НИЕСН

АДАМ НАТАН

WPF 4

ПОДРОБНОЕ РУКОВОДСТВО



СИМВОЛ®

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-196-7, название «WPF 4. Подробное руководство» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

15 лет вместе
с профессионалами

с 1996 года
books.ru
всегда к слову

WPF 4

Unleashed

Adam Nathan

SAMS

WPF 4

Подробное руководство

Адам Натан

Санкт-Петербург — Москва
2011

Адам Натан

WPF 4. Подробное руководство

Перевод А. Слинкин

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*Е. Тульсанова*
Корректоры	*С. Минин, О. Макарова*
Верстка	*Д. Орлова*

Натан А.

WPF 4. Подробное руководство. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 880 с., ил.

ISBN 978-5-93286-196-7

Windows Presentation Foundation (WPF) – рекомендуемая технология реализации пользовательских интерфейсов для Windows-приложений. Она позволяет создавать такие функционально насыщенные и визуально привлекательные приложения, о которых вы раньше не могли и мечтать. WPF дает возможность естественно объединять в одной программе традиционные интерфейсы, трехмерную графику, аудио и видео, анимацию, динамическую смену обложек, мультисенсорный ввод, форматированные документы и распознавание речи.

Книгу Адама Натана, известного гуру в области WPF, отличают полнота освещения, практические примеры и понятный язык. Издание содержит сведения о XAML – расширяемом языке разметки приложений; детально рассматриваются функциональные возможности WPF: элементы управления, компоновка, ресурсы, привязка к данным, стили, графика, анимация; уделено внимание новейшим средствам: мультисенсорному вводу, усовершенствованной визуализации текста, новым элементам управления, дополнениям языка XAML, программе Visual State Manager, переходным функциям в анимации; рассматриваются трехмерная графика, синтез и распознавание речи, документы и эффекты; демонстрируется создание популярных элементов пользовательского интерфейса, например галерей и экранных подсказок, а также создание более сложных механизмов организации пользовательского интерфейса, например выдвигающихся и стыкуемых панелей, как в Visual Studio; описывается, как создавать полноценные элементы управления WPF; демонстрируется создание гибридных приложений, в которых WPF сочетается с Windows Forms, DirectX и ActiveX; объясняется, как задействовать в WPF-приложении новые средства Windows 7, например списки переходов, и как обойти некоторые присущие WPF ограничения.

ISBN 978-5-93286-196-7
ISBN 978-0-672-33119-0 (англ)

© Издательство Символ-Плюс, 2011

Authorized translation of the English edition © 2010 Pearson Education. This translation is published and sold by permission of Pearson Education, the owner of all rights to publish and sell the same.

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

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 04.11.2011. Формат 70×100 1/16. Объем 55 печ. л.

Оглавление

Введение 19

I. Базовые сведения 27

1. Почему именно WPF и как насчет Silverlight? 29

Взгляд в прошлое 30
Появление WPF 32
Эволюция WPF 35
Усовершенствования в WPF 3.5 и WPF 3.5 SP1 36
Усовершенствования в WPF 4 38
Что такое Silverlight 40
Резюме 42

2. Все тайны XAML 43

Определение XAML 45
Элементы и атрибуты 47
Пространства имен 48
Элементы свойств 51
Конвертеры типов 52
Расширения разметки 55
Дочерние объектные элементы 58
Свойство Content 58
Элементы коллекций 59
Еще о преобразовании типов 61
Сочетание XAML и процедурного кода 63
Загрузка и разбор XAML во время выполнения 63
Компиляция XAML 67
Введение в XAML2009 72
Полная поддержка универсальных классов 73
Словарные ключи произвольного типа 74
Встроенные системные типы данных 75
Создание объектов с помощью конструктора с аргументами . . . 75
Создание экземпляров с помощью фабричных методов 76
Гибкость присоединения обработчиков событий 76

Определение новых свойств 77
Трюки с классами чтения и записи XAML 77
Обзор 78
Циклы обработки узлов 81
Чтение XAML 82
Запись в объекты 86
Запись в формате XML 88
XamlServices 89
Ключевые слова XAML 92
Резюме 96
Возражение 1: XML слишком многословен, долго набирать 97
Возражение 2: системы, основанные на XML, низкопроизводительны 97

3. Основные принципы WPF 98
Обзор иерархии классов 98
Логические и визуальные деревья 100
Свойства зависимости 106
Реализация свойства зависимости 107
Уведомление об изменении 109
Наследование значений свойств 111
Поддержка нескольких поставщиков 113
Присоединенные свойства 117
Резюме 121

II. Создание WPF-приложения 123

4. Задание размера, положения и преобразований элементов 125
Управление размером 126
Свойства Height и Width 126
Свойства Margin и Padding 128
Свойство Visibility 131
Управление положением 132
Выравнивание 132
Выравнивание содержимого 133
Свойство FlowDirection 134
Применение преобразований 135
Преобразование RotateTransform 137
Преобразование ScaleTransform 139
Преобразование SkewTransform 141
Преобразование TranslateTransform 142
Преобразование MatrixTransform 142
Комбинирование преобразований 143

Резюме 144

5. Компоновка с помощью панелей 146

Панель Canvas 147
Панель StackPanel 150
Панель WrapPanel 152
Панель DockPanel 154
Панель Grid 158
Задание размеров строк и столбцов 162
Интерактивное задание размера с помощью GridSplitter 165
Задание общего размера для строк и столбцов 166
Сравнение Grid с другими панелями 169
Примитивные панели 171
Панель TabPanel 171
Панель ToolBarPanel 171
Панель ToolBarOverflowPanel 171
Панель ToolBarTray 171
Панель UniformGrid 172
Панель SelectiveScrollingGrid 172
Обработка переполнения содержимого 173
Отсечение 173
Прокрутка 175
Масштабирование 177
Все вместе: создание сворачиваемой, стыкуемой, изменяющей размер панели 182
Резюме 192

6. События ввода: клавиатура, мышь, стилус и мультисенсорные устройства 193

Маршрутизируемые события 193
Реализация маршрутизируемого события 194
Стратегии маршрутизации и обработчики событий 195
Маршрутизируемые события в действии 196
Присоединенные события 200
События клавиатуры 202
События мыши 205
Класс MouseEventArgs 206
Перетаскивание 207
Захват мыши 208
События стилуса 209
Класс StylusDevice 210
События 210
Мультисенсорные события 211
Простые события касания 212

События манипулирования, описывающие сдвиг, поворот и масштабирование . . . 216
Команды . . . 224
Встроенные команды . . . 225
Выполнение команд с помощью жестов ввода . . . 228
Элементы управления со встроенными привязками к командам . . . 229
Резюме . . . 230

7. Структурирование и развертывание приложения . . . 231
Стандартные приложения Windows . . . 231
Класс Window . . . 232
Класс Application . . . 235
Показ заставки . . . 242
Создание и показ диалоговых окон . . . 243
Сохранение и восстановление состояния приложения . . . 246
Развертывание: технология ClickOnce и установщик Windows . . . 247
Приложения Windows с навигацией . . . 249
Страницы и их навигационные контейнеры . . . 249
Переходы между страницами . . . 252
Передача данных между страницами . . . 258
Приложения-гаджеты . . . 261
XAML-приложения для браузера . . . 263
Ограниченный набор возможностей . . . 265
Интегрированная навигация . . . 268
Развертывание . . . 268
Автономные XAML-страницы . . . 271
Резюме . . . 272

8. Особенности Windows 7 . . . 273
Списки переходов . . . 273
Элемент JumpTask . . . 275
Элемент JumpPath . . . 282
Настройка элементов на панели задач . . . 287
Индикатор выполнения для элемента на панели задач . . . 287
Наложения для элементов на панели задач . . . 288
Настройка содержимого эскиза . . . 289
Добавление кнопок управления к эскизу на панели задач . . . 290
Функция Aero Glass . . . 292
Функция TaskDialog . . . 296
Резюме . . . 299

III. Элементы управления 301

9. Однодетные элементы управления 303
Кнопки 305
Класс Button 306
Класс RepeatButton 307
Класс ToggleButton 308
Класс CheckBox 308
Класс RadioButton 309
Простые контейнеры 311
Класс Label 311
Класс ToolTip 312
Класс Frame 314
Контейнеры с заголовками 316
Класс GroupBox 316
Класс Expander 318
Резюме 318

10. Многодетные элементы управления 319
Общая функциональность 320
DisplayMemberPath 321
ItemsPanel 322
Управление поведением прокрутки 325
Селекторы 325
Элемент ComboBox 326
Элемент ListBox 332
Элемент ListView 335
Элемент TabControl 336
Элемент DataGrid 337
Меню 345
Элемент Menu 345
Элемент ContextMenu 347
Другие многодетные элементы управления 349
Элемент TreeView 349
Элемент ToolBar 351
Элемент StatusBar 354
Резюме 355

11. Изображения, текст и другие элементы управления 356
Элемент управления Image 356
Элементы управления Text и Ink 358
Элемент TextBlock 360
Элемент TextBox 362
Элемент RichTextBox 364

Элемент PasswordBox 364
Элемент InkCanvas 365
Документы 367
Создание потоковых документов 367
Отображение потоковых документов 378
Добавление комментариев 380
Диапазонные элементы управления 383
Элемент ProgressBar 384
Элемент Slider 385
Календарные элементы управления 386
Элемент Calendar 386
Элемент DatePicker 388
Резюме 389

IV. Средства для профессиональных разработчиков 391

12. Ресурсы 393
Двоичные ресурсы 393
Определение двоичного ресурса 394
Доступ к двоичным ресурсам 395
Локализация двоичных ресурсов 400
Логические ресурсы 402
Поиск ресурса 406
Статические и динамические ресурсы 406
Взаимодействие с системными ресурсами 411
Резюме 413

13. Привязка к данным 414
Знакомство с объектом Binding 414
Использование объекта Binding в процедурном коде 414
Использование объекта Binding в XAML 417
Привязка к обычным свойствам .NET 419
Привязка ко всему объекту 420
Привязка к коллекции 422
Обобществление источника с помощью DataContext 426
Управление визуализацией 428
Форматирование строк 428
Шаблоны данных 431
Конвертеры значений 434
Настройка представления коллекции 440
Сортировка 440
Группировка 443
Фильтрация 446
Навигация 447

Дополнительные представления 449
Поставщики данных 451
Класс XmlDataProvider 452
Класс ObjectDataProvider 456
Дополнительные вопросы 459
Настройка потока данных 459
Добавление в привязку правил проверки 461
Работа с несколькими источниками 466
А теперь все вместе: клиент Twitter на чистом XAML 469
Резюме 471

14. Стили, шаблоны, обложки и темы 472
Стили 473
Обобществление стилей 475
Триггеры 481
Шаблоны 488
Введение в шаблоны элементов управления 489
Обеспечение интерактивности с помощью триггеров 490
Ограничение типа целевого элемента 492
Учет свойств шаблона-родителя 493
Учет визуальных состояний с помощью триггеров 500
Учет визуальных состояний
с помощью менеджера визуальных состояний 505
Комбинирование шаблонов со стилями 514
Обложки 517
Темы 524
Системные цвета, шрифты и параметры 524
Стили и шаблоны тем 525
Резюме 529

V. Мультимедиа 531

15. Двумерная графика 533
Класс Drawing 534
Класс Geometry 537
Класс Pen 548
Пример изображения 550
Класс Visual 552
Наполнение DrawingVisual содержимым 553
Отображение объекта Visual на экране 556
Проверка попадания в Visual 559
Класс Shape 566
Класс Rectangle 568
Класс Ellipse 569

Класс Line . . . 570
Класс Polyline . . . 571
Класс Polygon . . . 572
Класс Path . . . 572
Изображение, составленное из объектов Shape . . . 573
Кисти . . . 575
Цветные кисти . . . 575
Мозаичные кисти . . . 584
Кисти как маски непрозрачности . . . 592
Эффекты . . . 594
Повышение производительности визуализации . . . 597
Класс RenderTargetBitmap . . . 597
Класс BitmapCache . . . 598
Класс BitmapCacheBrush . . . 601
Резюме . . . 601

16. Трехмерная графика . . . 602

Введение в трехмерную графику . . . 603
Камеры и системы координат . . . 607
Свойство Position . . . 608
Свойство LookDirection . . . 611
Свойство UpDirection . . . 614
Классы OrthographicCamera и PerspectiveCamera . . . 617
Класс Transform3D . . . 620
Преобразование TranslateTransform3D . . . 623
Преобразование ScaleTransform3D . . . 623
Преобразование RotateTransform3D . . . 627
Комбинирование преобразований Transform3D . . . 630
Класс Model3D . . . 631
Класс Light . . . 632
Класс GeometryModel3D . . . 639
Класс Model3DGroup . . . 654
Класс Visual3D . . . 656
Класс ModelVisual3D . . . 656
Класс UIElement3D . . . 658
Класс Viewport2DVisual3D . . . 660
Проверка попадания в трехмерном пространстве . . . 662
Класс Viewport3D . . . 663
Преобразование двумерных и трехмерных систем координат . . . 666
Метод Visual.TransformToAncestor . . . 666
Методы Visual3D.TransformToAncestor и Visual3D.
TransformToDescendant . . . 670
Резюме . . . 674

17. Анимация 675
Анимация в процедурном коде 676
Выполнение анимации «вручную» 676
Введение в классы анимации 677
Простые приемы работы с анимацией 685
Анимация в XAML-коде 690
Триггеры событий и раскадровки 690
Использование раскадровки как временной шкалы 698
Анимация с опорными кадрами 699
Линейные опорные кадры 700
Сплайновые опорные кадры 702
Дискретные опорные кадры 703
Переходные опорные кадры 706
Переходные функции 706
Встроенные переходные функции 707
Другие встроенные переходные функции 708
Написание своей переходной функции 710
Анимация и менеджер визуальных состояний 712
Переходы 716
Резюме 720

18. Аудио, видео и речь 722
Аудио 722
Класс SoundPlayer 723
Класс SoundPlayerAction 724
Класс MediaPlayer 724
Классы MediaElement и MediaTimeline 725
Видео 727
Управление визуальными аспектами класса MediaElement . . . 728
Управление мультимедийным содержимым 730
Речь 734
Синтез речи 734
Распознавание речи 737
Резюме 743

VI. Дополнительные вопросы 745

19. Интероперабельность с другими технологиями 747
Встраивание элементов управления Win32
в WPF-приложения 750
Элемент управления Win32 Webcam 750
Использование элемента управления Webcam в WPF 753
Поддержка навигации с помощью клавиатуры 760

Встраивание элементов управления WPF в Win32-приложения . . . 764
Введение в HwndSource . . . 765
Обеспечение правильной компоновки . . . 768
Встраивание элементов управления Windows Forms в WPF-приложения . . . 772
Встраивание PropertyGrid с помощью процедурного кода . . . 773
Встраивание элемента PropertyGrid с помощью XAML . . . 775
Встраивание элементов управления WPF в приложения Windows Forms . . . 777
Сочетание содержимого DirectX с содержимым WPF . . . 781
Встраивание элементов управления ActiveX в WPF-приложения . . . 788
Резюме . . . 792

20. Пользовательские и нестандартные элементы управления . . . 794
Создание пользовательского элемента управления . . . 796
Создание пользовательского интерфейса элемента управления . . . 796
Наделение пользовательского элемента управления поведением . . . 799
Включение в пользовательский элемент управления свойств зависимости . . . 802
Включение в пользовательский элемент управления маршрутизируемых событий . . . 804
Создание нестандартного элемента управления . . . 806
Программирование поведения нестандартного элемента . . . 806
Создание пользовательского интерфейса нестандартного элемента управления . . . 813
Некоторые соображения о более сложных элементах управления . . . 817
Резюме . . . 824

21. Компоновка с помощью нестандартных панелей . . . 825
Взаимодействие между родителями и потомками . . . 826
Этап измерения . . . 826
Этап размещения . . . 828
Создание панели SimpleCanvas . . . 830
Создание панели SimpleStackPanel . . . 834
Создание панели OverlapPanel . . . 837
Создание панели FanCanvas . . . 842
Резюме . . . 847

Алфавитный указатель . . . 848

Об авторе

Адам Натан – ведущий разработчик системы Microsoft Visual Studio, последняя версия которой представляет собой полноценное WPF-приложение. Ранее Адам был основателем, архитектором и разработчиком сайта Popfly, первого продукта корпорации Microsoft, построенного на базе технологии Silverlight, которая вошла в число 25 самых инновационных продуктов 2007 года по версии журнала *PCWorld Magazine*. Начав карьеру в составе коллектива разработчиков общеязыковой среды выполнения Microsoft (Common Language Runtime), Адам постоянно находился в гуще событий, связанных с созданием технологий .NET и WPF.

Многие сотрудники Microsoft и других компаний, занимающихся разработкой ПО, считают книги Адама обязательными для прочтения. Он автор бестселлера «WPF Unleashed» (Sams, 2006), который номинировался на премию Jolt Award в 2008 году, а также книг «Silverlight 1.0 Unleashed» (Sams, 2008) и «.NET and COM: The Complete Interoperability Guide» (Sams, 2002). Кроме того, Адам является одним из соавторов книг «ASP.NET: Tips, Tutorials, and Code» (Sams, 2001), «.NET Framework Standard Library Annotated Reference, Volume 2» (Addison-Wesley, 2005) и «Windows Developer Power Tools» (O'Reilly, 2006). Натан также создал сайт PINVOKE.NET и связанную с ним надстройку над Visual Studio. Связаться с Адамом можно через сайт *www.adamnathan.net* или по адресу @adamnathan в Twitter.

Посвящается

Ли ндсей, Тайлеру и Райану

Благодарности

Как всегда, я благодарю свою чудесную супругу Линдсей за невероятную поддержку и понимание. Нескончаемый процесс написания книг здорово сказывается на нашей жизни, и никто бы не удивился, если бы ее терпение наконец иссякло. Однако же никогда раньше ее поддержка не была столь ощутимой, как во время работы над этой книгой. Линдсей, что бы я без тебя делал!

Хотя создание любой книги, и этой в том числе, – по большей части глубоко личное занятие, она все же является плодом совместного труда многих талантливых и трудолюбивых людей. Не откажу себе в удовольствии назвать их поименно.

Я искренне благодарен Дуэйну Ниду (Dwayne Need), старшему менеджеру команды разработчиков WPF, – он потрясающий технический редактор. Его глубокие и проницательные рецензии на черновые варианты позволили значительно улучшить книгу. Выражаю признательность Роберту Хогу (Robert Hogue), Джо Кастро (Joe Castro) и Джордану Паркеру (Jordan Parker) за полезные отзывы. Дэвид Тейтельбаум (David Teitlebaum), специалист по трехмерной графике из команды разработчиков WPF, заслуживает самой горячей благодарности за согласие подкорректировать замечательную главу о 3D-графике, первоначально написанную Дэниелом Лехенбауэром (Daniel Lehenbauer). Ознакомиться с методологией и советами Дэниела и Дэвида – большая удача для любого читателя, подумывающего о том, чтобы заняться трехмерной графикой.

Хочется также поблагодарить следующих людей (в алфавитном порядке): Брайана Чэпмена (Brian Chapman), Беатрис де Оливейра Коста (Beatriz de Oliveira Costa), Эфианию Эчеруо (Ifeanyi Echeruo), Дэна Глика (Dan Glick), Нила

Кронлейга (Neil Kronlage), Рико Мариани (Rico Mariani), Майка Мюллера (Mike Mueller), Олега Овечкина, Лори Пирс (Lori Pearce), С. Рамини (S. Ramini), Роба Рилайи (Rob Relyea), Тима Райса (Tim Rice), Бена Ронко (Ben Ronco), Адама Смита (Adam Smith), Тима Снита (Tim Sneath), Дэвида Тредуэлла (David Treadwell) и Парамеша Вайдиянатана (Paramesh Vaidyanathan).

Я также выражаю признательность коллективу издательства Sams, а особенно Нилу Роуи (Neil Rowe) и Бетси Харрис (Betsy Harris), с которыми мне всегда приятно работать. Лучшей команды для подготовки книги не найти. Никто ни разу не сказал мне, что текст слишком длинный, или слишком короткий, или слишком отличается по стилю от типичной книги из серии «Подробное руководство». Мне предоставили свободу писать такую книгу, какую я хотел написать.

Спасибо также маме, папе и брату, которые раскрыли передо мной мир программирования, когда я еще учился в начальной школе. Если у вас есть дети, то посвятите их в магию создания программ, когда они еще прислушиваются к вашим словам! (А WPF и Silverlight помогут превратить этот опыт в незабываемое удовольствие!)

И наконец, спасибо вам за то, что вы взяли в руки эту книгу и прочитали ее хотя бы до этого места! Надеюсь, что вы на этом не остановитесь и для вас погружение в мир WPF 4 будет таким же завораживающим, как и для меня!



Нам важно ваше мнение!

Вы, читатель этой книги, – наш самый важный критик и комментатор. Мы ценим ваше мнение и хотим знать, что мы сделали правильно, что могли бы улучшить, на какие темы нам стоило бы выпускать книги. В общем, нам интересны любые мысли, которыми вы хотели бы с нами поделиться.

Вы можете писать мне по обычной или электронной почте о том, что понравилось или не понравилось в этой книге. А также о том, что мы могли бы еще сделать, чтобы наши книги стали лучше.

Пожалуйста, имейте в виду, что я не в состоянии ответить на технические вопросы по теме данной книги и что из-за большого количества получаемой почты я не всегда имею возможность ответить на каждое сообщение.

Если будете писать, не забудьте указать название и автора книги, а также свое имя и телефон или адрес электронной почты. Я внимательно изучу ваши замечания и направлю их автору и редакторам, работавшим над книгой.

Электронная почта: feedback@samspublishing.com

Почтовый адрес: Neil Rowe
Executive Editor
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

В помощь читателям

Посетите наш сайт и зарегистрируйте свой экземпляр книги по адресу *informit.com/register*, чтобы получить доступ к обновлениям, загружаемым материалам и перечню замеченных опечаток.

Введение

Благодарим за выбор книги «WPF 4. Подробное руководство»! Windows Presentation Foundation (WPF) – самая современная из предлагаемых корпорацией Microsoft технологий создания графических интерфейсов пользователя в ОС Windows, будь то простые формы, документо-ориентированные окна, анимированные изображения, видео, 3D-среды с эффектом погружения или все вышеперечисленное. Технология WPF позволяет разрабатывать самые разнообразные приложения проще, чем когда бы то ни было ранее. Кроме того, она лежит в основе технологии Silverlight, которая распространяет WPF на Сеть и мобильные устройства, например телефоны на базе ОС Windows.

С момента анонсирования WPF в 2003 году (под кодовым названием Avalon) эта технология привлекла к себе пристальное внимание благодаря революционному изменению процесса разработки ПО – особенно со стороны программистов Windows, привыкших к Windows Forms и GDI. WPF сравнительно легко позволяет создавать интересные и полезные приложения, демонстрирующие разнообразные возможности, которые трудно реализовать с помощью других технологий. В версии WPF 4, выпущенной в апреле 2010 года, существенно улучшены практически все аспекты этой технологии.

WPF знаменует собой отход от предшествующих технологий в плане модели программирования, основополагающих идей и базовой терминологии. Даже просмотр исходного кода WPF-приложения (например, путем декомпиляции его компонентов с помощью программы .NET Reflector или ей подобной) может стать источником сюрпризов, потому что интересующий вас код часто находится не там, где вы ожидаете. А если добавить сюда еще и тот факт, что любую задачу в WPF можно решить несколькими способами, то легко прийти к разделяемому многими выводу: *изучить WPF очень трудно*.

Вот тут-то и приходит на помощь эта книга. Когда WPF еще только разрабатывалась, было понятно, что не будет недостатка в книгах, посвященных этой технологии. Но лично меня беспокоило другое: смогут ли авторы соблюсти баланс между изложением самой технологии со всеми ее своеобразными идеями и демонстрацией использования ее на практике. Поэтому, работая над первым изданием этой книги, «Windows Presentation Foundation Unleashed», я ставил перед собой следующие цели:

- Познакомить читателя с базовыми концепциями в доступной форме, не покидая практическую почву

- Ответить на вопросы, возникающие у большинства изучающих технологию, и показать, как решаются типичные задачи
- Предложить авторитетный источник информации благодаря участию членов команды разработчиков WPF, которые проектировали, реализовывали и тестировали эту технологию
- Четко очертить границы применимости технологии, не делая вид, что она представляет собой решение всех проблем
- Предложить удобное справочное руководство, к которому можно возвращаться снова и снова

Успех первого издания превзошел самые смелые мои ожидания. Теперь, по прошествии четырех лет, я полагаю, что и во втором издании мне удалось достичь тех же целей, только с большей глубиной. Помимо освещения новых возможностей, появившихся в WPF 3.5, WPF 3.5 SP1 и WPF 4, я более подробно рассказываю о средствах, имевшихся еще в первой версии WPF. Надеюсь, что любой читатель – неважно, приступает он к изучению WPF впервые или имеет солидный опыт работы с этой технологией, – согласится, что книга отвечает всем заявленным критериям.

Предполагаемая аудитория

Эта книга адресована разработчикам, заинтересованным в создании пользовательских интерфейсов для Windows. Неважно, что именно вы разрабатываете: программы для бизнеса или для массового потребителя, повторно используемые элементы управления, – здесь вы найдете сведения, позволяющие извлечь максимум пользы из платформы. Книга написана так, что ее смогут понять даже читатели, совсем не знакомые с каркасом .NET Framework. Но и те, кто уверенно владеет WPF, тоже найдут интересную для себя информацию. Для них эта книга станет как минимум ценным справочным руководством.

Поскольку в основе WPF и Silverlight лежат одни и те же технология и концепции, то, прочитав эту книгу, вы заодно повысите свою квалификацию как разработчика приложений на платформе Windows Phone 7 и веб-приложений.

Хотя книга и не предназначена специально для графических дизайнеров, знакомство с ней поможет лучше понять, что на самом деле представляют собой такие продукты, как Microsoft Expression Blend.

Подведем итоги. В этой книге:

- Содержится все, что необходимо знать об основанном на XML языке eXtensible Application Markup Language (XAML) для декларативного создания пользовательских интерфейсов, допускающих применение стилей.
- Весьма детально рассматриваются различные функциональные возможности WPF: элементы управления, компоновка, ресурсы, привязка к данным, стили, графика, анимация и многое другое.

- Особое внимание уделено новейшим средствам, в том числе мультисенсорному вводу, усовершенствованной визуализации текста, новым элементам управления, дополнениям языка XAML, программе Visual State Manager, переходным кривым в анимации и т. д.
- Освещаются вопросы, не затрагиваемые в большинстве других книг: трехмерная графика, синтез и распознавание речи, документы, эффекты и пр.
- Демонстрируется создание популярных элементов пользовательского интерфейса, например галерей, экранных подсказок, нестандартных способов компоновки элементов.
- Демонстрируется создание более сложных механизмов организации пользовательского интерфейса, например выдвигающихся и стыкуемых панелей, как в Visual Studio.
- Объясняется, как писать и развертывать приложения любых типов, в том числе со встроенной навигацией, исполняемых в браузере и содержащих эффектные непрямоугольные окна.
- Описывается, как создавать полноценные элементы управления WPF.
- Демонстрируется создание гибридных приложений, в которых WPF сочетается с Windows Forms, DirectX, ActiveX и другими технологиями.
- Объясняется, как задействовать в WPF-приложении новые средства Windows 7, например списки переходов, и как обойти некоторые присущие WPF ограничения.

Нельзя сказать, что в этой книге описаны абсолютно все возможности WPF (в частности, вопросы спецификации XML Paper Specification (XPS) лишь слегка затронуты). Их так много, что в одной книге рассмотреть все, на мой взгляд, невозможно. Но думаю, что вам понравятся широта и глубина охвата материала.

Примеры, приведенные в книге, написаны на XAML и C#; при обсуждении вопросов интероперабельности встречается также код на C++/CLI. Повсеместное использование языка XAML объясняется рядом причин: зачастую это самый быстрый способ записать исходный код; фрагменты, написанные на XAML, можно копировать в инструментальные средства и видеть результат, не прибегая к компиляции; основанные на WPF инструменты генерируют код на XAML, а не на процедурных языках; наконец, XAML не зависит от того, на каком .NET-совместимом языке вы пишете: Visual Basic, C# или еще каком-то. В тех случаях, когда соответствие между XAML и C# неочевидно, приводятся эквивалентные представления кода на обоих языках.

