возвращает объект типа `LayoutEngine`, позволяющий управлять расположением элементов, как в режиме разработки, так и в режиме выполнения. Посмотрим, как это выглядит на практике.

Сам класс `LayoutEngine` очень простой и имеет всего два виртуальных метода:

```
public abstract class LayoutEngine
{
    protected LayoutEngine();
    public virtual void InitLayout(
        object child, BoundsSpecified specified);
    public virtual bool Layout(
        object container, LayoutEventArgs layoutEventArgs);
}
```

Попробуем реализовать собственный механизм управления, который будет располагать элементы управления построчно¹. Все, что для этого нужно, — это перекрыть свойство `LayoutEngine`, которое должно возвращать экземпляр нового класса управления `AutoFlowLayout` (листинг 11.1). Реализацию самого класса я привел в листинге 11.2. Основные действия управления расположением элементов происходят в методе `Layout()`. Подробно обсуждать алгоритм распределения элементов, думаю, не стоит. Главный принцип: элементы располагаются в одну "строку" до тех пор, пока они помещаются по ширине. Если больше элементов в одну строку не помещается, то панель начинает распределение на следующую "строку" (рис. 11.1).

Замечу, что этот механизм перестраивает элементы и при добавлении, и при удалении элементов на панели, а также при изменении размеров панели как в режиме разработки, так и в режиме выполнения.

Листинг 11.1. Класс панели с автоматическим управлением расположением элементов

```
public class AutoFlowPanel : Panel
{
    private AutoFlowLayout layoutEngine;

    public AutoFlowPanel()
    {
    }
}
```

¹ Я знаю, что такой элемент управления уже существует, но мне нужно показать пример создания собственного механизма управления расположением элементов.

```
// Механизм управления расположением элементов
public override LayoutEngine LayoutEngine
{
    get
    {
        if (layoutEngine == null)
        {
            layoutEngine = new AutoFlowLayout();
        }

        return layoutEngine;
    }
}
}
```

Листинг 11.2. Класс автоматического расположения элементов

```
public class AutoFlowLayout : LayoutEngine
{
    public override bool Layout(object container,
                               LayoutEventArgs layoutEventArgs)
    {
        // Панель, на которой располагаются элементы
        Control parent = container as Control;

        // Область для расположения элементов
        Rectangle parentDisplayRectangle = parent.DisplayRectangle;

        // Пробуем расположить элементы
        // на панели так, чтобы они все были видимы
        Point nextControlLocation = parentDisplayRectangle.Location;
        nextControlLocation.Offset(parent.Margin.Left,
                                parent.Margin.Top);

        foreach (Control c in parent.Controls)
        {
            // Невидимые элементы пропускаем
            if (!c.Visible)
            {
                continue;
            }

            // Элемент не будет виден в этой "строке".
            // Переходим к следующей.
        }
    }
}
```

```
if ((nextControlLocation.X + c.Width) > parent.Width)
{
    nextControlLocation.X =
        parentDisplayRectangle.X + parent.Margin.Right;
    nextControlLocation.Y +=
        c.Height + parent.Margin.Bottom;
}

// Располагаем компонент
c.Location = nextControlLocation;

// Передвигаемся к следующей позиции в "строке"
nextControlLocation.X += c.Width + parent.Margin.Right;
}
return false;
}
```

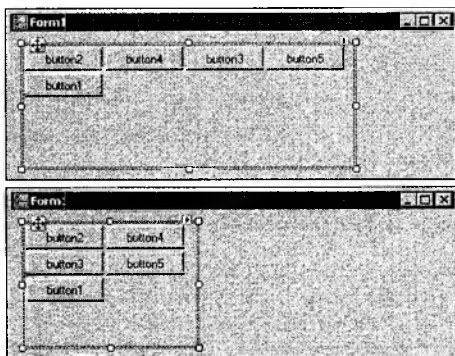
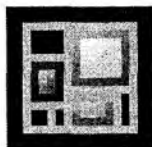


Рис. 11.1. Расположение элементов в зависимости от размера панели

ГЛАВА 12



Копаем глубже

Чем сложнее задача, тем проще ее не решать.

Эта глава содержит информацию о многих полезных классах, доступных в режиме разработки. С их помощью можно добиться удивительной гибкости и удобства разработки.

12.1. Компоненты-расширители и глобальные свойства

Компонент `Tooltip` интересен тем, что, не неся самостоятельной функциональности, он добавляет дополнительное свойство "строка подсказки" ко всем визуальным компонентам. Аналогично, компонент-панель `TableLayoutPanel` добавляет всем компонентам, расположенным на этой панели свойства `Cell`, `Column`, `Row`, `ColSpan` и `RowSpan` (рис. 12.1). В этом разделе я хочу рассказать, каким образом создаются такие свойства.

Компоненты, добавляющие свойства ко всем компонентам одновременно, называются *компонентами-расширителями* (`component extender`), а сами свойства я буду для простоты называть *глобальными свойствами*. Добавить их позволяет специальный интерфейс `IExtenderProvider`. При добавлении на форму компонента, реализующего этот интерфейс, Visual Studio автоматически добавляет глобальные свойства ко всем компонентам (вообще говоря, не ко всем подряд, но об этом позже).

Компонент, описывающий глобальные свойства, должен подчиняться трем правилам: реализовывать интерфейс `IExtenderProvider`, иметь один или несколько атрибутов `ProvideProperty` и, конечно, определять сами свойства.

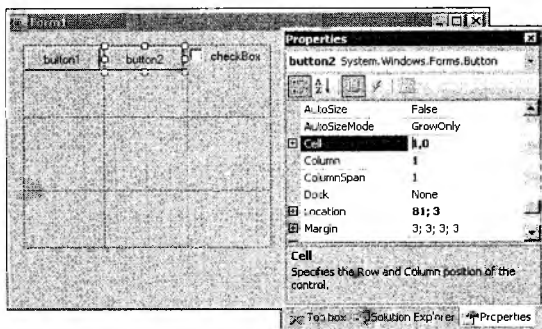


Рис. 12.1. Свойства Cell, Column, ColumnSpan, Row и RowSpan добавляются ко всем компонентам, расположенным на TableLayoutPanel

Атрибут `ProvideProperty` определяет имя глобального свойства. Он имеет два конструктора:

```
public ProvidePropertyAttribute(
    string propertyName, string receiverTypeName);
public ProvidePropertyAttribute(
    string propertyName, Type receiverType);
```

Первый параметр определяет имя свойства, а второй — тип компонентов, к которым должно быть добавлено это свойство (может быть указан либо тип, либо его имя). Один компонент может определять один или несколько глобальных свойств. Атрибут определяет только имя свойства, а само свойство должно быть описано с помощью методов `Get<имя_свойства>` и `Set<имя_свойства>`. Эти методы должны иметь специальное имя и формат. Например, если атрибут задает свойство `Title` следующим образом:

```
[ProvideProperty("Title", typeof(Control))]
```

В этом случае должны быть описаны два метода:

```
public string GetTitle(Control parent)
{
    return .....;
}
public void SetTitle(Control parent, string value)
{
    ... ..
}
```

Параметр `parent` передает компонент, к которому применяется свойство.

К методу `Get` можно добавлять дополнительные атрибуты, как для обычного свойства:

```
[Description("Дополнительный заголовок")]
[Category("Глобальные свойства")]
[DefaultValue("")]
[DisplayName("GlobalTitle")]
public string GetTitle(Control parent)
{
    return title;
}
```

По умолчанию имя глобального свойства имеет вид:

`<имя_свойства> on <имя_компонента>`

Например, `Tooltip on ToolTip1`. Но с помощью атрибута `DisplayName` можно задать любое другое имя.

Интерфейс `IExtenderProvider` имеет всего один метод:

```
bool CanExtend(Object extendee)
```

Метод должен возвращать `true`, если дополнительные свойства применимы для компонента, заданного параметром `extendee`. Таким образом, этот метод является фильтром, определяющим, к каким типам компонентов применимы данные глобальные свойства. Например, определить добавление свойства только к кнопкам можно так:

```
public bool CanExtend(object extendee)
{
    return extendee is Button;
}
```

Компонент, реализующий добавление глобального свойства `ExtenderIndex`, показан в листинге 12.1. Те же действия можно выполнить с помощью сервиса `IExtenderProviderService`, который позволяет добавить глобальное свойство в список "расширяющих" модулей (листинг 12.2). В принципе, кроме использования сервиса, этот код ничем не отличается от того, что я описал выше. Но, т. к. здесь используется дизайнер, то глобальное свойство будет доступно только в режиме разработки, что иногда тоже бывает полезно.

Листинг 12.1. Добавление глобального свойства с помощью сервиса `IExtenderProviderService` (вариант 1)

```
using System;
using System.Collections;
```

```
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;

namespace ExtenderServiceExample
{
    // Реализация IExtenderProvider
    [ProvidePropertyAttribute("ExtenderIndex", typeof(IComponent))]
    public class ComponentExtender : Component, IExtenderProvider
    {
        // Значение свойства
        public int index = 0;

        public ComponentExtender()
        {
        }

        public bool CanExtend(object extendee)
        {
            // разрешаем добавлять свойство к любым компонентам
            return true;
        }

        // Метод Get
        [DisplayName("ExtenderIndex")]
        public int GetExtenderIndex(IComponent component)
        {
            return index;
        }

        // Метод Set
        public void SetExtenderIndex(IComponent component, int index)
        {
            this.index = index;
        }
    }
}
```

Листинг 12.2 Добавление глобального свойства с помощью сервиса **IExtenderProviderService** (вариант 2)

```
using System;
using System.Collections;
```

```
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;

namespace ExtenderServiceExample
{
    // Компонент, связанный с дизайнером ExtenderServiceDesigner
    [DesignerAttribute(typeof(ExtenderServiceDesigner))]
    public class ExtenderServiceTestControl : Component
    {
        public ExtenderServiceTestControl()
        {
        }
    }

    // Дизайнер, добавляющий глобальное свойство ExtenderIndex,
    // описанное в классе ComponentExtender,
    // с помощью сервиса IExtenderProviderService.
    public class ExtenderServiceDesigner : ComponentDesigner
    {
        // Ссылка на сервис
        IExtenderProviderService extenderServiceReference;
        // Ссылка на ComponentExtender
        private ComponentExtender extender;

        public ExtenderServiceDesigner()
        {
        }

        // Инициализация
        public override void Initialize(IComponent component)
        {
            base.Initialize(component);

            // Получаем интерфейс IExtenderProviderService
            IExtenderProviderService extenderService =
                (IExtenderProviderService)
                component.Site.GetService(
                    typeof(IExtenderProviderService));
            if (extenderService != null)
            {
                // Создаем ComponentExtender и добавляем
                // его в список "расширителей"
            }
        }
    }
}
```

```
        extender = new ComponentExtender();
        extenderService.AddExtenderProvider(extender);
        extenderServiceReference = extenderService;
    }
    else
        MessageBox.Show
            ("Не удалось получить IExtenderProviderService.");
}

protected override void Dispose(bool disposing)
{
    // Удаляем глобальное свойство
    if (extenderServiceReference != null)
    {
        extenderServiceReference.RemoveExtenderProvider(extender);
        extenderServiceReference = null;
    }
}

// Реализация IExtenderProvider
[ProvidePropertyAttribute("ExtenderIndex", typeof(IComponent))]
public class ComponentExtender : IExtenderProvider
{
    // Значение свойства
    public int index = 0;

    public ComponentExtender()
    {
    }

    public bool CanExtend(object extendee)
    {
        // Разрешаем добавлять свойство к любым компонентам
        return true;
    }

    // Метод Get
    public int GetExtenderIndex(IComponent component)
    {
        return index;
    }
}
```

```
// Метод Set
public void SetExtenderIndex(IComponent component, int index)
{
    this.index = index;
}
}
```

Итак, посмотрим, что у нас получилось. Глобальное свойство действительно появилось в списке свойств всех компонентов, но есть одна большая "недоработка" — наше свойство имеет единственное значение для всех компонентов. Значит, нужно каким-то образом хранить таблицу значений нашего свойства для каждого компонента, к которому добавляется это свойство. Такую таблицу мы реализуем с помощью класса `Hashtable`. В методе `Set` мы будем сохранять в нее значения, а в методе `Get` — извлекать. Новая реализация класса `ComponentExtender` показана в листинге 12.3. Теперь, каждый компонент, к которому добавлено глобальное свойство, имеет свое значение этого свойства.

Листинг 12.3. Реализация таблицы свойств для `IExtenderProviderService`

```
// Реализация IExtenderProvider
[ProvidePropertyAttribute("ExtenderIndex", typeof(IComponent))]
public class ComponentExtender : IExtenderProvider
{
    // Таблица значений свойства Index
    private Hashtable properties;

    public ComponentExtender()
    {
        properties = new Hashtable();
    }

    public bool CanExtend(object extendee)
    {
        // Разрешаем добавлять только к кнопкам
        return extendee is Button;
    }

    // Метод Get
    public int GetExtenderIndex(IComponent component)
    {
        // Если компонент уже есть в таблице,
        // возвращаем сохраненное значение
    }
}
```

```
        if (properties.Contains(component))
        {
            return (int)properties[component];
        }
        else
        {
            // Компонента нет — возвращаем значение по умолчанию
            return 0;
        }
    }

    // Метод Set
    public void SetExtenderIndex(IComponent component, int index)
    {
        // Если компонент уже есть в таблице,
        // обновляем сохраненное значение
        if (properties.Contains(component))
        {
            properties[component] = index;
        }
        else
        {
            // Компонента нет — добавляем его в таблицу
            {
                properties[component] = index;
            }
        }
    }
}
```

Но добавление глобального свойства — это только половина дела, ведь компонент не умеет сам использовать это свойство. Дальнейшие действия зависят от того, каким образом предполагается использовать новое свойство. Если, например, нужен обработчик события, такой как `Paint`, то лучший способ внедрения своего обработчика — это использование интерфейса `ISupportInitialize` (см. разд. 4.6). Реализацию метода начала инициализации можно оставить пустой, а в методе завершения инициализации создавать свой обработчик, например, `Paint`:

```
public void EndInit()
{
    foreach (DictionaryEntry de in properties)
    {
        Button button = de.Key as Button;
```



```
if (button != null)
{
    button.Paint += new PaintEventHandler(ComponentExtender_Paint);
}
}
```

Как видно из кода, мы пробегаем по всей таблице компонентов, к которым добавлено глобальное свойство. Для каждого компонента регистрируется обработчик события `Paint`. Кроме того, при изменении свойства (метод `Set`) нужно перерисовать компонент. Полный код реализации глобального свойства и обработчика `Paint` показан в листинге 12.4.

Листинг 12.4. Использование `ISupportInitialize` для установки обработчика `Paint`

```
using System;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;
using System.Drawing;

namespace ExtenderServiceExample
{
    [ProvidePropertyAttribute("ExtenderIndex", typeof(IComponent))]
    public class ComponentExtender :
        Component, IExtenderProvider, ISupportInitialize
    {
        // Таблица значений свойства Index
        private Hashtable properties;

        public ComponentExtender()
        {
            properties = new Hashtable();
        }

        public bool CanExtend(object extendee)
        {
            // Разрешаем добавлять только к кнопкам
            return extendee is Button;
        }
    }
}
```

```
// Метод Get
public int GetExtenderIndex(IComponent component)
{
    // Если компонент уже есть в таблице,
    // возвращаем сохраненное значение
    if (properties.Contains(component))
    {
        return (int)properties[component];
    }
    else
    {
        // Компонента нет — возвращаем значение по умолчанию
        return 0;
    }
}

void ComponentExtender_Paint(object sender, PaintEventArgs e)
{
    if (properties.Contains(sender))
    {
        int index = (int)properties[sender];

        e.Graphics.DrawString(index.ToString(), Control.DefaultFont,
                               SystemBrushes.ControlText, 5, 5);
    }
}

// Метод Set
public void SetExtenderIndex(IComponent component, int index)
{
    // Если компонент уже есть в таблице,
    // обновляем сохраненное значение
    if (properties.Contains(component))
    {
        properties[component] = index;
    }
    else
    {
        // Компонента нет — добавляем его в таблицу
        {
            // Регистрируем обработчик
            properties[component] = index;
        }
    }
}
```

```
// Перерисовать компонент
(component as Button).Invalidate();
}

public void BeginInit()
{
}

public void EndInit()
{
    // Регистрируем обработчик OnPaint
    // для всех компонентов, имеющих наше глобальное свойство
    foreach (DictionaryEntry de in properties)
    {
        Button button = de.Key as Button;

        if (button != null)
        {
            button.Paint += new PaintEventHandler(
                ComponentExtender_Paint);
        }
    }
}
}
```

Вообще говоря, можно использовать не только внедрение и перехват свойств, но и интерфейсы самого компонента `ComponentExtender`, к которому точно так же, как и к обычному компоненту, можно добавлять контекстное меню, смарт-теги и т. п.

Теперь перейдем к следующему вопросу. Посмотрите на код, из листинга 12.3:

```
public void SetExtenderIndex(IComponent component, int index)
{
    properties[component] = index;
}
```

При задании глобального свойства компонент и значение свойства сохраняются в таблице `properties`. Что произойдет, если один из компонентов будет удален с формы? Компонент исчезнет, а ссылка на него в нашей таблице останется. Исправить эту проблему можно с помощью событий сервиса `IComponentChangeService` (см. разд. 7.4). Получать интерфейс этого сервиса лучше всего в перекрытом свойстве `Site`:

```
public override ISite Site
{
    get
    {
        return base.Site;
    }

    set
    {
        base.Site = value;
        // Получаем интерфейс сервиса
        componentChangeService = GetService(
            typeof(IComponentChangeService)) as IComponentChangeService;
        // Подписываемся на события добавления и удаления компонентов
        if (componentChangeService != null)
        {
            componentChangeService.ComponentAdded +=
                new ComponentEventHandler(
                    componentChangeService_ComponentAdded);
            componentChangeService.ComponentRemoved +=
                new ComponentEventHandler(
                    componentChangeService_ComponentRemoved);
        }
    }
}
```

В обработчике удаления компонента можно удалять ссылку на него из нашей таблицы:

```
void componentChangeService_ComponentRemoved(
    object sender, ComponentEventArgs e)
{
    properties.Remove(e.Component);
}
```

В обработчике добавления компонента на форму не обязательно выполнять какие-либо действия. Его реализация зависит от функциональности компонента-расширителя. Например, в нашем случае не требуется никакой дополнительной функциональности:

```
void componentChangeService_ComponentAdded(
    object sender, ComponentEventArgs e)
{
    // Пока никаких действий не нужно
}
```

В случае, если компонент-расширитель занимался бы, к примеру, управлением расположения элементов, возможно, потребовалась бы инициализация глобального свойства добавляемого на форму компонента.

Получившийся код показан в листинге 12.5.

Листинг 12.5. Обработка добавления и удаления компонентов в компоненте-расширителе

```
using System;
using System.Collections;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;
using System.Drawing;

namespace ExtenderServiceExample
{
    [ProvidePropertyAttribute("ExtenderIndex", typeof(IComponent))]
    public class ComponentExtender :
        Component, IExtenderProvider, ISupportInitialize
    {
        // Таблица значений свойства Index
        private Hashtable properties;
        IComponentChangeService componentChangeService;

        public ComponentExtender()
        {
            properties = new Hashtable();
        }

        public override ISite Site
        {
            get
            {
                return base.Site;
            }
            set
            {
                base.Site = value;

                componentChangeService =
                    GetService(typeof(IComponentChangeService))
                    as IComponentChangeService;
            }
        }
    }
}
```

```

        if (componentChangeService != null)
        {
            componentChangeService.ComponentAdded +=
                new ComponentEventHandler(
                    componentChangeService_ComponentAdded);
            componentChangeService.ComponentRemoved +=
                new ComponentEventHandler(
                    componentChangeService_ComponentRemoved);
        }
    }

    // Обработчик удаления компонента
    void componentChangeService_ComponentRemoved(
        object sender, ComponentEventArgs e)
    {
        properties.Remove(e.Component);
    }

    // Обработчик добавления компонента
    void componentChangeService_ComponentAdded(
        object sender, ComponentEventArgs e)
    {
        // Пока ничего не делаем
    }
    ... ..
}

```

Последний вопрос, который я хотел бы обсудить в этом разделе, — это сериализация глобальных свойств. Действительно, ведь реально эти свойства "не существуют", и, соответственно, сам компонент сохранить их не может. Зато за него это делает создатель свойства. Код сериализации выглядит так:

```
this.componentExtender1.SetExtenderIndex(компонент, значение);
```

Например, для нашего компонента, добавляющего глобальное свойство целого типа, этот код будет таким:

```
this.componentExtender1.SetExtenderIndex(this.button2, 1);
```

В качестве еще одного примера в *разд. 12.5* я покажу, как реализовать свой компонент всплывающей подсказки `BalloonToolTip`, добавляющей свойство `BalloonText` ко всем визуальным компонентам (см. листинг 12.12).

12.2. Транзакции дизайнера

В разд. 6.5 я приводил код компонента градиентной заливки `GradientLabel`. С помощью контекстного меню цвета начала и завершения заливки менялись местами. Код этого метода выглядел следующим образом:

```
PropertyDescriptor start = GetPropertyByName("StartColor");
PropertyDescriptor end = GetPropertyByName("EndColor");
Color tmp = (Color)start.GetValue(glabel);
start.SetValue(glabel, end.GetValue(glabel));
end.SetValue(glabel, tmp);
```

Как видно из этого фрагмента, перестановка цветов состоит из трех шагов: цвет начала сохраняется в промежуточную переменную `tmp`, цвет начала заливки устанавливается равным цвету конца заливки и, наконец, цвет конца заливки устанавливается равным цвету начала заливки. С точки зрения компонента установка свойств состоит из двух частей: установка цвета начала заливки и затем установка цвета завершения. Выполнение этого кода в Visual Studio показывает, что среда разработки видит этот процесс точно так же. Действительно, меню **Undo** будет содержать две операции: отменив первую из них, мы получим компонент с одинаковыми цветами начала и завершения заливки, и только отменив второй, мы вернем компонент в предыдущее состояние. Но ведь с точки зрения пользователя это была одна операция! Более того, приведенный выше код достаточно прост, а ведь последовательность операций может быть значительно длиннее. Проблема с откатом изменений не единственная. Если на одном из шагов произойдет ошибка, мы получим совершенно непредсказуемое и некорректное состояние компонента.

Транзакции (transaction) позволяют объединить несколько операций в один набор. Транзакция может быть либо выполнена целиком успешно, либо не выполнена вообще. Кроме того, редактор видит транзакцию как единую операцию и позволяет откатить только всю транзакцию полностью, что сохраняет компонент в непротиворечивом состоянии.

Транзакция создается с помощью метода `CreateTransaction()` класса `DesignerHost`:

```
IDesignerHost host = (IDesignerHost) GetService(typeof(IDesignerHost));
DesignerTransaction trans = host.CreateTransaction();
```

Использование транзакции выглядит следующим образом:

```
using (trans)
{
    try
```

```
{
    // Последовательность действий

    // Подтверждаем транзакцию
    trans.Commit();
}
catch
{
    // Ошибка — откатываем транзакцию
    trans.Cancel();
}
}
```

Как вариант, можно создать именованную транзакцию, используя второй вариант конструктора:

```
DesignerTransaction trans = host.CreateTransaction(имя_транзакции);
```

Новый код метода `OnInvertColors()` показан в листинге 12.6. Теперь редактор увидит только одну операцию и, соответственно, откат изменений будет работать корректно.

Листинг 12.6. Использование транзакции

```
private void OnInvertColors(object sender, System.EventArgs e)
{
    IDesignerHost host =
        (IDesignerHost)glabel.Site.GetService(typeof(IDesignerHost));

    if (host == null)
        return;

    DesignerTransaction trans =
        host.CreateTransaction("Invert colors " + glabel.Name);

   PropertyDescriptor start = GetPropertyByName("StartColor");
    PropertyDescriptor end = GetPropertyByName("EndColor");

    // Меняем цвета местами
    using (trans)
    {
        try
        {
            Color tmp = (Color)start.GetValue(glabel);
            start.SetValue(glabel, end.GetValue(glabel));
            end.SetValue(glabel, tmp);
        }
    }
}
```



```

        trans.Commit();
    }
    catch
    {
        trans.Cancel();
    }
}
}

```

12.3. Добавление закладки в редактор свойств

Стандартный редактор свойств имеет две кнопки, переключающие его закладки. При нажатии на одну из них открывается список свойств компонента, а при нажатии на другую — список обработчиков событий. В этом разделе я хочу рассказать, как добавить свою кнопку и закладку в редактор свойств.

Сама закладка редактора свойств описывается с помощью класса `System.Windows.Forms.Design.PropertyTab`. Для реализации своей закладки нужно перекрыть несколько свойств и методов (листинг 12.7):

- ❑ метод `bool CanExtend(object extendee)` должен возвращать `true`, в случае, когда закладку нужно показывать. Параметр `extendee` представляет выбранный компонент, поэтому чаще всего реализация этого метода сводится к проверке типа этого параметра;
- ❑ метод `PropertyDescriptorCollection GetProperties(object component, Attribute[] attributes)` возвращает коллекцию дескрипторов свойств, которые будут показаны на закладке. В нашем случае я собираю все свойства, имеющие добавленный мной атрибут `MyTabDisplayAttribute`;
- ❑ свойство `string TabName` возвращает имя закладки;
- ❑ свойство `Bitmap Bitmap` возвращает картинку закладки.