Требования к программному обеспечению

В этой книге рассматриваются окончательная версия Windows Presentation Foundation 4.0, соответствующий пакет Windows SDK и Visual Studio 2010.

Должно быть установлено следующее программное обеспечение:

- Версия ОС Windows, поддерживающая .NET Framework 4.0, например: Windows XP с пакетом обновлений Service Pack 2 (включая Media Center, Tablet PC и версию x64), Windows Server 2003 с пакетом обновлений Service Pack 1 (включая версию R2), Windows Vista и более поздние версии ОС.
- Каркас .NET Framework 4.0, который устанавливается по умолчанию начиная с Windows Vista. Для предыдущих версий Windows его можно бесплатно загрузить с сайта *http://msdn.com*.

Кроме того, рекомендуется иметь следующее программное обеспечение:

- Пакет средств разработки Windows Software Development Kit (SDK) и прежде всего включенные в него средства для .NET. Его также можно бесплатно загрузить с сайта *http://msdn.com*.
- Visual Studio 2010 или более позднюю версию; подойдет и бесплатная версия Express, имеющаяся на сайте *http://msdn.com*.

Для поддержки графического дизайна в среде WPF очень полезно иметь комплект программ Microsoft Expression (конкретно Expression Blend).

Некоторые из включенных в книгу примеров ориентированы на системы Windows Vista, Windows 7 или компьютер с поддержкой мультисенсорного ввода, но в большинстве своем примеры будут работать во всех перечисленных выше версиях Windows.

Примеры кода

Исходный код всех примеров, встречающихся в этой книге, можно загрузить со страницы *http://informit.com/title/9780672331190* или *http://adamnathan.net/wpf*.

Организация материала

Книга состоит из шести частей, в которых последовательно излагается материал, необходимый для эффективного использования WPF. Но если вам не терпится забежать вперед и сразу перейти к конкретной теме, например трехмерной графике или анимации, то можно читать и не по порядку. Ниже кратко описано содержание каждой части.

Часть I «Базовые сведения»

Эта часть состоит из следующих глав:

- Глава 1 «Почему именно WPF и как насчет Silverlight?»
- Глава 2 «Все тайны XAML»
- Глава 3 «Основные принципы WPF»

В главе 1 WPF сопоставляется с альтернативными технологиями, чтобы вам было проще решить, отвечает ли она вашим нуждам. В главе 2 подробно рассматривается язык XAML с целью заложить фундамент для понимания

XAML-кода, который встретится вам в этой книге и в реальной практике. В главе 3 освещаются уникальные особенности модели программирования WPF, выходящие за пределы того, что уже известно программистам, работающим с .NET.

Часть II «Создание WPF-приложения»

Эта часть состоит из следующих глав:

- Глава 4 «Задание размера, положения и преобразований элементов»
- Глава 5 «Компоновка с помощью панелей»
- Глава 6 «События ввода: клавиатура, мышь, стилус и мультисенсорные устройства»
- Глава 7 «Структурирование и развертывание приложения»
- Глава 8 «Особенности Windows 7»

В части II вы узнаете, как собрать и развернуть традиционное приложение (хотя затрагиваются и некоторые дополнительные механизмы, например преобразования, непрямоугольные окна и технология Aero Glass). В главах 4 и 5 обсуждается компоновка элементов управления (и других элементов) в пользовательском интерфейсе программы. Глава 6 посвящена событиям ввода, в том числе поддержке новых устройств с мультисенсорным вводом. В главе 7 рассматриваются различные способы пакетирования и развертывания пользовательских интерфейсов на базе WPF для получения законченного приложения. В последней главе этой части речь пойдет об использовании некоторых возможностей Windows 7, позволяющих создавать приложения с современным внешним видом.

Часть III «Элементы управления»

Эта часть состоит из следующих глав:

- Глава 9 «Однодетные элементы управления»
- Глава 10 «Многодетные элементы управления»
- Глава 11 «Изображения, текст и другие элементы управления»

Часть III представляет собой обзор элементов управления, встроенных в WPF. Среди них много хорошо знакомых, но есть и несколько неожиданных. Две категории элементов управления – однодетные и многодетные[1] – настолько важные и глубокие темы, что заслуживают отдельных глав. Прочие элементы управления рассматриваются в главе 11.

---

[1] Термины «однодетный» и «многодетный элемент управления» (content control и items control) могут показаться непривычными, однако же какой-то эквивалент предложить необходимо. Термин «content control» буквально означает «элемент управления со свойством `Content`», а «items control» – «элемент управления со свойством `Items`». Вариант «элемент управления содержимым», встречающийся в локализованных продуктах Microsoft, совершенно не отражает сути дела. – *Прим. перев.*

Часть IV «Средства для профессиональных разработчиков»

Эта часть состоит из следующих глав:

- Глава 12 «Ресурсы»
- Глава 13 «Привязка к данным»
- Глава 14 «Стили, шаблоны, обложки и темы»

Средства, рассматриваемые в части IV, не относятся к активно используемым в WPF-приложениях, но их применение может существенно повысить качество процесса разработки. Они незаменимы для профессиональных разработчиков, серьезно относящихся к созданию надежных и удобных для сопровождения приложений или компонентов. Речь идет не столько о результатах, видимых конечному пользователю, сколько о рекомендуемых способах достижения желаемого результата.

Часть V «Мультимедиа»

Эта часть состоит из следующих глав:

- Глава 15 «Двумерная графика»
- Глава 16 «Трехмерная графика»
- Глава 17 «Анимация»
- Глава 18 «Аудио, видео и речь»

В этой части рассматриваются те возможности WPF, которые обычно вызывают наибольший интерес. Поддержка двумерной и трехмерной графики, анимации, видео и пр. позволяет создавать приложения, поражающие воображение пользователя. Именно эти средства наряду со способами их использования и отличают WPF от предшествующих технологий. WPF снижает барьеры, стоящие на пути включения такого содержимого в приложения, позволяя браться за задачи, о которых раньше вы и помыслить не могли!

Часть VI «Дополнительные вопросы»

Эта часть состоит из следующих глав:

- Глава 19 «Интероперабельность с другими технологиями»
- Глава 20 «Пользовательские и нестандартные[1] элементы управления»
- Глава 21 «Компоновка с помощью нестандартных панелей»

В части VI рассматриваются вопросы, интересные для разработчиков более сложных WPF-приложений и элементов управления. Уже имеющиеся элементы управления WPF допускают применение стилей в очень широких пределах, поэтому потребность в создании дополнительных элементов не так насущна.

---

[1] В терминологии Microsoft – «настраиваемые». – *Прим. перев.*

Типографские соглашения

В этой книге новые термины и иные специальные элементы выделяются с помощью шрифтов, а именно:

Шрифт	Назначение
Курсив	Используется при первом определении термина и иногда для акцентирования внимания, а также для имен файлов и интернет-адресов
Моноширинный	Используется для записи выводимых на экран сообщений, листингов и команд кода. В листингах *курсивное моноширинное начертание* применяется для обозначения заменяемого текста

В книге встречаются следующие виды врезок.

FAQ

Что такое врезка FAQ?

В такой врезке формулируется вопрос, который может возникнуть у читателя в данном месте, и дается краткий ответ на него.

КОПНЕМ ГЛУБЖЕ

Врезки «Копнем глубже»

В такой врезке представлена более подробная информация по теме в дополнение к содержащейся в основном тексте. Можно сказать, что это добавочные сведения для особо любознательных.

СОВЕТ

Это описание приемов, которые могут пригодиться на практике, например самый быстрый способ достижения цели или альтернативный подход, дающий более качественный результат либо позволяющий решить задачу скорее и проще.

ПРЕДУПРЕЖДЕНИЕ

Такие врезки привлекают внимание к действию или условию, способному привести к неожиданному либо непредсказуемому результату, – с объяснением того, как избежать подобных последствий.

I

Базовые сведения

Глава 1 «Почему именно WPF и как насчет Silverlight?»
Глава 2 «Все тайны XAML»
Глава 3 «Основные принципы WPF»

- Взгляд в прошлое
- Появление WPF
- Эволюция WPF
- Что такое Silverlight

1

Почему именно WPF и как насчет Silverlight?

В кино и на телевидении главные герои обычно не похожи на обычных людей, которые встречаются нам в повседневной жизни. Они внешне более привлекательные, обладают мгновенной реакцией и почему-то всегда точно знают, что делать дальше. То же самое можно сказать и про компьютерные программы, которые показывают в фильмах.

Впервые меня это поразило в 1994 году при просмотре фильма «Disclosure» (Разоблачение) с Майклом Дугласом и Деми Мур. Почтовая программа, которой они пользовались, выглядела совершенно не так, как Microsoft Outlook! По ходу фильма мы дивились различным визуальным эффектам: вращающееся трехмерное «е»; сообщения, которые разворачиваются при открытии и комкаются при удалении; намеки на поддержку рукописного ввода и симпатичная анимация при распечатке сообщения. (Эта почтовая программа еще не самая нереалистичная из встречающихся в фильме. Достаточно лишь вспомнить «базу данных виртуальной реальности».)

Голливуд уже давно говорит нам, что реальные программы вовсе не такие впечатляющие, какими должны быть, и речь здесь идет отнюдь не о функциональности. Вы, наверное, и сами сможете вспомнить несколько примеров забавных и фантастичных программ из известных фильмов и сериалов. Однако в последние годы реальные программы стали подтягиваться к голливудским стандартам! Это наблюдается и в традиционных операционных системах (да, и в Windows тоже), и в веб-приложениях, и в ПО для таких устройств, как iPhone, iPad, Zune, TiVo, Wii, Xbox, Windows Phone и многих-многих других. Пользователи ожидают от программ большего, а компании-производители тратят массу времени и денег, чтобы превзойти конкурентов в области разработки пользовательского интерфейса. И это касается не только программ, рассчитанных на массового потребителя. Даже бизнес-приложения

и инструменты для внутреннего использования могут здорово выиграть от улучшения интерфейса.

Однако при возрастании требований к пользовательскому интерфейсу традиционного подхода и старых технологий разработки приложений часто оказывается недостаточно. Современные программы обычно нуждаются в быстром и кардинальном изменении интерфейса по инициативе различных сторон: профессиональных дизайнеров, проектировщиков пользовательских интерфейсов или начальства, которое хочет, чтобы приложение выглядело более эффектно и включало анимацию. Но для этого необходима технология, позволяющая естественным образом отделить пользовательский интерфейс от реализации приложения, а визуальное поведение – от внутренней программной логики. У разработчиков должна быть возможность создавать внешне аскетичные, но вместе с тем полнофункциональные приложения, которые впоследствии могут быть красиво оформлены дизайнерами без привлечения программистов. Однако присущий Win32 стиль программирования, при котором элементы управления содержат код собственной визуализации, как правило, сильно затрудняет быструю смену интерфейса.

В 2006 году корпорация Microsoft выпустила в свет технологию, которая позволила разработчикам создавать приложения XXI века, отвечающие возросшим требованиям. Она называется Windows Presentation Foundation (WPF). С выходом версии WPF 4 в 2010 году эта технология позволила добиваться еще более впечатляющих результатов при разработке практически любых программ. Всего через десять лет после того, как Том Круз поспособствовал популяризации идеи компьютера с мультисенсорным интерфейсом ввода в фильме «Особое мнение» (Minority Report), и после реализации такого интерфейса в самых разных устройствах (из которых наиболее известен iPhone), WPF 4 и Windows 7 его применение стало массовым. Голливуду пора придумывать что-нибудь новенькое!

Взгляд в прошлое

Базовые технологии большинства интерфейсов в Windows – интерфейс графического устройства (Graphics Device Interface, GDI) и подсистема USER – появились в Windows 1.0 еще в 1985 году. В мире технологий это смело можно назвать доисторическим периодом! В начале 1990-х годов компания Silicon Graphics разработала ставшую популярной графическую библиотеку OpenGL для двумерной и трехмерной графики как в Windows, так и в других системах. Она была с восторгом принята компаниями, работающими в сфере создания систем автоматизированного проектирования, программ визуализации научных данных и игр. Технология Microsoft DirectX, представленная в 1995 году, обеспечила высокоскоростную альтернативу для 2D-графики, ввода, сетевого взаимодействия, работы со звуком, а со временем и 3D-графики (которая стала возможной с версией DirectX 2, вышедшей в 1996 году).

Впоследствии и в GDI, и в DirectX было внесено много существенных улучшений. Например, технология GDI+, представленная в Windows XP, добавила

поддержку прозрачности и градиентные кисти. Однако ввиду большой сложности и в отсутствие аппаратного ускорения она работает медленнее, чем GDI. Что касается технологии DirectX (кстати, используемой в Xbox), то постоянно выходят новые версии, раздвигающие пределы возможностей компьютерной графики. После появления каркаса .NET и управляемого кода (в 2002 году) разработчики получили очень продуктивную модель для создания Windows- и веб-приложений. Включенная в нее технология Windows Forms (основанная на GDI+) стала основным способом создания пользовательских интерфейсов в Windows для разработчиков на C#, Visual Basic и (в меньшей степени) C++. Она пользовалась успехом и оказалась весьма продуктивной, но имела фундаментальные ограничения, уходящие корнями в GDI+ и подсистему USER.

Начиная с версии DirectX 9 Microsoft стала поставлять эту систему для управляемого кода (подобно тому, как в прошлом поставлялись библиотеки специально для Visual Basic), которая впоследствии была заменена каркасом XNA Framework. Хотя это и позволило разработчикам на C# использовать DirectX без многих проблем, связанных с интероперабельностью .NET и COM, однако работать с управляемыми каркасами оказалось не намного проще по сравнению с неуправляемыми альтернативами. Исключение составляет только разработка игр в среде XNA Framework, поскольку она включает в себя специализированные для этой цели библиотеки и работает с такими мощными инструментами, как XNA Framework Content Pipeline и XNA Game Studio Express.

Поэтому, хотя разработать в Windows почтовую программу с 3D-эффектами (как в фильме «Разоблачение») можно было уже в середине 90-х годов с помощью альтернативных GDI технологий (фактически комбинируя DirectX или OpenGL с GDI), на практике этот способ очень редко применялся даже и десять лет спустя. На то было несколько причин: аппаратное обеспечение, позволяющее достичь нужных результатов, было не так распространено вплоть до последнего времени; работать с альтернативными технологиями на порядок сложнее; и к тому же использование GDI считалось «вполне приемлемым».

Графические подсистемы компьютеров продолжали совершенствоваться и дешеветь, ожидания потребителей росли, но до появления WPF проблеме сложности построения выразительных пользовательских интерфейсов не уделяли должного внимания. Некоторые разработчики самостоятельно брались за ее решение, стремясь сделать свои приложения и элементы управления более привлекательными. Простым примером тут является использование растровых изображений вместо стандартных кнопок. Однако мало того что подобные нестандартные решения было трудно реализовывать, они еще зачастую оказывались ненадежными. Основанные на них приложения не всегда доступны людям с ограниченными возможностями, плохо адаптируются к высокому разрешению и имеют другие визуальные огрехи.

Появление WPF

Корпорация Microsoft понимала, что необходимо нечто совершенно новое, свободное от ограничений GDI+ и подсистемы USER, но не менее продуктивное и удобное в использовании, чем Windows Forms. И с учетом роста популярности кроссплатформенных приложений, основанных на HTML и JavaScript, Windows отчаянно нуждалась в столь же простой технологии, которая при этом еще и позволяла бы задействовать все возможности локального компьютера. И Windows Presentation Foundation (WPF) дала в руки разработчиков ПО и графических дизайнеров тот инструмент, который был необходим для создания современных решений и не требовал освоения сразу нескольких сложных технологий. И хотя слово Presentation (представление) – всего лишь высокопарный синоним привычного «пользовательского интерфейса», возможно, оно лучше отражает более высокий уровень визуального совершенства, которого ждут от современных приложений, равно как и обширную новую функциональность, включенную в WPF!

Перечислим основные возможности, которые предоставляет WPF.

- **Широкая интеграция.** До WPF разработчикам в Windows, которые хотели использовать одновременно 3D-графику, видео, речь и форматированные документы в дополнение к обычной двумерной графике и элементам управления, приходилось изучать несколько независимых технологий, плохо совместимых между собой и не имеющих встроенных средств сопряжения. А в WPF все это входит в состав внутренне согласованной модели программирования, поддерживающей композицию и визуализацию разнородных элементов. Одни и те же эффекты применимы к различным видам мультимедийной информации, а один раз освоенная техника может использоваться и для других целей.
- **Независимость от разрешающей способности.** Только представьте себе мир, в котором переход к более высокому разрешению экрана или принтера не означает, что все уменьшается. Вместо этого графические элементы и текст только становятся более четкими! Представьте себе пользовательский интерфейс, который прекрасно выглядит и на маленьком нетбуке, и на полутораметровом экране телевизора! WPF все это обеспечивает и дает возможность уменьшать или увеличивать элементы на экране независимо от его разрешения. Это стало возможным благодаря тому, что WPF основана на использовании векторной графики.
- **Аппаратное ускорение.** Поскольку WPF основана на технологии Direct3D, то все содержимое в WPF-приложении, будь то двумерная или трехмерная графика, изображения или текст, преобразуется в трехмерные треугольники, текстуры и другие объекты Direct3D, а потом отрисовывается аппаратной графической подсистемой компьютера. Таким образом, WPF-приложения задействуют все возможности аппаратного ускорения графики, что позволяет добиться более качественного изображения и одновременно повысить производительность (поскольку часть работы перекладывается с центральных процессоров на графические). При этом от применения

новых графических ускорителей и их драйверов выигрывают все WPF-приложения (а не только высококлассные игры). Но WPF *не требует* обязательного наличия высокопроизводительной графической аппаратуры. В ней имеется и собственный программный конвейер визуализации. Это позволяет использовать возможности, которые пока еще не поддерживаются аппаратно (например, осуществлять высокоточное отображение любого содержимого на экране). Программная реализация используется и как запасной вариант в случае отсутствия аппаратных ресурсов (например, если в системе стоит устаревшая графическая карта, или карта современная, но GPU не хватает ресурсов, скажем, видеопамяти).

- **Декларативное программирование.** Декларативное программирование не является уникальной особенностью WPF, поскольку в программах на платформе Win16/Win32 сценарии описания ресурсов для определения компоновки диалоговых окон и меню применяются вот уже 25 лет. И в .NET-приложениях часто используются декларативные атрибуты наряду с конфигурационными и ресурсными XML-файлами. Однако в WPF применение декларативного программирования вышло на новый уровень благодаря языку XAML (eXtensible Application Markup Language – расширяемый язык разметки *приложений*) (произносится «заммел»). Сочетание WPF и XAML аналогично использованию HTML для описания пользовательского интерфейса, но с гораздо более широкими выразительными возможностями. И эта выразительность выходит далеко за рамки описания интерфейса. В WPF язык XAML применяется в качестве формата документов, для представления 3D-моделей и многого другого. В результате дизайнер может непосредственно влиять на внешний вид приложения и некоторые особенности его поведения; раньше для этого, как правило, приходилось писать код. В следующей главе мы будем рассматривать XAML подробно.
- **Богатые возможности композиции и настройки.** В WPF элементы управления могут сочетаться немыслимыми ранее способами. Можно создать комбинированный список, содержащий анимированные кнопки, или меню, состоящее из видеоклипов! Конечно, сама мысль о таком интерфейсе может привести в ужас, но важно то, что для оформления элемента способом, о котором его автор и не помышлял, не понадобится писать никакой (!) код (и в этом коренное отличие от предшествующих технологий, где отрисовка элементов жестко задавалась создателем кода). Кроме того, отметим, что WPF позволяет безо всякого труда радикально изменять обложку (скин) приложения (мы рассмотрим этот вопрос в главе 14 «Стили, шаблоны, обложки и темы»).

Короче говоря, цель WPF – соединить в себе все лучшее, имеющееся в DirectX (трехмерная графика и аппаратное ускорение), Windows Forms (продуктивность разработки), Adobe Flash (развитая поддержка анимации) и HTML (декларативная разметка). Надеюсь, эта книга убедит вас в том, что WPF повышает производительность труда, дает больше возможностей и более увлекательна, чем любая другая технология, с которой вам доводилось работать прежде!

КОПНЕМ ГЛУБЖЕ

GDI и аппаратное ускорение графики

На самом деле в технологии GDI в Windows XP тоже использовалось аппаратное ускорение графики. Модель драйвера видеокарты явно поддерживает ускорение наиболее распространенных операций GDI. В Windows Vista реализована новая модель драйвера видеокарты без аппаратного ускорения примитивов GDI. Вместо этого применяется программная реализация устройства канонического отображения для поддержки операций GDI в унаследованных драйверах. Однако в Windows 7 восстановлено частичное аппаратное ускорение для примитивов GDI.

FAQ

Позволяет ли WPF сделать что-то, чего нельзя было сделать ранее?

Если быть совсем точным, то нет. Точно так же ни C#, ни .NET не позволяют сделать ничего, что нельзя было бы реализовать на языке ассемблера. Вопрос лишь в том, сколько времени и сил потребуется для достижения желаемого результата!

Если вы попытаетесь создать эквивалент WPF-приложения с нуля без использования WPF, то придется не только заниматься отрисовкой пикселов на экране и взаимодействием с устройствами ввода, но также проделать массу дополнительной работы для поддержки доступности и локализации, тогда как в WPF все это уже встроено. Кроме того, WPF обеспечивает простой способ задействовать все возможности Windows 7, например определить списки перехода с помощью коротенького кода на XAML (см. главу 8 «Особенности Windows 7»).

Поэтому я полагаю, что с учетом времени и финансовых затрат большинство людей утвердительно ответят на поставленный вопрос.

FAQ

Когда следует использовать DirectX вместо WPF?

DirectX больше подходит для разработчиков зрелищных игр или приложений со сложными 3D-моделями, в которых требуется максимальная производительность. Отметим, однако, что очень легко написать такое приложение DirectX, которое будет работать гораздо медленнее аналогичного WPF-приложения.

DirectX – это низкоуровневый интерфейс к графическому оборудованию, который делает явными все особенности конкретного графического процессора. DirectX можно назвать «языком ассемблера в мире графики»: вы можете делать все, что поддерживает данный GPU, но учитывать капризы разнообразной аппаратуры придется самостоятельно. Это трудно, зато такой низкоуровневый интерфейс позволяет опытным разработчикам достичь желаемого компромисса между высоким качеством графики и скоростью работы. Кроме того, DirectX позволяет работать с последними достижениями в области GPU, а они появляются гораздо быстрее, чем новые версии WPF.

С другой стороны, WPF обеспечивает более высокий уровень абстракции. Вы передаете системе описание сцены, а она уже сама решает, как оптимально визуализировать ее с учетом имеющихся аппаратных ресурсов. (Это система, работающая в *режиме запоминания*, а не в *режиме непосредственной визуализации*.) В WPF основное внимание уделено двумерной графике, а трехмерная графика ограничена сценариями визуализации данных и интеграцией с 2D без претензии на полную поддержку всех возможностей DirectX.

Однако, отдавая предпочтение DirectX, следует иметь в виду потенциально астрономическое увеличение стоимости разработки. По большей части затраты связаны с необходимостью тестировать приложение для всех возможных комбинаций драйверов и GPU, которые вы намереваетесь поддерживать. Одно из основных преимуществ построения приложения на базе WPF состоит в том, что Microsoft уже провела такое тестирование за вас! Вы же можете сосредоточиться на измерении производительности, для чего достаточно относительно дешевой системы. А тот факт, что WPF-приложение способно использовать установленный на компьютере пользователя GPU даже в условиях частичного доверия, – еще один аргумент в пользу выбора этой технологии.

Отметим также, что WPF и DirectX можно использовать совместно в одном приложении. В главе 19 «Интероперабельность с другими технологиями» описано, как это делается.

Эволюция WPF

Как ни странно, WPF 4 действительно является четвертой основной версией WPF. Странность в том, что первый выпуск имел номер 3.0! Он увидел свет в ноябре 2006 года и получил название WPF 3.0, потому что вошел в состав каркаса .NET Framework 3.0. Второй выпуск – WPF 3.5 – состоялся почти год спустя (если быть точным, то на день раньше). Третья основная версия вышла еще через год (в августе 2008 года). Она была включена в пакет обновлений Service Pack 1 (SP1) для .NET 3.5, но в части WPF это был не обычный пакет обновлений, поскольку появилось много новых возможностей и улучшений.

Кроме основных версий в августе 2008 года на сайте *http://wpf.codeplex.com* Microsoft представила набор инструментов WPF Toolkit, который содержит многочисленные инструментальные средства и примеры использования и обновляется несколько раз в год. WPF Toolkit предназначен для ускоренного внедрения новых возможностей, правда, в экспериментальной форме (и зачастую вместе с исходным кодом). Нередко средства, впервые появившиеся в WPF Toolkit, со временем включаются в новые версии WPF – в зависимости от мнения пользователей об их желательности и степени готовности.

На момент выпуска первой версии для WPF не существовало практически никакой инструментальной поддержки. В последующие месяцы были написаны примитивные WPF-расширения для Visual Studio 2005, а затем состоялся первый выпуск Expression Blend для публичного ознакомления. Теперь же среда Visual Studio 2010 включает не только полноценную поддержку WPF, но и сама была существенно переписана и ныне является WPF-приложением!

Программа Expression Blend, полностью написанная средствами WPF, также получила множество новых функций для проектирования и создания прототипов замечательных пользовательских интерфейсов. Кроме того, за последние несколько лет немало WPF-приложений было выпущено такими компаниями, как Autodesk, SAP, Disney, Blockbuster, Roxio, AMD, Hewlett Packard, Lenovo и многими другими. Сама корпорация Microsoft, конечно же, тоже разработала многочисленные приложения, основанные на WPF (Visual Studio, Expression, Test and Lab Manager, Deep Zoom Composer, Songsmith, Surface, Semblio, Robotics Studio, LifeCam, Amalga, Games for Windows LIVE Marketplace, Office Communicator Attendant, Active Directory Administrative Center, Dynamics NAV, Pivot, PowerShell ISE и многие другие).

СОВЕТ

Чтобы узнать, какие WPF-элементы применяются в конкретном WPF-приложении, можно воспользоваться программой Snoop, представленной на сайте *http://snoopwpf.codeplex.com*.

Давайте более подробно рассмотрим, как WPF менялась со временем.

Усовершенствования в WPF 3.5 и WPF 3.5 SP1

В версиях WPF 3.5 и 3.5 SP1 произошли следующие существенные изменения:

- **Интерактивная 3D-графика** – интеграция двумерной и трехмерной графики улучшилась благодаря базовому классу `UIElement3d`, который позволил 3D-элементам принимать данные, получать фокус клавиатуры и события; классу со странным названием `Viewport2DVisual3D`, который позволил помещать любой интерактивный 2D-элемент управления на 3D-сцену; и другим нововведениям. Подробнее см. в главе 16 «Трехмерная графика».
- **Полноценная интероперабельность с DirectX** – ранее WPF-приложение могло взаимодействовать с DirectX только на уровне общей для обеих технологий платформы Win32. Теперь с помощью класса `D3DImage` можно работать непосредственно с поверхностью Direct3D, а не с ее описателями `HWND`. Среди прочего это позволяет размещать WPF-содержимое поверх DirectX-содержимого и наоборот. См. главу 19.
- **Улучшенная привязка к данным** – в WPF появилась поддержка технологии привязки XLINQ, улучшены способы контроля данных и отладки, а также форматирование выводимых строк средствами XAML, что позволяет в ряде случаев обойтись без написания процедурного кода. См. главу 13 «Привязка к данным».
- **Улучшенные спецэффекты** – уже самая первая версия WPF поставлялась с полезными спецэффектами для рисунков (размывание, тени, внешнее свечение, выдавливание и выпуклость), но пользоваться ими не рекомен-

довалось из-за крайне низкой производительности. Теперь все изменилось – добавлен новый набор эффектов с поддержкой аппаратного ускорения, а также реализована совершенно иная архитектура, позволяющая подключать собственные эффекты с аппаратным ускорением с помощью пиксельных построителей текстур (pixel shaders). См. главу 15 «Двумерная графика».