О последнем свойстве стоит рассказать чуть подробнее. Картинку закладки можно загрузить из файла, как обычный файл изображения. Но такой способ не очень хорош, т. к. файл придется прикладывать к компоненту отдельно. Более удачный вариант — хранить картинку как ресурс. Для этого ее нужно подключить к проекту и выставить свойство `Build Action` в значение `Embedded Resource`. Теперь файл будет прикреплен к компоненту. Загрузить его можно так:

```
new Bitmap(typeof(MyControl), "MyPropertyTab.bmp")
```

Кроме того, класс `PropertyTab`, по умолчанию, ищет в ресурсах файл, совпадающий с именем типа, плюс расширение, поэтому можно добавить в проект

такой файл, а само свойство вообще не перегружать. В нашем примере я на всякий случай преобразую размер картинки к нужному (а именно 16×16):

```
new Bitmap(base.Bitmap, new Size(16, 16))
```

В коде самого компонента (листинг 12.8) свойствам, которые иужо вынести на отдельную закладку, следует задать атрибут `Browsable(false)`, чтобы свойства не были видны на стандартных закладках, и атрибут `MyTabDisplay(true)`, чтобы свойство появилось на новой закладке.

Вид новой закладки показан на рис. 12.2.

Листинг 12.7. Класс закладки для редактора свойств

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms.Design;
using System.ComponentModel;
using System.Drawing;

namespace MyControl
{
    public class MyPropertyTab : PropertyTab
    {
        // Показывать закладку, если это TestComponent
        public override bool CanExtend(object extendee)
        {
            return extendee is TestComponent;
        }

        // На новой закладке показываем свойства,
        // отмеченные атрибутом MyTabDisplayAttribute
        public override PropertyDescriptorCollection GetProperties(
            object component, Attribute[] attributes)
        {
            return TypeDescriptor.GetProperties(
                component, new Attribute[] {
                    new BrowsableAttribute(false),
                    new MyTabDisplayAttribute(true) });
        }

        // Имя закладки
        public override string TabName
```

```

    {
        get
        {
            return "New Tab";
        }
    }

    // Картинка. Берется из ресурсов.
    // Имя файла должно совпадать с
    // именем класса, т. е. MyPropertyTab.bmp
    public override Bitmap Bitmap
    {
        get
        {
            return new Bitmap(base.Bitmap, new Size(16, 16));
        }
    }
}

```

Листинг 12.6. Описание свойства, расположенного на новой закладке

```

using System;
using System.ComponentModel;
using System.Windows.Forms;

namespace MyControl
{
    [PropertyTab(typeof(MyPropertyTab), PropertyTabScope.Component)]
    public class TestComponent : UserControl
    {
        private int test;

        [Browsable(false)]
        [MyTabDisplayAttribute(true)]
        public int Test
        {
            get { return test; }
            set { test = value; }
        }
    }
}

```

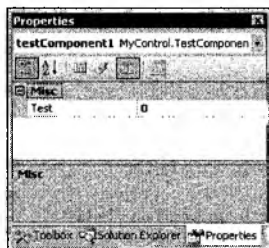


Рис. 12.2. Новая закладка в редакторе свойств

12.4. Обобщения

Чтобы не путаться с понятиями, я буду переводить слово *generic* как "обобщение". По сути обобщения похожи и на шаблоны (*templates*), и на макросы (*macros*), но будет проще, если использовать для них отдельное понятие. Упрощенно говоря, обобщения обеспечивают программисту возможность определения "заполнителей" (параметров типа) для аргументов методов и определений типов, которые будут конкретизированы во время вызова обобщенного метода или при создании обобщенного типа (см. [6]).

Попробуем описать компонент-обобщение:

```
public class TestControl<T> : Control
{
    private T test;

    public T Test
    {
        get { return test; }
        set { test = value; }
    }
}
```

Тип *T* используется и для типа компонента, и для описания свойства. Разумеется, такой компонент не появится в палитре компонентов, зато его наследники, реализованные для конкретного типа, вполне работоспособны:

```
public class Component1 : TestControl<int>
{
}
```

```
public class Component2 : TestControl<string>
{
}
```

Компонент `Component1` имеет свойство типа `int`, а компонент `Component2` — свойство типа `string` (рис. 12.3).

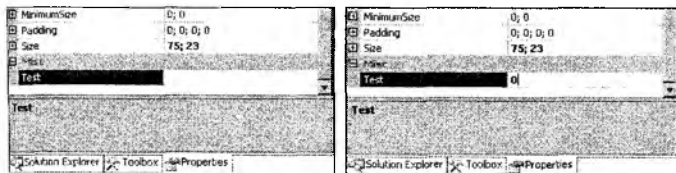


Рис. 12.3. Generic-свойство

Теперь попробуем сделать свойства-обобщения, представляющие собой списки. Один список будет содержать строки, а второй — числа:

```
private IList<string> list1 = new List<string>();
private IList<int> list2 = new List<int>();
```

Для редактирования таких списков проще всего использовать готовые редакторы `ListControlStringCollectionEditor` и `IntegerCollectionEditor`, находящиеся в пространстве имен `System.Windows.Forms.Design`. Но есть одна проблема — оба этих класса имеют уровень доступа `internal`, поэтому получить на них прямую ссылку не получится и нужно обращаться к ним по имени:

```
[DesignerSerializationVisibility(
    DesignerSerializationVisibility.Content)]
[Editor("System.Windows.Forms.Design.
    ListControlStringCollectionEditor, System.Design,
    Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a", typeof(UITypeEditor))]
public IList<string> List1
{
    get { return list1; }
}

[DesignerSerializationVisibility(
    DesignerSerializationVisibility.Content)]
[Editor("System.Windows.Forms.Design.IntegerCollectionEditor,
    System.Design, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a", typeof(UITypeEditor))]
```

```
public IList<int> List2  
{  
    get { return list2; }  
}
```

И для строк, и для чисел редакторы выглядят почти одинаково (рис. 12.4). Не могу сказать, что для числовой коллекции редактор выглядит удобным, но зато это самый быстрый способ при минимуме кода.

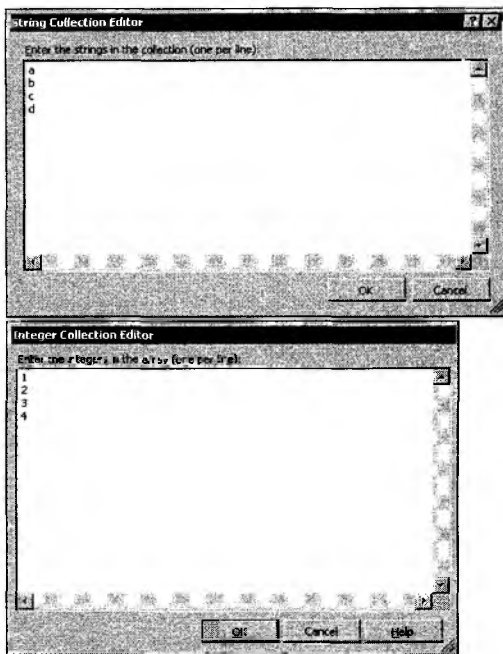


Рис. 12.4. Редактор для коллекции строк и коллекции чисел

Обратите внимание, что у свойства нет аксессуара `set`! Обобщение передается по ссылке, поэтому необходимость создавать список заново отсутствует. Ат-

рибут `DesignerSerializationVisibility.Content` указывает, что сериализация будет производиться по содержимому (см. главу 10). Соответственно, свойства должны быть инициализированы. Код, получившийся при сериализации, выглядит примерно так:

```
this.component21 = new TestControl.Component2();
this.SuspendLayout();
//
// component21
//
this.component21.List1.Add("a");
this.component21.List1.Add("b");
this.component21.List1.Add("c");
this.component21.List1.Add("d");
this.component21.List2.Add(1);
this.component21.List2.Add(2);
this.component21.List2.Add(3);
this.component21.List2.Add(4);
```

Еще один вариант использования обобщений — свойство, представляющее собой список объектов:

```
private IList<Person> list3 = new List<Person>();
public IList<Person> List3
{
    get { return list3; }
}
```

Класс `Person` очень простой (не так важно, какие в нем поля, он нужен только для тестов):

```
public class Person
{
    int id;
    string name;

    public int Id
    {
        get { return id; }
        set { id = value; }
    }

    public string Name
    {
        get { return name; }
    }
}
```

```

        set { name = value; }
    }
}

```

Разумеется, стандартного редактора для этого свойства нет, и для редактирования нужно реализовать собственный редактор (почти то же самое мы делали в разд. 9.3.4):

```

[DesignerSerializationVisibility(
    DesignerSerializationVisibility.Content)]
[Editor(typeof(PersonCollectionEditor), typeof(UITypeEditor))]
public IList<Person> List3
{
    get { return list3; }
}

```

Код редактора `PersonCollectionEditor` приведен в листинге 12.9. Собственно, весь код сводится к переопределению метода `EditValue()`, который вызывает редактор `CollectionEditor` для типа `List<Person>`:

```

return (IList<Person>)
    this.editor.EditValue(context, provider, value);

```

Вид редактора показан на рис. 12.5.

Листинг 12.9. Редактор `PersonCollectionEditor`

```

internal class PersonCollectionEditor : UITypeEditor
{
    private CollectionEditor editor =
        new CollectionEditor(typeof(List<Person>));

    public override object EditValue(ITypeDescriptorContext context,
        IServiceProvider provider, object value)
    {
        if (value != null)
        {
            value = this.editor.EditValue(context, provider, value);
            IList<person> list = (IList<Person>)value;
            return list;
        }

        return base.EditValue(context, provider, value);
    }
}

```



```
public override UITypedEditorEditStyle
    GetEditStyle(ITypeDescriptorContext context)
{
    return this.editor.GetEditStyle(context);
}
```

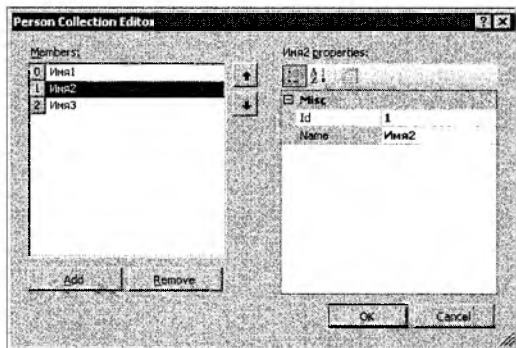


Рис. 12.5. Редактор Person Collection Editor

12.5. Поток

Часто длительные операции, такие как поиск файлов, анализ данных и т. п., выносят в отдельный поток (thread). Если же при этом нужно отображать прогресс выполнения операции, то возникает проблема доступа к компонентам (например, к компоненту `ProgressBar`) из одного или нескольких потоков, с которой мы попробуем разобраться в этом разделе.

В принципе можно просто воспользоваться компонентом `BackgroundWorker`, который позволяет запускать длительные операции в отдельном потоке и получать статус выполнения этих операций. В листинге 12.10 представлен небольшой пример использования этого компонента. Напрямую к разработке компонентов эта тема не относится, поэтому мы не будем долго останавливаться на работе с этим компонентом, но сам пример нам еще пригодится.

Основное правило, о котором следует помнить при разработке потоков, — доступ к компонентам не является потокобезопасным! Посмотрите, что про-

исходит, если в листинге 12.10 я попытаюсь установить значение `progressBar1` прямо из потока вычисления:

```
private int Calculation(BackgroundWorker bw, int arg)
{
    ...
    double progress = result * 100 / I.EN;
    progressBar1.Value = Convert.ToInt32(progress);
    progressBar1.Invalidate(); ;
    ...
}
```

Если запустить получившееся приложение без отладчика, все будет работать так, как положено. А вот в режиме отладки Visual Studio выдаст исключение `InvalidOperationException` (рис. 12.6), сообщающее о том, что была произведена попытка доступа к компоненту из другого потока: "Control 'progressBar1' accessed from a thread other than the thread it was created on".

На самом деле, и в режиме выполнения такой код будет хранить потенциальную ошибку, которая может возникнуть в любой момент. Вызов исключения в режиме отладки можно заблокировать с помощью специального флага:

```
System.Windows.Forms.Control.CheckForIllegalCrossThreadCalls= false;
```

Но только для отладки! Правильный путь — это исправить потенциальную проблему.

Листинг 12.10. Использование компонента `BackgroundWorker`

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace BackgroundWorkerExample
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

```
private void btnStart_Click(object sender, EventArgs e)
{
    // Запускаем операцию с аргументом.
    // Если не надо, то аргумент можно не передавать.
    this.backgroundWorker1.RunWorkerAsync(20000);
}

private void btnCancel_Click(object sender, EventArgs e)
{
    // Прервать операцию
    this.backgroundWorker1.CancelAsync();
}

private void backgroundWorker1_DoWork(
    object sender, DoWorkEventArgs e)
{
    // Здесь нельзя использовать прямую ссылку на backgroundWorker1.
    // Правильно использовать параметр sender.
    BackgroundWorker bw = sender as BackgroundWorker;

    // Получаем аргумент
    int arg = (int)e.Argument;

    // Собственно операция
    e.Result = Calculation(bw, arg);

    // Если операция прервана, то выставляем флаг Cancel
    if (bw.CancellationPending)
    {
        e.Cancel = true;
    }
}

// Вызывается при завершении операции
private void backgroundWorker1_RunWorkerCompleted(
    object sender, RunWorkerCompletedEventArgs e)
{
    // Пользователь нажал кнопку Прервать
    if (e.Cancelled)
    {
        MessageBox.Show("Операция была прервана");
        return;
    }
}
```

```
// Была ошибка во время выполнения операции
if (e.Error != null)
{
    string msgError = String.Format("Ошибка: {0}", e.Error.Message);
    MessageBox.Show(msgError);
    return;
}

// Операция завершилась нормально
string msg = String.Format("Результат = {0}", e.Result);
MessageBox.Show(msg);
}

// Отображение прогресса выполнения
private void backgroundWorker1_ProgressChanged(
    object sender, ProgressChangedEventArgs e)
{
    progressBar1.Value = e.ProgressPercentage;
    progressBar1.Invalidate(); ;
}

const int LEN = 20;

// Длительная операция
private int Calculation(BackgroundWorker bw, int arg)
{
    int result = 0;

    while (!bw.CancellationPending)
    {
        result += 1;
        Thread.Sleep(1000);

        double progress = result * 100 / LEN;
        bw.ReportProgress(Convert.ToInt32(progress));

        if (result >= LEN) // отсчитываем LEN секунд
            break;
    }
    return result;
}
}
```

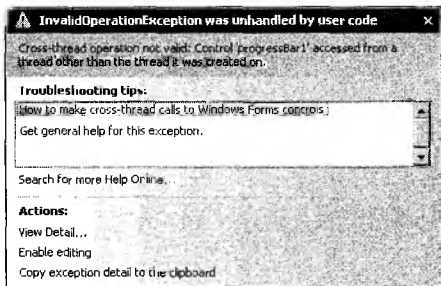


Рис. 12.6. Ошибка при попытке доступа

Для потокобезопасного доступа к компонентам используются методы `InvokeRequired()` и `Invoke()`. Код отображения прогресса выполнения будет выглядеть следующим образом:

```
private int Calculation(BackgroundWorker bw, int arg)
{
    ... ..
    double progress = result * 100 / LEN;
    // Вызываем метод для установки нового значения
    SetProgress(Convert.ToInt32(progress));
    ... ..
}

// Специальный делегат, используемый в методе Invoke
delegate void SetProgressCallback(int progress);

// Метод для установки нового значения progress. Он же
// используется как параметр для делегата
private void SetProgress(int progress)
{
    // Требуется вызов Invoke
    if (progressBar1.InvokeRequired)
    {
        // Создаем делегат и вызываем Invoke
        SetProgressCallback d = new SetProgressCallback(SetProgress);
        this.Invoke(d, new object[] { progress});
    }
}
```

```

else
{
    // Можно устанавливать значение напрямую
    progressBar1.Value = progress;
    progressBar1.Invalidate();
}
}

```

Теперь вызов будет потокобезопасным.

Еще один способ организации выполнения асинхронных операций (операций, выполняемых в других потоках) — использование класса `AsyncOperationManager`. Описание работы этого класса не совсем укладывается в тему книги, поэтому я расскажу о нем достаточно кратко, точнее, просто приведу полезный пример его использования. Пусть, например, на форме имеется компонент `listBox`, использующийся для регистрации действий некоторых методов, выполняющихся в других потоках (журнал). При загрузке формы мы будем создавать шесть потоков: один поток, в котором будет выполняться метод `LongOperation()`, и пять отдельных потоков, представленных классом `SomeOperation`. Каждый поток должен "отчитываться" о ходе работы, добавляя строки в компонент `listBox` основной формы. Добавление строк выполняется с помощью метода `OutMessage()`:

```

private void OutMessage(object item)
{
    listBox1.SelectedIndex = listBox1.Items.Add(item);
}

```

Для решения этой задачи мы создадим специальный класс `CallbackInvoker`, метод `Invoke()` которого будет запускать асинхронную операцию:

```

asyncOperation = AsyncOperationManager.CreateOperation(null);

```

```

public void Invoke(object userData)
{
    asyncOperation.Post(callback, userData);
}

```

Экземпляр этого класса мы будем передавать во все методы, требующие работы с журналом:

```

callbackInvoker = new CallbackInvoker(OutMessage);
SomeOperation client = new SomeOperation(i + 1, callbackInvoker);

```

Сами методы нужны нам только для примера, поэтому они очень просты: все действия сводятся к некоторой задержке и выводу нескольких строк в жур-

нал. Самое главное в них — это то, что при необходимости добавить строку в журнал вызывается метод `Invoke()` нашего класса-обертки, а не прямое добавление строки в компонент `listBox1` основной формы:

```
callbackInvoker.Invoke("SomeOperation: " + id);
```

Полный код описанного механизма приведен в листинге 12.11. Более детальное описание асинхронных операций и многопоточности можно найти в соответствующей литературе.

Листинг 12.11. Использование компонента `AsyncOperationManager`

```
public partial class Form1 : Form
{
    private CallbackInvoker callbackInvoker;

    public Form1()
    {
        InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Создаем класс-обертку для OutMessage,
        // внутри которого будет стартовать асинхронный вызов
        callbackInvoker = new CallbackInvoker(OutMessage);

        // Запустить выполнение метода LongOperation в отдельном потоке
        new MethodInvoker(delegate() {
            this.LongOperation(callbackInvoker);
        }).BeginInvoke(null, null);

        // Создать клиентов, которые выполняют
        // свою работу в отдельных потоках
        for (int i = 0; i < 5; i++)
        {
            // Клиент уведомляет о ходе
            // работ через общий callbackInvoker
            SomeOperation client = new SomeOperation(i + 1, callbackInvoker);
            client.Run();
        }
    }
}
```

```
// Метод, добавляющий строку в listBox1
private void OutMessage(object item)
{
    listBox1.SelectedIndex = listBox1.Items.Add(item);
}

// Выполнение длительной операции
private void LongOperation(CallbackInvoker callbackInvoker)
{
    for (int i = 0; i < 5; i++)
    {
        // Немного подождать
        System.Threading.Thread.Sleep(new Random().Next(300, 600));

        // Вызвать метод OutMessage
        callbackInvoker.Invoke(
            "LongOperation: " + DateTime.Now.ToString());
    }
    callbackInvoker.Invoke("Операция LongOperation завершена");
}

// Специальный класс, сохраняющий callback-метод
// и запускающий асинхронную операцию
public class CallbackInvoker
{
    private SendOrPostCallback callback;
    private AsyncOperation asyncOperation;

    public CallbackInvoker(SendOrPostCallback callback)
    {
        // Ссылка на callback-метод
        this.callback = callback;
        // Асинхронная операция
        asyncOperation = AsyncOperationManager.CreateOperation(null);
    }

    public void Invoke(object userData)
    {
        asyncOperation.Post(callback, userData);
    }
}
```



```
// Класс, выполняющий некую длительную операцию в методе Run
public class SomeOperation
{
    private int id;
    private CallbackInvoker callbackInvoker;

    public SomeOperation(int id, CallbackInvoker callbackInvoker)
    {
        this.id = id;
        this.callbackInvoker = callbackInvoker;
    }

    public void Run()
    {
        // Запустить отдельный поток
        new MethodInvoker(delegate()
        {
            // Немного подождать
            Thread.Sleep(new Random().Next(100, 300));

            // Вызвать метод Form1.OutMessage
            for (int i = 0; i < 3; i++)
            {
                // Сообщение в основной поток
                callbackInvoker.Invoke("SomeOperation: " + id);
                // Немного подождать
                Thread.Sleep(new Random().Next(10, 30));
            }
        }).BeginInvoke(null, null);
    }
}
```

12.6. Взаимодействие с Win32 API

Библиотека .NET Framework позиционируется как мультиплатформенная, поэтому в ней не содержится код, специфичный только для одной платформы. Соответственно, функции, существующие только в Windows, не входят в стандартную библиотеку. Кроме того, функции Win32 API не являются управляемыми и не могут быть использованы в .NET-коде напрямую. Для обращения к таким функциям необходимо подключать или, другими словами, импортировать их.

Импорт функций Win32 API производится с помощью атрибута `DllImport`, например:

```
[DllImport("User32.dll")]  
private static extern int SendMessage(IntPtr hWnd,  
    int Msg, int wParam, IntPtr lParam);
```

Этот атрибут находится в пространстве имен `System.Runtime.InteropServices`. В минимальном варианте достаточно просто имени библиотеки, из которой производится импорт. Но вообще говоря, атрибут `DllImport` имеет еще несколько полезных полей:

- поле `CharSet` (тип `CharSet`) позволяет задавать кодировку символов строк, используемых функцией;
- поле `EntryPoint` (тип `string`) может задавать имя импортируемой функции. По умолчанию имя импортируемой функции совпадает с именем описываемой функции;
- поле `SetLastError` (тип `bool`) указывает, что после вызова функции будет вызываться функция `Win32 GetLastError`. Код ошибки можно получить с помощью метода `Marshal.GetLastWin32Error`.

Углубляться в тонкости импорта функций Win32 я не буду. Достаточно подробное описание можно найти в [3], а на сайтах custom.programming-in.net и pinvoke.net можно найти уже готовые обертки для функций Win32. Здесь я расскажу ровно столько, сколько потребуется нам для реализации примера.

В качестве примера я покажу, как с помощью импортированных функций можно сделать свое окно подсказки, имеющее нестандартную форму. В .NET 2.0 такой компонент входит в список стандартных, но сейчас наша задача состоит не в изучении стандартной библиотеки. Нам просто надо разобраться как использовать возможности Win32 API.

Итак, мы имеем описание импортированных функций Win32. Первая сложность заключается в том, что в .NET нет описания констант, часто необходимых для работы с Win32, и нам придется сделать такие описания самостоятельно. Обычно константы просто берут из соответствующих заголовочных файлов C++. С простыми константами все проходит без проблем:

```
private const int TTS_ALWAYSSTIP = 0x01;  
private const int TTS_NOPREFIX   = 0x02;  
private const int TTS_BALLOON    = 0x40;
```

А вот с константами, имеющими большие значения, например `0x80000000`, возникает сложность, заключающаяся в том, что компилятор C# не может допустить произвола преобразования беззнакового значения к знаковому, как это допускает компилятор C++. При попытке описать такую константу с типом `int` мы получим ошибку "Cannot implicitly convert type 'uint' to 'int'" ("Невозможно неявно привести тип `uint` к типу `int`").

Прямое приведение типа тоже не помогает: "Constant value '2147483648' cannot be converted to a 'int'" ("Константа '2147483648' не может быть преобразована к типу int").

Вариантов два — либо изменить описание импортированной функции, изменив в заголовке функции тип `int` на `uint` (что не всегда удобно), либо с помощью ключевого слова `unchecked` "уговорить" компилятор не обращать внимание на такое преобразование:

```
private const int WS_POPUP = unchecked{(int)0x80000000};
```

Следующая проблема — описание структуры `ToolInfo`, которая используется для описания окна всплывающей подсказки:

```
private struct ToolInfo
{
    public int size;
    public int flag;
    public IntPtr parent;
    public int id;
    public Rectangle rect;
    public int nullvalue;
    [MarshalAs(UnmanagedType.LPStr)]
    public string text;
    public int param;
}
```

Так как мы собираемся передавать эту структуру в методы Win32, то, во-первых, нужно "совместить" типы и длины полей структуры так, чтобы нужные байты оказались на одинаковых местах и для среды .NET, и для ядра Windows. Для простых типов, таких как `int`, это делается автоматически, а вот для строк, которые в C# работают совсем не так, как в C++, придется указать проецирование типа явным образом, с помощью атрибута `MarshalAs`:

```
[MarshalAs(UnmanagedType.LPStr)]
public string text;
```

Наиболее используемыми являются следующие значения этого атрибута (кроме самого типа могут указываться дополнительные параметры):

- ☐ `Bool` — четырехбайтовый тип `Boolean`. Соответствует Win32-типу `BOOL`. `true` означает ненулевое значение, `false` — ноль;
- ☐ `ByValArray` — массив. Для этого типа должно быть указано значение поля `SizeConst`, равное числу элементов в массиве. Дополнительно параметр `ArraySubType` может указывать тип элементов массива;

- `ByValTStr` — строки конечной длины в стиле Си (соответствуют описанию вида `char s[5]`). Для этого значения должны дополнительно указываться параметр кодировки `StructLayoutAttribute.CharSet` и параметр размера `SizeConst`;
- `Error` — соответствует Win32-типу `HRESULT`;
- `I1` — соответствует типу `bool` языка Си. Значение `true` равно 1, значение `false` равно 0;
- `I2, I4, I8` — соответственно 2-, 4- и 8-байтовые целые числа со знаком;
- `R4, R8` — 4- и 8-байтовые числа с плавающей точкой;
- `U1, U2, U4, U8` — целые беззнаковые числа;
- `VariantBool` — аналог `VARIANT_BOOL`. Значение `true` соответствует 1, значение `false` соответствует 0.

Например, в данном случае неуправляемый код не может определить размер массива данных по описанию, поэтому нужно указать его явно:

```
[MarshalAs(UnmanagedType.ByValArray, SizeConst=8)]
public byte[] data4;
```

Во-вторых, проблема со структурами заключается в том, что политика работы в Win32 подразумевает частичное заполнение полей структуры. Чаще всего, для одних вызовов требуется заполнить одну часть полей, для других — другую. В C++ это считается вполне нормальным, а вот компилятор C# предупреждает, что часть полей не используется: "Field 'CustomTooltip.TooltipsClassWin32.ToolInfo.id' is never assigned to, and will always have its default value 0" ("Полю `CustomTooltip.TooltipsClassWin32.ToolInfo.id` нинкогда не присваивается значение и оно всегда будет иметь значение 0").

Отключить это предупреждение можно с помощью опции препроцессора:

```
#pragma warning disable 649
```

Теперь "обернем" все вызовы Win32 в класс `TooltipsClassWin32`, для того чтобы скрыть детали вызова неуправляемого кода от других классов (листинг 12.12). Теперь остается только сделать компонент, реализующий уже известный нам интерфейс `IExtenderProviderService` (см. листинг 12.1), с помощью которого мы добавим глобальное свойство `BalloonText`. В соответствии с этим нам потребуются методы `GetBalloonText()` и `SetBalloonText()`, реализующие само свойство, и метод `CanExtend()`, говорящий дизайнеру, что наше глобальное свойство применимо ко всем визуальным компонентам, кроме самой формы. Кроме того, несколько свойств потребуется для редактирования текста (`Title`), размера (`MaximumWidth`), значка (`Icon`) и других свойств самой подсказки. Каждое свойство делает двойную работу — уста-

навливает значение внутреннего поля компонента и сразу же передает вызов в неуправляемый код через класс-обертку:

```
toolWindow = new TooltipsClassWin32(max);
```

```
public Color ForeColor
{
    get
    {
        return fgcolor;
    }
    set
    {
        fgcolor = value;
        toolWindow.SetForeColor(fgcolor);
    }
}
```

Полный код компонента показан в листинге 12.13. Я думаю, что приведенной информации вполне достаточно, чтобы разобраться в этом коде. Еще раз подчеркну, что цель этого раздела состоит не в изучении конкретного класса, а в показе возможности взаимодействия с Win32. Надеюсь, поставленной цели я добился.

Листинг 12.12. Класс-обертка для Win32-функций управления окном подсказки

```
using System;
using System.Runtime.InteropServices;
using System.Drawing;
using System.Windows.Forms;

namespace CustomTooltip
{
    public class TooltipsClassWin32
    {
        private IntPtr toolwindow;
        private ToolInfo toolInfo;

        // Конструктор
        public TooltipsClassWin32(int max)
        {
            // Создаем окно класса tooltips_class32
            toolwindow = CreateWindowEx(
                0, "tooltips_class32", string.Empty,
                WS_POPUP | TTS_BALLOON | TTS_NOPREFIX |
```

```

        TTS_ALWAYSSTIP, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        IntPtr.Zero, 0, 0, 0);
SetWindowPos(toolwindow, TOPMOST, 0, 0, 0, 0,
        SWP_NOMOVE | SWP_NOSIZE | SWP_NOACTIVATE);
SendMessage(toolwindow, TTM_SETMAXTIPWIDTH,
        0, new IntPtr(max));

// Специальная структура ToolInfo для управления
// окном подсказки
toolInfo = new ToolInfo();
toolInfo.flag = TTF_SUBCLASS | TTF_TRANSPARENT;
toolInfo.size = Marshal.SizeOf(typeof(ToolInfo));
}

// Добавляем текст подсказки
public void AddBalloonText(Control parent, string value)
{
    SetBalloonText(parent, value, false);
}

// Обновляем текст подсказки
public void UpdateBalloonText(Control parent, string value)
{
    SetBalloonText(parent, value, true);
}

// Удаляем текст подсказки
public void RemoveBalloonText(Control parent)
{
    toolInfo.parent = parent.Handle;

    IntPtr tempPtr = Marshal.AllocHGlobal(toolInfo.size);
    Marshal.StructureToPtr(toolInfo, tempPtr, false);
    SendMessage(toolwindow, TTM_DELTOOL, 0, tempPtr);
    Marshal.FreeHGlobal(tempPtr);
}

// Максимальная ширина
public void SetMaxWidth(int max)
{
    SendMessage(toolwindow, TTM_SETMAXTIPWIDTH,
        0, new IntPtr(max));
}

```

```
// Активность окна
public void SetEnabled(bool enabled)
{
    SendMessage(toolwindow, TTM_ACTIVATE,
        Convert.ToInt32(enabled), new IntPtr(0));
}

// Заголовок и значок
public void SetTitle(string title, BalloonIcons icon)
{
    IntPtr tempPtr = Marshal.StringToHGlobalUni(title);
    SendMessage(toolwindow, TTM_SETTITLE, (int)icon, tempPtr);
    Marshal.FreeHGlobal(tempPtr);
}

// Время показа окна подсказки
public void SetAutoPop(int autopop)
{
    SendMessage(toolwindow, TTM_SETDELAYTIME, TTDI_AUTOPOP,
        new IntPtr(autopop));
}

// Время ожидания до отображения окна подсказки
public void SetInitial(int initial)
{
    SendMessage(toolwindow, TTM_SETDELAYTIME, TTDI_INITIAL,
        new IntPtr(initial));
}

// Фон окна
public void SetBackColor(Color backColor)
{
    SendMessage(toolwindow, TTM_SETTIPBKCOLOR,
        ColorTranslator.ToWin32(backColor),
        new IntPtr(0));
}

// Цвет текста
public void SetForeColor(Color foreColor)
{
    SendMessage(toolwindow, TTM_SETTIPTEXTCOLOR,
        ColorTranslator.ToWin32(foreColor),
        new IntPtr(0));
}
```

```
// Вызывается при изменении размеров
public void Resize(object sender)
{
    toolInfo.parent = ((Control)sender).Handle;

    IntPtr tempPtr = Marshal.AllocHGlobal(toolInfo.size);
    Marshal.StructureToPtr(toolInfo, tempPtr, false);

    SendMessage(toolwindow, TTM_GETTOOLINFO, 0, tempPtr);

    toolInfo = (ToolInfo)Marshal.PtrToStructure(
        tempPtr, typeof(ToolInfo));
    toolInfo.rect = ((Control)sender).ClientRectangle;

    Marshal.StructureToPtr(toolInfo, tempPtr, false);

    SendMessage(toolwindow, TTM_SETTOOLINFO, 0, tempPtr);

    Marshal.FreeHGlobal(tempPtr);
}

// Установка или обновления текста подсказки
private void SetBalloonText(Control parent,
    string value, bool update)
{
    toolInfo.parent = parent.Handle;
    toolInfo.rect = parent.ClientRectangle;
    toolInfo.text = value;
    IntPtr tempPtr = Marshal.AllocHGlobal(toolInfo.size);
    Marshal.StructureToPtr(toolInfo, tempPtr, false);

    if (update)
    {
        SendMessage(toolwindow, TTM_UPDATETIPTEXT, 0, tempPtr);
    }
    else
    {
        SendMessage(toolwindow, TTM_ADDTOOL, 0, tempPtr);
    }

    Marshal.FreeHGlobal(tempPtr);
}
```



```
// Деструктор
~TooltipsClassWin32()
{
    DestroyWindow(toolwindow);
}

#region Win32-константы
private const int TTS_ALWAYSSTIP = 0x01;
private const int TTS_NOPREFIX   = 0x02;
private const int TTS_BALLOON    = 0x40;

private const int TTF_SUBCLASS   = 0x0010;
private const int TTF_TRANSPARENT = 0x0100;

private const int SWP_NOSIZE      = 0x0001;
private const int SWP_NOMOVE     = 0x0002;
private const int SWP_NOACTIVATE = 0x0010;

private const int TTDT_AUTOPOP = 2;
private const int TTDT_INITIAL = 3;

private const int WS_POPUP = unchecked((int)0x80000000);
private const int CW_USEDEFAULT = unchecked((int)0x80000000);

private IntPtr TOPMOST = new IntPtr(-1);

private const int WM_USER          = 0x0400;
private const int TTM_ADDTOOL      = WM_USER + 50;
private const int TTM_DELTOOL      = WM_USER + 51;
private const int TTM_ACTIVATE     = WM_USER + 1;
private const int TTM_SETMAXTIPWIDTH = WM_USER + 24;
private const int TTM_SETTITLE     = WM_USER + 33;
private const int TTM_SETDELAYTIME = WM_USER + 3;
private const int TTM_UPDATETIPTEXT = WM_USER + 57;
private const int TTM_SETTIPBKCOLOR = WM_USER + 19;
private const int TTM_SETTIPTHEXTCOLOR = WM_USER + 20;
private const int TTM_GETTOOLINFO   = WM_USER + 53;
private const int TTM_SETTOOLINFO   = WM_USER + 54;
#endregion

#region Win32-функции
[DllImport("user32.dll")]
private static extern IntPtr CreateWindowEx(
    int exstyle, string classname,
    string windowtitle, int style, int x, int y,
```

```

        int width, int height,
        IntPtr parent, int menu,
        int nullvalue, int nullptr);

[DllImport("user32.dll")]
private static extern int DestroyWindow(IntPtr hwnd);

[DllImport("User32.dll")]
private static extern bool SetWindowPos(IntPtr hwnd,
    IntPtr hWndInsertAfter,
    int X, int Y, int cx, int cy, int uFlags);

[DllImport("User32.dll")]
private static extern int SendMessage(IntPtr hwnd,
    int Msg, int wParam,
    IntPtr lParam);
#endregion

#region Win32 структуры
#pragma warning disable 649
private struct ToolInfo
{
    public int size;
    public int flag;
    public IntPtr parent;
    public int id;
    public Rectangle rect;
    public int nullvalue;
    [MarshalAs(UnmanagedType.LPStr)]
    public string text;
    public int param;
}
#pragma warning restore 649
#endregion
}

public enum BalloonIcons : int
{
    None    = 0,
    Info    = 1,
    Warning = 2,
    Error   = 3
}
}

```

Листинг 12.13. Класс BalloonToolTip, реализующий интерфейс IExtenderProvider

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Windows.Forms.Design;

namespace CustomToolTip
{
    // Атрибут ProvideProperty указывает, что свойство BalloonText
    // будет добавлено ко всем компонентам. Для этого используется
    // класс IExtenderProvider.
    [ProvideProperty("BalloonText", typeof(Control))]
    public class BalloonToolTip : Component, IExtenderProvider
    {
        private int max;
        private int autopop;
        private int initial;
        private bool enabled;
        private string title;
        private Hashtable toolTexts;
        private BalloonIcons icon;
        private Color bgcolor;
        private Color fgcolor;

        // Win32-обертка для управления
        // окном всплывающей подсказки
        private TooltipsClassWin32 toolWindow;

        // Конструктор
        public BalloonToolTip()
        {
            // Инициализация свойств
            max = 200;
            autopop = 5000;
            initial = 500;
            title = string.Empty;
            bgcolor = Color.FromKnownColor(KnownColor.Info);
        }
    }
}
```

```
fgcolor = Color.FromKnownColor(KnownColor.InfoText);
toolTexts = new Hashtable();
enabled = true;
icon = BalloonIcons.None;

// Создаем Win32-класс всплывающей подсказки
toolWindow = new TooltipsClassWin32(max);
)

// Деструктор
~BalloonToolTip()
{
    Dispose(false);
}

// Реализация метода CanExtend интерфейса IExtenderProvider.
// Глобальное свойство может быть добавлено к любому
// компоненту, кроме самого компонента BalloonToolTip и формы
public bool CanExtend(object extendee)
{
    if (extendee is Control &&
        !(extendee is BalloonToolTip) &&
        !(extendee is Form))
    {
        return true;
    }
    return false;
}

// Метод Get свойства BalloonText
[DefaultValue("")]
public string GetBalloonText(Control parent)
{
    if (toolTexts.Contains(parent))
    {
        return toolTexts[parent].ToString();
    }
    else
    {
        return null;
    }
}
```

```
// Метод Set свойства BalloonText
public void SetBalloonText(Control parent, string value)
{
    if (value == null)
    {
        value = string.Empty;
    }

    // Указание пустой строки
    // означает удаление всплывающей подсказки
    if (value == string.Empty)
    {
        toolTexts.Remove(parent);
        toolWindow.RemoveBalloonText(parent);
        parent.Resize -= new EventHandler(Control_Resize);
    }
    else
    {
        // Если всплывающее окно уже создано, обновляем значение
        if (toolTexts.Contains(parent))
        {
            toolTexts[parent] = value;
            toolWindow.UpdateBalloonText(parent, value);
        }
        else
        {
            // Если окно еще не создано, то создаем его
            toolTexts.Add(parent, value);
            toolWindow.AddBalloonText(parent, value);
        }

        parent.Resize += new EventHandler(Control_Resize);
    }
}

[Description("Значок окна подсказки")]
[DefaultValue(BalloonIcons.None)]
public BalloonIcons Icon
{
    get
    {
        return icon;
    }
}
```

```
set
{
    icon = value;
    toolWindow.SetTitle(title, icon);
}

[Description("Максимальная ширина")]
[DefaultValue(200)]
public int MaximumWidth
{
    get
    {
        return max;
    }
    set
    {
        max = value;
        toolWindow.SetMaxWidth(max);
    }
}

[Description("Активность")]
[DefaultValue(true)]
public bool Enabled
{
    get
    {
        return enabled;
    }
    set
    {
        enabled = value;
        toolWindow.SetEnabled(enabled);
    }
}

[Description("Строка подсказки")]
public string Title
{
    get
    {
        return title;
    }
}
```

```

        set
        {
            title = value;
            toolWindow.SetTitle(title, icon);
        }
    }

    [Description("Время показа окна (в сек.)")]
    [DefaultValue(5)]
    public int AutoPop
    {
        get
        {
            return autopop / 1000;
        }
        set
        {
            . autopop = value * 1000;
            toolWindow.SetAutoPop(autopop);
        }
    }

    [Description("Время ожидания до отображения окна подсказки (мс)")]
    [DefaultValue(500)]
    public int Initial
    {
        get
        {
            return initial;
        }
        set
        {
            initial = value;
            toolWindow.SetInitial(initial);
        }
    }

    [Description("Фон окна")]
    public Color BackColor
    {
        get
        {
            return bgcolor;
        }
    }

```

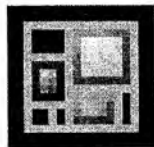
```
set
{
    bgcolor = value;
    toolWindow.SetBackColor(bgcolor);
}

[Description("Цвет текста")]
public Color ForeColor
{
    get
    {
        return fgcolor;
    }
    set
    {
        fgcolor = value;
        toolWindow.SetForeColor(fgcolor);
    }
}

// Освобождение ресурсов
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        toolTexts.Clear();
        toolTexts = null;
    }
    toolWindow = null;
    base.Dispose(disposing);
}

// Изменение размеров компонента
private void Control_Resize(object sender, EventArgs e)
{
    toolWindow.Resize(sender);
}
}
```


ГЛАВА 13



Компоненты работы с данными

Чтобы слова не расходились с делом,
нужно молчать и ничего не делать.

Привязка данных (data binding) предоставляет программисту возможность не только создать связь между компонентами и источником данных (например, базой данных), но и дает мощный механизм связи компонентов между собой. Модель привязки данных C# 2.0 позволяет, во-первых, использовать в качестве источника данных массивы, коллекции и т. д., а во-вторых, легко и быстро связывать компоненты между собой. Конечно, я не собираюсь рассматривать внутреннее устройство самого механизма привязки данных. Здесь я расскажу только об использовании механизма привязки данных в рамках создания собственных компонентов.

13.1. Несколько слов о привязке данных

Основная идея *привязки данных* — создать связь между источником данных и компонентом отображения (редактирования). Под связью имеется в виду автоматическое обновление отображения при изменении данных и обновление данных в источнике при редактировании их в компоненте. Привязка данных может быть простой (simple data binding) и сложной (complex data binding). Калька английских терминов не совсем очевидно отражает суть дела. Проще сказать, что простая привязка создает связь между источником данных и одной записью, а сложная — между источником данных и списком (рис. 13.1).

Общая схема работы привязки данных показана на рис. 13.2. Источником данных может служить любой класс, как минимум, реализующий интерфейс `IList`. Класс `PropertyManager` описывает связь свойства и источника данных, а класс `CurrencyManager` — связь источника данных и списка. Класс `BindingContext`

отвечает за хранение и управление привязкой. О прямой привязке свойств компонентов между собой я расскажу в разд. 13.5.

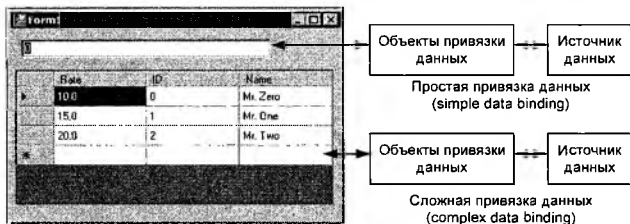


Рис. 13.1. Простая и сложная привязки данных

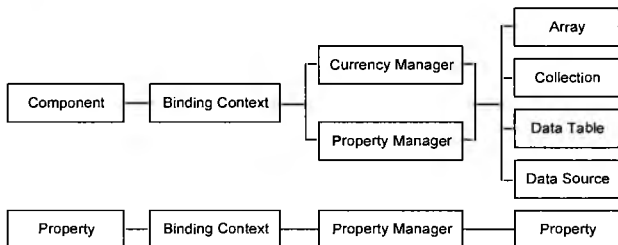


Рис. 13.2. Модель привязки данных

Чтобы не углубляться в "теорию", я покажу как работает механизм привязки данных на небольшом примере. Форма, показанная на рис. 13.3, содержит шесть компонентов `textBox` и один компонент `dataGridView`. Для организации источника данных мы будем использовать класс `Customer` (листинг 13.1), содержащий три поля: `ID` (уникальный номер), `Name` (имя), `Rate` (стоимость). Начнем с простой привязки. Компоненты `textBox1`, `textBox2` и `textBox3` привяжем к некоторому экземпляру класса `Customer` следующим образом:

```
Customer customer = new Customer(0, "Mr. Zero", 10.0M);
this.textBox1.DataBindings.Add("Text", customer, "ID", true);
this.textBox2.DataBindings.Add("Text", customer, "Name", true);
this.textBox3.DataBindings.Add("Text", customer, "Name", true);
```

	Rate	ID	Name
1	10.0	0	Mr. Zero
2	15.0	1	Mr. One
3	20.0	2	Mr. Two

Рис. 13.3. Привязка данных

Листинг 13.1. Класс Customer

```
public class Customer
{
    private int id;
    private string name;
    private Decimal rate;

    // Конструктор
    public Customer()
    {
        this.ID = -1;
        this.Name = string.Empty;
        this.Rate = 0.0M;
    }

    public Customer(int id, string name, Decimal rate)
    {
        this.ID = id;
        this.Name = name;
        this.Rate = rate;
    }

    // Номер
    public int ID
```

```

{
    get { return id; }
    set { id = value; }
}

// Имя
public string Name
{
    get { return name; }
    set { name = value; }
}

// Стоимость
public Decimal Rate
{
    get { return rate; }
    set { rate = value; }
}
}

```

Итак, свойство `Text` первого компонента привязано к полю `ID`, а два других компонента привязаны к одному и тому же свойству `Name`. Запустив приложение, можно увидеть, что, во-первых, в каждом из текстовых полей отображается привязанное к нему свойство, и во-вторых, при изменении текстового поля `textBox2` эти изменения отображаются в компоненте `textBox3`! Это происходит из-за того, что оба этих компонента привязаны к одному полю, а значит, при изменении значения `textBox2` происходит изменение самого поля, что, в свою очередь, вызывает изменение значения `textBox3`:

`textBox2` → поле `Name` → `textBox3`

Сложная привязка данных позволяет привязать список объектов к некоторому компоненту, предназначенному для отображения (и, возможно, редактирования) этого списка. Например, компонент `dataGridView1` мы привяжем к списку объектов `Customer`:

```

BindingList<Customer> list = new BindingList<Customer>>();
list.Add(new Customer(0, "Mr. Zero", 10.0M));
list.Add(new Customer(1, "Mr. One", 15.0M));
list.Add(new Customer(2, "Mr. Two", 20.0M));
this.dataGridView1.DataSource = list;

```

Запустив приложение, мы увидим, что список отображается в сетке данных. Привяжем еще компонент `textBox4` к тому же списку:

```

this.textBox4.DataBindings.Add("Text", list, "Name", true);

```

Теперь, перемещаясь по списку, мы можем видеть, как меняется значение в `textBox4`. Другими словами, мы создали связь между простой и сложной привязками данных.

И еще два небольших теста. Для компонента `textBox5` мы сначала создадим класс привязки, а затем уже выполним саму привязку:

```
Binding rateBinding = new Binding("Text", list, "Rate", true);
rateBinding.FormatString = "c";
this.textBox5.DataBindings.Add(rateBinding);
```

Свойство `FormatString` позволяет задать формат отображения данных. При необходимости более сложной обработки отображаемых и вводимых данных можно воспользоваться событиями `Format` и `Parse`:

```
Binding nameBinding = new Binding("Text", list, "Name");
nameBinding.Format += new ConvertEventHandler(nameBinding_Format);
nameBinding.Parse += new ConvertEventHandler(nameBinding_Parse);
this.textBox6.DataBindings.Add(nameBinding);
```

```
void nameBinding_Parse(object sender, ConvertEventArgs e)
```

```
{
    string value = e.Value.ToString();
    if (value.IndexOf("Имя: ") == 0)
        e.Value = e.Value.ToString().Remove(0, 5);
}
```

```
void nameBinding_Format(object sender, ConvertEventArgs e)
```

```
{
    e.Value = string.Format("Имя: {0}", e.Value);
}
```

Событие `Format` вызывается при необходимости отобразить значение, а событие `Parse` — при попытке получить реальное значение из введенных пользователем данных. Вообще, класс `Binding` имеет множество полезных свойств, но описывать их здесь я не буду. Для этого существует множество справочной литературы.

В конце раздела я покажу, как получить ссылки на объекты, которые я описывал ранее:

```
BindingManagerBase managerBase =
    this.textBox1.BindingContext[customer, "ID"];
PropertyManager propertyManager =
    (PropertyManager) this.textBox1.BindingContext[customer];
CurrencyManager currencyManager =
    (CurrencyManager) this.dataGridView1.BindingContext[list];
```

Например, для управления положением указателя данных можно использовать такой код:

```
private void button1_Click(object sender, EventArgs e)
{
    if (currencyManager.Position < currencyManager.Count - 1)
    {
        currencyManager.Position++;
    }
}

private void button2_Click(object sender, EventArgs e)
{
    if (currencyManager.Position > 0)
    {
        currencyManager.Position--;
    }
}
```

Далее я расскажу о стандартных свойствах привязки данных.

13.2. Стандартная привязка данных

Посмотрим еще раз на привязку списка записей `Customer`:

```
BindingList<Customer> list = new BindingList<Customer>();
list.Add(new Customer(0, "Mr. Zero", 10.0M));
list.Add(new Customer(1, "Mr. One", 15.0M));
list.Add(new Customer(2, "Mr. Two", 20.0M));
```

Два вида привязки я уже показал в предыдущем разделе. Привязка к компоненту `TextBox`:

```
this.textBox1.DataBindings.Add("Text", list, "Name");
```

Привязка к компоненту `GridView`:

```
this.dataGridView1.DataSource = list;
```

Итак, привязка к компоненту `TextBox` осуществляется с помощью свойства `DataBinding`, а привязка к компоненту `DataGridView` — с помощью свойства `DataSource`. Для выпадающего списка `ComboBox` привязка выглядит почти так же:

```
this.comboBox2.DataSource = list;
```

Но ведь компонент `ComboBox` отображает только одно поле! Что же произойдет в этом случае? Выпадающий список отобразит нужное количество строк,

содержащих одно и то же значение — `DataBindingTest.Customer`. Не трудно догадаться, что это значение получено вызовом метода `ToString()` класса `Customer`. Действительно, перекрыв этот метод (листинг 13.2), мы получим более осмысленные данные в списке (рис. 13.4).

Листинг 13.2. Метод `ToString()` класса `Customer`

```
public class Customer
{
    ... ..
    public override string ToString()
    {
        return string.Format("ID={0}, Name={1}, Rate={2}", ID, Name, Rate);
    }
}
```

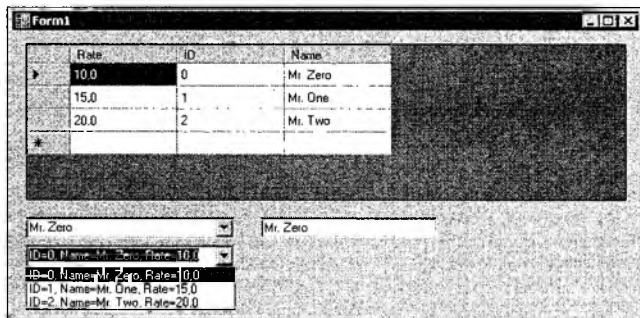


Рис. 13.4. Привязка данных: `GridView`, `ComboBox` и `TextBox`

Еще один способ привязки данных к компоненту `ComboBox` — указание поля, по которому будет осуществляться привязка:

```
this.comboBox1.DisplayMember = "Name";
this.comboBox1.DataSource = list;
```

Свойство `DisplayMember` указывает, какое именно поле списка будет отображено в выпадающем списке.

Теперь посмотрим, как эти свойства выглядят в редакторе свойств. Свойства, относящиеся к привязке данных, располагаются в категории **Data** (рис. 13.5):

- ☐ **DataSource** — определяет источник данных;
- ☐ **DataMember** — определяет поле, которое будет отображаться. Если это свойство не задано, то вызывается метод `ToString()` для вычисления отображаемого значения;
- ☐ **ValueMember** — определяет поле, которое будет возвращаться в свойстве текущего выбранного элемента `SelectedValue`. Если это свойство не задано, то вызывается метод `ToString()` для вычисления возвращаемого значения.