- **Высокопроизводительное произвольное рисование** – раньше в WPF не имелось хорошего механизма произвольного рисования, масштабируемого на тысячи точек или фигур, поскольку даже примитивы самого низкого уровня были для этого слишком медленны. Теперь класс `WriteableBitmap` модернизирован таким образом, что при рисовании можно указывать изменившиеся области, а не обновлять весь растр в каждом кадре! Впрочем, поскольку этот класс позволяет только задавать отдельные пикселы, то рисованием такой процесс можно назвать с большой натяжкой.
- **Улучшения в области обработки текста** – повышена производительность, улучшена поддержка интернационализации (усовершенствован редактор методов ввода (IME) и улучшена поддержка индийских языков). Модернизации подверглись также классы `TextBox` и `RichTextBox`. См. главу 11 «Изображения, текст и другие элементы управления».
- **Модернизация приложений с частичным доверием** – .NET-приложениям с частичным доверием стало доступно гораздо больше функциональности, например возможность обращаться к WCF (Windows Communication Foundation) для вызова веб-служб (с привязкой `basicHttpBinding`) и возможность читать и записывать файлы cookie. Кроме того, технология XAML Browser Applications (XBAPs) – основной механизм запуска WPF-приложений с частичным доверием, – ранее доступная только для Internet Explorer, теперь распространена и на браузер Firefox (но необходимая для этого надстройка больше не устанавливается по умолчанию).
- **Улучшенное развертывание приложений и каркаса .NET** – проявляется в различных формах: ускорение процедуры установки и уменьшение объема .NET за счет внедрения технологии «клиентского профиля», которая позволяет исключить части .NET, нужные только для серверов (например, ASP.NET); новый компонент-«загрузчик», который обрабатывает зависимости .NET, ранее установленные компоненты и появившиеся обновления и позволяет осуществлять установку нестандартных дистрибутивов; а также разнообразные новые возможности в технологии ClickOnce.
- **Улучшенная производительность** – в WPF и в общеязыковой среде исполнения реализован ряд изменений, которые существенно повысили скорость исполнения WPF-приложений без каких либо изменений в коде. Например, заметно уменьшилось время загрузки приложения (особенно первой), анимация (особенно медленная) стала гораздо более плавной, привязка данных в некоторых ситуациях начала работать быстрее, а полупрозрачные окна (описанные в главе 8) теперь поддерживают аппаратное ускорение. Имеются и другие усовершенствования в области производительности, которые пользователь должен включать самостоятельно из-за ограничений

совместимости, например улучшенная виртуализация и отложенная прокрутка в многодетных элементах управления (см. главу 10 «Многодетные элементы управления»).

Усовершенствования в WPF 4

В версию WPF 4 дополнительно внесены следующие изменения:

- **Поддержка мультисенсорного ввода** – на компьютерах с поддержкой мультисенсорного ввода, работающих под управлением ОС Windows 7 или более поздней, элементы WPF могут получать различные события ввода: низкоуровневые данные, простые преобразования (например, поворот и масштабирование) или высокоуровневые (в том числе нестандартные) жесты. Все встроенные в WPF элементы управления модернизированы для работы с мультисенсорными устройствами ввода. Разработчики WPF воспользовались результатами работы над проектом Microsoft Surface (который, в свою очередь, основан на WPF). В итоге поддержка мультисенсорного ввода в WPF 4 совместима с версией 2 Surface SDK, что стало отличной новостью для всех, кто собирался разрабатывать приложения для Windows и Surface. См. главу 6 «События ввода: клавиатура, мышь, стилус и мультисенсорные устройства».
- **Полноценная поддержка прочих возможностей Windows 7** – мультисенсорный ввод – конечно, ценное добавление в Windows 7, но есть и много других, не требующих наличия специального оборудования и, следовательно, доступных широкому кругу пользователей. WPF обеспечивает оптимальный способ интеграции приложений с такими новыми возможностями панели задач, как списки переходов (Jump List) и многослойные значки (icon overlays), а также с обновленными стандартными диалоговыми окнами и многими другими нововведениями. См. главу 8.
- **Новые элементы управления** – WPF 4 включает такие элементы управления, как `DataGrid`, `Calendar` и `DatePicker`, которые первоначально появились в WPF Toolkit. См. главу 11.
- **Новые функции для анимации переходов** – появилось одиннадцать новых классов анимации, в том числе `BounceEase`, `ElasticEase` и `SineEase`, которые позволяют декларативно задавать замысловатые траектории анимации с настраиваемым ускорением и замедлением. Эти «переходные функции» и поддерживающая их инфраструктура впервые были представлены в Silverlight 3 и только потом вошли в WPF 4.
- **Улучшенная стилизация с помощью Visual State Manager** – менеджер визуальных состояний впервые появился в Silverlight 2; он предлагает новый способ организации визуальных элементов и их интерактивного поведения: в виде «визуальных состояний» и «переходов между состояниями». Это упрощает дизайнерам работу с элементами управления в таких инструментальных средах, как Expression Blend, а также позволяет создавать общие шаблоны для WPF и Silverlight.

- **Улучшенное поведение на границе пикселов** – WPF всегда стремилась соблюдать баланс между независимостью от разрешающей способности устройства (для чего требуется игнорировать границы физических пикселов) и обеспечением четкости визуальных элементов (что предполагает привязку к границам пикселов, особенно для мелких элементов). С самого начала WPF поддерживала свойство `SnapsToDevicePixels`, позволяющее принудительно осуществлять «привязку элементов к пикселам». Но использовать его было довольно сложно, а в некоторых случаях оно не давало никакого эффекта. Тогда в Silverlight был сделан шаг назад, в направлении обычной чертежной доски, и реализовано свойство `UseLayoutRounding`, которое работало более естественным образом. Теперь это свойство появилось и в WPF 4. Если задать его равным `true` для корневого элемента, то координаты этого элемента и всех его потомков будут округляться (с недостатком или с избытком), так чтобы они совпали с границами ближайших пикселов. В результате пользовательский интерфейс сохраняет способность к масштабированию, оставаясь при этом четким!
- **Более четкий текст** – стремление WPF обеспечить независимость от разрешающей способности устройства и масштабируемость интерфейса всегда терпело неудачу при отображении небольших фрагментов текста, которые превалируют в традиционных интерфейсах на экранах с разрешением 96 точек на дюйм (DPI). Это очень огорчало многих пользователей и разработчиков. Я даже говорил, что всегда смогу отличить интерфейс, созданный с помощью WPF, просто по размытости текста. В WPF 4 разработчики наконец-то решили эту проблему, предложив альтернативный способ визуализации текста, при котором текст выглядит так же четко, как выведенный с помощью GDI, но с сохранением почти всех преимуществ WPF. Например, этот режим используется в Visual Studio 2010 для отображения текстовых документов. Но поскольку новый способ визуализации имеет ряд ограничений, то включать этот режим следует явно. См. главу 11.
- **Усовершенствования в области развертывания приложений** – теперь клиентский профиль .NET может устанавливаться на одной машине с полным каркасом .NET и использоваться почти во всех сценариях, характерных для WPF-приложений. На самом деле проекты для .NET 4.0 в Visual Studio 2010 по умолчанию ориентированы на более компактный клиентский профиль.
- **Дальнейшее повышение производительности** – для максимального ускорения векторной графики WPF может кэшировать результаты рендеринга в виде растровых изображений и затем использовать их повторно. Этим поведением можно управлять с помощью свойства `CacheMode`; см. главу 15. Стимулом для многочисленных улучшений в области производительности послужило то, что WPF активно используется в Visual Studio 2010, но ощутить эффект теперь могут все WPF-приложения.

FAQ

Что будет добавлено в WPF после версии 4?

Во время написания этой книги никаких анонсов еще не было, но без особого риска можно предположить, что повышение производительности и дальнейшее сближение с Silverlight по-прежнему будут оставаться в центре внимания разработчиков WPF. Дополнительным источником информации о том, что может быть включено в ядро, служит WPF Toolkit. Речь может идти об элементах управления для построения графиков, а также об элементах `BreadcrumbBar`, `NumericUpDown` и др.

FAQ

Зависит ли поведение WPF от версии Windows?

Среди прочего в WPF реализованы API, которые относятся только к Windows 7 (или более поздним версиям), в частности мультисенсорный ввод и другие функции, описанные в главе 8. Кроме того, WPF ведет себя несколько иначе при запуске в Windows XP (самой старой версии Windows, которую поддерживает WPF). Например, не производится сглаживание для 3D-объектов.

И, конечно же, для элементов управления WPF по умолчанию применяются разные темы – в зависимости от операционной системы (Aero в Windows Vista и Windows 7, Luna в Windows XP).

Кроме того, в Windows XP используется старая модель драйверов, что может негативно сказаться на работе WPF-приложений. Модель драйверов в последних версиях Windows подразумевает виртуализацию и распределение ресурсов графических процессоров, что повышает общую производительность системы при исполнении нескольких программ, активно работающих с графикой. Запуск нескольких приложений WPF или DirectX в Windows XP может затормозить систему, но в более поздних версиях Windows таких проблем быть не должно.

Что такое Silverlight

Silverlight – это компактная, облегченная версия каркаса .NET, рассчитанная на насыщенные веб-приложения (в качестве альтернативы Adobe Flash и Flex). Подход Silverlight к созданию пользовательских интерфейсов такой же, как WPF, что дает массу преимуществ. Первая версия Silverlight вышла в 2007 году, а теперь, как и WPF, она находится на уровне четвертой версии. Silverlight 4 была выпущена в апреле 2010 года, через несколько дней после выхода WPF 4.

Взаимоотношения между WPF и Silverlight выглядят несколько запутанно, как и вопрос о том, когда применять одну технологию, а когда другую. Еще больше осложняет ситуацию тот факт, что WPF-приложения можно запускать в браузере (с помощью технологии XBAPs), то есть они практически «готовы

для Сети». И наоборот, приложения Silverlight можно запускать вне браузера, даже в автономном режиме.

По существу, Silverlight является подмножеством WPF и дополнительно включает несколько фундаментальных классов из каркаса .NET Framework (встроенные типы данных, классы коллекций и т. д.). Каждая новая версия Silverlight содержит все больше функциональности WPF. И хотя совместимость с WPF и полным каркасом .NET по-прежнему является целью разработчиков Silverlight, они не упустили возможность учесть ошибки, допущенные при создании WPF и .NET. Внесены некоторые изменения и поддерживаются возможности, которые еще не вошли в полный каркас .NET. Кое-какие изменения и дополнения впоследствии были включены в состав WPF и .NET Framework (например, Visual State Manager и привязка элементов к границам пикселов), другие – нет (видеокисти и построение перспективы). И наоборот, в WPF и .NET есть такие функции, которые Silverlight, вероятно, никогда не будет поддерживать.

Короче говоря, следует задавать не вопрос «Что использовать: WPF или Silverlight?», а вопрос «Использовать полный каркас .NET или облегченный?». Если требуется функциональность, которая существует только в полной версии .NET, то выбор очевиден. В таком случае рекомендуется использовать WPF. Напротив, если необходима возможность выполнять приложение на компьютерах Mac или устройствах, отличных от стандартного ПК, то ответ тоже ясен – выбирайте Silverlight. На любой платформе Silverlight обеспечивает единую технологию построения интерфейса (хотя прекрасно взаимодействует с HTML). Во всех остальных случаях выбор зависит от природы приложения и целевой аудитории.

В идеале было бы желательно отложить решение о выборе конкретной технологии. Хотелось бы использовать один и тот же исходный код (и даже скомпилированные двоичные файлы) и иметь возможность легко адаптировать приложение к параметрам устройства, на котором оно выполняется, – будь то мобильное устройство либо обычный ПК под управлением Windows или Mac. Возможно, когда-нибудь так и будет, но пока создание единого кода, который работал бы и в WPF, и в Silverlight, требует дополнительной работы. Обычно создают совместимый с Silverlight код, в котором части, специфичные для WPF, обрамлены директивами `#ifdef`. Это позволяет компилировать разные версии приложения для Silverlight и для WPF, сводя расхождения в исходном коде к минимуму.

Лично я ожидаю (и надеюсь), что различия между WPF и Silverlight со временем будут стираться. Конечно, Silverlight звучит гораздо выразительнее, чем Windows Presentation Foundation, но само наличие двух названий порождает определенные проблемы и искусственные различия. Гораздо продуктивнее рассматривать Silverlight и WPF как две реализации одной и той же базовой технологии. На самом деле в корпорации Microsoft эти продукты разрабатываются практически одни и теми же людьми. И Microsoft постоянно говорит о построении «клиентского континуума», который позволил бы создавать приложения для любой платформы и устройства программистам, обладаю-

щим определенными навыками (о которых вы узнаете в этой книге), с помощью одних и тех же инструментов (Visual Studio, Expression Blend и т. д.) При этом желательно, чтобы если не *двоичный*, то хотя бы *исходный код* (на .NET-совместимом языке типа C# или VB в сочетании с XAML) был один и тот же. Назвать эту книгу «WPF и Silverlight. Полное руководство» было бы, пожалуй, чересчур смело, но рад сообщить, что знания, которые вы получите, прочитав ее, сделают вас специалистом и по WPF, и по Silverlight.

Резюме

В последнее время растет число программ, имеющих высококачественный интерфейс – иногда даже как в кино, – а те, что не следуют этой тенденции, рискуют показаться старомодными. Однако еще не так давно создание подобного интерфейса (особенно в Windows) требовало значительных усилий.

Технология WPF существенно упрощает создание любых пользовательских интерфейсов, будь то традиционное Windows-приложение или иммерсивная (создающая эффект присутствия) трехмерная программа, сравнимая с летним блокбастером. При этом впечатляющий интерфейс может создаваться относительно независимо от остального приложения, что позволяет дизайнерам гораздо эффективнее участвовать в процессе разработки приложения. Но не ограничивайтесь мечтами. Читайте дальше, и вы научитесь все это делать!

- Определение XAML
- Элементы и атрибуты
- Пространства имен
- Элементы свойств
- Конвертеры типов
- Расширения разметки
- Дочерние объектные элементы
- Сочетание XAML и процедурного кода
- Введение в XAML2009
- Трюки с классами чтения и записи XAML
- Ключевые слова XAML

2

Все тайны XAML

Язык XML активно используется в технологиях .NET для выражения различной функциональности в ясном, декларативном стиле. В этом смысле языку XAML (диалекту XML), появившемуся уже в первой версии WPF в 2006 году, отведена особая роль. Часто его неправильно считают всего лишь средством описания пользовательского интерфейса, чем-то вроде HTML. Однако, прочитав эту главу, вы поймете, что XAML может гораздо больше, чем просто расставлять элементы управления на экране.

В технологиях WPF и Silverlight XAML *действительно* применяется главным образом для описания пользовательского интерфейса (но не только). Однако в Windows Workflow Foundation (WF) и Windows Communication Foundation (WCF) язык XAML используется для описания операций и конфигураций, которые не имеют ничего общего с пользовательскими интерфейсами.

Основное назначение XAML заключается в том, чтобы предоставить программистам и специалистам в других областях возможность совместной работы. То есть XAML становится единым языком общения, как правило, опосредованным инструментами разработки и предметно-ориентированными средствами проектирования. Но поскольку язык XAML (и вообще XML) может восприниматься человеком, то с ним можно работать, даже не имея ничего, кроме редактора Блокнот.

В контексте WPF и Silverlight «специалистами в предметной области» являются графические дизайнеры, которые могут с помощью таких средств разработки, как Expression Blend, создавать элегантные пользовательские интерфейсы независимо от программистов, пишущих код приложения. Такая кооперация дизайнеров и программистов стала возможной не только благодаря общему языку XAML, но и потому, что значительная часть функциональности, обеспечиваемой различными API, сделана доступной для декларативного объявления. В результате для использования всей выразительной мощи

средств проектирования (например, для описания сложных анимаций или переходов состояний) не требуется писать процедурный код.

Но, даже если вы не планируете работать совместно с дизайнерами, все равно стоит ознакомиться с XAML по следующим причинам:

- Язык XAML может стать очень компактным средством описания пользовательского интерфейса и других иерархий объектов.
- XAML позволяет легко отделить внешний вид приложения от его внутренней логики, что сильно упрощает последующее сопровождение, даже если команда разработчиков состоит всего из одного человека.
- Код на XAML легко скопировать в различные средства разработки, например Visual Studio, Expression Blend или какую-нибудь небольшую автономную программу, и сразу увидеть результат без какой либо компиляции.
- Именно код XAML генерируют практически все средства разработки, связанные с WPF.

В этой главе мы подробно рассмотрим структуру и синтаксис языка XAML и покажем, как он соотносится с процедурным кодом. В отличие от предыдущей главы, погружение в тему будет по-настоящему глубоким. Располагая этими основополагающими знаниями, вы сможете не только читать примеры кода, но и понять, почему API различных классов спроектирован именно так, а не иначе. Это понимание будет полезно также при разработке WPF-приложений, элементов управления, библиотек классов, поддерживающих XAML,

СОВЕТ

Существует несколько способов выполнить написанные на XAML примеры из данной главы (их код можно скачать в электронном виде вместе с прочим исходным кодом, прилагаемым к книге), например:

- Сохраните текст примера в файле с расширением *.xaml* и откройте его в Internet Explorer (для Windows Vista или более поздней либо для Windows XP с установленным каркасом .NET 3.0 или более поздней версией). Можно также использовать Firefox, если установлено соответствующее дополнение. Но по умолчанию браузер будет использовать версию WPF, установленную вместе с операционной системой, а не WPF 4.
- Скопируйте текст примера в какую-нибудь простую инструментальную программу, например XAMLPAD2009, предлагаемую вместе с исходным кодом к этой главе, или Kaxaml (*http://kaxaml.com*), хотя на момент написания книги последняя еще не была модернизирована для работы с WPF 4.
- Создайте WPF-проект в Visual Studio и замените содержимое главного окна `Window` или элемента `Page` текстом примера; иногда может понадобиться внести в код некоторые изменения.

Первые два варианта открывают замечательную возможность ознакомиться с XAML и немного поэкспериментировать. Сочетание XAML с другим содержимым в проекте Visual Studio рассматривается в конце этой главы.

или инструментальных средств, получающих на входе или порождающих на выходе XAML-код (например, программы проверки и локализации, конвертеры форматов, конструкторы и т. д.).

FAQ

Что случилось с XamlPad?

В состав прежних версий Windows SDK входила простая программа XamlPad, которая позволяла вводить (или копировать) WPF-совместимый код на XAML и смотреть, как он визуализируется в виде пользовательского интерфейса. К сожалению, это средство больше не поставляется ввиду недостатка ресурсов. (Да, несмотря на распространенное убеждение, ресурсы Microsoft не безграничны!) Однако есть несколько альтернативных инструментов для экспериментов с XAML, в том числе:

- XAMLPAD2009 – включен в состав исходного кода к данной книге. В нем нет изысков, имеющихся в других инструментах, зато прилагается исходный код. К тому же, это единственная программа, поддерживающая версию XAML2009 (описанную далее в этой главе).
- Kaxaml – удобный инструмент, написанный Робби Ингебретсеном (Robby Ingebretsen), бывшим членом команды разработчиков WPF; доступен по адресу *http://kaxaml.com*.
- XamlPadX – довольно развитая программа, доступная по адресу *http://blogs.msdn.com/llobo/archive/2008/08/25/xamlpadx-4-0.aspx*. Автором является Лестер Лобо (Lester Lobo), один из разработчиков WPF.
- XAML Cruncher – ClickOnce-приложение, которое можно найти на странице *http://charlespetzold.com/wpf/XamlCruncher/ XamlCruncher.application*. Разработано Чарльзом Петцольдом (Charles Petzold), очень плодовитым автором книг и блогером.

Определение XAML

XAML – сравнительно простой декларативный язык программирования общего назначения, предназначенный для конструирования и инициализации объектов. На самом деле XAML представляет собой диалект XML с добавлением ряда правил, относящихся к элементам, атрибутам и их отображению на объекты, их свойства и значения свойств (помимо всего прочего).

Поскольку XAML – это просто механизм для использования различных API каркаса .NET, все попытки сравнения его с HTML, SVG (Scalable Vector Graphics) и другими предметно-ориентированными языками и форматами некорректны. XAML содержит правила интерпретации XML синтаксическими анализаторами и компиляторами, а также ряд ключевых слов, но сам по себе он не определяет никаких существенных элементов. Поэтому говорить о XAML вне связи с конкретной средой типа WPF – все равно что обсуждать C# вне каркаса .NET Framework. Вместе с тем Microsoft формализовала понятие «словарей

XAML», то есть наборов допустимых элементов для различных областей. Таким образом, можно говорить о WPF XAML, о Silverlight XAML и о других типах XAML-файлов.

КОПНЕМ ГЛУБЖЕ

Спецификации XAML и словарей XAML

Подробные спецификации XAML и двух словарей XAML можно найти по следующим адресам:

- XAML Object Mapping Specification 2006 (MS-XAML): *http://go.microsoft.com/fwlink/?LinkId=130721*
- WPF XAML Vocabulary Specification 2006 (MS-WPFXV): *http://go.microsoft.com/fwlink/?LinkId=130722*
- Silverlight XAML Vocabulary Specification 2008 (MS-SLXV): *http://go.microsoft.com/fwlink/?LinkId=130707*

Соотношение между XAML и WPF часто понимают неправильно, поэтому важно подчеркнуть, что WPF и XAML могут использоваться независимо друг от друга. И хотя XAML изначально был разработан именно для WPF, сейчас он используется и в других технологиях. Ведь благодаря своей универсальности XAML может применяться практически в любой объектно-ориентированной технологии. Более того, использовать XAML в WPF-проектах необязательно. Практически все, что вы способны сделать с помощью XAML, можно реализовать на любом процедурном .NET-совместимом языке (обратное не верно). Однако из-за тех преимуществ XAML, которые были упомянуты в начале этой главы, трудно встретить реальный WPF-проект, где этот язык не используется.

КОПНЕМ ГЛУБЖЕ

Функциональность XAML, недоступная из процедурного кода

Существует ряд приемов, которые можно выполнить на XAML, но не посредством процедурного кода. Они малоизвестны и более подробно рассматриваются в главах 12 и 14.

- Создание полного набора шаблонов. В процедурном коде можно создавать шаблоны с помощью класса `FrameworkElementFactory`, но выразительные возможности этого подхода ограничены.
- Использование конструкции `x:Shared="False"`, заставляющей WPF возвращать новый экземпляр при каждом обращении к элементу из словаря ресурсов.
- Отложенное создание объектов внутри словаря ресурсов. Это важно для оптимизации производительности и доступно только с помощью скомпилированного XAML-кода.

Элементы и атрибуты

В спецификации XAML определены правила отображения пространств имен, типов, свойств и событий .NET на пространства имен, элементы и атрибуты XML. Они иллюстрируются ниже на примере простого (но полного) XAML-файла, где определяется WPF-объект `Button`, и эквивалентного ему кода на C#:

XAML:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK"/>
```

C#:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
```

Оба фрагмента эквивалентны, но результат визуализации XAML можно сразу увидеть в Internet Explorer – это кнопка, заполняющая все окно браузера (рис. 2.1), а код на C# придется предварительно откомпилировать и скомпоновать с дополнительными библиотеками.



Рис. 2.1. *Простая кнопка WPF Button, определенная в XAML-файле*

Объявление XML-элемента в XAML-коде (он называется *объектным элементом*) эквивалентно созданию экземпляра соответствующего класса .NET с помощью конструктора по умолчанию. Задание атрибута объектного элемента эквивалентно заданию одноименного свойства (такие атрибуты называются *атрибутами свойств*) или подключению обработчика одноименного события (*атрибуты событий*). Изменим код создания кнопки, задав не только свойство `Content`, но и обработчик события `Click`.

XAML:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK" Click="button_Click"/>
```

C#:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Click += new System.Windows.RoutedEventHandler(button_Click);
b.Content = "OK";
```

Здесь предполагается, что где-то определен метод `button_Click` с правильной сигнатурой, то есть XAML-файл уже не получится визуализировать автономно, как на рис. 2.1. В разделе «Сочетание XAML и процедурного кода» в конце главы объясняется, как работать с XAML-файлом, нуждающимся в дополнительном коде. Отметим, что язык XAML, как и C#, чувствителен к регистру.

КОПНЕМ ГЛУБЖЕ

Порядок обработки свойств и событий

Во время выполнения обработчики событий присоединяются *до* установки свойств объектов, объявленных в XAML-коде (за исключением описанного ниже свойства `Name`, которое устанавливается сразу после конструирования объекта). Поэтому при генерации событий в ответ на установку свойств можно не думать о порядке записи атрибутов в XAML.

Установка нескольких свойств и присоединение нескольких обработчиков событий обычно производятся в порядке задания атрибутов свойств и событий в объектном элементе. К счастью, на практике этот порядок не важен, поскольку согласно принципам проектирования в .NET порядок установки свойств объекта класса не должен иметь значения, и то же самое относится к порядку присоединения обработчиков событий.

Пространства имен

Наименее очевидный аспект при сравнении XAML-кода с эквивалентным кодом на C# – отображение пространства имен XML `http://schemas.microsoft.com/winfx/2006/xaml/presentation` на пространство имен .NET `System.Windows.Controls`. В действительности отображение на это и другие пространства имен WPF жестко зашито в сборках WPF, точнее в нескольких экземплярах атрибута `XmlnsDefinitionAttribute`. (На всякий случай отметим, что указанный URL-адрес в домене *schemas.microsoft.com* не соответствует реальной веб-странице, это всего лишь произвольная строка, как и любое пространство имен.)

В корневом объектном элементе XAML-файла должно быть указано по крайней мере одно пространство имен XML; оно используется для квалификации самого элемента и его потомков. Можно объявлять и дополнительные пространства имен XML (в корневом элементе или его потомках) при условии, что для каждого пространства имен задан уникальный префикс, который будет сопровождать все идентификаторы из этого пространства. Так, в XAML-файлах для WPF обычно указывается второе пространство имен с префиксом `x` (записывается в виде `xmlns:x`, а не просто `xmlns`):

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Это пространство имен языка XAML, которое отображается на типы в пространстве имен System.Windows.Markup; кроме того, в нем определено несколько специальных директив для компилятора или синтаксического анализатора XAML. Эти директивы обычно выглядят, как атрибуты XML-элементов, то есть как свойства объемлющего элемента, хотя на самом деле таковыми не являются. Список ключевых слов XAML приведен в разделе «Ключевые слова XAML» ниже.

КОПНЕМ ГЛУБЖЕ

Неявные пространства имен .NET

WPF отображает следующие определенные в различных сборках WPF пространства имен .NET на пространство имен XML, соответствующее WPF (http://schemas.microsoft.com/winfx/2006/xaml/presentation):

- System.Windows
- System.Windows.Automation
- System.Windows.Controls
- System.Windows.Controls.Primitives
- System.Windows.Data
- System.Windows.Documents
- System.Windows.Forms.Integration
- System.Windows.Ink
- System.Windows.Input
- System.Windows.Media
- System.Windows.Media.Animation
- System.Windows.Media.Effects
- System.Windows.Media.Imaging
- System.Windows.Media.Media3D
- System.Windows.Media.TextFormatting
- System.Windows.Navigation
- System.Windows.Shapes
- System.Windows.Shell

Поскольку это отображение типа многие-к-одному, то разработчикам WPF пришлось позаботиться о том, чтобы в разных пространствах имен не встречались классы с одинаковыми именами.

Использование пространства имен WPF XML (http://schemas.microsoft.com/winfx/2006/xaml/presentation) в качестве основного, а пространства имен XAML (http://schemas.microsoft.com/winfx/2006/xaml) в качестве дополнительного, с префиксом x, – не более чем соглашение, такое же, как привычка начинать файл с кодом на C# директивой using System;. Можно было бы записать XAML-файл и как показано ниже, смысл от этого не изменился бы:

```
<WpfNamespace:Button
  xmlns:WpfNamespace="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK"/>
```

Чтобы код было проще читать, лучше объявлять наиболее часто используемое пространство имен XML (называемое также *основным*) без префикса, а для всех дополнительных пространств имен выбирать короткие префиксы.

СОВЕТ

В большинстве примеров XAML в этой главе пространства имен указаны явно, но в остальной части книги обычно предполагается, что пространство имен WPF XML (http://schemas.microsoft.com/winfx/2006/xaml/presentation) объявлено как основное, а пространство имен языка XAML (http://schemas.microsoft.com/winfx/2006/xaml) – как дополнительное, с префиксом x. Поэтому, если вы захотите посмотреть, как примеры выглядят в браузере или в простой программе просмотра типа XAMLPAD2009, то не забудьте добавить эти объявления.

КОПНЕМ ГЛУБЖЕ

В процессе развития образовалось несколько пространств имен WPF XML

В реальном XAML-коде для WPF практически всегда в качестве основного используется пространство имен WPF XML, но так сложилось, что на важнейшие типы WPF, определенные в пространстве имен System.Windows и вложенных в него, отображается более одного пространства имен XML.

В версии WPF 3.0 поддерживалось пространство имен http://schemas.microsoft.com/**winfx/2006**/xaml/presentation, а в WPF 3.5 было определено новое пространство имен XML – http://schemas.microsoft.com/**netfx/2007**/xaml/presentation, которое отображается на те же самые типы WPF. (Дело в том, что первоначально название WinFX использовалось для всего набора технологий, добавленных в .NET 3.0: WPF, WCF и WF. Потом от этого названия отказались, что и привело к изменению названия пространства имен.) В WPF 4 определено очередное пространство имен XML, которое отображается на те же самые типы WPF: http://schemas.microsoft.com/**netfx/2009**/xaml/presentation.

Но лучше не обращать внимания на этот разнобой, а придерживаться первоначального пространства имен http://schemas.microsoft.com/winfx/2006/xaml/presentation, поскольку оно применимо к любой версии WPF. (Но это вовсе не означает, что сам XAML-код будет работать со всеми версиями WPF; для этого необходимо пользоваться только возможностями, присутствующими в WPF 3.0) Отметим, что Silverlight также поддерживает пространство имен http://schemas.microsoft.com/winfx/2006/xaml/presentation, чтобы было проще использовать в Silverlight-проекте XAML-код, разработанный для WPF. При этом в Silverlight определено и собственное альтернативное пространство имен http://schemas.microsoft.com/client/2007, которое не поддерживается WPF.

С пространствами имен XML легко запутаться. Вопреки выбранным названиям, это не *схемы*. Они не описывают какой-то замкнутый набор типов, существовавший на момент публикации данного пространства имен. Просто в каждой новой версии WPF все предыдущие пространства имен дополняются новыми парами сборка/пространство имен. Таким образом, пространство имен `winfx/2006` на самом деле означает «версия 3.0 *или более поздняя*», пространство имен `netfx/2007` – «версия 3.5 *или более поздняя*» и т. д. Однако в WPF 4 по оплошности из пространства имен `netfx/2009` были исключены некоторые пары пространство имен/сборка, поэтому пользоваться пострадавшими из-за этого типами (например, `TextOptions`) проблематично!

Загрузка автономного XAML-файла в Internet Explorer в реальности производится программой *PresentationHost.exe*, которая решает, какую версию .NET Framework загрузить, основываясь на пространстве имен XML корневого элемента. Если это `netfx/2009`, то загружается версия 4.0, в противном случае – та версия 3.*x*, которая установлена в системе.

Элементы свойств

В предыдущей главе упоминалось, что одной из самых впечатляющих особенностей WPF является развитый механизм композиции. Это можно продемонстрировать на примере простой кнопки, изображенной на рис. 2.1, в которую можно поместить произвольное содержимое, а не только текст! Так, показанный ниже код помещает в кнопку квадрат, делая ее похожей на кнопку Стоп в медиаплеере:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
System.Windows.Shapes.Rectangle r = new System.Windows.Shapes.Rectangle();
r.Width = 40;
r.Height = 40;
r.Fill = System.Windows.Media.Brushes.Black;
b.Content = r; // Делаем квадрат содержимым Button
```

Свойство `Content` объекта `Button` – объект типа `System.Object`; в частности, это может быть объект `Rectangle`, то есть прямоугольник 40×40. Результат представлен на рис. 2.2.

Красиво, конечно, но как выразить то же самое на XAML с помощью синтаксиса атрибутов свойств? Что нужно написать, чтобы свойству `Content` был присвоен объект `Rectangle`, как в показанном выше коде на C#? Записать это одной строкой невозможно, но XAML предлагает альтернативный (менее компактный) синтаксис для установки составных свойств – *элементы свойств*. Выглядит это следующим образом:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  <Rectangle Height="40" Width="40" Fill="Black"/>
</Button.Content>
</Button>
```



***Рис. 2.2.** Кнопка Button со сложным содержимым*

Теперь свойство `Content` устанавливается с помощью XML-элемента, а не XML-атрибута, а результат эквивалентен коду на C#. Точка в выражении `Button.Content` позволяет отличить элемент свойства от объектного элемента. Элементы свойств всегда имеют вид *ИмяТипа.ИмяСвойства* и обязательно вложены в объектный элемент вида *ИмяТипа*. У элементов свойств не может быть собственных атрибутов (за одним исключением – атрибут `x:Uid` используется для локализации).

Синтаксис элементов свойств можно использовать и для простых значений свойств. В следующем примере для `Button` с помощью атрибутов устанавливаются два свойства (`Content` и `Background`).

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK" Background="White"/>
```

То же самое можно записать иначе, задав упомянутые свойства с помощью элементов:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  OK
</Button.Content>
<Button.Background>
  White
</Button.Background>
</Button>
```

Конечно, при вводе XAML-кода вручную лучше использовать форму с атрибутами – это короче.

Конвертеры типов

Рассмотрим C#-код, эквивалентный предыдущему объявлению `Button`: установка свойств `Content` и `Background`:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
b.Background = System.Windows.Media.Brushes.White;
```

Минуточку! Как это строка "White" в показанном выше XAML-файле может быть эквивалентна статическому полю System.Windows.Media.Brushes.White (типа System.Windows.Media.SolidColorBrush)? Действительно, здесь мы воспользовались одной хитростью, позволяющей с помощью строк в XAML-коде задавать значения свойств, тип которых отличается от System.String или System.Object. В таких случаях компилятор или анализатор XAML должен найти *конвертер типа*, который умеет преобразовывать строковое значение в нужный тип.

WPF предоставляет конвертеры типов для многих часто используемых типов данных: Brush, Color, FontWeight, Point и т. д. Все это – классы, производные от TypeConverter (BrushConverter, ColorConverter и т. д.) Вы можете написать собственный конвертер для произвольного типа данных. В отличие от самого языка XAML, конвертеры типов обычно не чувствительны к регистру символов.

Если бы не существовало конвертера типа Brush, то в XAML-коде для присваивания свойству Background значения пришлось бы применить синтаксис элементов свойств:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK">
<Button.Background>
  <SolidColorBrush Color="White"/>
</Button.Background>
</Button>
```

Но даже это возможно только потому, что конвертер типа для Color умеет интерпретировать строку "White". Если бы такого конвертера не было, пришлось бы писать следующий код:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK">
<Button.Background>
  <SolidColorBrush>
  <SolidColorBrush.Color>
    <Color A="255" R="255" G="255" B="255"/>
  </SolidColorBrush.Color>
  </SolidColorBrush>
</Button.Background>
</Button>
```

Но и *это* возможно лишь благодаря наличию конвертера типа, который умеет преобразовывать строку "255" в значение типа Byte, ожидаемое свойствами A, R, G и B типа Color. Не будь его, мы оказались бы в тупике. Конвертеры типов не только делают XAML-код понятнее, но и позволяют записывать значения, которые иначе записать было бы невозможно.

КОПНЕМ ГЛУБЖЕ

Использование конвертеров типов в процедурном коде

Хотя код на C#, в котором свойству `Background` присваивается значение `System.Windows.Media.Brushes.White`, дает тот же результат, что XAML-код, где этому свойству присваивается строка `"White"`, механизм преобразования типов в нем не применяется. Ниже показан код, который более точно отражает действия среды выполнения, когда она ищет и выполняет подходящий конвертер типа:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
b.Background = (Brush)System.ComponentModel.TypeDescriptor.GetConverter(
  typeof(Brush)).ConvertFromInvariantString("White");
```

В отличие от предыдущего кода на C#, опечатка в слове `"White"` не приведет к ошибке компиляции, но вызовет исключение во время выполнения, как и в случае XAML. (Хотя Visual Studio на этапе компиляции XAML-кода предупреждает об ошибках такого рода.)

КОПНЕМ ГЛУБЖЕ

Поиск конвертеров типов

А как все-таки компилятор или анализатор XAML находит подходящий конвертер типа для значения свойства? Он смотрит, снабжено ли определение данного свойства или определение типа данных этого свойства атрибутом `System.ComponentModel.TypeConverterAttribute`.

Например, при установке свойства `Background` кнопки `Button` в XAML-коде применяется конвертер типа `BrushConverter`, поскольку свойство `Background` имеет тип `System.Windows.Media.Brush`, в определении которого задан следующий атрибут:

```
[TypeConverter(typeof(BrushConverter)), ...]
public abstract class Brush : ...
{
  ...
}
```

С другой стороны, для установки свойства `FontSize` кнопки используется конвертер типа `FontSizeConverter`, потому что это свойство (определенное в базовом классе `Control`) снабжено следующим атрибутом:

```
[TypeConverter(typeof(FontSizeConverter)), ...]
public double FontSize
{
  get { ... }
  set { ... }
}
```

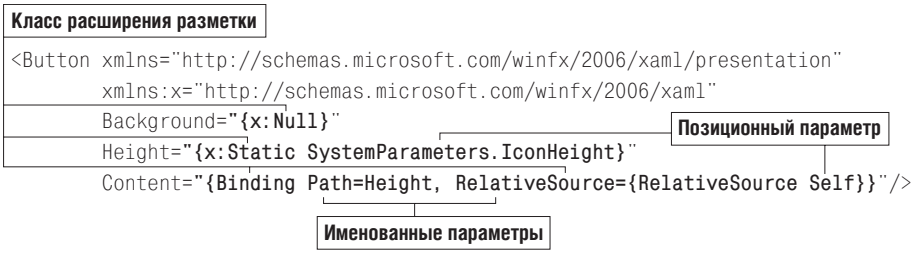
В этом случае конвертер типа необходимо соотносить именно со свойством, поскольку его тип данных (`double`) слишком общий и всегда ассоциировать его с `FontSizeConverter` неразумно. На самом деле в WPF тип `double` часто ассоциируется с другим конвертером типа – `LengthConverter`.

Расширения разметки

Расширения разметки, как и конвертеры типов, позволяют улучшить выразительность языка XAML. Оба механизма могут интерпретировать строковые атрибуты во время выполнения (за исключением нескольких встроенных расширений разметки, которые в настоящее время вычисляются во время компиляции из соображений производительности) и создавать объекты, соответствующие строкам. Помимо стандартных конвертеров типов в дистрибутиве WPF имеется несколько встроенных расширений разметки.

Но, в отличие от конвертеров типов, для расширений разметки в XAML предусмотрен явный логичный синтаксис. Поэтому именно последним следует отдать предпочтение. Кроме того, расширения разметки позволяют обойти потенциальные ограничения, присущие существующим конвертерам типов, изменить которые вы не в силах. Например, с помощью расширения разметки можно установить в качестве фона элемента управления градиентную кисть, заданную в виде строки, хотя встроенный конвертер `BrushConverter` не умеет этого делать.

Если значение атрибута заключено в фигурные скобки {}, то компилятор или анализатор XAML считает его значение расширением разметки, а не обычной строкой (или чем-то, нуждающимся в конвертере типов). В показанном ниже элементе `Button` используется три разных расширения разметки в трех различных свойствах:



Первый идентификатор в каждом заключенном в фигурные скобки значении – имя класса расширения разметки, который должен наследовать классу `MarkupExtension`. По принятому соглашению имена таких классов оканчиваются словом `Extension`, но в XAML его можно опускать. В данном примере `NullExtension` (записано в виде `x:Null`) и `StaticExtension` (записано в виде `x:Static`) – классы из пространства имен `System.Windows.Markup`, поэтому для их поиска необходимо указывать префикс `x`. Но класс `Binding` (имя которого не оканчивается словом `Extension`) находится в пространстве имен `System.Windows.Data`, поэтому его следует искать в пространстве имен XML, подразумеваемом по умолчанию.

Если расширение разметки поддерживает такой синтаксис, ему можно передавать параметры, разделенные запятой. Позиционные параметры (например, `SystemParameters.IconHeight`) рассматриваются как строковые аргументы для соответствующего конструктора класса расширения. Именованные пара-

метры (в данном примере Path и RelativeSource) позволяют устанавливать в конструируемом объекте расширения разметки свойства с соответствующими именами. Значением такого свойства может быть еще одно расширение разметки (задается с помощью вложенных фигурных скобок, как в случае RelativeSource) или литерал, который можно подвергнуть обычной процедуре конвертации типов. Если вы знакомы с атрибутами .NET (это популярный механизм расширения каркаса), то, возможно, заметили что синтаксис и использование расширений разметки в XAML очень напоминает способ определения атрибутов. Это неслучайно.

В предыдущем объявлении элемента Button расширение NullExtension позволяет в качестве кисти Background (свойство Background) задавать null, хотя конвертер типа BrushConverter (и, кстати, многие другие конвертеры) такую возможность не поддерживает. Это сделано только для примера, поскольку фон null на практике бесполезен. Расширение StaticExtension позволяет использовать в XAML статические свойства, поля, константы и элементы перечисления вместо явного прописывания литералов. В данном случае высота Height кнопки Button устанавливается равной текущей высоте значков в операционной системе, которую можно получить с помощью статического свойства IconHeight класса System.Windows.SystemParameters. Расширение Binding подробно рассматривается в главе 13 «Привязка к данным». Оно позволяет присвоить свойству Content значение, равное значению свойства Height.

КОПНЕМ ГЛУБЖЕ

Экранирование фигурных скобок

Если строковое значение атрибута свойства начинается открывающей фигурной скобкой, то ее следует экранировать, чтобы анализатор не счел ее началом расширения разметки. Это можно сделать, поставив перед ней пустую пару фигурных скобок, как в следующем примере:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Content="{}{Это не расширение разметки!}"/>
```

Или воспользоваться обратной косой чертой, которая экранирует открывающую фигурную скобку, одиночную или двойную кавычку.

В качестве альтернативы можно использовать синтаксис элемента свойства, поскольку в этом контексте фигурные скобки не имеют специального смысла. Предыдущий пример может быть переписан в следующем виде:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  {Это не расширение разметки!}
</Button.Content>
</Button>
```

В механизме привязки к данным (см. главу 13) такой способ экранирования используется для задания спецификаторов формата, где фигурные скобки являются стандартной частью синтаксиса.

Поскольку расширения разметки – это просто классы с конструкторами по умолчанию, то их можно использовать в элементах свойств. Следующее описание кнопки Button полностью эквивалентно предыдущему:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<Button.Background>
  <x:Null/>
</Button.Background>
<Button.Height>
  <x:Static Member="SystemParameters.IconHeight"/>
</Button.Height>
<Button.Content>
  <Binding Path="Height">
  <Binding.RelativeSource>
    <RelativeSource Mode="Self"/>
  </Binding.RelativeSource>
  </Binding>
</Button.Content>
</Button>
```

Такая трансформация допустима, поскольку у всех этих расширений разметки имеются свойства, соответствующие аргументам конструкторов с параметрами (то есть позиционным параметрам в синтаксисе атрибутов свойств). Например, в классе StaticExtension определено свойство Member, которое имеет тот же смысл, что и аргумент, ранее передававшийся конструктору с параметрами, а в классе RelativeSource есть свойство Mode – ему также соответствует аргумент конструктора.

КОПНЕМ ГЛУБЖЕ

Расширения разметки и процедурный код

Каждое расширение разметки выполняет некоторую специфическую функцию. Например, следующий код на C# эквивалентен XAML-описанию кнопки Button с использованием расширений NullExtension, StaticExtension и Binding:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
// Установить Background:
b.Background = null;
// Установить Height:
b.Height = System.Windows.SystemParameters.IconHeight;
// Установить Content:
System.Windows.Data.Binding binding = new System.Windows.Data.Binding();
binding.Path = new System.Windows.PropertyPath("Height");
binding.RelativeSource = System.Windows.Data.RelativeSource.Self;
b.SetBinding(System.Windows.Controls.Button.ContentProperty, binding);
```

Однако этот код работает иначе, чем компилятор или анализатор XAML, который предполагает, что любое расширение разметки устанавливает нужные значение

во время выполнения (вызывая метод ProvideValue). Эквивалентный процедурный код часто оказывается сложным и иногда требует знания контекста, известного только анализатору (например, как разрешать префикс пространства имен, который может встречаться в свойстве Member расширения StaticExtension). К счастью, работать с расширениями разметки таким образом в процедурном коде необязательно!

Дочерние объектные элементы

XAML-файл, как и любой XML-файл, должен иметь единственный корневой объектный элемент. Поэтому неудивительно, что объектные элементы могут поддерживать наличие дочерних объектных элементов (а не только элементов свойств, которые с точки зрения XAML не являются дочерними). Объектный элемент может иметь потомков трех разных типов: значение свойства содержимого, элементы коллекции или значение, тип которого может быть преобразован в тип объектного элемента.

Свойство Content

В большинстве классов WPF имеется свойство (задаваемое с помощью атрибута), значением которого является содержимое данного XML-элемента. Оно называется *свойством содержимого* и в действительности представляет собой просто удобный способ сделать XAML-представление более компактным. В некотором смысле свойство содержимого похоже на свойства по умолчанию в старых версиях Visual Basic (вызывавшие много нареканий).

Для свойства Content кнопки Button имеется специальное соглашение (что очень кстати), поэтому описание

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK"/>
```

можно представить в следующем виде:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  OK
</Button>
```

А составное содержимое Button, например

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  <Rectangle Height="40" Width="40" Fill="Black"/>
</Button.Content>
</Button>
```

можно переписать так:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Rectangle Height="40" Width="40" Fill="Black"/>
</Button>
```

Нигде не требуется, чтобы свойство содержимого называлось именно Content; так, в классах ComboBox, ListBox и TabControl (все из пространства имен System.Windows.Controls) свойство содержимого названо Items.

Элементы коллекций

Язык XAML позволяет добавлять элементы в два основных вида коллекций, поддерживающих индексирование: списки и словари.

Списки

Списком считается любая коллекция, в которой реализован интерфейс System.Collections.IList, например System.Collections.ArrayList и многочисленные классы коллекций, определенные в WPF. Следующий XAML-код добавляет два элемента в список ListBox, свойство Items которого имеет тип ItemsCollection, реализующий интерфейс IList:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<ListBox.Items>
  <ListBoxItem Content="Item 1"/>
  <ListBoxItem Content="Item 2"/>
</ListBox.Items>
</ListBox>
```

Этот XAML-код эквивалентен такому коду на C#:

```
System.Windows.Controls.ListBox listbox = new System.Windows.Controls.ListBox();
System.Windows.Controls.ListBoxItem item1 =
  new System.Windows.Controls.ListBoxItem();
System.Windows.Controls.ListBoxItem item2 =
  new System.Windows.Controls.ListBoxItem();
item1.Content = "Item 1";
item2.Content = "Item 2";
listbox.Items.Add(item1);
listbox.Items.Add(item2);
```

Далее, поскольку Items – свойство содержимого для ListBox, то XAML-код можно еще сократить:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ListBoxItem Content="Item 1"/>
  <ListBoxItem Content="Item 2"/>
</ListBox>
```

Этот код работает потому, что свойство Items класса ListBox автоматически инициализируется пустой коллекцией. Если бы оно инициализировалось значением null (и, в отличие от доступного только для чтения свойства Items класса ListBox, допускало чтение и запись), то пришлось бы поместить все элементы внутрь явно заданного элемента XAML, который создает экземпляр коллекции. Поскольку встроенные в WPF элементы управления устроены не так, то продемонстрируем этот подход для воображаемого элемента OtherListBox:

```
<OtherListBox>
<OtherListBox.Items>
  <ItemCollection>
    <ListBoxItem Content="Item 1"/>
    <ListBoxItem Content="Item 2"/>
  </ItemCollection>
</OtherListBox.Items>
</OtherListBox>
```

Словари

Коллекция System.Windows.ResourceDictionary используется в WPF очень часто, в чем мы убедимся в главе 12 «Ресурсы». Этот класс реализует интерфейс System.Collections.IDictionary, а значит, поддерживает добавление, удаление и перечисление пар ключ/значение в процедурном коде, как любая хеш-таблица. В XAML в любую коллекцию, реализующую интерфейс IDictionary, можно добавить пару ключ/значение. Например, следующий XAML-код добавляет в словарь ResourceDictionary два цвета Color:

```
<ResourceDictionary
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Color x:Key="1" A="255" R="255" G="255" B="255"/>
  <Color x:Key="2" A="0" R="0" G="0" B="0"/>
</ResourceDictionary>
```

Здесь используется ключевое слово XAML Key (определенное в дополнительном пространстве имен XML), которое обрабатывается специальным образом и позволяет связать с каждым значением Color некий ключ. (В самом типе Color свойство Key не определено.) Следовательно, этот XAML-код эквивалентен такому коду на C#:

```
System.Windows.ResourceDictionary d = new System.Windows.ResourceDictionary();
System.Windows.Media.Color color1 = new System.Windows.Media.Color();
System.Windows.Media.Color color2 = new System.Windows.Media.Color();
color1.A = 255; color1.R = 255; color1.G = 255; color1.B = 255;
color2.A = 0;   color2.R = 0;   color2.G = 0;   color2.B = 0;
d.Add("1", color1);
d.Add("2", color2);
```

КОПНЕМ ГЛУБЖЕ

Списки, словари и анализатор XAML2009

Анализатор WPF XAML всегда поддерживал только коллекции IList и IDictionary, но функциональность анализатора XAML2009 (описанного далее в разделе «Введение в XAML2009») несколько расширена. Сначала он проверяет интерфейсы IList и IDictionary, потом – ICollection<T> и IDictionary<K,V>, а затем – наличие методов Add и GetEnumerator.

Отметим, что значение, заданное в XAML с помощью атрибута `x:Key`, рассматривается как строка, если только не используется расширение разметки и мы не работаем с анализатором XAML2009 (см. далее раздел «Введение в XAML2009»). В указанных случаях попытка преобразовать тип не предпринимается.

Еще о преобразовании типов

Потомком объектного элемента может быть обычный текст, как в следующем объявлении элемента `SolidColorBrush` на XAML:

```
<SolidColorBrush>White</SolidColorBrush>
```

Эта запись эквивалентна следующей:

```
<SolidColorBrush Color="White"/>
```

даже несмотря на то, что `Color` не описано как свойство содержимого. В данном случае первый фрагмент работает потому, что существует конвертер типа, умеющий преобразовывать такие строки, как `"White"` (или `"white"`, или `"#FFFFFF"`) в объект типа `SolidColorBrush`.

Хотя конвертеры типов играют важнейшую роль в обеспечении удобочитаемости XAML, у них есть и негативная сторона: может показаться, что в XAML творится какое-то волшебство, поскольку не всегда понятно, как элементы отображаются на объекты .NET. Опираясь на уже известные нам факты, можно предположить, что в XAML нельзя определить элемент, соответствующий абстрактному классу, потому что невозможно создать экземпляр такого класса. Однако, несмотря на то, что `System.Windows.Media.Brush` является абстрактным базовым классом для `SolidColorBrush`, `GradientBrush` и других конкретных классов кистей, мы все же можем переписать предыдущий фрагмент XAML-кода в таком виде:

```
<Brush>White</Brush>
```

поскольку конвертер типов `Brush` понимает, что речь идет о `SolidColorBrush`. Выглядит несколько необычно, но такая возможность очень важна для записи в XAML примитивных типов, как будет показано ниже, во врезке «Расширяемая часть XAML».

КОПНЕМ ГЛУБЖЕ

Расширяемая часть XAML

Поскольку XAML предназначен для работы с системой типов .NET, то его можно использовать практически с любым объектом .NET (и даже с COM-объектами благодаря интероперабельности с COM), в том числе определенным вами. При этом совершенно неважно, относятся ли эти объекты к пользовательскому интерфейсу.

Однако классы необходимо определять с учетом возможности использования в декларативном коде. Например, если в классе нет ни конструктора по умолчанию, ни полезных открытых свойств, то им нельзя будет напрямую воспользоваться в XAML (если только вы не работаете с XAML2009). Программные интерфейсы WPF тщательно продуманы с тем, чтобы они отвечали декларативной модели XAML (обычных принципов разработки для .NET недостаточно).

Сборки WPF помечены атрибутом `XmlnsDefinitionAttribute`, который отображает содержащиеся в них пространства имен .NET на пространство имен XML в XAML-файле. Но что делать со сборками, которые разрабатывались без ориентации на XAML и, следовательно, не содержат этого атрибута? Находящиеся в них типы все равно можно использовать – нужно лишь добавить специальную директиву, описывающую пространство имен XML. Например, вот обычный код на C#, в котором используются классы .NET из сборки `mscorlib.dll`:

```
System.Collections.Hashtable h = new System.Collections.Hashtable();
h.Add("key1", 7);
h.Add("key2", 23);
```

А вот как его можно представить в XAML:

```
<collections:Hashtable
  xmlns:collections="clr-namespace:System.Collections;assembly=mscorlib"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <sys:Int32 x:Key="key1">7</sys:Int32>
  <sys:Int32 x:Key="key2">23</sys:Int32>
</collections:Hashtable>
```

Директива `clr-namespace` позволяет использовать пространство имен .NET непосредственно в XAML. Спецификация сборки в конце необходима только в случае, когда нужные типы не находятся в той же сборке, где хранится скомпилированный XAML-код. Обычно достаточно простого имени сборки (как в случае `mscorlib`), но можно использовать и каноническое представление, поддерживаемое методом `System.Reflection.Assembly.Load` (правда, без пробелов), которое включает дополнительную информацию, например номер версии и/или маркер открытого ключа.

Отметим два важных момента, которые проливают свет на интеграцию не только с системой типов .NET, но и с конкретными типами .NET Framework:

- Дочерние элементы можно добавлять в родительскую хеш-таблицу `Hashtable` с помощью стандартного синтаксиса XAML x:key, поскольку `Hashtable`, как и другие классы коллекций в .NET Framework, реализует интерфейс `IDictionary` начиная с версии 1.0.
- Тип `System.Int32` можно использовать столь простым образом, поскольку уже существует конвертер типа, умеющий преобразовывать строку в целое число. Объясняется это тем, что конвертеры типов, поддерживаемые XAML, – всего лишь подклассы класса `System.ComponentModel.TypeConverter`, который также существует со времени версии .NET Framework 1.0. Это тот же механизм преобразования типов, что используется в Windows Forms (и позволяет, например, вводить в сетке свойств в Visual Studio строки, которые преобразуются в подходящий тип).

КОПНЕМ ГЛУБЖЕ

Правила обработки потомков объектных элементов в XAML

Мы рассмотрели все три типа потомков объектных элементов. Во избежание неоднозначности каждый компилятор или анализатор XAML при разборе и интерпретации дочерних элементов должен придерживаться следующих правил:

1. Если тип реализует интерфейс `IList`, вызвать `IList.Add` для каждого дочернего элемента.
2. Иначе, если тип реализует интерфейс `IDictionary`, вызвать `IDictionary.Add` для каждого дочернего элемента, используя в качестве ключа значение атрибута `x:Key`, а в качестве значения – сам элемент. (Правда, анализатор XAML2009 проверяет `IDictionary` *раньше* `IList` и поддерживает также другие интерфейсы коллекций, о чем упоминалось выше.)
3. Иначе, если у родителя есть свойство содержимого (помеченное атрибутом `System.Windows.Markup.ContentPropertyAttribute`) и тип дочернего элемента совместим с этим свойством, считать дочерний элемент значением этого свойства.
4. Иначе, если дочерний элемент является простым текстом и существует конвертер типа, который может преобразовать этот текст в тип родителя (*и* при этом для родителя не установлены никакие свойства), подать дочерний элемент на вход конвертера типа, а полученный результат считать экземпляром родителя.
5. Иначе считать содержимое неизвестным, что может являться поводом для возбуждения исключения.