Эти же свойства отображаются в меню смарт-тега при включении режима **Use data bound items** (Использовать привязку к данным) (рис. 13.6).

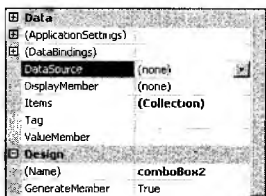


Рис. 13.5. Список свойств в категории **Data**

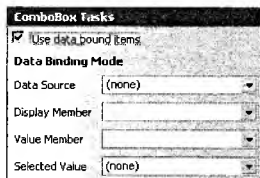


Рис. 13.6. Меню смарт-тега компонента **ComboBox**

Для подключения источника данных нужно выбрать значение свойства **DataSource** из существующих источников или, нажав на ссылку **Add Project Data Source**, добавить новый источник данных (рис. 13.7). Источником данных может быть выбрана база данных, Web-сервис или класс, реализующий нужные интерфейсы (рис. 13.8). Например, в качестве источника данных можно выбрать класс `CustomerDataSource` (листинг 13.3), аналогичный тому, что мы уже использовали в предыдущем разделе (рис. 13.9). После выбора источника данных можно выбрать остальные поля привязки (рис. 13.10).

Листинг 13.3. Класс `CustomerDataSource`

```
public class CustomerDataSource : BindingList<Customer>
{
    public CustomerDataSource()
    {
        this.Add(new Customer(0, "Mr. Zero", 10.0M));
        this.Add(new Customer(1, "Mr. One", 15.0M));
        this.Add(new Customer(2, "Mr. Two", 20.0M));
    }
}
```




Рис. 13.7. Редактор свойства DataSource

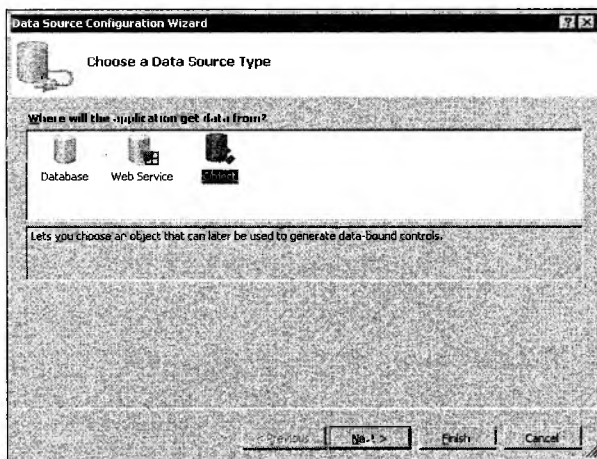


Рис. 13.8. Выбор типа источника данных

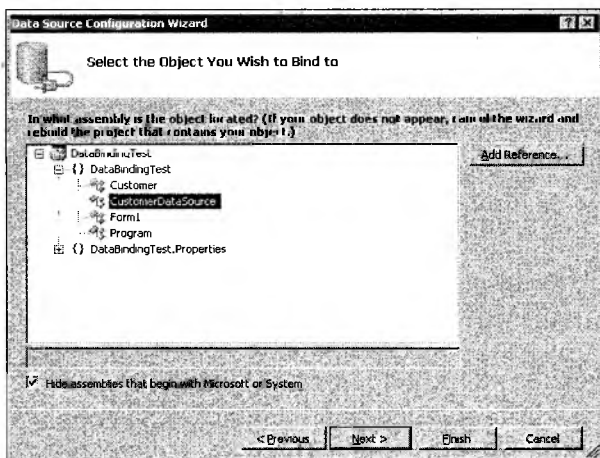


Рис. 13.9. Класс CustomerDataSource как источник данных

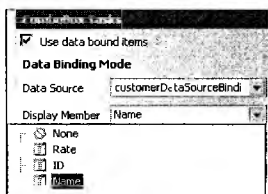


Рис. 13.10. Выбор значения поля DisplayMember

13.3. Интерфейсы источника данных

В предыдущем разделе мы использовали в качестве источника данных класс, наследованный от класса `BindingList`. На самом деле источником данных может являться любой класс, реализующий интерфейс `IList`. Этого вполне достаточно для отображения списка данных. Сейчас я хочу рассказать о некоторых других интерфейсах, доступных для источников данных. Детальный

рассказ о них выходит за рамки разработки компонентов, но общее представление нужно.

Интерфейс `IBindingList` управляет редактированием данных как простой привязки данных, так и привязки списка данных:

```
public interface IBindingList : IList, ICollection, IEnumerable
{
    // Возвращает true, если источник
    // данных поддерживает редактирование данных
    bool AllowEdit { get; }
    // Возвращает true, если источник данных поддерживает
    // добавление новых данных
    bool AllowNew { get; }
    // Возвращает true, если источник данных поддерживает
    // удаление данных
    bool AllowRemove { get; }
    // Возвращает true, если источник данных был отсортирован
    // с помощью вызова метода ApplySort
    bool IsSorted { get; }
    // Возвращает направление сортировки ListSortDirection.Ascending
    // или Ascending.Descending.
    ListSortDirection SortDirection { get; }
    // Возвращает дескриптор свойства, по которому
    // производится сортировка
    PropertyDescriptor SortProperty { get; }
    // Возвращает true, если при изменении списка
    // данных будет вызываться метод ListChanged
    bool SupportsChangeNotification { get; }
    // Возвращает true, если источник данных поддерживает
    // поиск с помощью метода Find
    bool SupportsSearching { get; }
    // Возвращает true, если источник данных поддерживает сортировку
    bool SupportsSorting { get; }
    // Вызывается при изменении списка данных
    event ListChangedEventHandler ListChanged;
    // Добавление индекса для поиска по указанному свойству
    void AddIndex(PropertyDescriptor property);
    // Добавление нового элемента в список
    object AddNew();
    // Сортировка данных по указанному свойству
    // в указанном направлении
    void ApplySort(PropertyDescriptor property,
        ListSortDirection direction);
}
```

```
// Поиск данных по указанному свойству и значению
int Find(PropertyDescriptor property, object key);
// Удаляет индекс
void RemoveIndex(PropertyDescriptor property);
// Удаляет сортировку данных
void RemoveSort();
}
```

Интерфейс `ICancelAddNew` управляет возможностями добавления новых элементов:

```
public interface ICancelAddNew
{
    // Отменить добавление нового элемента
    void CancelNew(int itemIndex);
    // Завершить добавление нового элемента
    void EndNew(int itemIndex);
}
```

Интерфейс `IRaiseItemChangedEvents` предоставляет событие, вызываемое при изменении списка данных:

```
public interface IRaiseItemChangedEvents
{
    bool RaisesItemChangedEvents { get; }
}
```

Интерфейс `IEditableObject` предоставляет интерфейсы управления редактированием:

```
public interface IEditableObject
{
    // Начало редактирования элемента
    void BeginEdit();
    // Откат результатов редактирования
    void CancelEdit();
    // Сохранение результатов редактирования
    void EndEdit();
}
```

Интерфейс `IChangeTracking` предоставляет механизмы для запроса изменений данных:

```
public interface IChangeTracking
{
    // Возвращает true, если данные были изменены
    bool IsChanged { get; }
```

```
// Указывает принять изменения данных
void AcceptChanges();
}
```

Интерфейс `IDataErrorInfo` предоставляет механизм управления ошибками:

```
public interface IDataErrorInfo
{
    // Возвращает строку ошибки или
    // пустую строку, если ошибок нет
    string Error { get; }
    // Возвращает строку ошибки для каждого свойства
    // или пустую строку, если ошибок нет
    string this[string columnName] { get; }
}
```

13.4. Разработка компонентов работы с данными

В качестве примера я покажу компонент `BoundListView`, который является наследником обычного `ListView`, но с возможностью привязки данных.

Для начала мне потребуются свойства `DataSource` и `DataMember`. Их описание достаточно стандартно, и его можно использовать независимо от типа компонента. Описание `DataSource` выглядит так:

```
private object dataSource;

[TypeConverter("System.Windows.Forms.Design.DataSourceConverter,
               System.Design")]
[Category("Data")]
[DefaultValue(null)]
public object DataSource
{
    get
    {
        return this.dataSource;
    }
    set
    {
        if (this.dataSource != value)
        {
            this.dataSource = value;
        }
    }
}
```

```

        TryDataBinding();
    }
}

```

Атрибут `TypeConverter` подключает стандартный конвертер, предназначенный для источников данных. В таком варианте редактор этого свойства будет выглядеть как выпадающий список, содержащий все доступные источники данных (рис. 13.11). В этом диалоге нет возможности создать новый источник данных, как на рис. 13.7. Такой редактор можно получить, наследовав атрибуты свойства от класса `IDataSource`:

```

[Category("Data")]
[DefaultValue(null)]
[AttributeProvider(typeof(IDataSource))]
[Browsable(true)]
public object DataSource
{
    get/set
}

```

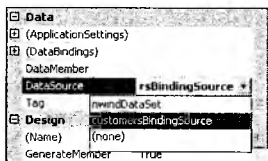


Рис. 13.11. Конвертер типа для свойства `DataSource`

Теперь редактор будет выглядеть именно так, как показано на рис. 13.7. Какой из способов лучше, зависит от конкретной ситуации.

Описание свойства `DataMember` выглядит примерно так же, только вместо конвертера здесь мы подключаем редактор типа `DataMemberListEditor`:

```

private string dataMember;

[Category("Data")]
[Editor("System.Windows.Forms.Design.DataMemberListEditor,
        System.Design",
        "System.Drawing.Design.UITypeEditor, System.Drawing")]
[DefaultValue(null)]
public string DataMember

```

```
{
    get
    {
        return this.dataMember;
    }
    set
    {
        if (this.dataMember != value)
        {
            this.dataMember = value;
            TryDataBinding();
        }
    }
}
```

Как не трудно догадаться, метод `TryDataBinding()` должен получать данные из источника данных. Основной вопрос заключается в том, как именно это делать? Можно, конечно, использовать код наподобие такого:

```
DataSet ds = this.dataSource as DataSet;
DataTable dt = ds.Tables[0];
foreach(DataRow dr in dt.Rows)
{
    ListViewItem lvi = new ListViewItem(dr[DataMember].ToString());
    lvi.Tag = dr;
    Items.Add(lvi);
}
```

Код, в общем-то, неплохой, но только при условии, что источником данных является именно `DataSet`. Но ведь это далеко не всегда так. Источником данных может быть и `ArrayList`, и `Web-сервис`... Именно поэтому нужно использовать другой механизм подключения источника — класс `CurrencyManager`, о котором я уже рассказывал. Соответствующее свойство может выглядеть следующим образом:

```
private CurrencyManager currencyManager;
protected CurrencyManager CurrencyManager
{
    get
    {
        return this.currencyManager;
    }
}
```

А заполнять это свойство мы будем как раз в методе `TryDataBinding()`, перед тем как обновить данные в компоненте:

```

private void TryDataBinding()
{
    ... ..
    // Получаем CurrencyManager
    cm = (CurrencyManager) base.BindingContext[
        this.DataSource, this.DataMember];

    ... ..
    // Источник данных изменился - обновляем данные
    if (this.currencyManager != cm)
    {
        ... ..
        this.currencyManager = cm;
        ... ..
        this.currencyManager.ListChanged += listChangedHandler;
        this.currencyManager.PositionChanged += positionChangedHandler;
    }
    // Обновляем информацию о колонках
    UpdateColumns();
    // Обновляем данные
    UpdateAllData();
}
}

```

Как я уже говорил, все обновления данных мы должны проводить через `currencyManager`. Обновление информации о колонках (этот метод вызывается в конце метода `TryDataBinding()`) выглядит следующим образом:

```

private void UpdateColumns()
{
    // Удаляем все колонки
    this.Columns.Clear();
    // Нет данных - нет колонок
    if (currencyManager == null)
        return;
    // Добавляем все колонки
    foreach (PropertyDescriptor prop in
        CurrencyManager.GetItemProperties())
    {
        ColumnHeader column = new ColumnHeader();
        column.Text = prop.Name;
        this.Columns.Add(column);
    }
}

```


Как видно из кода, мы просто перебираем все доступные свойства источника данных и добавляем имена этих свойств в список колонок нашего компонента. Обновление данных выглядит тоже несложно:

```
private void UpdateAllData()
{
    this.Items.Clear();
    for (int i = 0; i < CurrencyManager.Count; i++)
    {
        ListViewItem item = GetListViewItem(i);
        this.Items.Insert(i, item);
    }
}
```

Метод `GetListViewItem()` возвращает элемент `ListViewItem` по номеру элемента.

Остается осуществить привязку источника данных и данных компонента. Сделать это позволяет событие `ListChanged`, которое вызывается при изменении данных в источнике. Обработчик события выглядит следующим образом:

```
private void CurrencyManager_ListChanged(object sender,
    ListChangedEventArgs e)
{
    switch (e.ListChangedType)
    {
        case ListChangedType.Reset:
        case ListChangedType.ItemMoved:
            // Обновить все данные
            UpdateAllData();
            break;
        case ListChangedType.ItemAdded:
            // Добавить элемент
            AddItem(e.NewIndex);
            break;
        case ListChangedType.ItemChanged:
            // Обновить элемент
            UpdateItem(e.NewIndex);
            break;
        case ListChangedType.ItemDeleted:
            // Удалить элемент
            DeleteItem(e.NewIndex);
            break;
    }
}
```

```

        default:
            // Обновить все
            UpdateColumns();
            UpdateAllData();
            break;
    }
}

```

Параметр `e.ListChangedType` передает тип произошедшего изменения, и в соответствии с ним выполняется определенное действие над данными. Единственный нереализованный метод — это метод перемещения элемента. По аналогии с остальными методами можно реализовать и его, но можно оставить как есть — просто перечитывать все данные. Значение `Reset` передается при "большом" изменении данных, например, сортировке. В этом случае также надо перечитать все данные заново. Не буду приводить каждый из методов обновления, их можно найти в полном листинге.

Теперь давайте разберемся с обратной синхронизацией, т. е. изменением данных в компоненте. В компоненте `ListView` событие `OnAfterLabelEdit` вызывается при редактировании данных. Именно его и нужно обрабатывать для передачи изменений в источник данных:

```

protected override void OnAfterLabelEdit(LabelEditEventArgs e)
{
    base.OnAfterLabelEdit(e);

    if (e.Label == null)
    {
        // Нажать <Esc> - ничего не отредактировано
        e.CancelEdit = true;
        return;
    }

    if (CurrencyManager.List.Count > e.Item)
    {
        object row = CurrencyManager.List[e.Item];

        // В ListView можно редактировать только первую колонку
        PropertyDescriptor col =
            CurrencyManager.GetItemProperties().Find(
                this.Columns[0].Text, false);

        try
        {
            if (row != null && col != null)

```

```
{
    col.SetValue(row, e.Label);
}
CurrencyManager.EndCurrentEdit();
}
catch (Exception ex)
{
    // Ошибочный формат данных, слишком
    // длинная строка и т. д.
    MessageBox.Show("Ошибка:\r\n" + ex.Message,
        "Сбой редактирования", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    // Отменяем редактирование
    currencyManager.CancelCurrentEdit();
    e.CancelEdit = true;
}
}
```

Если пользователь отменил редактирование, нажав клавишу <Esc>, то поле `e.Label` будет равно `null`. Иначе это поле содержит новое значение отредактированного элемента. В компоненте `ListView` можно редактировать только первую колонку, поэтому наша задача состоит в записи нового значения в эту колонку. В случае ошибки (например, формат данных не совпадает) нужно отменить редактирование, вызвав метод `CancelCurrentEdit()`.

Кроме синхронизации данных, требуется и синхронизация указателя данных. Для этого мы будем обрабатывать события `PositionChanged` класса `CurrencyManager` и `SelectedIndexChanged` класса `ListView`. Первое событие вызывается при изменении позиции указателя в источнике данных и перемещает указатель в компоненте:

```
private void CurrencyManager_PositionChanged(object sender, EventArgs e)
{
    if (this.Items.Count > CurrencyManager.Position)
    {
        this.Items[CurrencyManager.Position].Selected = true;
        this.EnsureVisible(CurrencyManager.Position);
    }
}
```

Второе, наоборот, вызывается при изменении позиции указателя в компоненте и, соответственно, должно изменять позицию указателя в источнике данных:

```
private void ListViewDataBinding_SelectedIndexChanged(
    object sender, EventArgs e)
{
    try
    {
        if (this.SelectedIndices.Count > 0 &&
            CurrencyManager.Position != this.SelectedIndices[0])
            CurrencyManager.Position = this.SelectedIndices[0];
    }
    catch
    {
    }
}
```

Остается добавить к компоненту специальный атрибут `ComplexBindingProperties`, обозначающий имя свойства, ответственного за привязку данных:

```
[ComplexBindingProperties("DataSource", "DataMember")]
public class BoundListView : ListView
{
}
```

Полный код компонента показан в листинге 13.4, в результат работы — на рис. 13.12.

Листинг 13.4. Компонент BoundListView

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Windows.Forms;

namespace ComplexDataBindingSample
{
    /// <summary>
    /// Класс ListView с возможностью привязки данных
    /// </summary>
    [ComplexBindingProperties("DataSource", "DataMember")]
    public class BoundListView : ListView
    {
        private ListChangedEventHandler listChangedHandler;
        private EventHandler positionChangedHandler;
```

```
private object dataSource;
private string dataMember;
private CurrencyManager currencyManager;

/// <summary>
/// Конструктор
/// </summary>
public BoundListView()
{
    listChangedHandler = new ListChangedEventHandler(
        CurrencyManager_ListChanged);
    positionChangedHandler =
        new EventHandler(CurrencyManager_PositionChanged);

    this.View = System.Windows.Forms.View.Details;
    this.FullRowSelect = true;
    this.HideSelection = false;
    this.MultiSelect = false;
    this.LabelEdit = true;

    this.SelectedIndexChanged += new EventHandler(
        ListViewDataBinding_SelectedIndexChanged);
}

/// <summary>
/// Источник данных
/// </summary>
[TypeConverter("System.Windows.Forms.Design.DataSourceConverter,
    System.Design")]
[Category("Data")]
[DefaultValue(null)]
public object DataSource
{
    get
    {
        return this.dataSource;
    }
    set
    {
        if (this.dataSource != value)
        {
            this.dataSource = value;
        }
    }
}
```

```
        TryDataBinding();
    }
}

/// <summary>
/// Поле привязки
/// </summary>
[Category("Data")]
[Editor("System.Windows.Forms.Design.DataMemberListEditor,
        System.Design",
        "System.Drawing.Design.UITypeEditor, System.Drawing")]
[DefaultValue(null)]
public string DataMember
{
    get
    {
        return this.dataMember;
    }
    set
    {
        if (this.dataMember != value)
        {
            this.dataMember = value;
            TryDataBinding();
        }
    }
}

/// <summary>
/// Возвращает CurrencyManager
/// </summary>
protected CurrencyManager CurrencyManager
{
    get
    {
        return this.currencyManager;
    }
}

/// <summary>
/// Обновить привязку данных при смене компонента-владельца (Parent)
/// </summary>
```

```
protected override void OnBindingContextChanged(EventArgs e)
{
    this.TryDataBinding();
    base.OnBindingContextChanged(e);
}

/// <summary>
/// Попытка выполнить привязку данных
/// </summary>
private void TryDataBinding()
{
    // Нет данных для привязки - выходим
    if (this.DataSource == null || base.BindingContext == null)
        return;

    CurrencyManager cm;
    try
    {
        // Получаем CurrencyManager
        cm = (CurrencyManager)base.BindingContext[
            this.DataSource, this.DataMember];
    }
    catch (System.ArgumentException)
    {
        // CurrencyManager не найден
        return;
    }

    // Источник данных изменился - обновляем данные
    if (this.currencyManager != cm)
    {
        // Отключаем "старый" currencyManager
        if (this.currencyManager != null)
        {
            this.currencyManager.ListChanged -= listChangedHandler;
            this.currencyManager.PositionChanged -= positionChangedHandler;
        }
        // Сохраняем новое значение
        this.currencyManager = cm;
        // Восстанавливаем подписку событий CurrencyManager
        if (this.currencyManager != null)
        {
            this.currencyManager.ListChanged += listChangedHandler;
        }
    }
}
```

```
        this.currencyManager.PositionChanged += positionChangedHandler;
    }

    // Обновляем информацию о колонках
    UpdateColumns();
    // Обновляем данные
    UpdateAllData();
}

}

/// <summary>
/// Обновить информацию о колонках
/// </summary>
private void UpdateColumns()
{
    // Удаляем все колонки
    this.Columns.Clear();
    // Нет данных - нет колонок
    if (currencyManager == null)
        return;
    // Добавляем все колонки
    foreach (PropertyDescriptor prop in
        CurrencyManager.GetItemProperties())
    {
        ColumnHeader column = new ColumnHeader();
        column.Text = prop.Name;
        this.Columns.Add(column);
    }
}

/// <summary>
/// Обновить все элементы
/// </summary>
private void UpdateAllData()
{
    this.Items.Clear();
    for (int i = 0; i < CurrencyManager.Count; i++)
    {
        AddItem(i);
    }
}
```



```
/// <summary>
/// Добавить новый элемент
/// </summary>
/// <param name="index">Индекс элемента</param>
private void AddItem(int index)
{
    ListViewItem item = GetListViewItem(index);
    this.Items.Insert(index, item);
}

/// <summary>
/// Обработчик изменения данных
/// </summary>
private void CurrencyManager_ListChanged(object sender,
    ListChangedEventArgs e)
{
    switch (e.ListChangedType)
    {
        case ListChangedType.Reset:
        case ListChangedType.ItemMoved:
            // Обновить все данные
            UpdateAllData();
            break;
        case ListChangedType.ItemAdded:
            // Добавить элемент
            AddItem(e.NewIndex);
            break;
        case ListChangedType.ItemChanged:
            // Обновить элемент
            UpdateItem(e.NewIndex);
            break;
        case ListChangedType.ItemDeleted:
            // Удалить элемент
            DeleteItem(e.NewIndex);
            break;
        default:
            // Обновить все
            UpdateColumns();
            UpdateAllData();
            break;
    }
}
```

```
/// <summary>
/// Обновить элемент по указанному индексу
/// </summary>
/// <param name="index">Индекс элемента</param>
private void UpdateItem(int index)
{
    if (index >= 0 && index < this.Items.Count)
    {
        ListViewItem item = GetListViewItem(index);
        this.Items[index] = item;
    }
}

/// <summary>
/// Возвращает элемент ListViewItem
/// </summary>
/// <param name="index">Индекс строки</param>
private ListViewItem GetListViewItem(int index)
{
    // Получить строку
    object row = CurrencyManager.List[index];
    // Получить список колонок
   PropertyDescriptorCollection propColl =
        CurrencyManager.GetItemProperties();

    // Список строк
    ArrayList items = new ArrayList();

    // Собираем значения каждой колонки
    foreach (ColumnHeader column in this.Columns)
    {
        PropertyDescriptor prop = propColl.Find(column.Text, false);
        if (prop != null)
        {
            items.Add(prop.GetValue(row).ToString());
        }
    }

    // Возвращаем ListViewItem
    return new ListViewItem((string[])items.ToArray(
        typeof(string)));
}
```

```
/// <summary>
/// Удалить элемент с указанным индексом
/// </summary>
/// <param name="index">Индекс элемента</param>
private void DeleteItem(int index)
{
    if (index >= 0 && index < this.Items.Count)
        this.Items.RemoveAt(index);
}

/// <summary>
/// Изменение позиции указателя в данных
/// </summary>
private void CurrencyManager_PositionChanged(
    object sender, EventArgs e)
{
    if (this.Items.Count > CurrencyManager.Position)
    {
        this.Items[CurrencyManager.Position].Selected = true;
        this.EnsureVisible(CurrencyManager.Position);
    }
}

/// <summary>
/// Изменение указателя в ListView
/// </summary>
private void ListViewDataBinding_SelectedIndexChanged(
    object sender, EventArgs e)
{
    try
    {
        if (this.SelectedIndices.Count > 0 &&
            CurrencyManager.Position != this.SelectedIndices[0])
            CurrencyManager.Position = this.SelectedIndices[0];
    }
    catch
    {
    }
}

/// <summary>
/// Вызывается при редактировании данных в ListView
/// </summary>
```

```
protected override void OnAfterLabelEdit(LabelEditEventArgs e)
{
    base.OnAfterLabelEdit(e);

    if (e.Label == null)
    {
        // Нажата клавиша <Esc> - ничего не отредактировано
        e.CancelEdit = true;
        return;
    }

    if (CurrencyManager.List.Count > e.Item)
    {
        object row = CurrencyManager.List[e.Item];