Правила 1 и 2 обеспечивают поведение, описанное в разделе «Элементы коллекций», правило 3 – поведение, описанное в разделе «Свойство Content», а правило 4 – поведение, описанное в разделе «Еще о преобразовании типов», которое чаще всего ставит в тупик.

Сочетание XAML и процедурного кода

WPF-приложение можно полностью написать на любом .NET-совместимом языке программирования. А для создания некоторых простых WPF-приложений достаточно одного лишь XAML благодаря механизму привязки к данным (см. главу 13), триггерам (см. следующую главу) и тому факту, что страницы, написанные на чистом XAML, можно просматривать в браузере. Однако большинство WPF-приложений представляют собой сочетание XAML и процедурного кода. В этом разделе мы рассмотрим два способа совместного использования кода XAML и процедурного языка программирования.

Загрузка и разбор XAML во время выполнения

В состав WPF входит анализатор XAML, работающий на этапе выполнения. Он представлен двумя классами в пространстве имен `System.Windows.Markup`: `XamlReader` и `XamlWriter`. Их API предельно прост. Класс `XamlReader` содержит несколько перегруженных вариантов статического метода `Load`, а класс `Xaml-`

Writer – несколько вариантов статического метода Save. Таким образом, программы, написанные на любом .NET-совместимом языке, могут без особых проблем воспользоваться XAML во время выполнения. В версию .NET Framework 4 включен новый набор классов для чтения и записи XAML, но в них немало подводных камней. Сейчас эти проблемы нам не очень интересны, но ниже, в разделе «Трюки с классами чтения и записи XAML», мы вернемся к этой теме.

Класс XamlReader

Перегруженные варианты метода XamlReader.Load разбирают XAML-код, создают соответствующие объекты .NET и возвращают экземпляр, представляющий корневой элемент. Так, если XAML-файл *MyWindow.xaml* в текущем каталоге содержит в качестве корневого узла объект Window (подробно рассматриваемый в главе 7 «Структурирование и развертывание приложения»), то для загрузки и получения объекта Window можно использовать следующий код:

```
Window window = null;
using (FileStream fs =
  new FileStream("MyWindow.xaml", FileMode.Open, FileAccess.Read))
{
  // Получить корневой элемент. Мы знаем, что это Window
  window = (Window)XamlReader.Load(fs);
}
```

В данном случае методу Load передается объект класса FileStream (из пространства имен System.IO). Когда Load вернет управление, в памяти будет представлена вся иерархия объектов из XAML-файла, так что сам файл больше не нужен. Поэтому поток FileStream закрывается сразу по выходе из блока using. Поскольку объекту XamlReader можно передать произвольный поток Stream (или – в другом перегруженном варианте – объект System.Xml.XmlReader), то для получения содержимого XAML-файла есть масса возможностей.

Имея экземпляр корневого элемента, можно получить его дочерние элементы, если воспользоваться соответствующими свойствами содержимого или свойствами коллекций. В показанном ниже коде предполагается, что содержимым Window является объект StackPanel, пятый дочерний элемент которого – кнопка OK:

```
Window window = null;
using (FileStream fs =
  new FileStream("MyWindow.xaml", FileMode.Open, FileAccess.Read))
{
  // Получить корневой элемент. Мы знаем, что это Window
  window = (Window)XamlReader.Load(fs);
}
// Найти кнопку OK, перебирая дочерние элементы (мы
// пользуемся априорными знаниями о структуре документа!)
StackPanel panel = (StackPanel)window.Content;
Button okButton = (Button)panel.Children[4];
```

Имея ссылку на кнопку Button, можно делать с ней все, что угодно: задавать дополнительные свойства (возможно, применяя логику, которую трудно или невозможно выразить на XAML), присоединять обработчики событий или выполнять какие-то действия, которые нельзя реализовать на XAML, например вызывать методы кнопки.

Конечно же, использование «зашитого» индекса и прочих предположений относительно структуры пользовательского интерфейса нельзя считать удовлетворительным решением, так как любое изменение XAML может привести к их нарушению. Вместо этого можно было бы написать код обработки элементов XAML в более общем виде и искать элемент Button, содержащий строку "OK", но тогда придется выполнить слишком много работы для такой простой задачи. Кроме того, если нас интересует кнопка с графическим содержимым, то как ее найти среди других кнопок?

К счастью, XAML поддерживает именование элементов, поэтому их можно находить и использовать посредством процедурного кода.

СОВЕТ

В классе XamlReader определен также метод экземпляра LoadAsync, который загружает и разбирает XAML-код асинхронно. Этим методом имеет смысл пользоваться, например, чтобы не «подвешивать» пользовательский интерфейс на время, пока загружается большой XAML-файл или производится загрузка по сети. Кроме того, имеется метод CancelAsync для прерывания обработки и событие LoadCompleted, информирующее о ее завершении.

Однако метод LoadAsync ведет себя несколько странно. Он работает в потоке пользовательского интерфейса, многократно обращаясь к методу Dispatcher.BeginInvoke (WPF пытается разбить работу на отрезки продолжительностью 200 мс). К тому же обработка производится асинхронно, только если в корневом узле XAML установлен атрибут x:SynchronousMode="Async". В противном случае LoadSync загружает XAML в синхронном режиме, ничего не сообщая об этом.

Именование элементов XAML

В пространстве имен XAML определено ключевое слово Name, которое позволяет назначить имя любому элементу. Для простой кнопки OK, которая, как мы предполагаем, находится где-то в окне Window, ключевое слово Name можно использовать следующим образом:

```
<Button x:Name="okButton">OK</Button>
```

Тогда приведенный выше код на C# можно переписать с использованием метода Window.FindName, который рекурсивно просматривает всех потомков и возвращает требуемый объект:

```
Window window = null;
using (FileStream fs =
```

```
    new FileStream("MyWindow.xaml", FileMode.Open, FileAccess.Read))
{
    // Получить корневой элемент. Мы знаем, что это Window
    window = (Window)XamlReader.Load(fs);
}
// Находим кнопку OK, зная только ее имя
Button okButton = (Button)window.FindName("okButton");
```

Метод `FindName` имеется не только в классе `Window`. Он определен в классах `FrameworkElement` и `FrameworkContentElement`, которые являются базовыми для многих важных классов WPF.

КОПНЕМ ГЛУБЖЕ

Именование элементов без x:Name

Синтаксис `x:Name` можно использовать для именования элементов, но в некоторых классах определено специальное свойство, которое можно рассматривать как имя элемента (оно назначается с помощью атрибута `System.Windows.Markup.RuntimeNamePropertyAttribute`). Например, в классах `FrameworkElement` и `FrameworkContentElement` имеется свойство `Name`, поэтому они помечены атрибутом `RuntimeNameProperty("Name")`. Это означает, что для таких элементов можно просто задать свойство `Name`, не используя синтаксис `x:Name`. Можно использовать любой из этих механизмов, но не оба сразу. Наличие двух способов задания имени элемента вносит некоторую путаницу, но иметь свойство `Name` удобно для использования в процедурном коде.

СОВЕТ

Во всех версиях WPF для ссылки на именованный элемент можно использовать расширение разметки `Binding` в значении свойства:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Label Target="{Binding ElementName=box}" Content="Enter _text:"/>
  <TextBox Name="box"/>
</StackPanel>
```

(В данном случае присваивание ссылки на поле ввода `TextBox` в качестве значения атрибута `Target` элемента `Label` передает этому полю фокус ввода при нажатии комбинации клавиш Alt+T.) WPF 4 включает новое, более простое расширение разметки `System.Windows.Markup.Reference`, которое позволяет искать элементы на этапе синтаксического разбора, а не выполнения. Его можно использовать следующим образом:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Label Target="{x:Reference box}" Content="Enter _text:"/>
  <TextBox Name="box"/>
</StackPanel>
```

Кроме того, если свойство помечено конвертером типа `System.Windows.Markup.NameReferenceConverter` (как в данном случае), то строковое имя неявно преобразуется в экземпляр, на который ведет ссылка:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Label Target="box" Content="Enter _text:"/>
  <TextBox Name="box"/>
</StackPanel>
```

Компиляция XAML

Загрузка и синтаксический анализ XAML во время выполнения представляют интерес для динамического изменения внешнего облика приложения или для использования в .NET-совместимых языках, не поддерживающих компиляции XAML. Однако в большинстве WPF-проектов используется механизм компиляции XAML, поддерживаемый MSBuild и Visual Studio. Компиляция XAML включает три шага: преобразование XAML-файла в специальный двоичный формат, включение результата в создаваемую сборку в качестве двоичного ресурса и создание инфраструктуры, которая автоматически подключает XAML к процедурному коду. В языках C# и Visual Basic поддержка компиляции XAML реализована лучше всего.

Если вы не возражаете против использования XAML-файла совместно с процедурным кодом, то для его компиляции нужно лишь добавить его в WPF-проект, созданный в Visual Studio, указав в качестве действия при построении (Build Action) значение Page (Страница). (В главе 7 объясняется, как воспользоваться таким содержимым в контексте приложения.) Однако в типичном случае, когда XAML-файл компилируется *и* сочетается с процедурным кодом, первым делом нужно задать подкласс для корневого элемента в XAML-файле. Это можно сделать с помощью ключевого слова `Class`, определенного в пространстве имен XAML. Например:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyNamespace.MyWindow">
  ...
</Window>
```

КОПНЕМ ГЛУБЖЕ

Поддержка откомпилированного XAML-кода в произвольном .NET-языке

Для использования откомпилированного XAML-кода в произвольном .NET-совместимом языке программирования необходимо выполнение двух требований: наличие соответствующего поставщика CodeDom и целевого файла MSBiuld. Кроме того, желательна (но не обязательна) поддержка в языке частичных классов.

В отдельном исходном файле (в том же самом проекте) можно определить этот подкласс и добавить в него любые члены:

```
namespace MyNamespace
{
  partial class MyWindow : Window
  {
    public MyWindow()
    {
      // Необходимо для загрузки содержимого, определенного в XAML-файле!
      InitializeComponent();
      ...
    }
    Здесь могут быть любые члены...
  }
}
```

Этот файл часто называют *застраничным* (*code-behind file*). Если в XAML-коде имеются ссылки на обработчики событий (в таких атрибутах событий, как `Click` для `Button`), то именно здесь их следует определить.

Ключевое слово `partial` в определении класса важно, поскольку реализация класса распределена по нескольким файлам. Если .NET-совместимый язык не поддерживает частичные классы (как, например, C++/CLI и J#), то в XAML-файле необходимо задать также ключевое слово `Subclass` в корневом элементе:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyNamespace.MyWindow" x:Subclass="MyNamespace.MyWindow2">
  ...
</Window>
```

При таком изменении XAML-файл полностью определяет подкласс (в данном случае `MyWindow2`), но в качестве базового класса используется класс, определенный в застраничном файле (`MyWindow`). Таким образом, разделение реализации между двумя файлами моделируется с помощью наследования.

При создании WPF-проекта на языке C# или Visual Basic в Visual Studio или при использовании пункта меню Add New Item... (Добавить новый элемент), чтобы добавить в проект какие-то WPF-элементы, Visual Studio автоматически создает XAML-файл с атрибутом `x:Class` в корневом элементе и застраничный исходный файл, содержащий частичное определение класса, а также связывает их между собой, чтобы они правильно обрабатывались при построении проекта.

Если вы пользуетесь программой MSBuild и хотите разобраться в том, что содержит файл проекта, предполагающего наличие застраничного кода, то можете открыть любой файл проекта на C# из числа прилагаемых к данной книге в обычном текстовом редакторе, например Блокноте. Интересующая нас часть типичного проекта выглядит следующим образом:

```
<ItemGroup>
  <Page Include="MyWindow.xaml"/>
</ItemGroup>
<ItemGroup>
  <Compile Include="MyWindow.xaml.cs">
    <DependentUpon>MyWindow.xaml</DependentUpon>
    <SubType>Code</SubType>
  </Compile>
</ItemGroup>
```

Для такого проекта система построения генерирует несколько файлов в ходе обработке *MyWindow.xaml*, в том числе:

- BAML-файл (*MyWindow.baml*), который по умолчанию внедряется в сборку в качестве двоичного ресурса.
- C#-файл (*MyWindow.g.cs*), который компилируется и включается в сборку, как любой другой файл с исходным кодом.

СОВЕТ

Атрибут `x:Class` разрешается использовать только в компилируемых XAML-файлах. Но иногда можно скомпилировать XAML-файл и без этого атрибута. Это просто означает, что соответствующий застраничный файл отсутствует, так что пользоваться средствами, нуждающимися в процедурном коде, нельзя. Поэтому добавление в проект Visual Studio XAML-файла без атрибута `x:Class` – хороший способ воспользоваться всеми преимуществами компиляции XAML в плане повышения производительности и удобства развертывания без создания ненужного застраничного файла.

BAML

Аббревиатура BAML расшифровывается как Binary Application Markup Language (Двоичный язык разметки приложений). Это просто XAML, который был разобран, разбит на лексемы и преобразован в двоичный формат. Хотя практически любой код на XAML можно представить в виде процедурного кода, компиляция XAML в BAML *не генерирует* исходный код на процедурном языке. В этом смысле BAML не похож на промежуточный язык MSIL; это всего лишь сжатый декларативный формат, который загружается и разбирается быстрее, чем простой XAML-файл (и к тому же меньше по размеру). По существу, BAML – это деталь реализации процедуры компиляции XAML. Тем не менее знать о его существовании полезно. На самом деле в WPF 4 даже имеется открытый класс для чтения BAML-файлов (см. далее раздел «Трюки с классами чтения и записи XAML»).

КОПНЕМ ГЛУБЖЕ

Когда-то здесь был CAML...

В предварительных версиях WPF была возможность компиляции XAML в формат BAML или MSIL. Получающийся при этом MSIL-код назывался CAML, что означает Compiled Application Markup Language (Скомпилированный язык разметки приложения). Идея заключалась в том, чтобы предоставить возможность выбора оптимизации по размеру (BAML) или скорости выполнения (CAML). Но потом разработчики решили не отягощать WPF поддержкой двух независимых реализаций, которые делали практически одно и то же. Формату BAML было отдано предпочтение, поскольку он имеет несколько преимуществ: более безопасен, чем MSIL, более компактен (поэтому объем загрузки при исполнении вебсценариев меньше) и может быть локализован даже после компиляции. Более того, CAML работал не настолько быстрее BAML, как ожидалось. При этом генерировался объемный код, выполняемый всего один раз. Это неэффективно, приводит к разбуханию DLL-библиотек, преимущества кэширования не используются и т. д.

Генерируемый исходный код

В процессе компиляции XAML кое-какой процедурный код все же генерируется (если использовался атрибут `x:Class`), но это всего лишь «клей», аналогичный тому, что пришлось бы писать для загрузки и разбора независимого XAML-файла во время исполнения программы. Таким файлам присваивается суффикс вида *.g.cs* (или *.g.vb*), где *g* означает «*сгенерированный*» (*generated*).

Каждый сгенерированный исходный файл содержит частичное определение класса, указанного в атрибуте `x:Class` корневого объектного элемента. В нем находятся поля (по умолчанию `internal`) для каждого именованного элемента в XAML-файле, при этом в качестве имени поля используется имя элемента. Там же находится метод `InitializeComponent`, который выполняет всю рутинную работу по загрузке внедренного BAML-ресурса, присваиванию полям экземпляров объектов, первоначально определенных в XAML-файле, и присоединению обработчиков событий (если они были специфицированы в XAML).

Поскольку этот «склеивающий» код, помещенный в сгенерированный файл, является частью того же класса, который определен вами в застраничном файле (и поскольку BAML внедряется в виде ресурса), то вам обычно не приходится задумываться о самом существовании BAML и о процедуре его загрузки и разбора. Вы просто пишете код, который ссылается на именованные элементы, как на любые другие члены класса, а система построения заботится о том, как связать все воедино. Нужно только не забыть вызвать метод `InitializeComponent` в конструкторе своего застраничного класса.

ПРЕДУПРЕЖДЕНИЕ

Не забывайте вызывать метод InitializeComponent в конструкторе своего застраничного класса!

Если вы забудете это сделать, то в корневом элементе не окажется содержимого, определенного в XAML-файле (потому что соответствующий ему BAML-код не загружен), а все поля, представляющие именованные объектные элементы, будут равны `null`.

КОПНЕМ ГЛУБЖЕ

Процедурный код внутри XAML

На самом деле XAML поддерживает еще и плохо документированную возможность «внутристраничного кода», помимо застраничного (как в ASP.NET). Для этого предназначено ключевое слово `Code` из пространства имен XAML:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyNamespace.MyWindow">
  <Button Click="button_Click">OK</Button>
  <x:Code><![CDATA[
    void button_Click(object sender, RoutedEventArgs e)
    {
      this.Close();
    }
  ]]></x:Code>
</Window>
```

При компиляции такого XAML-файла содержимое элемента `x:Code` копируется в частичный класс, находящийся в *.g.cs*-файле. Отметим, что процедурный язык в XAML-файле не указывается; он определяется проектом, содержащим этот файл.

Заключать процедурный код в скобки `<![CDATA[...]]>` необязательно, но это позволяет обойтись без замены знаков '<' на `<`, а амперсандов на `&`, поскольку секция `CDATA` полностью игнорируется анализаторами XML, а все остальное обрабатывается как XML-документ. (Правда, в качестве платы за это удобство вы не должны включать в код последовательность символов `]]>`, поскольку она закрывает секцию `CDATA`!)

Впрочем, не существует разумных причин засорять XAML-файлы таким «внутристраничным кодом». Мало того что при этом стирается различие между пользовательским интерфейсом и логикой приложения, так еще подобные файлы не будут отображаться в браузере, а Visual Studio не поддерживает для них стандартные средства работы с кодом, в том числе IntelliSense и цветовую подсветку синтаксиса.

FAQ

Можно ли BAML декомпилировать обратно в XAML?

Разумеется, да, поскольку на основе BAML можно построить граф объектов и затем сериализовать его в виде XAML независимо от того, как объекты были объявлены в первоначальном коде.

Первым делом необходимо найти объект, который будет корневым элементом XAML. Если его еще нет, то можно вызвать статический метод `System.Windows.Application.LoadComponent`, который загружает нужный объект из BAML:

```
System.Uri uri = new System.Uri("/WpfApplication1;component/MyWindow.xaml",
  System.UriKind.Relative);
Window window = (Window)Application.LoadComponent(uri);
```

Да, этот код загружает BAML, несмотря на суффикс *.xaml* в имени файла. Этим он отличается от предыдущего кода, где для загрузки XAML-файла использовался класс `FileStream`, поскольку в случае `LoadComponent` имя файла задается в виде универсального идентификатора ресурса (URI) и наличие физического файла с таким именем не требуется. Метод `LoadComponent` может автоматически загрузить BAML-код, внедренный в виде двоичного ресурса, если получит соответствующий URI (который по соглашению, принятому в MSBuild, должен совпадать с именем исходного XAML-файла). На самом деле автоматически генерируемый Visual Studio метод `InitializeComponent` вызывает именно метод `Application.LoadComponent` для загрузки внедренного BAML-кода, правда, другой перегруженный вариант. В главе 12 более подробно описан механизм получения внедренных ресурсов по URI.

Имея корневой элемент, мы можем воспользоваться классом `System.Windows.Markup.XamlWriter`, чтобы получить XAML-представление этого элемента (а следовательно, и всех его потомков). Класс `XamlWriter` содержит пять перегруженных вариантов статического метода `Save`; самый простой принимает экземпляр объекта и возвращает соответствующий XAML-код в виде строки:

```
string xaml = XamlWriter.Save(window);
```

Если вас пугает, что BAML-код легко «вскрывается», то вспомните, что то же самое можно сказать о любой программе, которая выполняется локально или локально отображает пользовательский интерфейс. (Например, нетрудно разобраться в HTML-коде, JavaScript-сценариях или CSS-стилях сайта.) Для популярной программы .NET Reflector имеется специальная надстройка BamlViewer (см. *http://codeplex.com/reflectoraddins*), которая показывает BAML-ресурс, внедренный в любую сборку, в виде декомпилированного XAML-кода.

Введение в XAML2009

Хотя XAML является языком общего назначения, сфера применения которого не ограничивается WPF, компилятор и анализаторы XAML для WPF архитектурно привязаны к WPF. Поэтому применение их в других технологиях

создает зависимость от WPF. В версии .NET Framework 4.0 это положение исправлено за счет новой сборки System.Xaml, которая содержит средства для работы с XAML. WPF (равно как WCF и WF) теперь зависят только от System.Xaml, но не друг от друга.

Одновременно в .NET Framework 4.0 появилось много улучшений самого языка XAML. Это второе поколение языка получило название XAML2009. (А чтобы не путаться, первое поколение иногда называют XAML2006.) Сборка System.Xaml поддерживает XAML2009, в отличие от прежних API (например, System.Windows.Markup.XamlReader и System.Windows.Markup.XamlWriter из предыдущего раздела), которые поддерживают только XAML2006.

Новые возможности XAML2009, описанные в этом разделе, не представляют собой ничего особо революционного, но составляют приятный набор последовательных улучшений XAML. Впрочем, радоваться рано: значительную часть новых возможностей не удастся использовать в WPF-проектах, поскольку компилятор XAML все еще основан на API XAML2006, равно как и конструктор и редактор WPF в Visual Studio, – не хватило времени на полную интеграцию.

Во время написания данной книги еще не было ясно, когда WPF полностью перейдет на XAML2009. (Отметим, что Silverlight также не поддерживает XAML2009; даже спецификация XAML2006 поддерживается не полностью!) Однако в WPF 4 новыми средствами можно пользоваться в автономных XAML-файлах при условии, что они загружаются в программу, построенную на базе API XAML2009. Таковыми, например, являются программа XAMLPAD2009, приложенная в качестве примера к этой книге, и Internet Explorer при использовании пространства имен XML netfx/2009.

Но изучить возможности XAML2009 интересно, пусть даже пока они не очень полезны. Большинство из них касается расширения множества типов, которые можно использовать в XAML напрямую. Это отличная новость для создателей библиотек классов, поскольку XAML2009 накладывает гораздо меньше ограничений на совместимость библиотек с XAML. Взятые по отдельности, новые средства не сильно повышают выразительность языка, но в совокупности упрощают решение реальных задач.

Полная поддержка универсальных классов

В XAML2006 корневой элемент может быть экземпляром универсального класса благодаря ключевому слову x:TypeArguments, значением которого является имя типа или список имен типов через запятую. Но поскольку атрибут x:TypeArguments разрешается использовать только в корневом элементе, то назвать XAML2006 дружественным к универсальным классам было бы преувеличением.

Традиционно это ограничение обходили, создавая обычный класс, наследующий универсальному. На такой класс уже можно ссылаться в XAML без ограничений. Например:

C#:

```
public class PhotoCollection : ObservableCollection<Photo> {}
```

XAML:

```
<custom:PhotoCollection>
  <custom:Photo .../>
  <custom:Photo .../>
</custom:PhotoCollection>
```

В XAML2009 атрибут x:TypeArguments может употребляться в любом элементе, поэтому, скажем, объекты класса ObservableCollection<Photo> допустимо создавать непосредственно в XAML:

```
<collections:ObservableCollection TypeArguments="custom:Photo">
  <custom:Photo .../>
  <custom:Photo .../>
</collections:ObservableCollection>
```

Здесь предполагается, что collections отображается на пространство имен System.Collections.ObjectModel, которое содержит класс ObservableCollection.

Словарные ключи произвольного типа

В XAML2009 преобразование типов применяется и к значениям атрибута x:Key, поэтому в словарь можно добавлять значения с нестроковыми ключами, не прибегая к расширениям разметки. Например:

```
<collections:Dictionary x:TypeArguments="x:Int32, x:String">
  <x:String x:Key="1">One</x:String>
  <x:String x:Key="2">Two</x:String>
</collections:Dictionary>
```

Здесь предполагается, что collections отображается на пространство имен System.Collections.Generic.

КОПНЕМ ГЛУБЖЕ

Отключение преобразования типов для нестроковых словарных ключей

Для обратной совместимости в классе XamlObjectWriter из XAML2009 имеется возможность отключить новый механизм автоматического преобразования типов. Это свойство XamlObjectWriterSettings.PreferUnconvertedDictionaryKeys. Принимая значение true, System.Xaml не будет конвертировать ключи, если словарь реализует неуниверсальный интерфейс IDictionary, при условии, что:

- System.Xaml уже потерпел неудачу при вызове метода IDictionary.Add для того же экземпляра, или
- словарь принадлежит известному типу .NET Framework, и System.Xaml точно знает, что для него необходимо преобразование.

Встроенные системные типы данных

В XAML2006 использовать встроенные типы данных .NET, например String или Int32, было неудобно, так как требовалось ссылаться на пространство имен System из сборки mscorlib; мы уже видели, как это выглядит:

```
<sys:Int32 xmlns:sys="clr-namespace:System;assembly=mscorlib">7</sys:Int32>
```

В XAML2009 в пространство имен языка XAML добавлено 13 наиболее употребительных типов данных .NET. В предположении, что данному пространству имен сопоставлен префикс x, это следующие типы: x:Byte, x:Boolean, x:Int16, x:Int32, x:Int64, x:Single, x:Double, x:Decimal, x:Char, x:String, x:Object, x:Uri, и x:TimeSpan. Следовательно, предыдущий фрагмент можно переписать в таком виде:

```
<x:Int32 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">7</x:Int32>
```

Но обычно в XAML-файле, где уже есть ссылка на пространство имен XAML, это записывается проще:

```
<x:Int32>7</x:Int32>
```

Создание объектов с помощью конструктора с аргументами

В XAML2009 появилось новое ключевое слово x:Arguments, которое позволяет задать один или несколько аргументов для передачи конструктору класса. Рассмотрим, к примеру, класс System.Version, в котором имеется конструктор по умолчанию и четыре конструктора с параметрами. В XAML2006 невозможно было создать экземпляр этого класса, если не существовало подходящего конвертера типа (при условии, конечно, что вас не устраивал конструктор по умолчанию, создающий версию 0.0).

В XAML2009 можно создать объект этого класса с помощью конструктора, принимающего в качестве параметра простую строку:

```
<sys:Version x:Arguments="4.0.30319.1"/>
```

При этом аргумент конструктора необязательно должен быть строкой; при необходимости тип значения атрибута преобразуется.

В отличие от x:TypeArguments, ключевое слово x:Arguments не позволяет задавать в качестве значения атрибута несколько аргументов в одной строке через запятую. Но можно задавать их в виде элементов, вложенных в x:Arguments. Например, вызвать конструктор класса System.Version, который принимает четыре целых числа, можно так:

```
<sys:Version>
<x:Arguments>
  <x:Int32>4</x:Int32>
  <x:Int32>0</x:Int32>
  <x:Int32>30319</x:Int32>
  <x:Int32>1</x:Int32>
</x:Arguments>
</sys:Version>
```

Создание экземпляров с помощью фабричных методов

С помощью нового ключевого слова x:FactoryMethod в XAML2009 можно создать экземпляр класса, вообще не имеющего открытых конструкторов. x:FactoryMethod позволяет указать произвольный открытый статический метод, который возвращает объект нужного типа. Например, в следующем XAML-коде используется объект типа Guid, возвращаемый статическим методом Guid.NewGuid:

```
<Label xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
       xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
       xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <sys:Guid x:FactoryMethod="sys:Guid.NewGuid"/>
</Label>
```

Если x:FactoryMethod используется совместно с x:Arguments, то аргументы передаются статическому фабричному методу, а не конструктору. Таким образом, в следующем примере вызывается статический метод Marshal.GetExceptionForHR, который принимает код ошибки HRESULT и возвращает соответствующее ему исключение .NET, которое слой интероперабельности CLR возбуждает при возникновении такой ошибки:

```
<Label xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
       xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
       xmlns:sys="clr-namespace:System;assembly=mscorlib"
       xmlns:interop=
         "clr-namespace:System.Runtime.InteropServices;assembly=mscorlib">
  <sys:Exception x:FactoryMethod="interop:Marshal.GetExceptionForHR">
  <x:Arguments>
    <x:Int32>0x80004001</x:Int32>
  </x:Arguments>
  </sys:Exception>
</Label>
```

На рис. 2.3 показано, как XAMLPAD2009 реагирует на размещение двух меток Label в одной панели StackPanel.