        // В ListView можно редактировать только первую колонку
       PropertyDescriptor col =
            CurrencyManager.GetItemProperties().Find(
                this.Columns[0].Text, false);
        try
        {
            if (row != null && col != null)
            {
                col.SetValue(row, e.Label);
            }
            CurrencyManager.EndCurrentEdit();
        }
        catch (Exception ex)
        {
            // Ошибочный формат данных,
            // слишком длинная строка и т. д.
            MessageBox.Show("Ошибка:\r\n" + ex.Message,
                "Сбой редактирования",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            // Отменяем
            currencyManager.CancelCurrentEdit();
            e.CancelEdit = true;
        }
    }
}
```

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
ALFKI	Allred's Futterli	Maria Anders	Sales R	Obers 5	Berlin		12203	Germany	(30) 077	(30) 007
ANATR	Ana Trullio Em	Ana Trullio	Owner	Avda. d	Mexico		05021	Mexico	(5) 555-	(5) 555
ANTON	Angebot Moren	Antonio Moe	Owner	Matazon	Mexico		05023	Mexico	(5) 555	
AROUT	Arnaud's Hote	Thomas Ha	Sales R	120 Ha	London		W81 1DP	UK	(171) 55	(171) 55
BERGS	Berglund's snab	Christina Be	Order A	Bergvgs	Uleå		S-968 22	Sweden	0921 12	0921 12
BLAUS	Blauer See Deje	Hanna Moos	Sales R	Forststr	Munich		68366	Germany	0621 08	0621 00
BOLNP	Bolndal pers et.	Pindborg	Manager	24, plac	Strasbourg		67000	France	88 60 1	88 60 1
BOLID	Boldo Comidas	Martin Som	Owner	C/Alaq	Madrid		28009	Spain	(91) 555	(91) 555
BONAP	Bon app	Lawrence L	Owner	12 rue	Marseille		13008	France	91 24 4	91 24 4
BOTTM	Bottom-Dollar	Elizabeth Li	Account	23 Tsa	Tsawas	BC	T2F 9M4	Canada	(604) 55	(604) 55
BSEBV	B's Beverages	Victoria Ash	Sales R	Fairfield	London		EC2 5HT	UK	(171) 55	
CACTU	Cactus Comida	Patricia Sim	Sales A	Centro 3	Buenos		1010	Argentina	(1) 135-	(1) 135
CEHTC	Cerezo comarc	Rancho C	Market	Santa	Mexico		05022	Mexico	(5) 555-	(5) 555
CHOPS	Chop's suya Cha	Yang Wang	Owner	Hauptst	Bern		3012	Switzerl	045 107	

Рис. 13.12. Вид компонента BoundListView

13.5. Связь свойств между собой

В этом разделе я хочу рассказать о не совсем очевидном способе привязки одного свойства компонента к свойству другого. В принципе, ничего нового в этом нет, и реальность такого действия следует из возможностей простой привязки данных. Тем не менее прямая связь свойств между собой настолько удобна, что стоит отдельных слов.

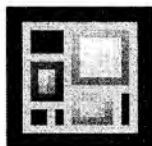
Добавим на форму два компонента — строковое поле `textBox1` и метку `label1`. Задача заключается в отображении в метке количества символов, введенных в `textBox1`. Самый очевидный вариант решения — обработка события `TextChanged`. Но есть и другой, более изящный вариант. С помощью простой привязки данных мы свяжем значение метки и длину текста в строковом поле:

```
label1.DataBindings.Add("Text", textBox1, "Text.Length");
```

В результате выполнения этого кода получится именно то, что мы хотели — текст метки будет меняться при изменении длины введенного текста!

Связь свойств между собой — очень мощный механизм, и часто его использование значительно упрощает код.

ГЛАВА 14



Распространение компонентов

Два уровня защиты: High и . . . и . . .

Как я уже говорил, компонентный подход хорош тем, что, написав компонент один раз, можно использовать его многократно. Более того, компонентом может пользоваться не только автор, но и любой другой программист (возможно, за деньги). В этой главе я расскажу, каким образом производится распространение, лицензирование и установка компонентов.

14.1. Три составляющие и позднее связывание

Создание программы заключается в написании некоторого кода, форм и т. п., создаваемых в режиме разработки. Логично предположить, что код, необходимый для работы в режиме разработки, не всегда нужен в режиме выполнения. Действительно, расположив элементы на форме, на самом деле мы создаем код, получающийся с помощью сериализации (см. главу 10). И именно этот код будет выполнен при запуске программы, а сам дизайнер в этот момент уже не нужен. То же самое рассуждение справедливо для кода компонента и кода его дизайнера.

Таким образом, код, относящийся к приложению, состоящему из компонентов, можно разделить на три составляющие: код самого приложения, код компонентов и код времени выполнения. Деление, конечно, весьма условное, но, тем не менее, последняя часть не нужна в режиме выполнения и, вообще говоря, является просто лишним грузом, увеличивающим размер приложения.

Наша задача будет состоять в том, чтобы, во-первых, отделить код, необходимый в режиме выполнения, от кода, работающего в режиме разработки.

Во-вторых, нужно разделить код самих компонентов и код приложения для того, чтобы появилась возможность распространять компоненты без исходного кода.

Сразу скажу, что при создании компонентов в этой книге я чаще всего выделял код компонента в отдельный модуль. Вполне можно было располагать класс компонента в том же проекте, где и форма, но это не лучший стиль. Таким образом, код компонента получается отделенным от кода самого приложения.

Теперь посмотрим, как мы подключали дизайнер к компоненту:

```
[DesignerAttribute(typeof(GradientLabelDesigner))]  
public partial class GradientLabel : Label  
{  
}
```

Основная проблема, связанная с таким кодом, — нельзя разделить код компонента и код дизайнера, т. к. компонент использует явное определение типа `typeof`. Если компилятор не сможет найти нужный тип, код компилироваться не будет. И здесь на помощь приходит второй конструктор атрибута `Designer`:

```
[Designer("MyControl_Designer.GradientLabelDesigner",  
          MyControl_Designer")]  
public partial class GradientLabel : Label  
{  
}
```

Привязка дизайнера к компоненту с помощью имени типа, а не самого типа, называется *поздним связыванием*. В этом случае нет необходимости держать код дизайнера рядом с кодом компонента. Прямой ссылки на тип нет, компилятор создаст код, а уже на этапе запуска дизайнера формы (не запуска программы, а именно *выполнения кода дизайнера!*) произойдет привязка указанного дизайнера к компоненту. В следующих разделах мы обсудим вопрос, где должна располагаться сборка, чтобы дизайнер Visual Studio смог ее "увидеть" и загрузить.

Теперь остается разделить компонент и код программы. Действительно, ведь я же обещал, что компоненты можно использовать в разных программах, а значит, храниться они должны независимо. Вообще говоря, нет никакой причины держать код компонента рядом с кодом программы, но... Как тогда компонент появится в палитре компонентов? Чтобы ответить на этот и предыдущий вопрос, мне нужно немного отвлечься и рассказать, где Visual Studio ищет сборки.

14.2. Где Visual Studio ищет сборки

Чтобы среда Visual Studio могла найти файл сборки, сборка должна быть расположена в одной из определенных папок:

- сборка может быть расположена в папке, где установлена сама среда Visual Studio. По умолчанию этот путь выглядит как `C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE`. Это не лучшее место для расположения сборок просто потому, что мусорить в системных папках — не очень хорошая идея;
- сборка может быть расположена в каталоге `PublicAssemblies` вышеуказанной папки. Этот каталог специально предназначается для дополнительных сборок;
- сборка может быть расположена в одном из каталогов, описанных в ключе реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\8.0\AssemblyFolders` (рис. 14.1);
- сборка может быть расположена в глобальном кэше сборок.

Глобальный кэш сборок (GAC, global assembly cache) был разработан для хранения сборок, предназначенных для общего пользования. Сборки в .NET могут иметь одинаковые названия, но разные версии, поэтому хранилище общедоступных сборок в виде простого каталога не подходит. Физически GAC представляет собой каталог на диске (обычно это скрытый каталог `C:\WINDOWS\assembly\GAC_32`), структура подкаталогов которого позволяет хранить все версии сборок (рис. 14.2). Разумеется, никто не копирует сборки в GAC вручную, для этого существуют специальные инструменты (см. разд. 14.2.2).

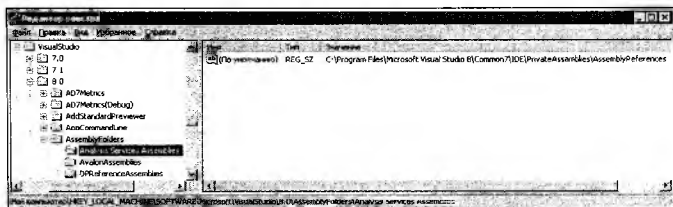


Рис. 14.1. Ключ `AssemblyFolders` описывает дополнительные каталоги для поиска сборок

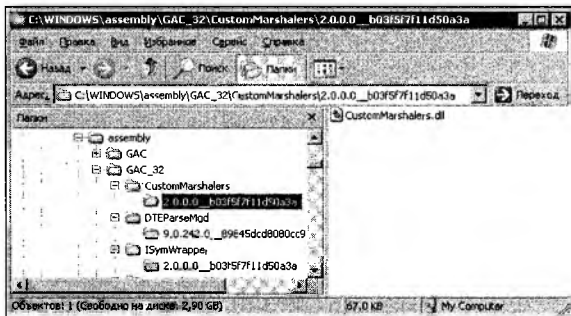


Рис. 14.2. Физическое расположение GAC

14.2.1. Подпись сборок

Как видно из рис. 14.2, глобальный кэш сборок хранит каждую версию сборки в специальном подкаталоге. Соответственно, для того чтобы сборка могла располагаться в GAC, она должна подчиняться двум правилам:

- ☐ сборка должна иметь цифровую подпись;
- ☐ сборка должна иметь версию.

Именно наличие подписи и версии позволяет строго идентифицировать сборку.

Версия сборки указывается в диалоге **Assembly Information** (рис. 14.3), который открывается при нажатии на кнопку **Assembly Information** на вкладке **Application** диалога свойств проекта.

Цифровая подпись представляет собой файл, имеющий расширение **snk**. Обычно имя файла совпадает с именем сборки, но это не обязательно. Для добавления подписи нужно выбрать вкладку **Signing** в свойствах проекта и установить флажок **Sign the assembly** (рис. 14.4). В выпадающем списке можно выбрать существующий файл цифровой подписи или создать новый (рис. 14.5).

Файл цифровой подписи можно также сгенерировать с помощью утилиты **SN.exe**, входящей в состав .NET Framework SDK.

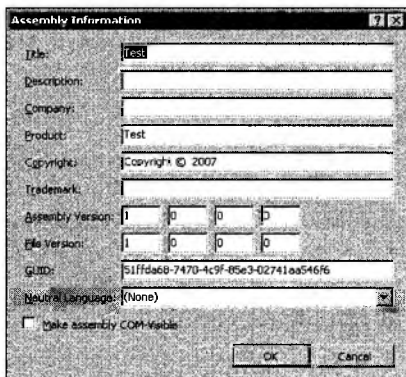


Рис. 14.3. Задание версии сборки

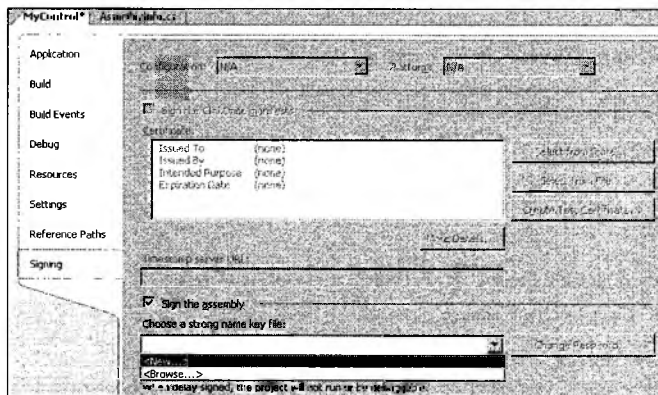


Рис. 14.4. Включение и выбор имени файла цифровой сборки

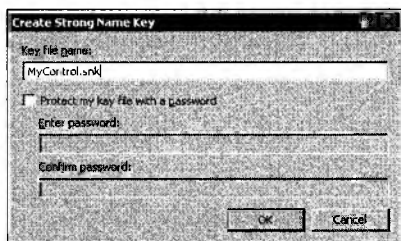


Рис. 14.5. Создание нового файла цифровой подписи

14.2.2. Установка сборок в GAC

Для установки сборок в GAC можно использовать несколько способов:

- ☐ с помощью утилиты GacUtil;
- ☐ с помощью консольной оснастки .NET Framework;
- ☐ программно.

Утилита GacUtil

Утилита GacUtil имеет множество ключей, но для установки и удаления сборок используются два ключа:

- ☐ /i <путь к сборке> — установка сборки в GAC;
- ☐ /u <отображаемое имя сборки> — удаление сборки из GAC.

Недостаток этого способа в том, что данная утилита входит в поставку .NET Framework SDK, но отсутствует в стандартном пакете .NET Framework.

Установка сборок в GAC с помощью консоли

Наиболее удобный способ для управления сборками вручную — это использование консоли. Для ее запуска нужно нажать кнопку **Пуск**, выбрать меню **Выполнить...** и в появившемся диалоге набрать **mmc**. В появившемся окне консоли нужно выбрать пункт **Консоль | Добавить или удалить оснастку** (рис. 14.6) и указать оснастку **.NET Framework 2.0 Configuration** (рис. 14.7). Управление GAC производится с помощью узла **Assembly Cache** (рис. 14.8). Меню, вызываемое по щелчку правой кнопки мыши, позволяет добавить или удалить сборки из GAC.

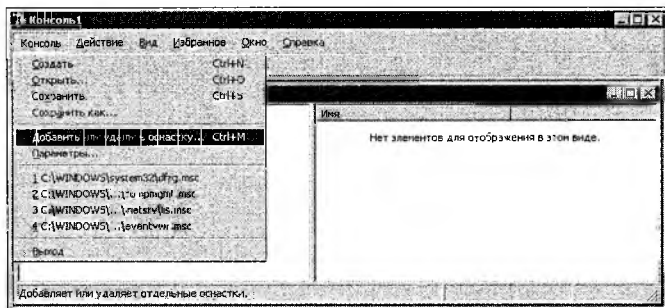


Рис. 14.6. Команда Добавить или удалить оснастку

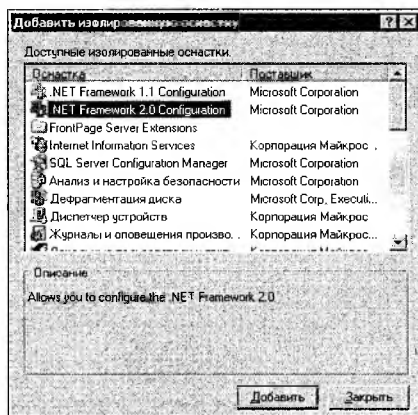


Рис. 14.7. Оснастка NET Framework Configuration

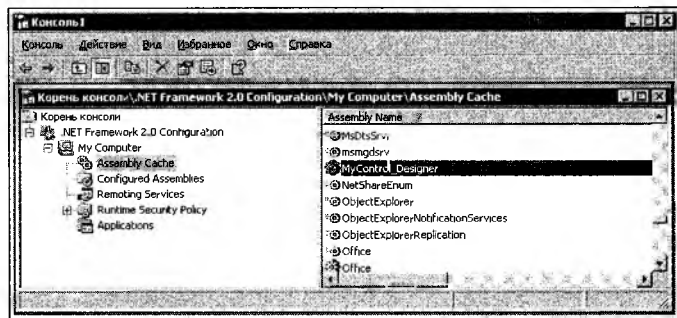


Рис. 14.8. Оснастка управления GAC

Программная установка сборок в GAC

Для программного управления сборками в GAC можно использовать методы класса `System.EnterpriseServices.Internal.Publish`. Метод `GacInstall()` устанавливает сборку в GAC, а метод `GacRemove()` удаляет ее (листинг 14.1).

Дополнительную информацию об GAC API можно найти в документации Microsoft по адресу <http://support.microsoft.com/default.aspx?scid=kb;en-us;317540>.

Листинг 14.1. Программное управление сборками в GAC

```
using System;
using System.Collections.Generic;
using System.Text;
// Для работы требуется подключить System.EnterpriseServices.dll
using System.EnterpriseServices.Internal;

namespace GacManagement
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length != 2)
                return;
        }
    }
}
```

```
publish publisher = new Publish();

switch (args[0])
{
    case "i":
        // Установка в GAC
        publisher.GacInstall(args[1]);
        break;
    case "u":
        // Удаление из GAC
        publisher.GacRemove(args[1]);
        break;
}
}
```

Работа со сборками в GAC

Хотя сборки, расположенные в GAC, доступны всем приложениям, среда Visual Studio не позволяет делать прямые ссылки на них. Если нужно как-то использовать сборку, расположенную в GAC, во время разработки, то придется скопировать ее в локальную папку и делать ссылку на локальную копию. Для сборок, требующихся только в режиме разработки, можно выставить свойство **Copy Local** в значение **False**. Такие сборки не будут копироваться в выходную папку проекта. В режиме выполнения проблем нет — сборки будут загружены из GAC.

14.3. Установка компонентов

Теперь можно вернуться к основному вопросу — установке компонентов. Добавление (и вообще управление) компонента производится с помощью пункта **Choose Items** контекстного меню (рис. 14.9). Кнопка **Browse** в появляющемся окне позволяет добавить новую DLL в палитру компонентов (рис. 14.10).

Контекстные меню **Add Tab**, **Delete Tab** и **Rename Tab** позволяют добавить, удалить и переименовать группу компонентов в палитре компонентов. С помощью этого меню я создал группу **My Controls** и зарегистрировал в ней наш компонент (рис. 14.11). Теперь он доступен в любом проекте типа **Windows Forms**, но не совсем полнофункционален, т. к. дизайнер компонента пока недоступен.

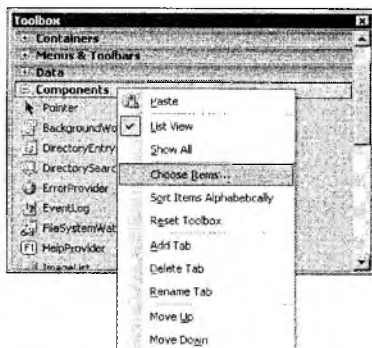


Рис. 14.9. Меню управления компонентами

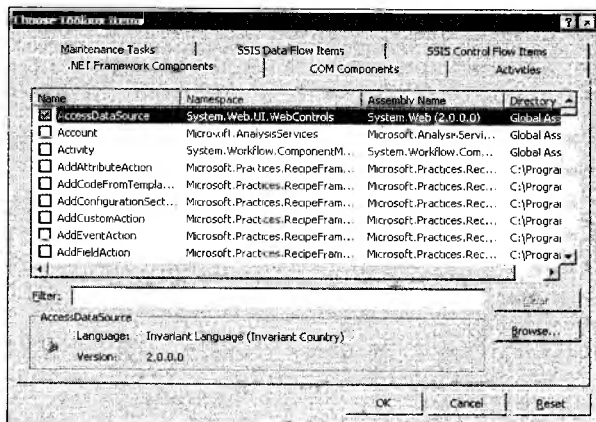


Рис. 14.10. Диалог управления компонентами

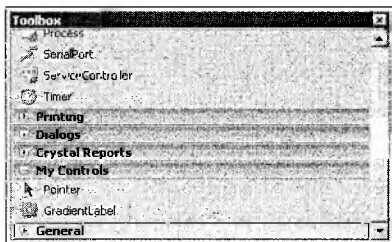


Рис. 14.11. Новый компонент в палитре компонентов

Как я уже говорил, у нас есть несколько вариантов расположения сборки дизайнера (файл `MyControl_Designer.dll`). Мне кажется, наилучшим расположением сборок дизайнера является GAC, т. к. расположив сборку в каталоге `PublicAssemblies`, мы сделаем ее доступной также из меню **Add Reference**, а это не совсем то, что нам нужно.

Итак, остановимся на таком расположении сборок (файлы и сборки проекта, в который будет добавляться компонент):

- ☐ сборка компонента добавляется в палитру компонентов с помощью меню **Choose Items** (с.м. также разд. 14.5);
- ☐ сборка, содержащая код времени разработки (дизайнеры, конвертеры и т. п.), располагается в GAC;
- ☐ сборки библиотек, не являющиеся компонентами, но, возможно, требующиеся для работы, располагаются в каталоге `PublicAssemblies`. Ссылки на них добавляются с помощью меню **Add Reference**.

Эти правила работают при разработке приложения, но не совсем удобны во время разработки самого компонента. С этим вопросом мы разберемся в разд. 14.4.

14.3.1. Программная установка компонентов

В разд. "Программная установка сборок в GAC" ранее в этой главе я описывал, как программно установить сборки в GAC. Сейчас я хочу рассказать, как программно установить компоненты в палитру компонентов. К сожалению, по непонятной мне причине в Visual Studio не существует прямого и простого способа это сделать. Как вы увидите далее, для установки компонентов используется прямое обращение к окнам среды, что нельзя назвать очень удобным способом.

Итак, сначала нам потребуется специальный класс, который выполняет такую последовательность действий:

1. Создает экземпляр среды Visual Studio.
2. Создает временный проект типа **Windows Forms** (для отображения палитры компонентов).
3. Активизирует редактор свойств.
4. Получает список категорий (вкладок).
5. Создает нужную категорию, если ее еще нет.
6. Регистрирует компонент.
7. Закрывает среду.

Код такого класса показан в листинге 14.2. Заметьте, что этот класс реализует интерфейс `IDisposable`, поэтому его использование может выглядеть следующим образом:

```
using (ComponentInstaller installer = new ComponentInstaller())
{
    installer.InstallComponent(
        имя вкладки,
        имя компонента,
        расположение сборки
    );
}
```

Расположение сборки может указываться либо полным путем к имени файла `dll`: `E:\Books\Deployment2\MyControl\bin\Debug\mycontrol.dll`, либо полным именем в `GAC`: `MyControl.GradientLabel, MyControl, Version=1.0.0.0, Culture=neutral, PublicKeyToken=bele7a1beb2652d8`.

Я не думаю, что в рамках этой книги стоит подробно рассматривать код этого класса. Просто используйте его как есть (класс `MessageFilter`, использующийся при работе с окнами среды, приведен в листинге 14.3).

Листинг 14.2. Класс `ComponentInstaller`

```
using System;
using System.Collections.Generic;
using System.Text;
using EnvDTE;
using EnvDTE80;
using System.IO;
```

```
namespace Installer
{
    public class ComponentInstaller : IDisposable
    {
        DTE dteObject;

        /// <summary>
        /// Конструктор
        /// </summary>
        public ComponentInstaller()
        {
            // Создаем экземпляр объекта Visual Studio
            dteObject = (DTE)Activator.CreateInstance(
                Type.GetTypeFromProgID("VisualStudio.DTE.8.0"), true);
            MessageFilter.Register();
        }

        /// <summary>
        /// Установка компонента
        /// </summary>
        /// <param name="tabName">Категория (имя закладки)</param>
        /// <param name="componentName">Имя компонента</param>
        /// <param name="location">Путь к сборке или имя в GAC</param>
        public void InstallComponent(string tabName,
                                     string componentName, string location)
        {
            // Создаем временный проект типа Windows Forms,
            // который потом надо будет удалить
            string tmpFile = Path.GetFileNameWithoutExtension(
                Path.GetTempFileName());
            string tmpDir = string.Format("{0}\\{1}",
                Path.GetTempPath(), tmpFile);
            Solution2 solution = dteObject.Solution as Solution2;
            string templatePath = solution.GetProjectTemplate(
                "Windows Application", "CSharp");
            Project proj = solution.AddFromTemplate(
                templatePath, tmpDir, "dummyproj", false);
            solution.Close(false);

            // Активизируем окно редактора свойств
            dteObject.ExecuteCommand("View.PropertiesWindow", "");
            Window window = dteObject.Windows.Item(
                Constants.vsWindowKindToolbox);
        }
    }
}
```

```

dteObject.Windows.Item(Constants.vsWindowKindProperties);
window.Activate();

// Получаем список категорий
ToolBoxTabs tabs = (window.Object as ToolBox).ToolBoxTabs;
dteObject.ExecuteCommand("View.PropertiesWindow", "");

// Добавляем новую вкладку, если ее еще нет
ToolBox toolbox = (EnvDTE.ToolBox>window.Object;
ToolBoxTab tab = FindOrCreateTab(tabs, tabName);
if (tab != null)
{
    tab.Activate();
    // Добавляем компонент
    tab.ToolBoxItems.Add(componentName, location,
        vsToolBoxItemFormat.
            vsToolBoxItemFormatDotNETComponent);
}
)

/// <summary>
/// Найти нужную вкладку или создать ее, если ее еще нет
/// </summary>
private ToolBoxTab FindOrCreateTab(ToolBoxTabs tabs,
                                   string tabName)
{
    if (tabs == null)
        return null;

    for (int i = 1; i <= tabs.Count; i++)
    {
        ToolBoxTab tab = tabs.Item(i);
        if (tab.Name == tabName)
        {
            return tab;
        }
    }

    return tabs.Add(tabName);
}

/// <summary>
/// Освобождение ресурсов
/// </summary>

```

```

    public void Dispose()
    {
        dteObject.Quit();
        MessageFilter.Revoke();
    }
}

```

Листинг 14.3. Класс MessageFilter

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;

namespace Installer
{
    // Определяем интерфейс IMessageFilter, который требуется
    // для использования CoRegisterMessageFilter API.
    [ComImport(), Guid("00000016-0000-0000-C000-000000000046"),
    InterfaceTypeAttribute(ComInterfaceType.InterfaceIsUnknown)]
    interface IOleMessageFilter // Интерфейс переименован,
                                // чтобы не путать
    {
        // ero c System.Windows.Forms.IMessageFilter
        [PreserveSig]
        int HandleInComingCall(int dwCallType, IntPtr hTaskCaller,
                                int dwTickCount, IntPtr lpInterfaceInfo);

        [PreserveSig]
        int RetryRejectedCall(IntPtr hTaskCallee,
                                int dwTickCount, int dwRejectType);

        [PreserveSig]
        int MessagePending(IntPtr hTaskCallee,
                                int dwTickCount, int dwPendingType);
    }

    internal class MessageFilter : IOleMessageFilter
    {
        [DllImport("Ole32.dll")]
        private static extern int CoRegisterMessageFilter(
            IOleMessageFilter newfilter,
            out IOleMessageFilter oldfilter);
        public static void Register()
        {
            IOleMessageFilter newfilter = new MessageFilter();

```

```
        IOleMessageFilter oldfilter = null;
        CoRegisterMessageFilter(newfilter, out oldfilter);
    }

    public static void Revoke()
    {
        IOleMessageFilter oldfilter = null;
        CoRegisterMessageFilter(null, out oldfilter);
    }

    int IOleMessageFilter.HandleInComingCall(
        int dwCallType,
        IntPtr hTaskCaller, int dwTickCount,
        IntPtr lpInterfaceInfo)
    {
        return 0;
    }

    int IOleMessageFilter.MessagePending(IntPtr hTaskCallee,
        int dwTickCount, int dwPendingType)
    {
        return 2;
    }

    int IOleMessageFilter.RetryRejectedCall(IntPtr hTaskCallee,
        int dwTickCount, int dwRejectType)
    {
        if (dwRejectType == 2)
        {
            return 99;
        }
        return -1;
    }
}
```

Для выполнения всех перечисленных действий мы создадим специальную консольную утилиту *Installer* (листинг 14.4), а нужные параметры вынесем в файл конфигурации (листинг 14.5).