Гибкость присоединения обработчиков событий

В XAML2006 нельзя было присоединять обработчики событий в автономном XAML-файле. В XAML2009 это стало возможно при условии, что удается найти корневой экземпляр и в нем имеется метод с указанным именем и подходящей сигнатурой. Кроме того, в XAML2009 значением атрибута события может быть любое расширение разметки, которое возвращает соответствующий делегат:

```
<Button Click="{custom:DelegateFinder Click}"/>
```

Как любое расширение разметки, оно может принимать произвольные входные данные и выполнять логические действия для поиска делегата.

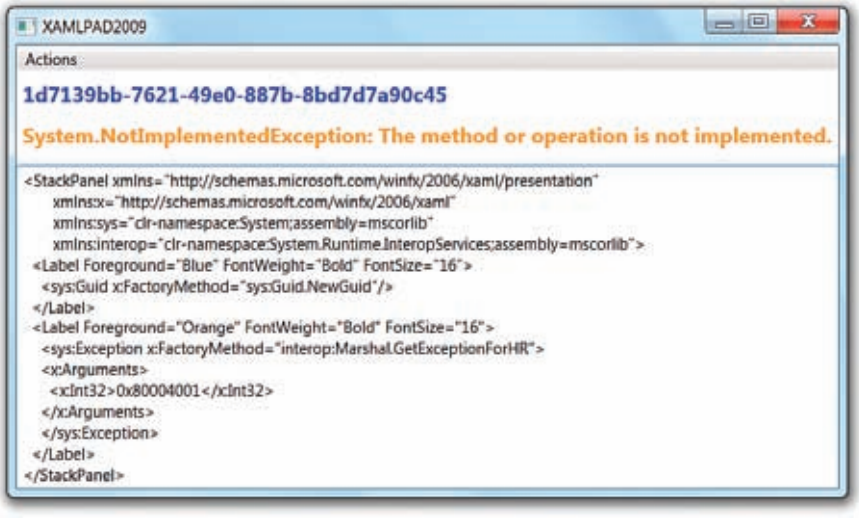


Рис. 2.3. *Отображение двух экземпляров, полученных с помощью статических фабричных методов*

Определение новых свойств

Основной задачей языка XAML является создание экземпляров существующих классов и установка значений уже определенных в них свойств. Тем не менее в XAML2009 появилось два новых элемента – `x:Members` и `x:Property`, – которые позволяют *определять* дополнительные свойства непосредственно на XAML. Однако эта функциональность неприменима к WPF. Как она используется в версии XAML для Windows Workflow Foundation, показано в следующем примере:

```
<Activity x:Class="ActivityLibrary1.Activity1" ...>
<x:Members>
  <x:Property Name="argument1" Type="InArgument(x:Int32)"/>
  <x:Property Name="argument2" Type="OutArgument(x:String)"/>
</x:Members>
...
</Activity>
```

Трюки с классами чтения и записи XAML

Мы уже видели, как читать и записывать XAML-код с помощью методов `XamlReader.Load` и `XamlWriter.Save` из пространства имен `System.Windows.Markup`. Эти API существовали еще в самой первой версии WPF и до сих пор прекрасно работают с содержимым WPF – при условии, что оно не выходит за рамки подмножества XAML2006.

Новая сборка System.Xaml содержит абстрактные базовые классы System.Xaml.XamlReader и System.Xaml.XamlWriter (не путайте с вышеупомянутыми классами для чтения/записи), лежащие в основе нового способа чтения и записи XAML. Классы из сборки System.Xaml гораздо более гибкие, чем конвертация по принципу «черного ящика», реализованная в их предшественниках. И при этом они поддерживают XAML2009.

Обзор

Класс XamlReader предназначен для генерации потока логических узлов XAML из произвольного источника (определяемого конкретной реализацией производного класса), а XamlWriter на входе получает такой поток узлов и выводит его произвольным способом. В настоящий момент поставляются следующие открытые производные классы для чтения и записи:

Считыватели (производные от System.Xaml.XamlReader):

- System.Xaml.XamlXmlReader – читает XML-код (из System.Xml.XmlReader, System.IO.TextReader, System.IO.Stream или из файла, заданного своим именем в виде строки).
- System.Xaml.XamlObjectReader – читает существующий граф объектов.
- System.Windows.Baml2006.Baml2006Reader – читает BAML-код (в WPF все еще используется формат 2006 года).
- System.Xaml.XamlBackgroundReader – обертывает другой объект XamlReader, реализуя двойную буферизацию, что позволяет считывателю работать не в том же потоке, что записыватель.

Записыватели (производные от System.Xaml.XamlWriter):

- System.Xaml.XamlXmlWriter – записывает XML (используя System.Xml.XmlWriter, System.IO.TextWriter или Stream).
- System.Xaml.XamlObjectWriter – создает граф объектов.

Средства чтения и записи XAML работают совместно, так же как и любые другие подобные средства в .NET Framework, например находящиеся в пространствах имен System.IO и System.Xml. В результате появилась своеобразная экосистема, в которой различные средства чтения и записи комбинируются друг с другом, а понятие логических узлов XAML играет роль общего звена. Это изображено на рис. 2.4, где показаны считыватели и записыватели, входящие в состав .NET Framework. Поток узлов XAML ассоциирован не столько с текстовым представлением XML, сколько с логическим понятием иерархии объектов, в которой различные члены могут принимать разные значения.

Части, обозначенные на рис. 2.4 многоточием, важны, – на этом месте могут находиться созданные сторонними производителями средства чтения и записи, которые расширяют возможности трансформации. За последние несколько лет различные организации предоставили целый ряд конвертеров для преобразования XAML в другие форматы файлов и наоборот (хотя пока не на

основе новых API). В их число входит более 40 форматов трехмерной графики (Autodesk 3ds Max и Maya, AutoCAD DXF, NewTek LightWave и т. д.), Adobe Illustrator/Photoshop/Flash/Fireworks, SVG, HTML 5 Canvas, Visio, PowerPoint, Windows Metafile (WMF), Enhanced Metafile (EMF), и даже формы Visual Basic 6!

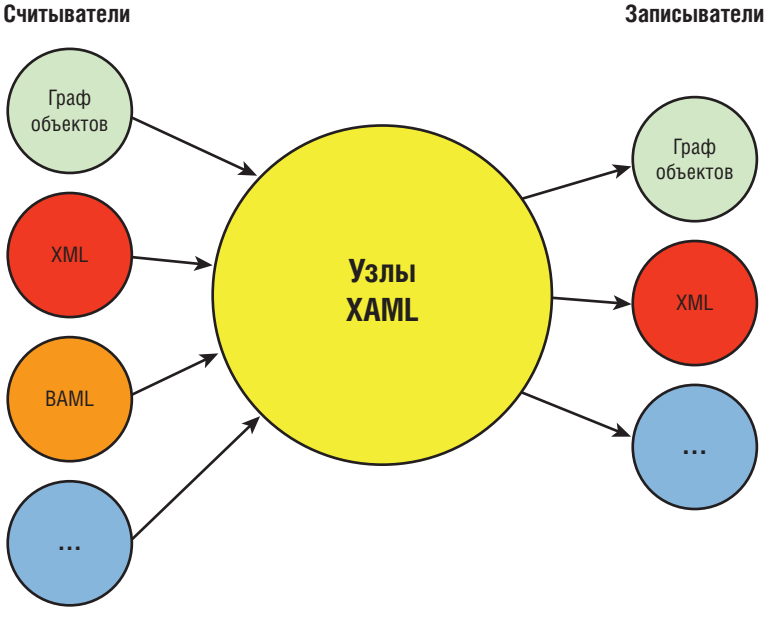


Рис. 2.4. *Средства чтения и записи работают совместно, обеспечивая разнообразные способы трансформации*

ПРЕДУПРЕЖДЕНИЕ

Описанная в этом разделе функциональность применима главным образом к вариантам XAML, не относящимся к WPF!

Данный раздел не без причины называется «Трюки с классами чтения и записи XAML». Конечно, с этими классами приятно работать, но пока рекомендуем ограничиться экспериментами. Текущая версия `XamlObjectReader` не поддерживает некоторые аспекты WPF, поэтому для сериализации в виде WPF XAML придется по-прежнему использовать класс `System.Windows.Markup.XamlWriter`. Но если вы используете XAML не для WPF, то все должно работать прекрасно.

FAQ



Почему для чтения XAML-файлов лучше использовать XamlXmlReader, а не просто XmlReader? Разве XAML не является диалектом XML?

`XamlXmlReader` в действительности пользуется классом `XmlReader`, но дополнительно обеспечивает еще две важные возможности.

- Абстрагирует различия в представлениях XML, эквивалентных с точки зрения XAML.
- Порождает поток узлов XAML, совместимый с любым записывателем XAML и содержащий дополнительную информацию, которая отсутствует в исходном XML.

Первая возможность существенна для сокращения работы, связанной с чтением XAML. Например, следующие три фрагмента XAML-кода выражают одну и ту же концепцию – кнопку `Button`, для которой свойство `Content` содержит строку `"OK"`:

```
<!-- Неявная установка свойства содержимого: -->
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  OK
</Button>
<-- Установка свойства с помощью синтаксиса элементов: -->
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  OK
</Button.Content>
</Button>
<!-- Установка свойства с помощью синтаксиса атрибутов: -->
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK"/>
```

Для `XmlReader` эти варианты выглядят совершенно по-разному, но `XamlXmlReader` преобразует их к одному виду. Именно это и нужно инструментам, читающим XAML (если только задачей инструмента не является нечто вроде принудительного применения стилистических правил к текстовому представлению XAML), и именно это требует большой дополнительной работы. Например, `XamlXmlReader` может определить, что первый вариант эквивалентен остальным двум, только после полного исследования определения кнопки `Button` и обнаружения в нем свойства содержимого под названием `Content`.

Что же касается второй возможности, то дополнительная информация, присутствующая в потоке узлов XAML, которую предоставляет `XamlXmlReader` (или любой другой считыватель XAML), – это результат комбинирования входных данных с определениями типов, на которые имеются ссылки. Например, с помощью `XamlXmlReader` можно узнать, что `Content` – действительно свойство содержимого и его тип – `System.Object`.

Циклы обработки узлов

Для конвертации из одного формата файла в другой необходимо сначала получить узлы XAML от соответствующего считывателя, а затем отправить их подходящему записывателю. Классы `XamlReader` и `XamlWriter` разработаны так, чтобы максимально упростить этот процесс, они позволяют написать тривиальный «цикл обработки узлов», в котором выполняются все операции чтения и записи от начала до конца. В предположении, что `reader` – считыватель XAML, а `writer` – записыватель, такой цикл выглядит следующим образом:

```
// Простой цикл обработки узлов
while (reader.Read())
{
  writer.WriteNode(reader);
}
```

Что на самом деле происходит в этом цикле, зависит от конкретного типа считывателя и записывателя. Например, задачей программы XAMLPAD2009 является чтение XAML в формате XML (представленного в виде строк) и создание графа объектов, который можно связать с ее собственным пользовательским интерфейсом (а значит, и визуализировать). В листинге 2.1 эта задача решается с помощью цикла обработки узлов, в котором роль считывателя играет `XamlXmlReader`, а записывателем выступает `XamlObjectWriter`. Самое трудоемкое здесь – прочитать с помощью `XamlXmlReader` строку XML. Проще всего сделать это, создав для чтения строки объект `System.IO.StringReader` и передав его объекту `XamlXmlReader` (это можно сделать, так как `StringReader` является подклассом `TextReader`).

Листинг 2.1. Простой цикл обработки узлов для преобразования XAML-кода, представленного строками XML, в граф объектов

```
public static object ConvertXmlStringToObjectGraph(string xmlString)
{
  // String -> TextReader -> XamlXmlReader
  using (TextReader textReader = new StringReader(xmlString))
  using (XamlXmlReader reader = new XamlXmlReader(textReader,
           System.Windows.Markup.XamlReader.GetWpfSchemaContext()))
  using (XamlObjectWriter writer = new XamlObjectWriter(reader.SchemaContext))
  {
    // Простой цикл обработки узлов
    while (reader.Read())
    {
      writer.WriteNode(reader);
    }
    // По завершении работы XamlObjectWriter здесь будет
    // находиться экземпляр корневого объекта
    return writer.Result;
  }
}
```

Объекту XamlObjectWriter передается контекст WPF-схемы, чтобы он лучше работал с XAML для WPF. Это позволяет использовать ряд возможностей и нюансов совместимости, которые неприменимы к XAML-файлам общего вида.

Чтение XAML

Средства чтения XAML предоставляют полезную информацию о результирующем потоке узлов XAML, поэтому можно не просто слепо выводить их в другой форме, но и делать что-то еще, например трансформировать содержимое в процессе преобразования.

Самым важным свойством XamlReader, которое имеет смысл анализировать при записи узла в цикле, является NodeType, способное принимать одно из восьми перечисляемых значений:

- StartObject – считыватель позиционирован в начале явно представленного объекта, например открывающего тега XML-элемента или расширения разметки, указанного в качестве значения свойства.
- GetObject – считыватель позиционирован в начале *неявного* объекта, например коллекции, которая в XAML явно не представлена, хотя ее элементы присутствуют (как в примере ListBox, приведенном в разделе «Элементы коллекций»).
- EndObject – считыватель позиционирован в конце объекта (который ранее встречался в виде StartObject или GetObject). Каждому узлу StartObject или GetObject соответствует узел EndObject, который встретится в потоке позже.
- StartMember – считыватель позиционирован в начале некоторого члена объекта: свойства (присоединенного или нет), события (присоединенного или нет) либо директивы XAML, например x:Key. Каждый атрибут принадлежит некоторому родительскому объекту, поэтому узлу StartMember обязательно предшествует узел StartObject или GetObject. Отметим, что в XML неважно, задан ли член с помощью синтаксиса атрибута свойства или элемента свойства, – в любом случае он является членом, а не объектом.
- EndMember – считыватель позиционирован в конце члена объекта (для которого ранее встречался узел StartMember). Каждому узлу StartMember обязательно соответствует находящийся далее в потоке узел EndMember.
- Value – считыватель позиционирован в начале значения члена объекта. Поскольку каждое значение ассоциировано с каким-то членом, то невозможно появление узла Value до соответствующего ему StartMember (и предшествующего ему узла StartObject или GetObject).
- NamespaceDeclaration – считыватель позиционирован на объявлении пространства имен XML (которое ассоциирует пространство имен с префиксом). Отметим, что такой узел непосредственно предшествует узлу StartObject, который «содержит» эти объявления. Это может показаться удивительным, но, учитывая, что объявления пространств имен предоставляют контекст даже для корневого элемента, важно, чтобы контекст был определен предварительно.

- None – считыватель позиционирован на чем-то, не являющемся реальным узлом, например находящемся в конце файла. Узлы такого типа можно без опаски игнорировать.

В классе XamlReader определены четыре важных свойства, которые позволяют извлекать нужные данные об узле любого типа: Type, Member, Value и Namespace. Возвращаемые ими данные зависят от типа узла в текущей позиции считывателя. Например, если свойство NodeType имеет значение StartObject, то в Type находится экземпляр класса XamlType, а остальные три свойства равны null. Если NodeType имеет значение StartMember, то Member содержит экземпляр XamlMember, а остальные три свойства равны null. Если NodeType равно Value, то лишь Value отлично от null. А если NodeType равно NamespaceDeclaration, то ненулевым будет только свойство Namespace.

Кроме того, все средства чтения XAML в .NET Framework 4.0 (за исключением XamlObjectReader) реализуют интерфейс IXamlLineInfo, который выдает информацию о номере строки, если она имеется. Когда свойство HasLineInfo равно true, можно получить данные о номере строки и позиции в ней, обратившись к свойствам LineNumber и LinePosition соответственно.

FAQ

Откуда берутся экземпляры XamlType и XamlMember, обнаруживаемые считывателями XAML?

Эти классы представляют собой специфическую для XAML форму отражения .NET.

Класс XamlType обертывает System.Type (который можно получить с помощью свойства UnderlyingType), добавляя такие специфические для XAML элементы, как свойства содержимого, присоединенные свойства и многое другое. Этот уровень абстракции также позволяет при необходимости представлять с помощью класса XamlType не только типы .NET.

А класс XamlMember расширяет System.Reflection.MemberInfo (который можно получить с помощью свойства UnderlyingMember, если свойство MemberInfo действительно существует). Он также добавляет такие специфические для XAML свойства, как IsDirective и PreferredXamlNamespace.

Чтобы продемонстрировать, как на самом деле выглядит работа со считывателем XAML, в табл. 2.1 показан весь поток узлов, порождаемых XamlXmlReader при чтении XAML-разметки в листинге 2.2. Отступы значений XamlNodeType иллюстрируют вложенность объектов, их членов и значений.

Листинг 2.2. Пример XAML-разметки для демонстрации поведения XamlXmlReader

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <!-- Задаем имена двумя разными способами -->
  <Button Name="okButton" Click="okButton_Click">OK</Button>
```

```
    <Button x:Name="cancelButton">Cancel</Button>
    <ListBox>
      <-- Задаем содержимое тремя разными способами -->
      <ListBoxItem Content="Item 1"/>
      <ListBoxItem>Item 2</ListBoxItem>
      <ListBoxItem>
      <ListBoxItem.Content>
        Item 3
      </ListBoxItem.Content>
      </ListBoxItem>
    </ListBox>
  </StackPanel>
```

Таблица 2.1. Поток узлов XAML, порождаемых XamlXmlReader при чтении разметки в листинге 2.2

XamlNodeType	Данные	Строка	Позиция
NamespaceDeclaration	Namespace="…/xaml/presentation", Prefix=""	1	13
NamespaceDeclaration	Namespace="…/xaml", Prefix="x"	2	13
StartObject	Type=StackPanel	1	2
StartMember	Member=**потомок типа** UIElementCollection	4	4
GetObject	null	4	4
StartMember	Member=_Items, XamlDirective **типа** List<Object>	4	4
StartObject	Type=Button	4	4
StartMember	Member=Name **типа** String	4	11
Value	Value="okButton"	4	11
EndMember	null	4	11
StartMember	Member=Click **типа** RoutedEventHander (IsEvent=true)	4	27
Value	Value="okButton_Click"	4	27
EndMember	null	4	27
StartMember	Member=Content **типа** Object	4	54
Value	Value="OK"	4	54
EndMember	null	4	54
EndObject	null	4	54
StartObject	Type=Button	5	4
StartMember	Member=Name, XamlDirective **типа** String	5	11

XamlNodeType	Данные	Строка	Позиция
Value	Value="cancelButton"	5	11
EndMember	null	5	11
StartMember	Member=Content **типа** Object	5	41
Value	Value="Cancel"	5	41
EndMember	null	5	41
EndObject	null	5	41
StartObject	Type=ListBox	6	4
StartMember	Member=Items **типа** ItemCollection	8	6
GetObject	null	8	6
StartMember	Member=_Items, XamlDirective **типа** List<Object>	8	6
StartObject	Type=ListBoxItem	8	6
StartMember	Member=Content **типа** Object	8	18
Value	Value="Item 1"	8	18
EndMember	null	8	18
EndObject	null	9	6
StartObject	Type=ListBoxItem	9	6
StartMember	Member=Content **типа** Object	9	26
Value	Value="Item 2"	9	26
EndMember	null	9	26
EndObject	null	9	26
StartObject	Type=ListBoxItem	10	6
StartMember	Member=Content **типа** Object	11	6
Value	Value="Item 3"	13	7
EndMember	null	13	7
EndObject	null	14	7
EndMember	null	15	5
EndObject	null	15	5
EndMember	null	15	5
EndObject	null	15	5
EndMember	null	16	3
EndObject	null	16	3
EndMember	null	16	3
EndObject	null	16	3

Обратите внимание, что все три элемента ListBoxItem в табл. 2.1 представлены одинаково, так же как и оба элемента Button, хотя и возможно провести различие между использованием свойства Name кнопки Button и директивы XAML x:Name. (В последнем случае XamlMember наследует типу XamlDirective, свойство IsDirective которого равно true.)

Также отметим, что узлы GetObject, EndMember и EndObject не сопровождаются никакой дополнительной информацией; ее следует получать из других узлов в потоке. Из-за этого для выполнения нетривиальных преобразований в формат XAML часто требуется создавать собственный стек для хранения данных, относящихся к объектам и/или их членам.

КОПНЕМ ГЛУБЖЕ

Совместимость разметки

Пространство имен совместимости разметки (http://schemas.openxmlformats.org/markup-compatibility/2006, обычно ему сопоставляется префикс mc) содержит атрибут Ignorable, информирующий процессор XAML о необходимости игнорировать все элементы/атрибуты из указанных пространств имен, которым нельзя сопоставить типы или члены типов .NET. (В этом пространстве имен имеется также атрибут ProcessContent, который отменяет действие Ignorable для некоторых типов в игнорируемых пространствах имен.)

Программа Expression Blend использует эту возможность для добавления в XAML-содержимое свойств, имеющих смысл только на этапе конструирования. На этапе выполнения эти свойства будут проигнорированы. Пример:

```
<StackPanel xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="100" d:DesignHeight="100">
  ...
</StackPanel>
```

Значением атрибута mc:Ignorable может быть список пространств имен, разделенных пробелами, а значением атрибута mc:ProcessContent — список элементов, также разделенных пробелами.

Когда XamlXmlReader встречает игнорируемое содержимое, которое не может быть разрешено, он не порождает для него никаких узлов. Если же игнорируемое содержимое можно разрешить, то узлы порождаются как обычно. Поэтому потребителям не нужно специально ничего делать для корректной обработки совместимости разметки.

Запись в объекты

Приложение XAMLPAD2009 не конвертирует XAML в объекты, находящиеся в памяти. Оно лишь модифицирует XAML-содержимое, чтобы гарантировать успешную визуализацию более широкого спектра конструкций WPF XAML. Точнее, производятся две модификации.

- Убираются все члены, относящиеся к событиям, так как если обработчик события не найден, то `XamlObjectWriter` возбуждает исключение, например, с таким сообщением: `Failed to create a 'Click' from the text 'button_Click'`. Отметим, что в классе `XamlObjectWriter` имеется свойство `RootObjectInstance`, которому можно присвоить объект с подходящими обработчиками событий, но проще всего эти события просто выкинуть – для инструмента экспериментирования с XAML такой подход вполне приемлем. Кроме этого, убирается атрибут `x:Class`, потому что в автономном XAML-коде он недопустим.
- Элемент `Window` конвертируется в `Page`. В главе 7 эти элементы рассматриваются подробно, но смысл в том, что элемент `Window` не может быть потомком другого элемента, а XAMLPAD2009 всегда пытается присоединить корневой объект в качестве непосредственного потомка своего собственного пользовательского интерфейса. Существуют и другие способы справиться с этой трудностью (например, увидев, что корневым элементом является `Window`, создавать из него окно), но описанной выше замены одного узла XAML другим для учебного примера достаточно.

В листинге 2.3 показан специализированный цикл обработки узлов, в котором преобразование содержимого XAML-строки в объекты сопровождается двумя дополнительными операциями.

Листинг 2.3. Цикл обработки узлов для преобразования XAML-строки в граф объектов с модификациями

```
public static object ConvertXmlStringToMorphedObjectGraph(string xmlString)
{
  // String -> TextReader -> XamlXmlReader
  using (TextReader textReader = new StringReader(xmlString))
  using (XamlXmlReader reader = new XamlXmlReader(textReader,
           System.Windows.Markup.XamlReader.GetWpfSchemaContext()))
  using (XamlObjectWriter writer = new XamlObjectWriter(reader.SchemaContext))
  {
    // Цикл обработки узлов
    while (reader.Read())
    {
      // Пропустить события и x:Class
      if (reader.NodeType == XamlNodeType.StartMember &&
          reader.Member.IsEvent || reader.Member == XamlLanguage.Class)
      {
        reader.Skip();
      }

      if (reader.NodeType == XamlNodeType.StartObject &&
          reader.Type.UnderlyingType == typeof(Window))
      {
        // Преобразовать Window в Page
        writer.WriteStartObject(new XamlType(typeof(Page),
          reader.SchemaContext));
      }
```

```
        else
        {
          // В противном случае вывести узел без изменений
          writer.WriteNode(reader);
        }
      }

      // По завершении работы XamlObjectWriter здесь будет
      // экземпляр корневого объекта
      return writer.Result;
    }
  }
```

В листинге 2.3 для пропуска членов-событий (`IsEvent = true`) и атрибутов `x:Class` применяется метод `Skip` класса `XamlReader`. (Атрибут `x:Class` опознается с помощью удобного статического класса `System.Xaml.XamlLanguage`, в котором все виды директив `XamlDirective` и встроенных значений `XamlType` определены как свойства, доступные только для чтения; это упрощает сравнение.) Когда считыватель позиционирован на узле `StartObject` или `StartMember`, метод `Skip` сдвигает указатель потока на узел, *следующий* за соответствующим узлом `EndObject`/`EndMember` (пропуская все вложенные объекты/члены, что нам и нужно). Если же считыватель позиционирован на узле любого другого типа, то вызов `Skip` эквивалентен повторному вызову `Read`: он переходит на следующий узел.

Для замены `Window` на `Page` нужно подменить только узел `StartObject`. Напомним, что с узлом `EndObject` не ассоциированы никакие данные, его интерпретация зависит от других узлов в потоке. Поэтому `EndObject` для `Window` вполне может стать `EndObject` для `Page`. Однако подобная подмена других членов `Window` членами `Page` некорректна, поскольку они уже были разрешены считывателем как члены `Window` до начала цикла обработки узлов. В исходном коде, прилагаемом к книге, дополнительно создается новый член `Page` для каждого члена `Window`, к которому такое преобразование применимо.

В листингах 2.1 и 2.3 по завершении цикла обработки узлов в свойство `XamlObjectWriter.Result` записывается экземпляр корневого объекта. Точнее, после вывода каждого узла `EndObject` в `XamlObjectWriter.Result` помещается ссылка на соответствующий ему объект. А так как последний записанный в поток узел `EndObject` соответствует корневому элементу, то окончательным значением `Result` оказывается корень графа объектов.

Запись в формате XML

Запись WPF-объектов в XAML-файл в формате XML – часто встречающаяся задача. Поскольку в настоящее время класс `XamlObjectReader` не поддерживает WPF-объекты, в листинге 2.4 показано, как можно конвертировать XML из одного варианта в другой, совместно используя `XamlObjectReader` и `XamlObjectWriter`. При этом получается простейший «очиститель XAML», который нормализует входной XML-документ, убирая комментарии и формируя разметку с единообразно расставленными пробелами.

Листинг 2.4. «Очиститель XAML», нормализующий входной XML

```
public static string RewriteXaml(string xmlString)
{
  // String -> TextReader -> XamlXmlReader
  using (TextReader textReader = new StringReader(xmlString))
  using (XamlXmlReader reader = new XamlXmlReader(textReader))
  // TextWriter -> XmlWriter -> XamlXmlWriter
  using (StringWriter textWriter = new StringWriter())
  using (XmlWriter xmlWriter = XmlWriter.Create(textWriter,
           new XmlWriterSettings { Indent = true, OmitXmlDeclaration = true }))
  using (XamlXmlWriter writer = new XamlXmlWriter(xmlWriter,
           reader.SchemaContext))
  {
    // Простой цикл обработки узлов
    while (reader.Read())
    {
      writer.WriteNode(reader);
    }

    return textWriter.ToString();
  }
}
```

Здесь практически вся работа сводится к настройке считывателя и записывателя. Экземпляр `XamlXmlReader` конструируется так же, как в предыдущем листинге, а `XamlXmlWriter` конструируется из объекта `System.IO.StringWriter`. (`XmlWriter` можно было сконструировать также из объекта `StringBuilder`.) Использование `XmlWriter` позволяет организовать аккуратную печать (каждый элемент на отдельной строке с отступами), а заодно удалить ненужные XML-объявления (`<?xml version="1.0" encoding="utf-16"?>`). Но если вам это неважно и вы готовы смириться с выводом содержимого в одну строку, то можно просто передать конструктору `XamlXmlWriter` объект `StringWriter` (поскольку он наследует `TextWriter`), не оборачивая его в `XmlWriter`.

```
// TextWriter -> XamlXmlWriter
using (StringWriter textWriter = new StringWriter())
using (XamlXmlWriter writer = new XamlXmlWriter(textWriter,
         reader.SchemaContext))
{
  ...
}
```

XamlServices

Чтобы пользователю приходилось писать меньше кода, самые распространенные случаи употребления средств чтения и записи XAML инкапсулированы в простые статические методы, определенные в классе `System.Xaml.XamlServices`, а именно:

- `Load` – есть несколько перегруженных вариантов, принимающих имя файла в виде строки, объекты `Stream`, `TextReader`, `XmlReader` или `XamlReader`. Все

они возвращают корень соответствующего графа объектов, как и прежний метод XamlReader.Load. Внутри Load вся работа производится объектами XamlXmlReader и XamlObjectWriter, как в листинге 2.1.

- Parse – как и Load, метод Parse возвращает корень графа объектов, но на входе принимает XAML-содержимое в виде строки. Внутри он создает из этой строки объект StringReader, затем XmlReader и наконец XamlXmlReader, от имени которого можно уже вызвать метод Load. Таким образом, Parse аналогичен методу ConvertXmlStringToObjectGraph, представленному в листинге 2.1.
- Save – принимает на входе объект и, в зависимости от перегруженного варианта, возвращает его содержимое в виде строки, объекта Stream, TextWriter, XmlWriter либо XamlWriter или даже сохраняет содержимое объекта прямо в текстовом файле. Внутри Save создает экземпляры XamlObjectReader и XamlXmlWriter (если только ему уже не передан объект XamlWriter). Он присваивает свойствам Indent и OmitXmlDeclaration объекта XamlWriter значение true, как в листинге 2.4.
- Transform – выполняет тривиальный цикл обработки узлов, применяя считыватель и записыватель, которые ему переданы.

На самом деле метод XamlServices.Transform работает чуть хитрее, чем показанный выше тривиальный цикл обработки узлов. Он сохраняет информацию о номере строки и позиции в ней, если считыватель и записыватель поддерживают интерфейсы для их создания и использования (IXamlLineInfo для считывателя и IXamlLineInfoConsumer для записывателя). Таким образом, Transform на самом деле выполняет следующее:

```
public static void Transform(XamlReader reader, XamlWriter writer)
{
  IXamlLineInfo producer = reader as IXamlLineInfo;
  IXamlLineInfoConsumer consumer = writer as IXamlLineInfoConsumer;
  bool transferLineInfo = (producer != null && producer.HasLineInfo &&
                           consumer != null && consumer.ShouldProvideLineInfo);
  // Улучшенный цикл обработки узлов
  while (reader.Read())
  {
    // Передать информацию о строке
    if (transferLineInfo && producer.LineNumber > 0)
      consumer.SetLineInfo(producer.LineNumber, producer.LinePosition);

    writer.WriteNode(reader);
  }
}
```

Следовательно, от цикла обработки узлов из листинга 2.1 можно отказаться (и немного улучшить результат), заменив его методом XamlServices.Transform, как показано в листинге 2.5. Впрочем, метод ConvertXmlStringToObjectGraph вообще не нужен, поскольку он дублирует XamlServices.Parse.

Листинг 2.5. Небольшое упрощение листинга 2.1

```
public static object ConvertXmlStringToObjectGraph(string xmlString)
{
  // String -> TextReader -> XamlXmlReader
  using (TextReader textReader = new StringReader(xmlString))
  using (XamlXmlReader reader = new XamlXmlReader(textReader,
           System.Windows.Markup.XamlReader.GetWpfSchemaContext()))
  using (XamlObjectWriter writer = new XamlObjectWriter(reader.SchemaContext))
  {
  // Цикл обработки узлов
  XamlServices.Transform(reader, writer);

    // По завершении работы XamlObjectWriter здесь будет
    // экземпляр корневого объекта
  return writer.Result;
  }
}
```

ПРЕДУПРЕЖДЕНИЕ

Берегитесь подводных камней XamlServices в WPF XAML!

Быть может, вы думаете, что комбинация `XamlServices.Parse` и `XamlServices.Save` позволит реализовать «очиститель XAML» из листинга 2.4 в следующем простом, хотя и неэффективном виде:



```
public static string RewriteXaml(string xmlString)
{
  return XamlServices.Save(XamlServices.Parse(xmlString));
}
```

Это неэффективно потому, что на внутреннем уровне строка сначала проходит через `XamlXmlReader`, потом записывается в объект с помощью `XamlObjectWriter` (корень которому возвращает `XamlServices.Parse`), затем эту иерархию объектов читает `XamlObjectReader` и только после этого окончательная строка записывается в `XmlWriter` с помощью `XamlXmlWriter`. Этот промежуточный шаг создания объектов проблематичен не только по причинам производительности. Он также требует специальной обработки на уровне XAML, в частности присоединения обработчиков событий или разрешения директивы `x:Class`.

Но еще хуже то, что приведенный код просто не работает, поскольку `XamlObjectWriter` в настоящее время не поддерживает WPF-объекты. Можно было бы вместо этого воспользоваться более старыми классами `XamlReader` и `XamlWriter`:

```
return System.Windows.Markup.XamlWriter.Save(
  System.Windows.Markup.XamlReader.Parse(xmlString));
```

Или, если нужна красивая печать:

```
using (StringWriter textWriter = new StringWriter())
using (XmlWriter xmlWriter = XmlWriter.Create(textWriter,
```

```
            new XmlWriterSettings { Indent = true, OmitXmlDeclaration = true }))
  {
    System.Windows.Markup.XamlWriter.Save(
      System.Windows.Markup.XamlReader.Parse(xmlString), xmlWriter);
    return textWriter.ToString();
  }
```

Но и этим подходам свойствен промежуточный шаг, связанный с преобразованием XAML-разметки в объекты.

СОВЕТ

Набор инструментов Microsoft XAML Toolkit (доступен по адресу *http://code.msdn.microsoft.com/XAML*), построенный на основе классов из пространства имен `System.Xaml`, предлагает несколько очень интересных возможностей, например интеграцию XAML с инструментом FxCop и объектную модель документа XAML. XAML DOM – это набор API, совместимых с LINQ, которые еще больше упрощают исследование и модификацию XAML-содержимого по сравнению со средствами чтения и записи, описанными в этой главе. Этот набор инструментов также включает дополнительные контексты схем: `SilverlightSchemaContext` для Silverlight XAML и `UISchemaContext`, в котором реализована общая абстракция для WPF XAML и Silverlight XAML.

Ключевые слова XAML

В пространстве имен языка XAML (`http://schemas.microsoft.com/winfx/2006/xaml`) определен ряд ключевых слов, которые должны особым образом обрабатываться компилятором или анализатором XAML. В основном они управляют различными аспектами того, как элементы интерпретируются в процедурном коде, но некоторые полезны и сами по себе. С некоторыми из них мы уже встречались (`Key`, `Name`, `Class`, `Subclass` и `Code`), а в табл. 2.2 перечислены они все. Мы используем традиционный префикс `x`, потому что именно так они обычно употребляются в XAML и в документации.

КОПНЕМ ГЛУБЖЕ

Специальные атрибуты, определенные консорциумом W3C

В дополнение к ключевым словам пространства имен язык XAML поддерживает также два специальных атрибута, определенных для XML консорциумом World Wide Web Consortium (W3C): `xml:space` для управления разбором пробелов и `xml:lang` для объявления языка и культуры документа. При этом префикс `xml` неявно отображается на стандартное пространство имен XML `http://www.w3.org/XML/1998/namespace`.

Таблица 2.2. Ключевые слова из пространства имен языка XAML со стандартным префиксом x

Ключевое слово	Допустим в качестве	Версия	Назначение
x:AsyncRecords	Атрибут корневого элемента	2006+	Управляет размером блока при асинхронной загрузке XAML
X:Arguments	Атрибут или вложенный элемент	2009	Задает аргумент (или несколько аргументов, если употребляется в качестве элемента), передаваемый конструктору элемента. При использовании в сочетании с x:FactoryMethod задает аргумент(ы) фабричного метода
X:Boolean	Элемент	2009	Представляет класс System.Boolean
X:Byte	Элемент	2009	Представляет класс System.Byte
x:Char	Элемент	2009	Представляет класс System.Char
x:Class	Атрибут корневого элемента	2006+	Определяет для корневого элемента класс, производный от типа элемента. Может сопровождаться необязательным префиксом пространства имен .NET
x:ClassAttributes	Атрибут корневого элемента, должен использоваться совместно с x:Class	2009	Не используется в WPF; содержит атрибуты, относящиеся к Windows Workflow Foundation
x:ClassModifier	Атрибут корневого элемента, должен использоваться совместно с x:Class	2006+	Определяет видимость класса, указанного в x:Class (по умолчанию открытого). Значение атрибута должно быть задано в терминах используемого процедурного языка (например public или internal для C#)
x:Code	Элемент в любом месте XAML, должен использоваться совместно с x:Class	2006+	Окружает процедурный код, включаемый в класс, указанный в x:Class.
x:ConnectionId	Атрибут	2006+	Не для открытого применения
x:Decimal	Элемент	2009	Представляет System.Decimal
x:Double	Элемент	2009	Представляет System.Double

Таблица 2.2 (продолжение)

Ключевое слово	Допустим в качестве	Версия	Назначение
x:FactoryMethod	Атрибут любого элемента	2009	Определяет статический метод, вызываемый для получения экземпляра элемента вместо конструктора
x:FieldModifier	Атрибут любого элемента, должен использоваться совместно с x:Name (или эквивалентом)	2006+	Определяет видимость поля, генерируемого для элемента (по умолчанию internal). Как и в случае x:ClassModifier, значение этого атрибута должно быть задано в терминах процедурного языка (например, public, private, ... для C#)
x:Int16	Элемент	2009	Представляет System.Int16
x:Int32	Элемент	2009	Представляет System.Int32
x:Int64	Элемент	2009	Представляет System.Int64
x:Key	Атрибут элемента, родитель которого реализует интерфейс IDictionary	2006+	Задает ключ элемента при добавлении в словарь родителя
x:Members	Не используется в WPF XAML	2009	Определяет дополнительные члены корневого класса, заданного в x:Class
x:Name	Атрибут любого не корневого элемента, должен использоваться совместно с x:Class	2006+	Задает имя поля, генерируемого для элемента, по которому на него можно ссылаться из процедурного кода
x:Object	Элемент	2009	Представляет System.Object
x:Property	Не используется в WPF XAML	2009	Определяет свойство внутри элемента x:Members
x:Shared	Атрибут любого элемента в ResourceDictionary, принимается во внимание только при компиляции XAML	2006+	Может принимать значение false для запрета использования одного экземпляра ресурса в нескольких местах (см. главу 12)
x:Single	Элемент	2009	Представляет System.Single
x:String	Элемент	2009	Представляет System.String
x:Subclass	Атрибут корневого элемента, должен использоваться совместно с x:Class	2006+	Определяет подкласс класса, заданного в x:Class, в котором хранится содержимое, определенное в XAML. В качестве необязательного префикса можно указать пространство имен

Ключевое слово	Допустим в качестве	Версия	Назначение
			.NET (используется с языками, не поддерживающими частичные классы)
x:SynchronousMode	Атрибут корневого элемента	2006+	Определяет, может ли содержимое XAML загружаться асинхронно
x:TimeSpan	Элемент	2009	Представляет System.TimeSpan
x:TypeArguments	В XAML2009 атрибут любого элемента, а в XAML2006 атрибут корневого элемента, используемый только совместно с x:Class	2006+	Делает класс универсальным (как List<T>), конкретизируемым указанными аргументами (например, List<Int32> или List<String>). Может содержать список аргументов конкретизации через запятую. Типам, отсутствующим в пространстве имен по умолчанию, должен предшествовать префикс пространств имен XML
x:Uid	Атрибут любого элемента	2006+	Помечает элемент идентификатором для локализации (см. главу 12)
x:Uri	Элемент	2009	Представляет System.Uri
x:XData	Элемент, используемый в качестве значения любого свойства типа IXmlSerializable	2006+	Произвольный остров данных XML, который остается непрозрачным для анализатора XAML (см. главу 13)

В табл. 2.3 перечислены дополнительные элементы пространства имен XAML, которые можно принять за ключевые слова, хотя на самом деле это расширения разметки (реальные классы .NET в пространстве имен System.Windows.Markup). Суффикс Extension в именах классов опущен, поскольку они обычно используются без суффикса.

Таблица 2.3. Расширения разметки в пространстве имен языка XAML в предположении, что префикс пространства имен x определен стандартным образом

Расширение	Назначение
x:Array	Представляет массив .NET. Потомками элемента x:Array являются элементы массива. В элементе должен присутствовать атрибут x:Type, определяющий тип массива
x:Null	Представляет ссылку null

Таблица 2.3 (продолжение)

Расширение	Назначение
x:Reference	Ссылка на именованный элемент. Должен присутствовать единственный позиционный параметр, задающий имя этого элемента
x:Static	Ссылка на любое статическое свойство, поле, константу или элемент перечисления, определенные в процедурном коде. При компиляции XAML это может быть даже неоткрытый член, определенный в той же сборке. Строка Member должна быть квалифицирована префиксом пространства имен XML, если тип не находится в пространстве имен по умолчанию
x:Type	Представляет экземпляр типа System.Type так же, как оператор typeof в C#. Строка TypeName должна быть квалифицирована префиксом пространства имен XML, если тип не находится в пространстве имен по умолчанию

Резюме

Мы рассмотрели, как XAML сочетается с WPF, и – что самое главное – теперь вы располагаете всей необходимой информацией для перевода почти всех примеров XAML на язык типа C# и наоборот. Однако, поскольку конвертеры типов и расширения разметки – «черные ящики», прямой перевод не всегда очевиден. Но в любом случае можно вызвать конвертер типа напрямую из процедурного кода, если непонятно, как именно выполняет преобразование компилятор! (Многие классы, для которых имеются конвертеры типов, даже содержат открытый статический метод Parse, который делает то же самое, – исключительно ради упрощения процедурного кода.)

Мне очень нравится, что даже простые элементы, которые можно было бы обработать в XAML специальным образом (например, null или именованные ссылки), выражаются с помощью того же механизма расширений разметки, который доступен сторонним разработчикам. Это позволяет сохранить простоту XAML и гарантирует, что механизм расширения работает действительно хорошо.

По мере дальнейшего изучения WPF вы, возможно, обратите внимание, что некоторые API WPF в процедурном коде выглядят громоздко, поскольку оптимизированы для использования совместно с XAML. Например, в WPF есть много мелких строительных блоков (что позволяет создавать развитые композиции, описанные в предыдущей главе), поэтому в WPF-приложении приходится вручную создавать гораздо больше объектов, чем, скажем, в Windows Forms. Мощь XAML особенно наглядно проявляется, когда нужно кратко описать глубокую иерархию объектов, поэтому команда разработчиков WPF потратила больше времени на реализацию средств, которые позволяют скрыть промежуточные объекты в XAML (например, конвертеры типов), а не средств для их сокрытия от процедурного кода (например, конструкторов, которые создают внутренние объекты от вашего имени).

Большинство людей хорошо понимают преимущества, которые дает наличие в WPF декларативной модели, предлагаемой XAML, но некоторые считают, что выбор XML в качестве формата представления неудачен. В следующих разделах я рассмотрю два типичных возражения и постараюсь на них ответить.

Возражение 1: XML слишком многословен, долго набирать

Это правда. Никому не нравится вводить длинный XML-код, но ведь есть же инструменты. Такие средства, как IntelliSense и визуальные конструкторы, могут избавить вас от необходимости дотошно вводить бесконечные угловые скобки! А прозрачная и детально разработанная спецификация XML позволяет легко интегрировать в процесс разработки новые инструментальные средства (например, создать программу экспорта XAML в формате вашего любимого инструмента), а также вносить в разметку изменения вручную или искать ошибки.

Более того, для некоторых областей применения WPF – построения сложных путей и фигур, 3D-моделей и т. д. – вводить XAML вручную практически нереально. На самом деле по мере развития XAML со времени появления его бета-версии некоторые ориентированные на человека приемы сокращенного ввода даже были исключены, чтобы сделать формат более устойчивым и расширяемым, то есть более удобным для поддержки со стороны инструментальных средств. Но я все же думаю, что знакомство с XAML и умение видеть API WPF сквозь призму как процедурного, так и декларативного кода остается лучшим способом изучить технологию. Это как понимание принципов работы HTML без использования визуальных средств разработки.

Возражение 2: системы, основанные на XML, низкопроизводительны

Язык XML создавался ради интероперабельности, а не ради максимально эффективного представления данных. Так зачем нагружать WPF-приложения кучей данных относительно большого объема, которые к тому же медленно разбираются?

Но ведь в типичном сценарии применения WPF XML компилируется в BAML, поэтому на этапе выполнения вы не платите полную цену за размер и низкую производительность разбора. BAML и меньше по размеру, чем исходный XAML, и оптимизирован для эффективного исполнения. Таким образом, все отрицательные стороны XML в плане производительности проявляются на этапе разработки, то есть там, где выгоды от использования XML нужнее всего.

- Обзор иерархии классов
- Логические и визуальные деревья
- Свойства зависимости

3

Основные принципы WPF

Перед тем как завершить часть I и перейти к *действительно* интересным вещам, будет полезно поговорить о некоторых важных концепциях WPF, с которыми программисты .NET ранее не были знакомы. Именно из-за обсуждаемых в этой главе вопросов у WPF установилась репутация технологии, очень сложной для изучения. Поэтому чем раньше мы разберемся с ними, тем увереннее вы будете чувствовать себя при чтении этой книги и другой документации по WPF.

Некоторые темы в этой главе не имеют никаких аналогов в прежнем опыте (например, логические и визуальные деревья), другие являются обобщениями хорошо известных понятий (скажем, свойства). По мере изложения мы будем демонстрировать применение изучаемых концепций на примере очень простого элемента пользовательского интерфейса – *диалогового окна* `About` (`О программе`).

Обзор иерархии классов

Классы, входящие в состав WPF, образуют очень глубокую иерархию наследования, поэтому сразу трудно уложить в голове назначение и взаимосвязи различных классов. Но есть несколько фундаментальных для работы WPF классов, о которых стоит упомянуть, прежде чем двигаться дальше. На рис. 3.1 показаны 12 наиболее важных классов и соотношения между ними.

- `Object` – базовый класс, которому наследуют все остальные классы .NET, и единственный из представленных на рисунке, не имеющий прямого отношения к WPF.
- `DispatcherObject` – базовый класс, предназначенный для объектов, к которым можно обращаться только из того потока, где они были созданы. Большинство классов WPF наследуют `DispatcherObject` и, следовательно, принципиально небезопасны относительно потоков. Слово `Dispatcher` в имени

класса относится к реализованному в WPF варианту цикла обработки сообщений Win32, который мы еще будем обсуждать в главе 7.

- `DependencyObject` – базовый класс, предназначенный для объектов, поддерживающих свойства зависимости; это одна из центральных тем данной главы.
- `Freezable` – базовый класс для объектов, которые можно «заморозить» в состоянии, разрешающем только чтение, – ради повышения производительности. К замороженным объектам можно безопасно обращаться из разных потоков, в отличие от объектов всех прочих классов, производных от `DispatcherObject`. Замороженный объект нельзя разморозить, однако можно клонировать, в результате чего получается незамороженная копия. По большей части объекты `Freezable` – это графические примитивы: кисти, перья, геометрические фигуры и классы анимации.
- `Visual` – базовый класс для объектов, имеющих двумерное визуальное представление. Визуальные объекты подробно рассматриваются в главе 15 «Двумерная графика».
- `UIElement` – базовый класс для двумерных визуальных объектов с поддержкой маршрутизации событий, привязки команд, компоновки и захвата фокуса. Эти механизмы обсуждаются в главе 5 «Компоновка с помощью панелей» и в главе 6 «События ввода: клавиатура, мышь, стилус и мультисенсорные устройства».
- `Visual3D` – базовый класс для объектов, имеющих трехмерное визуальное представление. Рассматривается в главе 16 «Трехмерная графика».
- `UIElement3D` – базовый класс для трехмерных визуальных объектов с поддержкой маршрутизации событий, привязки команд и захвата фокуса. Также рассматривается в главе 16.

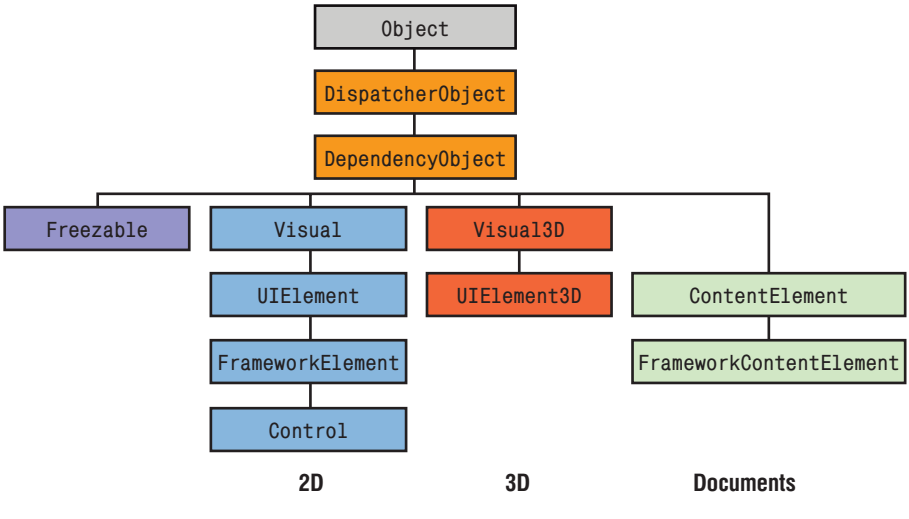


Рис. 3.1. *Важнейшие классы, составляющие основу WPF*

- `ContentElement` – базовый класс, аналогичный `UIElement`, но предназначенный для тех частей содержимого, которые относятся к документам и потому не имеют собственного механизма визуализации. Чтобы объект типа `ContentElement` появился на экране, им должен владеть объект класса, производного от `Visual`. Часто для правильной визуализации объект `ContentElement` нуждается в нескольких объектах `Visual` (охватывающих несколько строк, столбцов и страниц).
- `FrameworkElement` – базовый класс, добавляющий поддержку стилей, привязки к данным, ресурсов и нескольких механизмов, общих для всех элементов управления в Windows, в частности всплывающих подсказок и контекстных меню.
- `FrameworkContentElement` – аналог `FrameworkElement` для содержимого. Этот класс рассматривается в главе 11 «Изображения, текст и другие элементы управления».
- `Control` – базовый класс для таких хорошо знакомых элементов управления, как `Button`, `ListBox` и `StatusBar`. Класс `Control` добавляет к своему базовому классу `FrameworkElement` множество свойств, например `Foreground`, `Background` и `FontSize`, а также возможность тотального изменения стиля. Элементы управления WPF рассматриваются в части III.

В этой главе словом «*элемент*» без уточнений мы будем обозначать объект класса, производного от `UIElement` или `FrameworkElement`, а иногда от `ContentElement` или `FrameworkContentElement`. Разница между `UIElement` и `FrameworkElement` или `ContentElement` и `FrameworkContentElement` здесь несущественна, потому что в WPF нет никаких других открытых подклассов `UIElement` и `ContentElement`.

Логические и визуальные деревья

Естественность применения языка XAML для описания пользовательских интерфейсов объясняется его иерархической природой. В WPF пользовательский интерфейс представляет собой дерево объектов, которое называется *логическим деревом*.

В листинге 3.1 приведен первый вариант описания гипотетического диалогового окна About (О программе), в котором корнем логического дерева является объект `Window`. У `Window` имеется дочерний элемент `StackPanel` (см. главу 5), содержащий несколько простых элементов управления и еще один элемент `StackPanel`, который содержит две кнопки `Button`.

Листинг 3.1. Описание простого диалогового окна About на XAML

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4 Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4 Unleashed
    </Label>
```

```
    <Label>© 2010 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

На рис. 3.2 показано, как выглядит это диалоговое окно (можете убедиться в этом, скопировав текст листинга 3.1 в программу XAMLPAD2009, описанную в предыдущей главе), а на рис. 3.3 – логическое дерево окна.



Рис. 3.2. *Окно, соответствующее коду в листинге 3.1*

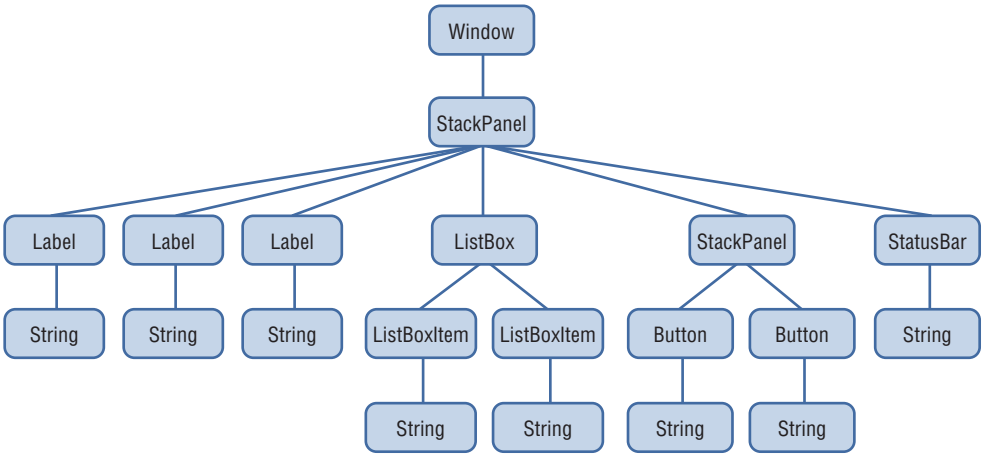


Рис. 3.3. *Логическое дерево, соответствующее коду в листинге 3.1*

Отметим, что логическое дерево существует даже для WPF-интерфейсов, созданных без участия XAML. Логику листинга 3.1 можно было бы реализовать чисто процедурно, а логическое дерево при этом не изменилось бы.

Идея логического дерева представляется очевидной, так зачем о нем вообще говорить? Затем, что поведение чуть ли не всех механизмов WPF (свойств, событий, ресурсов и т. д.) так или иначе связано с логическим деревом. Например, значения свойств иногда автоматически распространяются вниз по дереву на дочерние элементы, а генерируемые события могут распространяться как вниз, так и вверх. Такое поведение свойств обсуждается ниже в этой главе, а поведение событий – в главе 6.

Логическое дерево в WPF фактически дает упрощенную картину того, что в действительности происходит при визуализации элементов. Полное дерево, содержащее все визуализированные элементы, называется *визуальным деревом*. Можно представлять себе визуальное дерево как раскрытое логическое дерево, в котором каждый узел является вершиной поддерева, содержащего его визуальные компоненты. Иными словами, в визуальном дереве каждый элемент уже не «черный ящик», а раскрывает все детали своей визуальной реализации. Например, `ListBox` – логически единый элемент управления, однако по умолчанию его визуальное представление состоит из более простых WPF-элементов: рамки `Border`, двух полос прокрутки `ScrollBar` и др.

В визуальном дереве представлены не все узлы логического дерева, а лишь элементы, производные от классов `System.Windows.Media.Visual` или `System.Windows.Media.Visual3D`. Прочие элементы (в том числе простые строки, присутствующие в листинге 3.1) не включаются, потому что не обладают собственным поведением визуализации.

СОВЕТ

Некоторые несложные программы просмотра XAML, в частности XamlPadX, упомянутая в предыдущей главе, позволяют просматривать визуальное дерево (и значения свойств) объектов, визуализированных на основе XAML-кода.

На рис. 3.4 изображено визуальное дерево, получающееся при выполнении кода из листинга 3.1 в системе Windows 7 с темой Aero. Здесь представлены внутренние компоненты пользовательского интерфейса, которые в настоящий момент невидимы, например две полосы прокрутки `ScrollBar` элемента `ListBox` и рамки `Border` всех меток `Label`. Видно также, что элементы `Button`, `Label` и `ListBoxItem` составлены из одних и тех же элементов за одним исключением – в `Button` вместо `Border` используется скрытый элемент `ButtonChrome`. (У этих элементов управления имеются и другие визуальные различия, обусловленные тем, что по умолчанию подразумеваются разные значения свойств. Например, у кнопки `Button` поле `Margin` для всех четырех сторон по умолчанию равно `10`, а у метки `Label` поле равно `0`.)

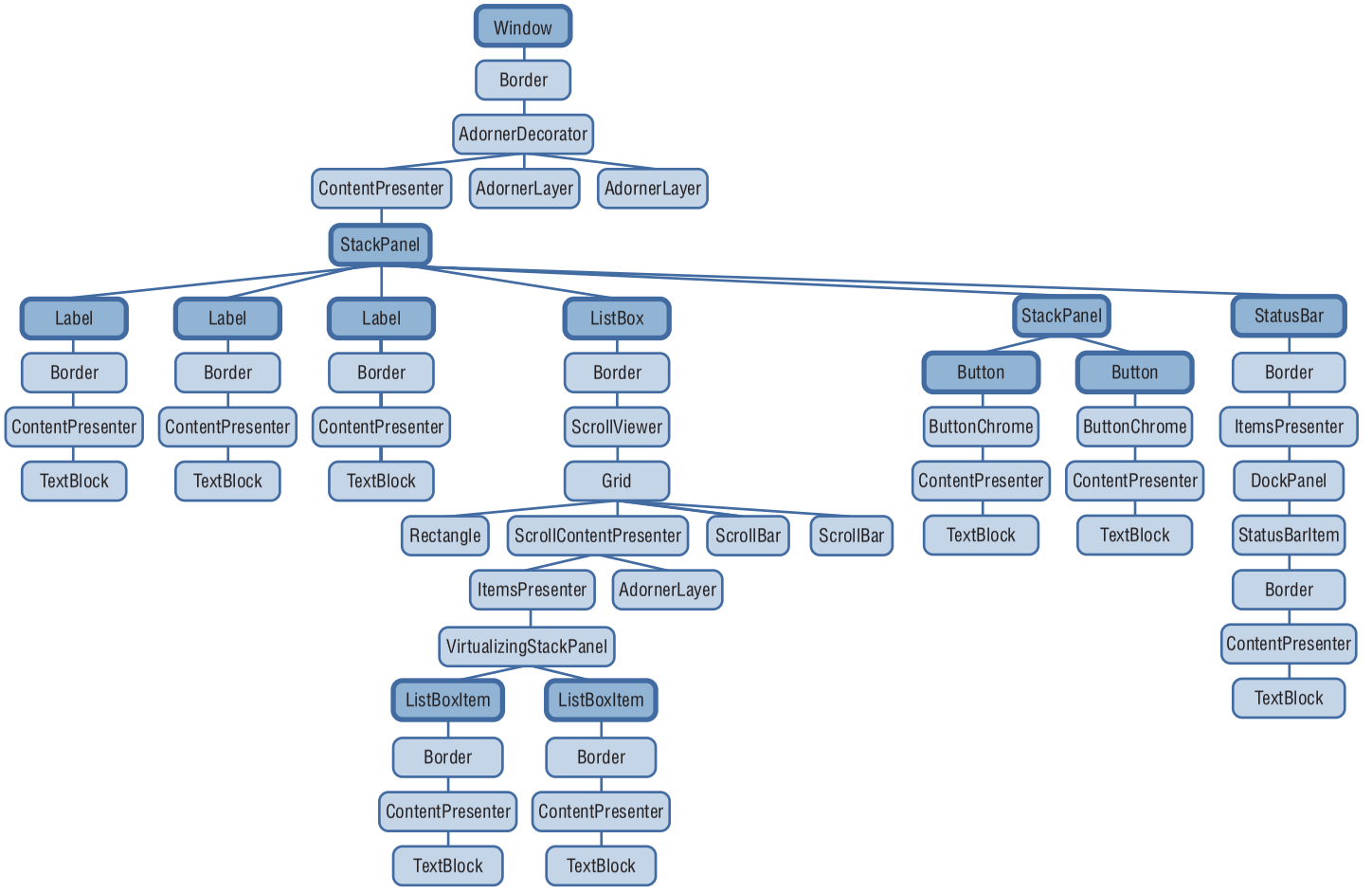


Рис. 3.4. *Визуальное дерево для листинга 3.1; выделены узлы логического дерева*

В визуальном дереве отражено внутреннее устройство WPF-элементов, поэтому оно может оказаться весьма сложным. К счастью, несмотря на то что визуальное дерево является существенной частью инфраструктуры WPF, задумываться о нем имеет смысл, только если вы собираетесь радикально изменять стили элементов управления (см. главу 14 «Стили, шаблоны, обложки и темы») или выполнять низкоуровневое рисование (см. главу 15). В частности, написание кода, зависящего от конкретной структуры визуального дерева элемента `Button`, противоречит одному из основополагающих принципов WPF – отделению внешнего вида от логики. Если кто-нибудь решит изменить стиль кнопки, применяя технику, описанную в главе 14, то ее стандартное визуальное дерево может быть заменено чем-то совершенно непохожим.

ПРЕДУПРЕЖДЕНИЕ

Если логическое дерево остается статичным, пока не вмешается программист (который может, например, динамически добавить или удалить элементы), то визуальное дерево может измениться просто потому, что пользователь выбрал другую тему Windows!

Однако логическое и визуальное деревья можно обойти с помощью взаимодополняющих классов `System.Windows.LogicalTreeHelper` и `System.Windows.Media.VisualTreeHelper`. В листинге 3.2 показан застраничный код для листинга 3.1, который при запуске в отладчике выводит простое представление логического и визуального деревьев диалогового окна `About` в порядке обхода в глубину. (Чтобы присоединить этот процедурный код, необходимо добавить в листинг 3.1 атрибут `x:Class="AboutDialog"` и соответствующую директиву `xmlns:x`.)

Листинг 3.2. Обход и распечатка логического и визуального деревьев