Листинг 14.4. Консольная утилита *Installer*

```
using System;
using System.Collections.Generic;
using System.Text;
```

```

using AppManager = System.Configuration.ConfigurationManager;

namespace Installer
{
    class Program
    {
        static void Main()
        {
            using (ComponentInstaller installer = new ComponentInstaller())
            {
                installer.InstallComponent(
                    AppManager.AppSettings["TabName"],
                    AppManager.AppSettings["ComponentName"],
                    AppManager.AppSettings["Location"]
                );
            }
        }
    }
}

```

Листинг 14.5. Файл конфигурации для утилиты Installer

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="TabName" value="MyComponents" />
    <add key="ComponentName" value="MyControl" />
    <add key="Location" value="MyControl.GradientLabel,
      MyControl, Version=1.0.0.0, Culture=neutral,
      PublicKeyToken=bele7a1beb2652d8" />

    <!--add key="Location"
      value="E:\Deployment2\MyControl\bin\Debug\mycontrol.dll" /-->
  </appSettings>
</configuration>

```

В принципе необходимый минимум у нас уже есть, но использовать exe-файл для установки компонентов не очень удобно. Поэтому добавим еще один проект типа **Setup Project** (рис. 14.12). По умолчанию система позволяет производить установку файлов в три папки: Application Folder (папка приложений), User's Desktop (рабочий стол пользователя), User's Program Menu (меню пользователя). Для наших целей этого мало, поэтому нужно выбрать меню **Add Special Folder** и добавить папку Global Assembly Cache Folder

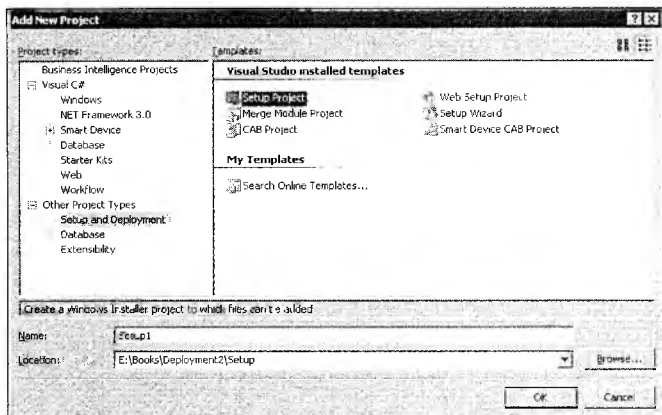


Рис. 14.12. Добавление проекта типа Setup Project

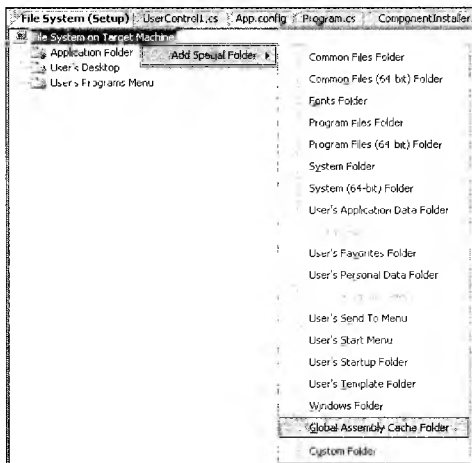


Рис. 14.13. Добавление папки GAC

(рис. 14.13). В эту папку нужно добавить результаты компиляции проектов, содержащих компонент и дизайнер компонента (рис. 14.14).

Теперь нужно отредактировать последовательность шагов инсталляции. Шаги выбора каталога нам не нужны, удаляем. Результат должен быть похож на приведенный на рис. 14.15.

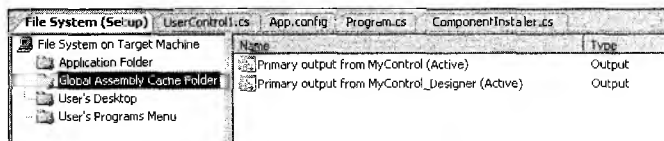


Рис. 14.14. Установка компонента и дизайнера в GAC



Рис. 14.15. Шаги инсталляции

Для того чтобы код, который мы выполняли в консольном приложении, мог быть выполнен при работе инсталлятора, он должен быть оформлен специальным образом. А именно — все действия должны выполняться в рамках метода `Commit()` класса `Installer`, который представляет собой шаг инсталляции. Реализация такого класса показана в листинге 14.6. Конфигурационный файл `app.config` также пришлось немного модифицировать (листинг 14.7). Регистрация нового шага инсталляции производится на вкладке **Custom Actions** (рис. 14.16).

Листинг 14.6. Шаг инсталляции

```
using System;
using System.Collections.Generic;
using System.ComponentModel;

using AppManager = System.Configuration.ConfigurationManager;

namespace InstallerStep
{
    [RunInstaller(true)]
    public partial class VSToolboxInstaller :
        System.Configuration.Install.Installer
    {
        public VSToolboxInstaller()
        {
        }

        public override void Commit(
            System.Collections.IDictionary savedState)
        {
            base.Commit(savedState);

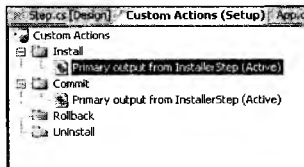
            Properties.Settings settings = Properties.Settings.Default;

            using (InstallerCore.ComponentInstaller installer =
                new InstallerCore.ComponentInstaller())
            {
                installer.InstallComponent(settings.TabName,
                    settings.ComponentName,
                    settings.Location
                );
            }
        }

        public override void Uninstall(
            System.Collections.IDictionary savedState)
        {
            base.Uninstall(savedState);
        }
    }
}
```

Листинг 14.7. Файл конфигурации для класса VSToolboxInstaller

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings"
      type="System.Configuration.ApplicationSettingsGroup, System,
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="InstallerStep.Properties.Settings"
        type="System.Configuration.ClientSettingsSection, System,
        Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        requirePermission="false" />
    </sectionGroup>
  </configSections>
  <applicationSettings>
    <InstallerStep.Properties.Settings>
      <setting name="TabName" serializeAs="String">
        <value>MyComponents</value>
      </setting>
      <setting name="ComponentName" serializeAs="String">
        <value>MyControl</value>
      </setting>
      <setting name="Location" serializeAs="String">
        <value>MyControl.GradientLabel, MyControl, Version=1.0.0.0,
        Culture=neutral, PublicKeyToken=bele7albeb2652d8</value>
      </setting>
    </InstallerStep.Properties.Settings>
  </applicationSettings>
</configuration>
```

**Рис. 14.16.** Регистрация нового шага инсталляции

Результатом компиляции проекта являются два файла: setup.exe и setup.msi. Для старта инсталляции можно запускать любой из них (рис. 14.17).

Аналогично можно реализовать удаление компонентов при запуске процедуры uninstall.

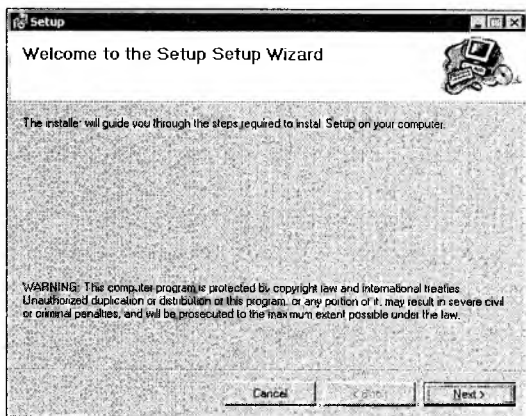


Рис. 14.17. Установка компонента

14.4. Процедура разработки

Правила расположения сборок, которые я описал ранее, не совсем годятся при разработке самого компонента. Дело в том, что загружая сборку в память, Visual Studio блокирует ее и, соответственно, не дает обновить. Получается, что при изменении в коде компонента придется выгружать среду разработки, обновлять сборки и запускать ее снова. Конечно, такой алгоритм неприемлем при разработке компонента. Но ведь когда я описывал создание компонентов, мы использовали прямое связывание типов. И проблем у нас не было!

Итак, у нас получаются два состояния кода:

- ☐ при разработке компонента лучше использовать прямое указание типа и, соответственно, разрабатывать дизайнер компонента и сам компонент в рамках одного проекта;
- ☐ после завершения разработки код компонента отделяется от кода дизайнера компонента, и сборки располагаются согласно правилам из предыдущего раздела.

При таком подходе у нас не будет проблем ни при разработке, ни при использовании компонента.

14.5. Лицензирование

Кроме повторного использования кода, цель разработки компонентов может заключаться в предоставлении компонентов другим разработчикам. Иными словами, в продаже компонентов. Лицензирование позволяет защитить компоненты от нелегального распространения. В этом разделе я хочу рассказать о средствах создания лицензирования, предоставляемых библиотекой .NET Framework.

Прежде чем начать описание библиотеки и классов, нужно сделать очень важное замечание. Классы, о которых я буду рассказывать, не осуществляют *реальную* защиту компонентов! Они только лишь предоставляют *механизм* реализации защиты.

14.5.1. Провайдер лицензий

Библиотека .NET Framework разделяет логику проверки лицензии от кода самого компонента с помощью провайдера лицензий (license provider), представленного классом `System.ComponentModel.LicenseProvider`.

Класс `LicenseProvider` является базовым абстрактным классом для всех провайдеров лицензий. Единственным его методом является метод `GetLicense()`, возвращающий экземпляр класса `License`:

```
public abstract License GetLicense(LicenseContext context,  
                                   Type type, object instance,  
                                   bool allowExceptions);
```

Первый параметр этого метода, `context`, предоставляет сервис для работы с лицензиями (о нем я расскажу чуть позже), второй параметр передает тип лицензируемого компонента, а третий параметр — экземпляр самого компонента. Последний параметр разрешает генерацию исключений, если файл лицензии некорректен.

14.5.2. Класс `LicFileLicenseProvider`

Класс `LicFileLicenseProvider` реализует простейшую защиту: если в каталоге, где расположен запускаемый файл, присутствует файл специального формата, лицензия считается корректной. Формат файла очень простой:

- имя файла должно быть в формате

`<пространство_имен>.<тип_компонента>.lic`

- файл должен содержать строку

`<пространство_имен>.<тип_компонента> is a licensed component.`

Например, для класса `MyControl.GradientLabel` файл должен называться `MyControl.GradientLabel.lic` и содержать строку `MyControl.GradientLabel is a licensed component.`

Разумеется, назвать это защитой сложно, но этот класс дает возможность реализовать собственный механизм защиты, перекрыв виртуальный метод `protected virtual bool IsKeyValid(string key, Type type);`

В реализации наследника можно (и нужно) использовать более сложный механизм, чем простое сравнение строк.

14.5.3. Подключение провайдера лицензий к компоненту

Подключение провайдера лицензий к компоненту производится с помощью атрибута `LicenseProvider`:

```
[LicenseProvider(typeof(LicFileLicenseProvider))]  
public partial class GradientLabel : Label  
{  
}
```

Само по себе определение атрибута еще не означает включение механизма проверки лицензии. Это просто определение провайдера лицензий. Для проверки правильности лицензии используется вызов метода `Validate()` класса `LicenseManager`:

```
License license = LicenseManager.Validate(this.GetType(), this);
```

О классах `License` и `LicenseManager` я расскажу в следующих разделах.

14.5.4. Класс *License*

Класс `License` является абстрактным базовым классом для лицензий. Он имеет всего одно абстрактное свойство и один метод:

```
public abstract string LicenseKey { get; }  
public abstract void Dispose();
```

Свойство `LicenseKey` возвращает строку лицензии (ключ), относящуюся к лицензируемому компоненту. В качестве ключа может быть любая строка. Разумеется, не имеет смысла возвращать лицензионную информацию открытым текстом.

Метод `Dispose()` (класс `License` реализует интерфейс `IDisposable`) должен освобождать все занятые ресурсы, если класс лицензии больше не нужен.

14.5.5. Класс *LicenseContext*

Теперь вернемся к классу `LicenseContext`, о котором я обещал рассказать в начале этого раздела. Этот класс имеет следующее описание:

```
public class LicenseContext : IServiceProvider
{
    public LicenseContext();
    public virtual LicenseUsageMode UsageMode { get; }
    public virtual string GetSavedLicenseKey(
        Type type, Assembly resourceAssembly);
    public virtual object GetService(Type type);
    public virtual void SetSavedLicenseKey(Type type, string key);
}
```

Виртуальное свойство `UsageMode` возвращает режим, в котором может использоваться лицензия. Режим задается перечислением `LicenseUsageMode`, которое содержит два значения:

```
public enum LicenseUsageMode
{
    Runtime      = 0,
    DesignTime   = 1,
}
```

Как не трудно догадаться из названия, `Runtime` означает, что лицензия может использоваться в режиме выполнения, а `DesignTime` означает возможность использования лицензии в режиме разработки. По умолчанию возвращается значение `Runtime`.

Виртуальный метод `GetSavedLicenseKey()` возвращает значение ключа лицензии для указанного типа компонента, сохраненное как встроенный ресурс в указанной сборке. По умолчанию метод возвращает `null`. Виртуальный метод `SetSavedLicenseKey()` задает лицензионный ключ.

Метод `GetService()` предоставляет доступ к сервисам (см. главу 7).

14.5.6. Класс *LicenseManager*

Класс `LicenseManager` предоставляет набор статических методов для работы с лицензиями:

```
public sealed class LicenseManager
{
    public static LicenseContext CurrentContext { get; set; }
    public static LicenseUsageMode UsageMode { get; }
```

```
public static object CreateWithContext(Type type,
    LicenseContext creationContext);
public static object CreateWithContext(Type type,
    LicenseContext creationContext, object[] args);
public static bool IsLicensed(Type type);
public static bool IsValid(Type type);
public static bool IsValid(Type type,
    object instance, out License license);
public static void Validate(Type type);
public static License Validate(Type type, object instance);
public static void LockContext(object contextUser);
public static void UnlockContext(object contextUser);
}
```

Свойство `CurrentContext` позволяет получить или установить контекст лицензии `LicenseContext` (см. разд. 14.5.5).

Свойство `UsageMode` возвращает режим, в котором может использоваться текущая лицензия (см. разд. 14.5.5).

Две реализации метода `CreateWithContext()` используются для создания лицензии указанного типа компонента (параметр `type`) и контекста (параметр `creationContext`). С помощью третьего параметра можно указать дополнительные параметры создания объекта.

Метод `IsLicensed()` возвращает `true`, если объект указанного типа имеет связанную лицензию.

Методы `IsValid()` возвращают `true`, если лицензия корректна. Первая реализация проверяет лицензию по типу (параметр `type`), а вторая — для указанного экземпляра компонента, возвращая при этом ссылку на лицензию или `null`, если лицензия некорректна. Если метод вернул объект лицензии, то программист должен вызвать метод `Dispose()` этого объекта после завершения использования. Метод возвращает `false`, если компонент не имеет лицензии или лицензия некорректна.

Методы `Validate()` аналогичны методам `IsValid()`.

Методы `LockContext()` и `UnlockContext()` позволяют заблокировать доступ к лицензии и разблокировать его. Эти методы используются при синхронизации работы с лицензиями из нескольких потоков.

14.5.7. Реализация собственного алгоритма лицензирования

Разумеется, проверка лицензионности с помощью текстовой строки известного содержания, как это делает класс `LicFileLicenseProvider`, — не реше-

ние. Зато, с помощью этого класса, можно легко реализовать собственный механизм проверки лицензий.

Например, достаточно перекрыть метод `IsValid()`, чтобы реализовать другой алгоритм контроля:

```
class MyLicenseProvider : LicFileLicenseProvider
{
    protected override bool IsValid(string key, Type type)
    {
        return string.Format("MyLicenseProvider: {0}",
                               type.ToString()).Equals(key);
    }
}
```

Теперь строка корректной лицензионной информации будет выглядеть так:

```
MyLicenseProvider: MyControl.GradientLabel
```

Реализация собственного алгоритма организации этой строки может включать разные алгоритмы защиты. Например, в ней может быть зашифрована дата, до которой работает лицензия, режим работы (разработки или выполнения) или какая-либо другая информация. Вообще говоря, сюда же может быть включена некоторая информация, идентифицирующая компьютер или пользователя. В качестве примера в следующем разделе я покажу, как получить информацию, идентифицирующую жесткий диск и BIOS компьютера.

14.5.8. Получение уникальной информации о компьютере

Уникальная информация о компьютере пользователя может быть использована для генерации ключа, который будет работать только на этом компьютере. По моему мнению, привязка ключа к компьютеру — не самый хороший путь, но для тех, кто считает иначе, информация этого раздела может пригодиться.

Для получения информации о характеристиках компьютера можно использовать данные реестра, которые можно получить, как показано в листинге 14.8.

Информацию о производителе компьютера можно получить с помощью методов WMI (листинг 14.9). Следует только помнить, что эти методы требуют администраторских прав на компьютере. Код программ, получающих другие характеристики с помощью методов WMI, можно найти в [4].

Листинг 14.8. Получение характеристик компьютера из реестра

```

using System;
using Microsoft.Win32;
namespace ReadRegistry
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            RegistryKey hkml = Registry.LocalMachine;
            hkml=hkml.OpenSubKey("HARDWARE\\DESCRIPTION\\System\\
                                CentralProcessor\\0");
            Object obp=hkml.GetValue("Identifier");
            Console.WriteLine("Processor Identifier :{0}",obp);
            RegistryKey hklp =Registry.LocalMachine;
            hklp=hklp.OpenSubKey("HARDWARE\\DESCRIPTION\\System\\
                                CentralProcessor\\0");
            Object obc=hklp.GetValue("VendorIdentifier");
            Console.WriteLine("Vendor Identifier   :{0}",obc);
            RegistryKey biosv =Registry.LocalMachine;
            biosv=biosv.OpenSubKey("HARDWARE\\DESCRIPTION\\System\\
                                MultiFunctionAdapter\\4");
            Object obv=biosv.GetValue("Identifier");
            Console.WriteLine("Bios Status           :{0}",obv);
            RegistryKey biosd =Registry.LocalMachine;
            biosd=biosd.OpenSubKey("HARDWARE\\DESCRIPTION\\System\\");
            Object obd=biosd.GetValue("SystemBiosDate");
            Console.WriteLine("Bios Date             :{0}",obd);
            RegistryKey bios =Registry.LocalMachine;
            bios=bios.OpenSubKey("HARDWARE\\DESCRIPTION\\System\\");
            Object obs=bios.GetValue("Identifier");
            Console.WriteLine("System Identifier    :{0}",obs);
        }
    }
}

```

Листинг 14.9. Получение характеристик компьютера с помощью WMI

```

using System;
using System.Management;

```

```
class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        WqlObjectQuery query = new WqlObjectQuery(
            "SELECT * FROM Win32_ComputerSystemProduct");
        ManagementObjectSearcher find =
            new ManagementObjectSearcher(query);
        foreach (ManagementObject mo in find.Get())
        {
            Console.WriteLine("Description." + mo["Description"]);
            Console.WriteLine("Identifying number (usually
                serial number)." + mo["IdentifyingNumber"]);
            Console.WriteLine("Commonly used product name." + mo["Name"]);
            Console.WriteLine("Universally Unique Identifier of
                product." + mo["UUID"]);
            Console.WriteLine("Vendor of product." + mo["Vendor"]);
        }
    }
}
```

Код, показанный в листинге 14.10, возвращает серийный номер диска. Правда, этот номер можно задать при форматировании диска, но и такой способ проверки может пригодиться:

```
VolumeInfo gv = new VolumeInfo();
string serialNumber = gv.GetVolumeSerial('C');
```

Заводской серийный номер можно прочесть только с помощью метода `DeviceIoControl()`, который позволяет передавать команды напрямую драйверу жесткого диска. Соответствующий код показан в листинге 14.11. Код работает только в Windows NT/2000 (подробный разбор листинга не входит в задачи этой книги). Вот как его можно использовать:

```
HardDiskInfo hdd = AtapiDevice.GetHddInfo(0);
string serialNumber = hdd.SerialNumber;
```

С помощью этих и многих других методов можно реализовать необходимый уровень лицензирования компонентов.

Листинг 14.10. Получение серийного номера логического диска

```
using System;
using System.Text;
using System.Runtime.InteropServices;
```

```

namespace Volume
{
    public class VolumeInfo
    {
        [DllImport("kernel32.dll")]
        private static extern long GetVolumeInformation(
            string PathName,
            StringBuilder VolumeNameBuffer,
            UInt32 VolumeNameSize,
            ref UInt32 VolumeSerialNumber,
            ref UInt32 MaximumComponentLength,
            ref UInt32 FileSystemFlags,
            StringBuilder FileSystemNameBuffer,
            UInt32 FileSystemNameSize);

        /// <summary>
        /// Возвращает серийный номер указанного диска
        /// </summary>
        /// <param name="strDriveLetter">Буква диска</param>
        public string GetVolumeSerial(char driveLetter)
        {
            string driveName = new string(driveLetter, 1);

            uint serialNumber = 0;
            uint maxCompLen = 0;
            StringBuilder volumeLabel = new StringBuilder(256);
            UInt32 volumeFlags = new UInt32();
            StringBuilder fileName = new StringBuilder(256);
            driveName += ":\\";

            GetVolumeInformation(driveName,
                volumeLabel, (UInt32)volumeLabel.Capacity,
                ref serialNumber, ref maxCompLen, ref volumeFlags,
                fileName, (UInt32)fileName.Capacity);

            return Convert.ToString(serialNumber);
        }
    }
}

```

Листинг 14.11. Получение заводского серийного номера диска

```

using System;
using System.Runtime.InteropServices;
using System.Text;

```

```
namespace HardwareVolumeInfo
{
    [Serializable]
    public struct HardDiskInfo
    {
        public string ModuleNumber;
        public string Firmware;
        public string SerialNumber;
        public uint Capacity;
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1)]
    internal struct GetVersionOutParams
    {
        public byte bVersion;
        public byte bRevision;
        public byte bReserved;
        public byte bIDEDeviceMap;
        public uint fCapabilities;
        [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
        public uint[] dwReserved;
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1)]
    internal struct IdeRegs
    {
        public byte bFeaturesReg;
        public byte bSectorCountReg;
        public byte bSectorNumberReg;
        public byte bCylLowReg;
        public byte bCylHighReg;
        public byte bDriveHeadReg;
        public byte bCommandReg;
        public byte bReserved;
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1)]
    internal struct SendCmdInParams
    {
        public uint cBufferSize;
        public IdeRegs irDriveRegs;
        public byte bDriveNumber;
        [MarshalAs(UnmanagedType.ByValArray, SizeConst = 3)]
```

```

    public byte[] bReserved;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
    public uint[] dwReserved;
    public byte bBuffer;
}

[StructLayout(LayoutKind.Sequential, Pack = 1)]
internal struct DriverStatus
{
    public byte bDriverError;
    public byte bIDStatus;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 2)]
    public byte[] bReserved;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 2)]
    public uint[] dwReserved;
}

[StructLayout(LayoutKind.Sequential, Pack = 1)]
internal struct SendCmdOutParams
{
    public uint cBufferSize;
    public DriverStatus DriverStatus;
    public IdSector bBuffer;
}

[StructLayout(LayoutKind.Sequential, Pack = 1, Size = 512)]
internal struct IdSector
{
    public ushort wGenConfig;
    public ushort wNumCyls;
    public ushort wReserved;
    public ushort wNumHeads;
    public ushort wBytesPerTrack;
    public ushort wBytesPerSector;
    public ushort wSectorsPerTrack;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 3)]
    public ushort[] wVendorUnique;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 20)]
    public byte[] sSerialNumber;
    public ushort wBufferType;
    public ushort wBufferSize;
    public ushort wECCSize;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 8)]

```

```

public byte[] sFirmwareRev;
[MarshalAs(UnmanagedType.ByValArray, SizeConst = 40)]
public byte[] sModelNumber;
public ushort wMoreVendorUnique;
public ushort wDoubleWordIO;
public ushort wCapabilities;
public ushort wReserved1;
public ushort wPIOTiming;
public ushort wDMATiming;
public ushort wBS;
public ushort wNumCurrentCyls;
public ushort wNumCurrentHeads;
public ushort wNumCurrentSectorsPerTrack;
public uint ulCurrentSectorCapacity;
public ushort wMultSectorStuff;
public uint ulTotalAddressableSectors;
public ushort wSingleWordDMA;
public ushort wMultiWordDMA;
[MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
public byte[] bReserved;

```

```

public class AtapiDevice
{
    [DllImport("kernel32.dll", SetLastError = true)]
    static extern int CloseHandle(IntPtr hObject);

    [DllImport("kernel32.dll", SetLastError = true)]
    static extern IntPtr CreateFile(
        string lpFileName,
        uint dwDesiredAccess,
        uint dwShareMode,
        IntPtr lpSecurityAttributes,
        uint dwCreationDisposition,
        uint dwFlagsAndAttributes,
        IntPtr hTemplateFile);

    [DllImport("kernel32.dll")]
    static extern int DeviceIoControl(
        IntPtr hDevice,
        uint dwIoControlCode,
        IntPtr lpInBuffer,
        uint nInBufferSize,

```

```

    ref GetVersionOutParams lpOutBuffer,
    uint nOutBufferSize,
    ref uint lpBytesReturned,
    [Out] IntPtr lpOverlapped);

[DllImport("kernel32.dll")]
static extern int DeviceIoControl(
    IntPtr hDevice,
    uint dwIoControlCode,
    ref SendCmdInParams lpInBuffer,
    uint nInBufferSize,
    ref SendCmdOutParams lpOutBuffer,
    uint nOutBufferSize,
    ref uint lpBytesReturned,
    [Out] IntPtr lpOverlapped);

const uint DFP_GET_VERSION = 0x00074080;
const uint DFP_SEND_DRIVE_COMMAND = 0x0007c084;
const uint DFP_RECEIVE_DRIVE_DATA = 0x0007c088;

const uint GENERIC_READ = 0x80000000;
const uint GENERIC_WRITE = 0x40000000;
const uint FILE_SHARE_READ = 0x00000001;
const uint FILE_SHARE_WRITE = 0x00000002;
const uint CREATE_NEW = 1;
const uint OPEN_EXISTING = 3;

public static HardDiskInfo GetHddInfo(byte driveIndex)
{
    GetVersionOutParams vers = new GetVersionOutParams();
    SendCmdInParams inParam = new SendCmdInParams();
    SendCmdOutParams outParam = new SendCmdOutParams();
    uint bytesReturned = 0;

    // Код работает только в Windows NT/2000
    IntPtr hDevice = CreateFile(
        string.Format(@"\\.\PhysicalDrive{0}", driveIndex),
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        IntPtr.Zero,
        OPEN_EXISTING,
        0,
        IntPtr.Zero);

```

```
if (hDevice == IntPtr.Zero)
{
    throw new Exception("Ошибка выполнения CreateFile.");
}
if (0 == DeviceIoControl(
    hDevice,
    DFP_GET_VERSION,
    IntPtr.Zero,
    0,
    ref vers,
    (uint)Marshal.SizeOf(vers),
    ref bytesReturned,
    IntPtr.Zero))
{
    CloseHandle(hDevice);
    throw new Exception(string.Format(
        "Диск {0} не существует.", driveIndex + 1));
}
if (0 == (vers.fCapabilities & 1))
{
    CloseHandle(hDevice);
    throw new Exception("Ошибка: IDE-команды не поддерживаются.");
}

if (0 != (driveIndex & 1))
{
    inParam.irDriveRegs.bDriveHeadReg = 0xb0;
}
else
{
    inParam.irDriveRegs.bDriveHeadReg = 0xa0;
}
if (0 != (vers.fCapabilities & (16 >> driveIndex)))
{
    CloseHandle(hDevice);
    throw new Exception(string.Format(
        "Диск {0} является ATAPI-устройством.",
        driveIndex + 1));
}
else
{
    inParam.irDriveRegs.bCommandReg = 0xec;
}
```



```
inParam.bDriveNumber = driveIndex;
inParam.irDriveRegs.bSectorCountReg = 1;
inParam.irDriveRegs.bSectorNumberReg = 1;
inParam.cBufferSize = 512;

if (0 == DeviceIoControl(
    hDevice,
    DFP_RECEIVE_DRIVE_DATA,
    ref inParam,
    (uint)Marshal.SizeOf(inParam),
    ref outParam,
    (uint)Marshal.SizeOf(outParam),
    ref bytesReturned,
    IntPtr.Zero))
{
    CloseHandle(hDevice);
    throw new Exception(
        "Ошибка DeviceIoControl: DFP_RECEIVE_DRIVE_DATA");
}
CloseHandle(hDevice);

return GetHardDiskInfo(outParam.bBuffer);
}

private static HardDiskInfo GetHardDiskInfo(IdSector phdinfo)
{
    HardDiskInfo hddInfo = new HardDiskInfo();

    ChangeByteOrder(phdinfo.sModelNumber);
    hddInfo.ModuleNumber = Encoding.ASCII.GetString(
        phdinfo.sModelNumber).Trim();

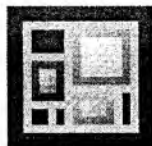
    ChangeByteOrder(phdinfo.sFirmwareRev);
    hddInfo.Firmware = Encoding.ASCII.GetString(
        phdinfo.sFirmwareRev).Trim();

    ChangeByteOrder(phdinfo.sSerialNumber);
    hddInfo.SerialNumber = Encoding.ASCII.GetString(
        phdinfo.sSerialNumber).Trim();

    hddInfo.Capacity = phdinfo.ulTotalAddressableSectors / 2 / 1024;
    return hddInfo;
}
```

```
private static void ChangeByteOrder(byte[] charArray)
{
    byte temp;
    for (int i = 0; i < charArray.Length; i += 2)
    {
        temp = charArray[i];
        charArray[i] = charArray[i + 1];
        charArray[i + 1] = temp;
    }
}
```

ГЛАВА 15



Атрибуты времени разработки

Это нормальные люди деревня сажают, а программисты их сначала строят, а потом обходят

В этой главе я хочу обобщить информацию об атрибутах, которые используются в режиме разработки (табл. 15.1), а также добавить еще несколько атрибутов, которые имеют довольно несущественную область применения, и поэтому я не описывал их в других главах.

Таблица 15.1. Атрибуты времени разработки

Название свойства	Описание	Область применения	Раздел
AmbientValue	Позволяет указать значение свойства, при котором нет смысла сохранять его реальное значение, т. к. оно берется из некоторого другого источника	Свойство	5.3
AttributeProvider	Позволяет указать набор атрибутов свойства, полученных от другого типа	Свойство	5.4
Browsable(bool)	Запрещает отображать свойство в редакторе свойств. Это может быть полезно, если свойство нельзя настроить во время разработки, или если свойство может быть настроено другим (более простым) путем. Например, сложное свойство можно скрыть, а вместо него добавить несколько виртуальных свойств простых типов	Свойство, событие	5.4

Таблица 15.1 (продолжение)

Название свойства	Описание	Область применения	Раздел
Bindable(bool)	Обозначает свойство, обычно используемое для привязки данных. Указание значения атрибута false не означает невозможность привязки данных, а только лишь, что это свойство не будет показываться в стандартных диалогах привязки данных	Свойство	Нет
Category(string categoryName)	Позволяет задать категорию, к которой относится свойство или событие. Редактор свойств имеет два режима отображения: по алфавиту и по категориям. Во втором режиме он группирует свойства и события по категориям	Свойство, событие	5.2
ComplexBindingProperties	Обозначает имя свойства, ответственного за привязку данных	Класс	13.4
DefaultProperty(string name)	Указывает свойство, которое редактор свойств должен активизировать по умолчанию	Класс	5.3
DefaultEvent(string name)	Указывает событие, которое редактор свойств должен активизировать по умолчанию. Обработчик этого события также будет автоматически создаваться при двойном щелчке по компоненту (если не переопределено поведение действия, предпринимаемого по умолчанию)	Класс	6.10
DefaultValue	Позволяет определить для свойства значение по умолчанию. Если этот атрибут задан, редактор свойств позволяет пользователю сбросить значение свойства в исходное состояние с помощью пункта Reset	Свойство	5.3
Description	Описание свойства или события. Редактор свойств выводит его в своей нижней части	Свойство, событие	5.2
Designer	Позволяет задать дизайнер для компонента	Класс	6.3
DesignerSerializer	Позволяет привязать сериализатор к компоненту	Класс	10.8
DesignerSerializationVisibility	Позволяет управлять сериализацией свойства	Свойство	10.5

Таблица 15.1 (продолжение)

Название свойства	Описание	Область применения	Раздел
DesignOnly	Указывает, что значение свойства может быть задано только в режиме разработки	Свойство	Нет
DisplayName	Задаёт имя свойства, которое будет отображено в редакторе свойств	Свойство, событие	5.4
DesignTimeVisible	Указывает видимость компонента в палитре компонентов	Класс	5.6
Editor	Позволяет назначить редактор для свойства или целого типа	Класс, свойство	9.1
EditorBrowsable	Указывает, будет ли объект (класс, свойство, событие) виден в среде Visual Studio: - EditorBrowsableState.Never — никогда не виден; - EditorBrowsableState.Always — всегда виден (по умолчанию), - EditorBrowsableState.Advanced — виден "продвинутым" разработчикам (т.е. C# и C++)	К чему угодно	Нет
LocalizableAttribute	Указывает, что свойство может быть локализовано. Любые свойства, имеющие этот атрибут, автоматически сохраняются в ресурсном файле, когда пользователь решает локализовать форму	Свойства	10.6
MergableProperty	Если выбрано более одного компонента, редактор свойств отображает (и, соответственно, даёт возможность изменить) только те свойства, для которых данный атрибут установлен в true	Свойство	Нет
NotifyParentProperty (bool)	Означает, что при изменении свойства нужно уведомить "вышестоящий" компонент	Свойство	8.1
PropertyTab	Позволяет задавать имя страницы редактора свойств, на которой будет отображено свойство. Можно добавлять новые страницы	Свойство	12.3
ParenthesizePropertyName	Отображать название свойства в круглых скобках	Свойство, событие	5.4

Таблица 15.1 (окончание)

Название свойства	Описание	Область применения	Раздел
PasswordPropertyText	Указывает, что текстовое поле, представленное таким свойством, будет скрыто в редакторе свойств звездочками, т. к. представляет собой поле ввода пароля и не может быть показано в открытом виде	Свойство	5.4
ReadOnly	Если этот атрибут установлен, свойство будет невозможно изменить во время разработки. Редактор свойств отображает такие свойства серым цветом	Свойство	5.4
RefreshProperties	Применяется, если от значения некоторого свойства зависят другие свойства и редактор свойств должен обновить их значения	Свойство	8.8
RunInstallerAttribute(bool)	Указывает, что класс будет использоваться при выполнении установки	Класс	14.3.1
SettingsBindable(bool)	Указывает, что свойство рекомендуется к вынесению в настройки программы	Свойство	10.7
TypeConverter	Задаёт класс-конвертер, используемый для преобразования одного типа в другой	Класс	8.2
ToolboxBitmap	Позволяет задать пиктограмму для компонента	Класс	5.6

ГЛАВА 16

FAQ

Прозаик пишет обдуманное, лирик — пережитое,
программист — прочитанное.

В этой главе я постарался собрать наиболее часто задаваемые вопросы. Этой главой можно пользоваться и как указателем на разделы книги (визуальный указатель представлен на рис. 16.1—16.3) для быстрого поиска нужной информации.

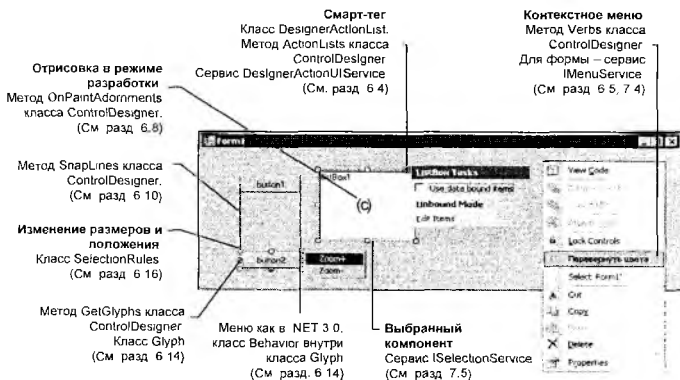


Рис. 16.1

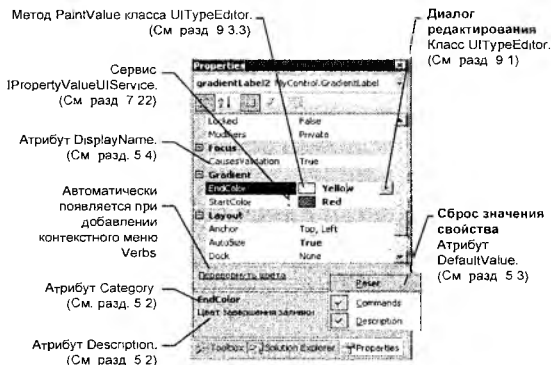


Рис. 16.2

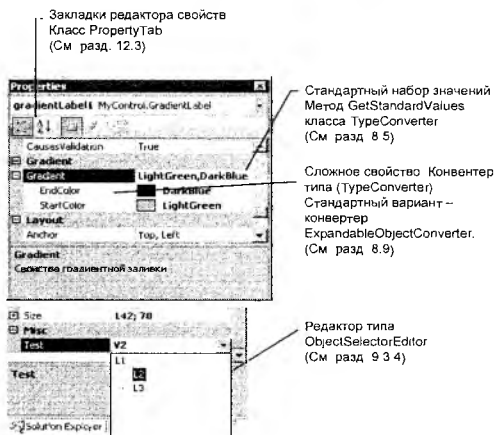


Рис. 16.3

16.1. Общие определения

Что такое палитра компонентов?

Окно **Toolbox** (Инструменты) во время редактирования формы превращается в набор компонентов. Название окна при этом никак не меняется, поэтому для удобства я называю представление этого окна *палитрой компонентов* (см. разд. 2.2).

Что такое редактор свойств?

Редактор свойств — окно **Properties** (Свойства) редактирования свойств и событий (см. разд. 2.4).

Что такое режим разработки?

Компоненты имеют две "стороны жизни" (см. разд. 4.1) — во время разработки формы в Visual Studio (design time) и во время выполнения готовой программы (run-time).

16.2. Режим разработки

Как определить, что компонент находится в режиме разработки?

Свойство `DesignMode` класса `Component` возвращает `true`, если компонент находится в режиме разработки. Это свойство нельзя использовать в конструкторе компонента (см. разд. 4.6).

Почему свойство `DesignMode` не работает в конструкторе?

Процесс создания компонента происходит в такой последовательности: сначала вызывается конструктор компонента, а затем компонент связывается с хостом (формой). Свойство `DesignMode` возвращает `true`, если свойство `Site` возвращает непустое значение. Таким образом, `DesignMode` в конструкторе всегда будет возвращать `false`.

Как проверить режим разработки в конструкторе?

Если по какой-то причине избежать проверки режима разработки в конструкторе не удастся (не могу придумать почему), можно использовать один из способов, указанных в разд. 4.6.

Как отлаживать компонент в режиме разработки?

См. разд. 4.10.

Как отобразить некоторые "тестовые" данные только в режиме разработки?

Часто компонент в режиме разработки выглядит значительно красивее, если содержит хоть какие-нибудь данные. Например, пустой `ListBox` выглядит простым прямоугольником. Добавление в него нескольких строк "оживляет" вид компонента (см. *разд. 6.14*). Конечно же нельзя добавить такие строки в конструкторе, т. к. они появятся и в режиме выполнения. Наиболее правильный способ — использовать метод `set` свойства `ISite` (см. *разд. 4.3*):

```
public class ListBoxControl : UserControl
{
    private ListBox listBox1;

    public override ISite Site
    {
        get
        {
            return base.Site;
        }
        set
        {
            base.Site = value;

            // Заполняем ListBox
            for (int i = 0; i < 100; i++)
                listBox1.Items.Add(i.ToString());
        }
    }
}
```

Как реализовать собственный дизайнер формы?

Код реализации дизайнера формы можно найти в MSDN по адресу:

<http://www.microsoft.com/Rus/Msdn/Magazine/2004/12/ApplicationCreator.mspx>.

Как компонент `ToolTip` добавляет свойства ко всем компонентам одновременно?

Добавление свойств ко всем компонентам осуществляется с помощью сервиса `IExtenderProviderService`. См. *разд. 12.1*.

Как сделать `public`-свойство невидимым в редакторе свойств?

По умолчанию все `public`-свойства показываются в редакторе свойств, что не всегда нужно. Отключить показ свойства помогает атрибут `Browsable` (см. *разд. 5.4*).

Как скопировать атрибуты одного свойства в другое?

Атрибут `AttributeProvider` позволяет указать набор атрибутов свойства, полученных от другого типа (см. разд. 5.4).

Как отобразить значок компонента?

Для указания значка компонента используются атрибуты `ToolboxItem` и `ToolboxBitmap` (см. разд. 5.6).

Где можно хранить дополнительную информацию во время разработки?

Сервис `IDictionaryServiceDesigner` предоставляет доступ к хэш-таблице данных компонента, доступных только в режиме разработки. Эта таблица хранит соответствие ключа (key) и значения (value). Ее можно использовать для хранения нужной информации (см. разд. 7.12). Причем сама форма тоже является компонентом, поэтому она тоже имеет такую таблицу, и ее можно использовать для буферизации общих данных компонентов.

Как получить информацию о проекте, которому принадлежит форма и компонент?

Для получения информации о проекте можно использовать класс `ProjectItem`. Получить его экземпляр можно недокументированным вызовом метода `Site.GetService(typeof(ProjectItem))`. См. разд. 7.13.

Как выводить отладочную информацию в окно *Output*?

Соответствующий код показан в разд. 7.14.

16.3. Дизайнер формы и компонентов

Как отследить изменение выбранного компонента на форме во время разработки?

Сервис `IServiceSelectionService` предоставляет события `SelectionChanging` и `SelectionChanged`, которые вызываются соответственно до и после изменения выбранного компонента (см. разд. 7.5).

Как в режиме разработки различить создание новых компонентов на форме от инициализации уже существующих?

Методы `InitializeExistingComponent` и `InitializeNewComponent` класса `ComponentDesigner` (см. главу 6) вызываются соответственно при инициализации существующих компонентов и при создании новых (см. разд. 6.21).

Как ограничить список типов компонентов, которые могут располагаться на своем компоненте, и список компонентов, на котором можно располагать свой компонент?

Методы `CanBeParentedTo` и `CanParent` класса `ParentControlDesigner` позволяют управлять расположением компонентов друг на друге (см. разд. 6.20).

Почему значок компонента не отображается?

Проверьте, что фактическое имя ресурса в сборке совпадает с именем, заданным в атрибуте. Получить все имена ресурсов в сборке можно, например, с помощью такого кода:

```
string[] names =
    Assembly.GetAssembly(this.GetType()).GetManifestResourceNames();
foreach (string name in names)
    MessageBox.Show(name);
```

См. также разд. 5.6.

Как перезагрузить все компоненты на форме заново?

Метод `Reload` сервиса `IDesignerLoaderService` позволяет перезагрузить все компоненты на форме заново:

```
// Проверяем, что мы в режиме дизайна
if (Site != null && Site.DesignMode)
{
    // Получаем сервис нужного типа
    IDesignerLoaderService service =
        (IDesignerLoaderService)Site.GetService(
            typeof(IDesignerLoaderService));
    // Перезагружаем дизайнер
    if (service != null)
        service.Reload();
}
```

Как сделать окно смарт-тега?

Для создания собственного смарт-тега нужно создать список элементов тега, являющийся наследником класса `DesignerActionList`, а в перекрытом свойстве `ActionLists` дизайнера вернуть экземпляр этого списка (см. разд. 6.4).

Как изменить порядок элементов в окне смарт-тега?

Сделать это можно, перекрыв метод `GetSortedActionItems`, возвращающий коллекцию элементов диалогового окна (см. разд. 6.4).

Как обновить окно смарт-тега?

Для обновления окна смарт-тега нужно вызвать метод `Refresh` сервиса `DesignerActionUIService` (см. разд. 6.4).

Как заставить форму пересоздать компонент?

Пересоздать компонент можно с помощью сервиса `IMenuCommandService` (см. разд. 7.5).

Как получить все компоненты на форме?

Для получения всех компонентов на форме используется вызов метода `GetReferences(typeof(IComponent))` сервиса `IReferenceService` (см. разд. 7.8).

Как программно добавить компонент на форму?

Метод `CreateComponents()` класса `ToolboxItem` создает компоненты на форме так же, как будто их перетащили из палитры компонентов (см. разд. 7.16).

16.4. Отрисовка компонентов

Как сделать рисование компонента только в режиме разработки?

Оптимальное решение для отрисовки некоторых элементов компонента (например, авторского логотипа) — использование метода `OnPaintAdornments` класса `ControlDesigner` (см. разд. 6.8).

Как сделать отрисовку значения в редакторе свойств, как, например, это делается в редакторе цветов (цветной квадратик слева от названия цвета)?

Небольшое изображение, которое рисуется слева от значения свойства, можно вывести с помощью метода `PaintValue` (см. разд. 9.3.3).

Как запретить отрисовку сетки дизайнера формы на своем компоненте?

Запретить отрисовку сетки дизайнера формы можно, перекрыв метод `DrawGrid` класса `ParentControlDesigner` (см. разд. 6.20):

```
class GradientLabelDesigner : ParentControlDesigner
{
    protected override bool DrawGrid
    (
        get
```

```

    {
        return false;
    }
}
}

```

16.5. Редактирование свойств и сериализация

Как правильно устанавливать значение свойства в режиме разработки?

Прямое присвоение значения свойства в режиме разработки может вызвать ряд неожиданных эффектов, т. к. в этом случае среда разработки не знает об изменениях. Правильный способ — использование метода `SetValue` дескриптора свойства (см. разд. 4.5).

Как сделать редактируемым свойство, которое является классом и содержит свои свойства?

Для редактирования сложных свойств нужно подключить к классу конвертер типа (см. главу 8). Наиболее простой способ — использовать стандартный конвертер типа `ExpandableObjectConverter` (см. разд. 8.9).

Как сделать свойство "только для просмотра"?

Атрибут `ReadOnly` позволяет указать, что свойство нельзя редактировать, и редактор свойств показывает такое свойство серым цветом (см. разд. 5.4).

Можно ли отобразить в редакторе свойств имя, отличное от реального имени свойства?

Да, можно. Нужно указать отображаемое имя в атрибуте `DisplayName` (см. разд. 5.4).

Как сделать свойство типа `StringCollection` редактируемым?

Самый простой способ — добавить атрибут, указывающий класс редактора свойств (см. разд. "Редактирование `StringCollection`" главы 9):

```

[Editor("System.Windows.Forms.Design.StringCollectionEditor,
    System.Design, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a",
    "System.Drawing.Design.UITypeEditor, System.Drawing,
    Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a"

```

```

    ]

```

Как выполнить редактирование списка объектов?

Для редактирования списка объектов (например, `ArrayList`) нужно реализовать наследник класса `CollectionEditor` (см. разд. "Редактирование списка объектов" главы 9).

Почему при изменении полей сложного свойства компонент не обновляется?

При изменении полей сложного свойства компонент "не видит" эти изменения. Исправить ситуацию позволяет атрибут `NotifyParentProperty`, указывающий сообщать объекту-владельцу об изменениях, происходящих во внутренних свойствах (см. разд. 8.1).

Как отобразить имя свойства в круглых скобках?

Атрибут `ParenthesizePropertyName` указывает редактору свойств отображать имя свойства в круглых скобках (см. разд. 5.4).

Какое свойство отвечает за ввод пароля?

Атрибут `PasswordPropertyText` позволяет указать, что текстовое поле, представленное таким свойством, будет скрыто в редакторе свойств звездочками (см. разд. 5.4).

Можно ли при редактировании коллекции объектов отображать описание (Description) свойств, как в обычном редакторе свойств?

Да, для этого нужно найти в редакторе свойство-компонент с типом `VsPropertyGrid` и включить свойство `HelpVisible` (см. разд. "Изменение формы редактора коллекций" главы 9).

Как привязать к свойству редактор выбора имени файла с определенным расширением?

Для указания нужного расширения должен быть перекрыт метод `InitializeDialog` класса `FileNameEditor` (см. разд. "Редактор выбора имени файла и каталога" главы 9).

Как обновить редактор свойств при изменении значения свойства?

У такого свойства должен быть атрибут `RefreshProperties` (`RefreshProperties.Repaint`). См. разд. 8.8.

Как запретить редактирование свойства, если выбрано несколько компонентов одновременно?

Для свойств, которые не должны быть редактируемыми одновременно у нескольких компонентов (например, имя компонента должно быть уникальным

и не может быть присвоено сразу всем компонентам), должен быть выставлен атрибут `MergableProperty(false)`. См. главу 15.

Можно ли заменить набор значений свойства?

Да, можно. Для этого нужно реализовать конвертер типа (см. главу 8). Например, для свойства типа `bool` можно отображать не `true/false`, а `Да/Нет`.

Как реализовать стандартный набор значений для свойства?

Существуют два варианта реализации. Если выбор значения свойства предполагает создание нового экземпляра значения, то для реализации стандартного набора значений свойства нужно использовать методы `GetStandardValuesSupported` и `GetStandardValues` класса `TypeConverter` (см. разд. 8.5). Если же выбор должен производиться из уже существующих значений (например, выбор одного из компонентов формы), то следует использовать класс `ObjectSelectorEditor` (см. разд. "Редактор-дерево" главы 9).

Как реализовать редактор свойства в виде дерева объектов (как, например, редактор `BorderStyle`)?

Следует использовать редактор типа `ObjectSelectorEditor` (см. разд. "Редактор-дерево" главы 9).

Можно ли запретить ручной ввод значения свойства и оставить только выбор из стандартного набора значений?

Да, для этого метод `GetStandardValuesExclusive` класса `TypeConverter` должен возвращать `true` (см. разд. 8.5).

Как менять набор вложенных свойств в зависимости от выбранного значения свойства?

См. разд. 8.10.

Как изменить порядок отображения свойств в редакторе свойств?

Поменять порядок отображения свойств можно, перекрыв метод `GetProperties` класса `TypeConverter` (см. разд. 8.11).

Как реализовать собственный редактор свойства?

Реализовать собственный редактор свойства можно с помощью класса `UITypeEditor` (см. главу 9).

Можно ли редактор свойства вызвать с параметрами?

Да, можно. Для этого нужно подменить дескриптор редактируемого свойства (см. разд. 9.3.5).

16.6. Редактирование размеров и положения компонентов

Как запретить изменение размеров компонента?

Для запрещения изменения размера компонента нужно перекрыть свойство `SelectionRules` класса `ControlDesigner` (с.м. *разд. 6.19*):

```
public override SelectionRules SelectionRules
{
    get
    {
        SelectionRules selectionRules = base.SelectionRules;
        selectionRules &= ~(SelectionRules.AllSizeable);
        return selectionRules;
    }
}
```

Как компонент может получать события мыши в режиме разработки?

Двойной щелчок мыши на компоненте можно обработать с помощью метода `DoDefaultAction` (с.м. *разд. 6.10*).

Еще один вариант — прямая обработка очереди сообщений Windows с помощью метода `WndProc` класса `ControlDesigner` (с.м. *разд. 6.13*).

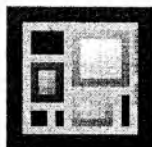
Как передавать события в режиме разработки компоненту, а не дизайнеру?

По умолчанию все события, происходящие во время разработки (например, движение мыши), обрабатываются дизайнером. В результате компоненты ведут себя "как нарисованные". Передавать события компонентам можно с помощью простого дизайнера (с.м. *разд. 6.14*):

```
public class WindowMessageDesigner : ControlDesigner
{
    public WindowMessageDesigner() {}

    // Обрабатываем WndProc
    protected override void WndProc(ref Message message)
    {
        if (message.HWnd == this.Control.Handle)
            base.WndProc(ref message);
        else
            this.DefWndProc(ref message);
    }
}
```

ПРИЛОЖЕНИЕ 1



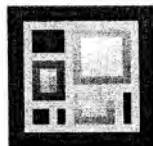
Полезные ссылки

- ☐ Сайт RSDN
www.rsdn.ru
- ☐ XML-сериализация сложных коллекций
<http://fahrain.blogspot.com/2007/09/xml.html>
- ☐ Разработка собственного дизайнера форм для .NET-приложений
<http://www.microsoft.com/Rus/Msdn/Magazine/2004/12/ApplicationCreator.aspx>
- ☐ Вся правда о PropertyGrid
<http://forum.sources.ru/index.php?showtopic=138155>
- ☐ Элементы управления Windows Forms и компоненты
<http://rsdn.ru/article/dotnet/winformsctrls.xml>
- ☐ Использование атрибутов в среде .NET
<http://rsdn.ru/article/dotnet/attributes.xml>
- ☐ Метаданные в среде .NET
<http://rsdn.ru/article/dotnet/refl.xml>
- ☐ Runtime Object Editor
<http://www.codeproject.com/KB/cs/RuntimeObjectEditor.aspx>
- ☐ A designable PropertyTree for VS.NET
<http://www.codeproject.com/KB/miscctrl/propertytree.aspx>
- ☐ Implementing complex data binding in custom controls
<http://www.codeproject.com/cs/database/DataBindCustomControls.asp>

- ❑ Data binding a ListView
http://www.codeproject.com/cs/miscctrl/ListView_DataBinding.asp
- ❑ Data Binding in .NET / C# Windows Forms
http://www.akadia.com/services/dotnet_databinding.html
- ❑ Complex data-binding со свойствами вложенных объектов
<http://rstdn.ru/forum/message/1924793.flat.aspx#1924793>
- ❑ .NET Shape Library: A Sample Designer
<http://windowsforms.net/articles/shapedesigner.aspx>
- ❑ Hosting Windows Forms Designers
<http://www.divil.co.uk/net/articles/designers/hosting.asp>
- ❑ Simplify UI Development with Custom Designer Actions in Visual Studio
<http://msdn.microsoft.com/msdnmag/issues/05/07/DesignerActions>
- ❑ Build a Property Editor That Can Edit Any .NET Type
<http://www.devx.com/dotnet/Article/20920/1954?pf=true>
- ❑ Deploying Controls to VS.NET ToolBox Programmatically
<http://www.codeproject.com/dotnet/AddItemToToolBar.asp?print=true>
- ❑ PropertyGrid FAQ
<http://rstdn.ru/article/dotnet/PropertyGridFAQ.xml>
- ❑ Give Your Everyday Custom Collections a Design-Time Makeover
<http://msdn.microsoft.com/msdnmag/issues/05/08/CollectionsandDataBinding/toc.asp?frame=true>
- ❑ Multithreading in Windows Forms Controls
[http://msdn2.microsoft.com/en-us/library/ms229730\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms229730(VS.80).aspx)
- ❑ Developing Windows Forms Controls at Design Time
[http://msdn2.microsoft.com/en-us/library/w29y3h59\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/w29y3h59(VS.80).aspx)
- ❑ How to: Implement a Component That Supports the Event-based Asynchronous Pattern
<http://msdn2.microsoft.com/en-us/library/9hk12d4y.aspx>
- ❑ George Shepherd's Windows Forms FAQ
<http://www.syncfusion.com/faq/windowsforms/Default.aspx>
- ❑ .NET FAQ collection
<http://www.serialcoder.net/FAQ/winforms/Default.aspx>

- ❑ Microsoft WindowsClient.NET FAQ
<http://windowsclient.net/blogs/faqs/default.aspx>
- ❑ How to Edit and Persist Collections with CollectionEditor
<http://www.codeproject.com/KB/cs/dzcollectioneditor.aspx>
- ❑ Using IDesigner and CodeDomSerializer to create read-only runtime properties
<http://www.codeproject.com/KB/miscctrl/RuntimeProperties.aspx>
- ❑ Extending the PropertyGrid with a new PropertyTab
<http://www.codeproject.com/KB/grid/PropertyGridExWinForms.aspx>

ПРИЛОЖЕНИЕ 2



Описание компакт-диска

Папка	Описание	Глава или раздел
!MicrosoftCodeExamples	Пример разработки собственного дизайнера с сайта Microsoft	16.1
=DataBinding	Примеры к главе "Компоненты работы с данными"	13
=Examples\BalloonToolTip	Пример взаимодействия с Win32 API — реализация всплывающей подсказки (на основе IExtenderProvider)	12.6
=Installation	Примеры к главе "Распространение компонентов"	14
=Serializer	Примеры к главе "Сериализация времени разработки"	10
=Services\ComponentChangeService	Обработка изменений компонентов и их свойств (сервис IComponentChangeService)	7.4
=Services\DesignerEventService	Нотификации о смене дизайнера (сервис IDesignerEventService)	7.11
=Services\DictionaryServiceControl	Пользовательские данные времени разработки (сервис IDictionaryService)	7.12
=Services\EventBindingService	Управление событиями компонента с помощью сервиса IEventBindingService	7.9
=Services\ExtenderService_Step1 =Services\ExtenderService_Step2 =Services\ExtenderService_Step3 =Services\ExtenderService_Step4 =Services\ExtenderService_Step5	Пример реализации глобальных свойств компонентов (на основе IExtenderProvider)	12.1

(продолжение)

Папка	Описание	Глава или раздел
=Services\HelpService	Реализация контекстной подсказки с помощью сервиса IHelpService	7.17
=Services\MenuCommandService	Реализация "горячих" клавиш дизайнера формы с помощью сервиса IMenuCommandService	7.7
=Services\ProjectItem	Получение информации о проекте с помощью класса ProjectItem	7.13
=Services\PropertyValueUIService	Пиктограмма значения компонента (сервис IPropertyValueUIService)	7.22
=Services\ReferenceService	Работа с именами и типами компонентов (сервис IReferenceService)	7.8
=Services\SelectionService	Управление выделенными компонентами на форме дизайнера (сервис ISelectionService)	7.5
=Services\SplitWindowService	Использование недокументированного сервиса ISplitWindowService	7.23
=Services\ToolBoxService	Взаимодействие с палитрой компонентов (сервис IToolboxService)	7.16
=Services\TypeDescriptorFilterService	Фильтрация свойств и событий (сервис ITypeDescriptorFilterService)	7.18
=Services\UIServiceTestDesigner	Создание диалоговых окон в режиме разработки (сервис IUIService)	7.20
AsyncOperationManager, BackgroundWorker	Работа с потоками	12.5
BooleanConverter	Редактирование свойства типа bool как выпадающего списка Да/Нет (класс BooleanConverter)	8.3
Container	Как контейнер освобождает свои ресурсы	4.3
ConverterTest	Использование BooleanConverter в режиме выполнения	8.3
DesignerTransaction	Пример использования транзакций дизайнера	12.2
EnumTypeConverter	Редактор перечисления (класс EnumConverter)	8.3
FileNameEditor	Как задать расширение при редактировании имени файла	9.3.4
FlagsTypeConverter	Редактирование флагов	8.7

(продолжение)

Папка	Описание	Глава или раздел
GenericPropertyTest	Редактирование обобщений	12.4
LayoutEngine	Автоматическое расположение элементов с помощью класса <code>LayoutEngine</code>	11.5
License	Пример реализации провайдера лицензий (класс <code>LicFileLicenseProvider</code>)	14.5
MyControl_AddDesigner	Привязка дизайнера к проекту	6.3
MyControl_Adorner	Работа со слоями	6.14
MyControl_Adornments	Отрисовка компонента в режиме разработки	6.8
MyControl_Beginning	Базовый проект компонента	5.1
MyControl_ContextMenu1, MyControl_ContextMenu2	Реализация контекстного меню компонента	6.5
MyControl_DinamProp	Динамическое управление списком свойств	8.8
MyControl_Glyph	Реализация собственного маркера компонента	6.14
MyControl_GlyphMenu	Реализация меню с помощью маркера	6.15
MyControl_LockResize	Блокировка изменения размера компонента	6.16
MyControl_Mouse	Обработка мыши в режиме разработки	6.9
MyControl_SmartTag	Реализация смарт-тега	6.4
MyControl_TypeConverter	Реализация конвертера типа для редактирования сложного свойства	8.4
MyControl_TypeConverter_Sort	Определение порядка отображения свойств	8.11
MyControl_TypeConverterSt	Реализация редактирования сложного свойства с помощью стандартного конвертера <code>ExpandableObjectConverter</code>	8.9
MyControl_VirtualProp	Реализация виртуального свойства	6.7
MyControl_WndProc	Прямая обработка очереди сообщений в режиме разработки	6.12
NestedContainer	Встроенные компоненты	7.15
NewPropertyTab	Реализация новой закладки в редакторе свойств	12.3

(окончание)

Панка	Описание	Глава или раздел
ObjectSelectorEditor	Редактор в виде дерева	9.3.4
SerializeTest0, SerializeTest1, SerializeTest2, SerializeTest3	Примеры различной сериализации	10
SettingsBindable	Использование атрибута SettingsBindable для обозначения настраиваемого свойства	10.7
StandartValues	Реализация стандартного набора значений	8.5
StandartValuesClass	Стандартный набор значений-классов	8.10
StandartValuesEx	Реализация стандартного набора значений (с исправлением проблемы двойного щелчка)	8.5
TableLayout	Глобальные свойства компонентов, расположенных на панели TableLayoutPanel	12.1
ThreadInvoke	Использование класса BackgroundWorker	12.5
TypeDescriptor	Получение списка свойств и событий компонента	4.5
UIEditor1, ..., UIEditor8	Редакторы типов	9
VolumeInfo	Получение серийного номера логического диска (может использоваться при лицензировании)	14.5.8
WindowMessageDesigner	Дизайнер, передающий сообщения в компоненты в режиме разработки	6.13

Литература

1. Matthew MacDonald, *User Interfaces in C#: Windows Forms and Custom Controls*. — APress, 2002.
2. James Henry, *Developing .NET Custom Controls and Designers using C#*. — BlueVision, 2005
3. Агуров П. В. *Практика программирования USB*. — СПб.: БХВ-Петербург, 2006.
4. Агуров П. В. *C#. Сборник рецептов*. — СПб.: БХВ-Петербург, 2007.
5. Рихтер Д., *CLR via C#*. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. — М.: Русская Редакция, 2007.
6. Троелсен Э. *Язык программирования C# 2005 и платформа .NET 2.0*. — М.: И.Д. Вильямс, 2007.

Предметный указатель

A

Accessor 31

Attribute 37

C

Container 47

D

Data binding 381

Default event 112

Delegate 31

E

Event 31

Extender 333

F

Form designer *C.m* дизайнер формы

G

GAC:

◊ определение 413

◊ удаление сборки 418

◊ установка сборки 418

GacUtil, утилита 416

Generic 352

Glyph 20, 117

H

Host 51

R

Reflection 35

S

Smart tag 89

Snap lines 22

T

Transaction 347

Type converter 214

U

UITypeEditor 255

V

Verbs 100

А

- Аксессор 31
- Архитектура времени разработки 43
- Атрибут:
 - ◊ AttributeProvider 78, 394, 457
 - ◊ AttributeUsageAttribute 37
 - ◊ Browsable 76, 456
 - ◊ Category 72, 278
 - ◊ ComplexBindingProperties 400
 - ◊ DefaultEvent 74, 112
 - ◊ DefaultProperty 74
 - ◊ DefaultValue 74, 312
 - ◊ Description 72, 79
 - ◊ Designer 88, 412
 - ◊ DesignerSerializationVisibility 231, 305, 312
 - ◊ DesignerSerializer 319
 - ◊ DesignTimeVisible 81
 - ◊ DisplayName 76, 231, 278, 335
 - ◊ DllImport 365
 - ◊ Editor 276, 288
 - ◊ EditorAttribute 255
 - ◊ LicenseProvider 434
 - ◊ MarshalAs 367
 - ◊ MergableProperty 77
 - ◊ NotifyParentProperty 213
 - ◊ ParenthesizePropertyName 77, 461
 - ◊ PasswordPropertyText 76, 461
 - ◊ ProvideProperty 334
 - ◊ ReadOnly 77, 460
 - ◊ RefreshProperties 242
 - ◊ RunInstallerAttribute 430
 - ◊ SettingsBindable 317
 - ◊ ToolboxBitmap 79
 - ◊ ToolboxItem 79, 81
 - ◊ TypeConverter 220, 394
 - ◊ определение 37
 - ◊ перебор 39
 - ◊ получение списка 39

Г

- Глобальный кэш сборки 413

Д

- Действие по умолчанию 112
- Делегат 31
- Дескриптор:
 - ◊ базовый 52
 - ◊ получение 53
 - ◊ свойства 52
 - ◊ события 52
- Дизайнер 85
 - ◊ определение 44
- Дизайнер формы 20

И

- Интерфейс:
 - ◊ IBindingList 391
 - ◊ ICancelAddNew 392
 - ◊ IChangeTracking 392
 - ◊ IComponent 45, 64
 - ◊ IContainer 47
 - ◊ IDataErrorInfo 393
 - ◊ IDesignerHost 51, 180
 - ◊ IDesignerSerializationManager 319
 - ◊ IDisposable 34, 45, 63
 - ◊ IEditableObject 392
 - ◊ IExtenderProvider 333, 335
 - ◊ INestedContainer 49, 137, 176
 - ◊ IRaiseItemChangedEvents 392
 - ◊ IResourceReader 195
 - ◊ IResourceWriter 195
 - ◊ ISite 48
 - ◊ ISupportInitialize 61, 340
 - ◊ определение 34

К

- Класс:
 - ◊ Activator 280
 - ◊ Adorner 117
 - ◊ AsyncOperationManager 362
 - ◊ Attribute 37
 - ◊ BackgroundWorker 357
 - ◊ BaseNumberConverter 215
 - ◊ Behavior 125
 - ◊ Binding 385

- ◊ BindingContext 381
 - ◊ BooleanConverter 215
 - ◊ CharConverter 215
 - ◊ CodeObject 320
 - ◊ CollectionConverter 215
 - ◊ CollectionEditor 272, 276, 285, 356
 - ◊ CollectionForm 285
 - ◊ ColorConverter 215
 - ◊ Component 45, 64
 - ◊ ComponentDesigner 135, 149
 - ◊ ComponentGlyph 118
 - ◊ Container 48
 - ◊ ControlBodyGlyph 118
 - ◊ ControlStyles 327
 - ◊ CSharpCodeProvider 300
 - ◊ CurrencyManager 381, 395
 - ◊ DateTimeConverter 215
 - ◊ DesignerActionHeaderItem 95
 - ◊ DesignerActionItem 95
 - ◊ DesignerActionItemCollection 95
 - ◊ DesignerActionList 458
 - ◊ DesignerActionMethodItem 95, 100
 - ◊ DesignerActionPropertyItem 95
 - ◊ DesignerActionTextItem 95
 - ◊ DesignerHost 347
 - ◊ DesignerOptionService 163
 - ◊ DesignerVerb 100
 - ◊ DesignerVerbCollection 100
 - ◊ EnumConverter 215, 217
 - ◊ EventArgs 32
 - ◊ EventDescriptor 52
 - ◊ EventDescriptorCollection 54
 - ◊ EventHandler 32
 - ◊ ExpandableObjectConverter 246, 247
 - ◊ FlowLayoutPanel 328
 - ◊ FontConverter 215
 - ◊ Glyph 117
 - ◊ GlyphCollection 118
 - ◊ Installer 429
 - ◊ InstanceDescriptor 302
 - ◊ IntegerCollectionEditor 353
 - ◊ LayoutEngine 329
 - ◊ License 433, 434
 - ◊ LicenseContext 435
 - ◊ LicenseManager 434, 435
 - ◊ LicenseProvider 433
 - ◊ LicFileLicenseProvider 433
 - ◊ ListControlStringCollectionEditor 353
 - ◊ MarshalByValueComponent 64
 - ◊ MemberDescriptor 52
 - ◊ MenuCommands 150
 - ◊ ObjectSelectorEditor 291
 - ◊ PaintEventArgs 323
 - ◊ ParentControlDesigner 131, 132
 - ◊ PointConverter 215
 - ◊ ProjectItem 137, 171, 457
 - ◊ PropertyDescriptor 52
 - ◊ PropertyDescriptorCollection 54
 - ◊ PropertyManager 381
 - ◊ PropertyTab 349
 - ◊ PropertyValueUIItem 202
 - ◊ Publish 418
 - ◊ RectangleConverter 215
 - ◊ Selector 291
 - ◊ SimplePropertyDescriptor 233
 - ◊ SizeConverter 215
 - ◊ SnapLine 111
 - ◊ StandardCommands 150
 - ◊ StandardValuesCollection 225
 - ◊ StringCollectionEditor 273
 - ◊ StringConverter 215
 - ◊ System.Attribute 37
 - ◊ ToolboxItem 179
 - ◊ TypeConverter 214, 233
 - ◊ TypeDescriptor 54
 - ◊ UIMTypeEditor 255, 258
 - ◊ VisualStyles 325
 - ◊ WindowsFormsDesignerOptionService 163
- Ключевое слово:
- ◊ event 31
 - ◊ interface 34
- Компонент:
- ◊ визуальный 23
 - ◊ добавление 139
 - на форму 179
 - ◊ изменение 139
 - размеров 129
 - ◊ имя 48
 - ◊ инициализация 133

Класс (*прод.*):

- ◊ контекстное меню 100
- ◊ метод 15
- ◊ невизуальный 23
- ◊ основной 22
- ◊ особенности разработки 17
- ◊ отладка 65
- ◊ переименование 139
- ◊ получение списка 157
- ◊ расширитель 333
- ◊ свойство 15, 29
- ◊ событие 15, 29
- ◊ удаление 139

Конвертер типа:

- ◊ BaseNumberConverter 215
 - ◊ BooleanConverter 215
 - ◊ CharConverter 215
 - ◊ CollectionConverter 215
 - ◊ ColorConverter 215
 - ◊ DateTimeConverter 215
 - ◊ EnumConverter 215
 - ◊ ExpandableObjectConverter 246, 247
 - ◊ FontConverter 215
 - ◊ PointConverter 215
 - ◊ RectangleConverter 215
 - ◊ SizeConverter 215
 - ◊ StringConverter 215
 - ◊ базовый класс 214
 - ◊ определение 44, 214
- Контейнер 47
- Контекстное меню 100

Л

Лицензирование 433

Локализация 76, 195, 313

М

Маркеры 117

- ◊ изменение размера:
 - компонента 20
 - формы 20

Метаинформация 35

Метод:

- ◊ AddContextAttribute 188

- ◊ CanBeParentedTo 131
- ◊ CanParent 131
- ◊ CreateCollectionForm 285
- ◊ CreateComponents 180, 459
- ◊ CreateTransaction 347
- ◊ CreateUniqueMethodName 158
- ◊ DoDefaultAction 109
- ◊ FilterProperties 191
- ◊ FindForm 180
- ◊ GetPropertiesSupported 223, 242
- ◊ GetService 51, 135
- ◊ GetStandardValues 225
- ◊ GetStandardValuesExclusive 229
- ◊ GetStandardValuesSupported 225
- ◊ InitializeExistingComponent 133
- ◊ InitializeNewComponent 133
- ◊ OnMouseEnter 109
- ◊ OnMouseLeave 109
- ◊ OnParentChanged 131
- ◊ ShouldSerialize 312
- ◊ ShowHelpFromKeyword 188
- ◊ ShowHelpFromUrl 188
- ◊ ShowMessage 196
- ◊ обратного вызова 31

О

Обобщение 352

Окно Properties 24

Отладка 65

Отражение 35

П

Палитра компонентов 24

Позднее связывание 412

Поток 357

Привязка данных 381

Провайдер лицензий 433

Р

Редактор свойств 25

Редактор типа 255

- ◊ CollectionEditor 272, 356

- ◊ DataMemberListEditor 394

- ◊ FileNameEditor 289
- ◊ FolderNameEditor 290
- ◊ IntegerCollectionEditor 353
- ◊ ListControlStringCollectionEditor 353
- ◊ ObjectSelectorEditor 291
- ◊ StringCollectionEditor 273
- ◊ коллекции 285
- ◊ массива 288
- ◊ определение 44
- ◊ списка объектов 275
- Рендеринг 323

С

Свойство:

- ◊ ActionLists 458
- ◊ CanRaiseEvents 45
- ◊ DesignMode 45
- ◊ GenerateMember 45
- ◊ LayoutEngine 329
- ◊ Locked 45
- ◊ Modifiers 45
- ◊ SelectionRules 129
- ◊ SnapLines 111
- ◊ Verbs 100, 149
- ◊ виртуальное 105
- ◊ глобальное 333
- ◊ дескриптор 90
- ◊ категория свойства 72
- ◊ определение 29
- ◊ по умолчанию 74
- ◊ скрытие 103

Сервис:

- ◊ BehaviorService 138
- ◊ DesignerActionUIService 93, 138
- ◊ IComponentChangeService 139, 343
- ◊ IDesignerEventService 164
- ◊ IDesignerOptionService 161
- ◊ IDictionaryServiceDesigner 168, 457
- ◊ IEventBindingService 158
- ◊ IExtenderProviderService 335
- ◊ IHelpService 188
- ◊ IMenuCommandService 148, 150, 459
- ◊ IMenuService 149

- ◊ INameCreationService 157
- ◊ IPropertyValueUIService 201
- ◊ IReferenceService 154
- ◊ IResourceService 195
- ◊ ISelectionService 120, 145
- ◊ ISplitWindowService 206
- ◊ ITypeDescriptorFilterService 190
- ◊ ITypeDiscoveryService 281
- ◊ IUIService 196
- ◊ IWindowsFormsEditorService 258
- ◊ ToolboxService 179
- ◊ определение 44
- ◊ список 135

Сериализация 300

- ◊ управление кодом 318

Смарт-тег 89, 138, 458

Событие:

- ◊ ActiveDesignerChanged 164
- ◊ ComponentAdded 139
- ◊ ComponentAdding 139
- ◊ ComponentChanged 139
- ◊ ComponentChanging 139
- ◊ ComponentRemoved 139
- ◊ ComponentRemoving 139
- ◊ ComponentRename 139
- ◊ DesignerCreated 164
- ◊ SelectionChanged 146, 164
- ◊ SelectionChanging 145
- ◊ по умолчанию 74
- ◊ управление 158
- Событие по умолчанию 112
- Стандартный набор значений 224

Т

Транзакции 347

Ф

Флаги 234

Х

Хост 51

Магазин-салон «НОВАЯ ТЕХНИЧЕСКАЯ КНИГА»

190005, Санкт-Петербург, Измайловский пр., 29

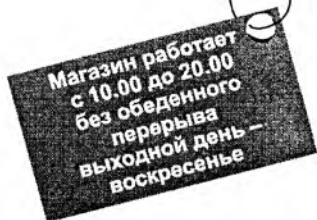
В МАГАЗИНЕ ПРЕДСТАВЛЕНА ЛИТЕРАТУРА

ПО

**компьютерным технологиям
радиотехнике и электронике
физике и математике
строительству и архитектуре
транспорту
машиностроению**

***и другим
естественно-научным
и техническим направлениям***

- ✓ Низкие цены
- ✓ Прямые поставки
от издательств
- ✓ Ежедневное пополнение
ассортимента
- ✓ Подарки и скидки
покупателям



**Тел.: (812) 251-41-10, e-mail: trade@techkniga.com
www.techkniga.com**

C#

РАЗРАБОТКА КОМПОНЕНТОВ В MS VISUAL STUDIO 2005/2008



Все что нужно для разработки компонентов



Агуров Павел Владимирович, менеджер проектов и ведущий программист компании EPAM Systems. Специалист с многолетним опытом работы, автор книг «Последовательные интерфейсы ПК. Практика программирования», «Интерфейс USB. Практика использования и программирования», «С#. Сборник рецептов» и др.

Компоненты предоставляют программисту множество замечательных возможностей, таких как повторное использование программного кода, согласованность интерфейса приложения и много других. Программист может использовать и модернизировать компоненты сторонних разработчиков (при наличии исходного кода и прав), а также создавать и продавать свои собственные. Между тем информации на эту тему очень мало, а документация в MSDN местами очень запутанна и порой недостаточна. Книга предоставляет все необходимое для разработки и распространения компонентов, написанных в среде Visual Studio 2005/2008 на языке C#. Весь код, содержащийся в книге, полностью работоспособен и готов к использованию.

КАТЕГОРИЯ:

C#



На компакт-диске
приведены примеры
из книги.

ISBN 978-5-9775-0295-5



9 785977 502955



БХВ-ИНТЕРБУРГ, 194354,
Санкт-Петербург, ул. Есенина, 56
E-mail: info@bhv.ru / lr@bhv.ru www.bhv.ru
Тел./факс: (812) 591-6243