```
using System;
using System.Diagnostics;
using System.Windows;
using System.Windows.Media;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
    PrintLogicalTree(0, this);
  }

  protected override void OnContentRendered(EventArgs e)
  {
    base.OnContentRendered(e);
    PrintVisualTree(0, this);
  }
```

```
  void PrintLogicalTree(int depth, object obj)
  {
    // Напечатать объект с предшествующими пробелами,
    // число которых соответствует глубине вложенности
    Debug.WriteLine(new string(' ', depth) + obj);

    // Иногда листовые узлы не принадлежат классу
    // DependencyObject (например, строки)
    if (!(obj is DependencyObject)) return;

    // Рекурсивный вызов для каждого логического
    // дочернего узла
    foreach (object child in LogicalTreeHelper.GetChildren(
      obj as DependencyObject))
      PrintLogicalTree(depth + 1, child);
  }

  void PrintVisualTree(int depth, DependencyObject obj)
  {
    // Напечатать объект с предшествующими пробелами,
    // число которых соответствует глубине вложенности
    Debug.WriteLine(new string(' ', depth) + obj);

    // Рекурсивный вызов для каждого визуального
    // дочернего узла
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
      PrintVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
  }
}
```

Если вызвать любой из этих методов с параметром `depth`, равным 0, и текущим экземпляром `Window` в качестве параметра `obj`, то получится текстовое представление дерева с тем же узлами, что на рис. 3.2 и 3.3. Логическое дерево можно обойти внутри конструктора, однако визуальное дерево остается пустым до первой компоновки элемента `Window`. Именно поэтому метод `PrintVisualTree` вызывается в обработчике события `OnContentRendered`, который выполняется после завершения компоновки.

СОВЕТ

Визуальные деревья, аналогичные изображенному на рис. 3.4, часто называют просто *деревьями элементов*, поскольку они содержат элементы как логического, так и визуального дерева. В таком случае термином «*визуальное дерево*» обозначают любое поддерево, содержащее только визуальные (нелогические?) элементы. Например, многие сказали бы, что стандартное визуальное дерево `Window` содержит `Border`, `AdornerDecorator`, два элемента `AdornerLayer`, `ContentPresenter` и больше ничего. Самая верхняя панель `StackPanel` на рис. 3.4 обычно *не* рассматривается как визуальный дочерний элемент `ContentPresenter`, хотя `VisualTreeHelper` и считает ее таковым.

Обход и того и другого дерева иногда можно выполнить с помощью методов экземпляра самих элементов. Например, в классе `Visual` есть три защищенных члена (`VisualParent`, `VisualChildrenCount` и `GetVisualChild`) для доступа к родителю и потомкам визуального элемента. А в классе `FrameworkElement`, базовом для таких элементов управления, как `Button` и `Label`, и в дополняющем его классе `FrameworkContentElement` имеются открытое свойство `Parent`, которое представляет логического родителя, и защищенное свойство `LogicalChildren`, описывающее список логических дочерних элементов. В подклассы этих классов часто включают открытые члены, обеспечивающие тот или иной доступ к логическим дочерним элементам, например открытую коллекцию `Children`. Некоторые классы, к примеру `Button` и `Label`, раскрывают свойство `Content` и гарантируют наличие только одного логического дочернего элемента.

СОВЕТ

В отладчике Visual Studio 2010 щелчок по значку лупы рядом с экземпляром класса, производного от `Visual`, позволяет исследовать его визуальное дерево.

Свойства зависимости

В WPF появился новый тип свойств – *свойства зависимости*; они повсеместно используются для реализации таких механизмов, как назначение стилей, автоматическая привязка к данным, анимация и др. Поначалу вы, возможно, отнесетесь к этой идее скептически, потому что она вносит дополнительные сложности в картину типов .NET, где имеются простые поля, свойства, методы и события. Но, поняв, какие именно задачи решают свойства зависимости, вы, скорее всего, сочтете их желанным нововведением.

Свойство зависимости *зависит* от нескольких поставщиков, которые определяют его значение во время выполнения. Поставщиком может быть анимация, постоянно изменяющая значение свойства, родительский элемент, распространяющий значение своего свойства на потомков, и т. д. Пожалуй, наиболее существенной особенностью свойства зависимости является встроенная возможность генерировать уведомления об изменениях своего значения.

Причина наделения свойств подобной интеллектуальностью – стремление описать развитую функциональность на уровне декларативной разметки. Механизм декларативного описания интерфейсов в WPF опирается на использование свойств. К примеру, в классе `Button` имеется 111 открытых свойств (из них 98 наследуются от класса `Control` и его предков)! Свойства можно задавать в XAML-коде (непосредственно или с помощью инструментов конструирования) без написания процедурного кода. Но в отсутствие механизмов, предлагаемых свойствами зависимости, было бы весьма трудно получить желаемый результат без дополнительного кода.

В этом разделе мы сначала вкратце ознакомимся с реализацией свойства зависимости, чтобы сделать обсуждение более предметным, а потом внимательнее рассмотрим, как свойства зависимости расширяют функциональность обычных свойств .NET в следующих направлениях:

- Уведомление об изменениях
- Наследование значений свойств
- Поддержка нескольких поставщиков

Понимать большинство нюансов свойств зависимости необходимо только авторам нестандартных элементов управления. Но даже обычный пользователь WPF должен знать, что такое свойства зависимости и как они работают. Например, применять стили и анимацию можно только к свойствам зависимости. Немного поработав с WPF, вы еще будете жалеть, что не все свойства элементов являются свойствами зависимости!

Реализация свойства зависимости

На практике свойство зависимости – это обычное свойство .NET, которое включено в состав дополнительной инфраструктуры, предоставляемой WPF. Для этого в WPF предусмотрены специальные API; ни один .NET-совместимый язык программирования (кроме XAML) ничего не знает о свойствах зависимости.

В листинге 3.3 показано, как в классе `Button` реализовано свойство зависимости `IsDefault`.

Листинг 3.3. Реализация стандартного свойства зависимости

```
public class Button : ButtonBase
{
  // Свойство зависимости
  public static readonly DependencyProperty IsDefaultProperty;

  static Button()
  {
    // Зарегистрировать свойство
  Button.IsDefaultProperty = DependencyProperty.Register("IsDefault",
    typeof(bool), typeof(Button),
    new FrameworkPropertyMetadata(false,
      new PropertyChangedCallback(OnIsDefaultChanged)));
    ...
  }

  // Обертка в виде обычного свойства .NET (необязательно)
  public bool IsDefault
  {
    get { return (bool)GetValue(Button.IsDefaultProperty); }
    set { SetValue(Button.IsDefaultProperty, value); }
  }
```

```
    // Метод, вызываемый при изменении свойства (необязательно)
    private static void OnIsDefaultChanged(
      DependencyObject o, DependencyPropertyChangedEventArgs e) { ... }
    ...
}
```

Статическое поле `IsDefaultProperty` типа `System.Windows.DependencyProperty` и является свойством зависимости. По принятому соглашению все поля типа `DependencyProperty` открытые, статические и имеют имя, оканчивающееся словом `Property`. Некоторые компоненты инфраструктуры требуют обязательного соблюдения этого соглашения, например средства локализации, загрузчики XAML и пр.

Обычно для создания свойства зависимости вызывается метод `DependencyProperty.Register` – ему передается имя свойства (`IsDefault`), его тип (`bool`) и тип класса, который будет владельцем свойства (`Button`). Дополнительно (с помощью перегруженных вариантов метода `Register`) можно передать метаданные, уточняющие, как WPF должна интерпретировать это свойство, а также обратные вызовы для обработки изменения значения свойства, приведения типа значения и проверки значения. В классе `Button` метод `Register` вызывается в статическом конструкторе; при этом свойству присваивается значение по умолчанию `false` и задается делегат, который будет вызываться в ответ на уведомление об изменении.

Далее приведено обычное свойство `IsDefault`. Его аксессоры вызывают методы `GetValue` и `SetValue`, унаследованные от `System.Windows.DependencyObject`, низкоуровневого базового класса, которому должны наследовать все свойства зависимости. Метод `GetValue` возвращает последнее значение, переданное `SetValue`, или, если метод `SetValue` еще ни разу не вызывался, значение по умолчанию, заданное при регистрации свойства. Обычное свойство .NET `IsDefault` (в этом контексте его иногда называют *обертывающим свойством*) определять необязательно; клиенты класса `Button` могут напрямую обращаться к методам `GetValue` и `SetValue`, поскольку они открыты. Однако наличие свойства .NET позволяет клиентам более естественно программировать чтение и изменение значения, а кроме того, только таким образом можно установить свойство в XAML. Со стороны WPF было бы правильно предоставить универсальные перегруженные варианты `GetValue` и `SetValue`. Но это не сделано – в первую очередь потому, что свойства зависимости появились до того, как универсальные типы .NET получили широкое распространение.

СОВЕТ

В дистрибутиве Visual Studio имеется сниппет `propdp`, который автоматически генерирует определение свойства зависимости. Это намного быстрее, чем вводить определение вручную!

ПРЕДУПРЕЖДЕНИЕ

Во время выполнения обертывающие свойства .NET не вызываются при задании значений свойств зависимости в XAML!

Хотя компилятор XAML требует, чтобы обертывающее свойство присутствовало, на этапе выполнения WPF напрямую обращается к методам `GetValue` и `SetValue`! Поэтому во избежание несогласованности между результатами установки свойства в XAML и в процедурном коде не следует помещать в обертывающее свойство какой-нибудь код помимо вызова `GetValue`/`SetValue`. Для реализации дополнительной логики предназначены методы обратного вызова, задаваемые при регистрации. Все стандартные обертывающие свойства в WPF следуют этому правилу, так что предупреждение адресовано авторам новых классов, содержащих свойства зависимости.

При поверхностном взгляде листинг 3.3 кажется излишне многословным способом представить простое булевское свойство. Однако поскольку в реализации `GetValue` и `SetValue` используется весьма эффективная система разреженного хранения, а `IsDefaultProperty` – статическое поле (а не поле экземпляра), то на практике свойства зависимости даже позволяют сэкономить память, выделяемую под экземпляр, по сравнению с обычными свойствами .NET. Если бы все свойства элементов управления WPF были обертками полей экземпляра (как большинство свойств .NET), то потребление памяти существенно возросло бы из-за объема связанных с каждым экземпляром локальных данных. Только представьте себе – 111 полей для каждой кнопки, 104 поля для каждой метки и т. д.! Но в действительности 89 из 111 открытых свойств класса `Button` и 82 из 104 открытых свойств класса `Label` – это свойства зависимости.

И экономией памяти достоинства свойств зависимости не исчерпываются. Реализация устроена так, что код для доступа к свойству из разных потоков для извещения элемента-владельца о необходимости повторной визуализации и многого другого централизован и стандартизован, так что авторам свойств писать его не придется. Например, если после изменения значения свойства необходимо перерисовать элемент (как в случае свойства `Background` класса `Button`), то достаточно указать флаг `FrameworkPropertyMetadataOptions.AffectsRender` при вызове перегруженного варианта метода `DependencyProperty.Register`. Кроме того, реализация поддерживает три вышеупомянутых механизма, которые мы теперь рассмотрим более подробно.

Уведомление об изменении

При изменении значения свойства зависимости WPF может автоматически инициировать некоторые действия в соответствии с метаданными свойства. Это может быть перерисовка элементов, пересчет компоновки, обновление привязки к данным и многое другое. Одна из самых интересных черт встроенного механизма уведомления об изменении – *триггеры свойств*, которые

позволяют ассоциировать с изменением запрограммированные вами действия без написания процедурного кода.

Пусть, например, требуется, чтобы при наведении указателя мыши на кнопку в диалоговом окне About в листинге 3.1 надпись на кнопке становилась синей. Без триггеров свойств для этого нужно было бы присоединить к каждой кнопке обработчики событий MouseEnter и MouseLeave:

```
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
        MinWidth="75" Margin="10">Help</Button>
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
        MinWidth="75" Margin="10">OK</Button>
```

Сами обработчики можно реализовать в застраничном коде на C# следующим образом:

```
// Сделать цвет фона синим, когда указатель находится над кнопкой
void Button_MouseEnter(object sender, MouseEventArgs e)
{
  Button b = sender as Button;
  if (b != null) b.Foreground = Brushes.Blue;
}

// Восстановить черный цвет фона, когда указатель покидает кнопку
void Button_MouseLeave(object sender, MouseEventArgs e)
{
  Button b = sender as Button;
  if (b != null) b.Foreground = Brushes.Black;
}
```

А триггер свойства позволяет реализовать такое же поведение целиком на XAML. Достаточно добавить такое коротенькое описание объекта Trigger:

```
<Trigger Property="IsMouseOver" Value="True">
  <Setter Property="Foreground" Value="Blue"/>
</Trigger>
```

Этот триггер срабатывает при изменении свойства IsMouseOver объекта Button, которое принимает значение true одновременно с генерацией события MouseEnter и значение false – при генерации события MouseLeave. Обратите внимание, что восстанавливать черный цвет фона, когда IsMouseOver становится равным false, не нужно. WPF сделает это автоматически!

Единственная проблема заключается в том, как ассоциировать этот триггер с каждой кнопкой. К сожалению, из-за досадного ограничения невозможно применять триггеры непосредственно к элементам, в частности к Button. Они могут располагаться только внутри объекта Style, поэтому подробное рассмотрение триггеров свойств мы отложим до главы 14. А пока, чтобы поэкспериментировать с применением триггера к кнопке, можете добавить несколько промежуточных XML-элементов:

```
<Button MinWidth="75" Margin="10">
<Button.Style>
  <Style TargetType="{x:Type Button}">
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Foreground" Value="Blue"/>
    </Trigger>
  </Style.Triggers>
  </Style>
</Button.Style>
  OK
</Button>
```

Триггеры свойств – лишь один из трех видов триггеров, поддерживаемых WPF. *Триггер данных* – разновидность триггера свойства, работающая для произвольных свойств .NET (а не только свойств зависимости); такие триггеры также рассматриваются в главе 14. *Триггер события* позволяет декларативно описывать, какие действия следует предпринять при генерации маршрутизируемого события (см. главу 6). Триггеры событий всегда подразумевают наличие анимации или звукового сопровождения, поэтому мы отложим их рассмотрение до главы 17 «Анимация».

ПРЕДУПРЕЖДЕНИЕ

Не обманывайтесь насчет свойства элемента Triggers!

Свойство `Triggers` класса `FrameworkElement` содержит допускающую чтение и запись коллекцию объектов типа `TriggerBase` (общий базовый класс всех трех типов триггеров) – на первый взгляд это очень простой способ присоединить триггеры свойств к таким элементам, как `Button`. Но, увы, эта коллекция может содержать только триггеры событий, так что ее название и тип обманчивы. Попытка добавить в коллекцию триггер свойства (или данных) приведет к исключению во время выполнения.

Наследование значений свойств

Словосочетание «*наследование значений свойств*» (или просто «*наследование свойств*») относится не к традиционному для объектно-ориентированного программирования наследованию классов, а к распространению значений свойств вдоль дерева элементов. В листинге 3.4 приведен простой пример, расширяющий код из листинга 3.1, – мы явно задали в элементе `Window` свойства зависимости `FontSize` и `FontStyle`. На рис. 3.5 показан результат такого изменения. (Отметим, что благодаря удобному атрибуту `SizeToContent` размер элемента `Window` автоматически подстраивается под размер содержимого!)

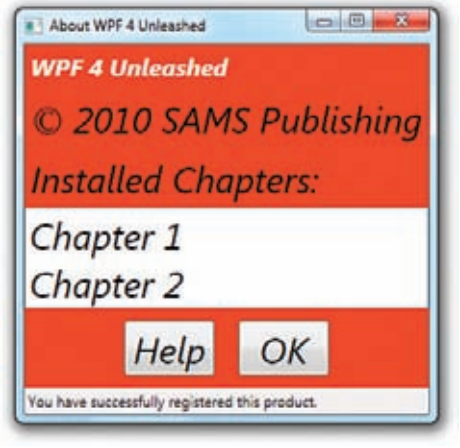


Рис. 3.5. *Диалоговое окно About, в котором для корневого элемента Window установлены свойства FontSize и FontStyle*

Листинг 3.4. Диалоговое окно About, в котором для корневого элемента Window установлены свойства шрифта

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4 Unleashed" SizeToContent="WidthAndHeight"
  FontSize="30" FontStyle="Italic"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4 Unleashed
    </Label>
    <Label>© 2010 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

В большинстве случаев оба эти свойства распространяются вниз по дереву и наследуются всеми потомками. Это относится даже к элементам `Button` и `ListBoxItem`, которые расположены на три уровня ниже в логическом дереве. Свойство `FontSize` первой метки `Label` не изменяется, потому что для нее явно указано значение `FontSize` `20`, отменяющее унаследованное значение `30`. Напротив,

значение FontStyle наследуется всеми элементами Label, ListBoxItem и Button, поскольку явно оно нигде не переопределено.

Отметим, что на текст в строке состояния StatusBar эти свойства не оказывают влияния, хотя класс StatusBar и поддерживает их, как, впрочем, любой элемент управления. В подобных случаях поведение механизма наследования свойств видоизменяется по двум причинам.

- Не всякое свойство зависимости принимает участие в наследовании свойств. (На самом деле желание подключиться к этому механизму выражается явно путем передачи флага FrameworkPropertyMetadataOptions.Inherits при вызове метода DependencyProperty.Register.)
- Могут существовать другие источники значения свойства с более высоким приоритетом (см. следующий раздел).

В данном случае наблюдаемое поведение обусловлено второй причиной. Некоторые элементы управления, в частности StatusBar, Menu и ToolTip, устанавливают для себя свойства шрифта в соответствии с текущими системными настройками. Это дает пользователю возможность настраивать шрифты привычным способом – с помощью панели управления. Но результат может обескуражить разработчика WPF-приложения, потому что такие элементы препятствуют распространению наследования на расположенные под ними части дерева элементов. Например, если в листинге 3.4 добавить Button в качестве логического дочернего элемента StatusBar, то свойства FontSize и FontStyle сохранят подразумеваемые по умолчанию значения 12 и Normal соответственно и, следовательно, эта кнопка будет отличаться от других кнопок, расположенных вне StatusBar.

КОПНЕМ ГЛУБЖЕ

Наследование значений свойств в других местах

Механизм наследования свойств первоначально был разработан для дерева элементов, а затем перенесен на некоторые другие контексты. Например, распространяющиеся вниз значения могут применяться к элементам, которые *выглядят* как потомки с точки зрения XML (благодаря синтаксису элементов свойств в XAML), но *не являются* потомками в понимании логических или визуальных деревьев. Такими псевдопотомками могут быть триггеры, присоединенные к элементам, или значения *любого* свойства (а не только Content или Children) при условии, что объект является производным от класса Freezable. Такое решение может показаться произвольным и к тому же плохо документировано, но смысл его в том, чтобы в некоторых сценариях XAML «просто работал» естественным образом, не требуя от вас никакого внимания.

Поддержка нескольких поставщиков

WPF содержит немало механизмов, каждый из которых пытается установить значения свойств зависимости. Если бы не было четко определенного способа упорядочить независимых поставщиков значений свойств, то система пре-

вратилась бы в хаос, не позволяющий уверенно предсказать значение свойства. Но, разумеется, такой способ существует.

На рис. 3.6 показаны пять шагов, которые WPF применяет при определении окончательного значения каждого свойства зависимости. Все это происходит автоматически благодаря встроенному механизму уведомления об изменении значений свойств.



Рис. 3.6. *Последовательность вычисления значения свойства зависимости*

Шаг 1: определение базового значения

Свой вклад в вычисление базового значения вносят почти все поставщики значений свойств. Ниже приведен перечень десяти поставщиков, которые могут устанавливать значения большинства свойств зависимости, – в порядке убывания приоритета:

1. Локальное значение
2. Триггер в шаблоне родителя
3. Шаблон родителя
4. Триггеры в стиле
5. Триггеры в шаблоне
6. Установщики стиля
7. Триггеры стиля темы
8. Установщики стиля темы
9. Наследование значения свойства
10. Значение по умолчанию

С некоторыми поставщиками значений свойств вы уже встречались, например с механизмом наследования свойств (9). *Локальное значение* (1) технически означает любое обращение к методу `DependencyObject.SetValue`, но обычно имеет вид простого присваивания свойства в XAML или процедурном коде (в силу способа определения свойств зависимости, проиллюстрированного выше на примере свойства `Button.IsDefault`). Под *значением по умолчанию* (10) понимается начальное значение, указанное при регистрации свойства зависимости; понятно, что его приоритет наименьший. Остальные поставщики связаны со стилями и шаблонами, поэтому их рассмотрение мы отложим до главы 14.

Описанная выше расстановка приоритетов объясняет, почему механизм наследования значений свойств не оказал влияния на свойства `FontSize` и `FontStyle` элемента `StatusBar` в листинге 3.4. Установка свойств шрифта `StatusBar` в соответствии с настройками системы производится на уровне установщиков стилей темы (8). Хотя приоритет этого поставщика выше, чем у механизма наследования свойств (9), переопределить параметры шрифта все равно можно –

достаточно воспользоваться поставщиком с еще более высоким приоритетом, например просто задать свойства локально в самом элементе StatusBar.

СОВЕТ

Если вы не можете понять, в какой именно момент данное свойство зависимости получает значение, то попробуйте воспользоваться статическим методом DependencyPropertyHelper.GetValueSource. Он возвращает структуру ValueSource, содержащую несколько полей: перечисление BaseValueSource, которое показывает источник базового значения (шаг 1) и булевские свойства IsExpression, IsAnimated и IsCoerced, содержащие информацию о шагах 2–4.

Если вызвать этот метод для элемента StatusBar в листинге 3.1 или 3.4, запросив сведения о свойстве FontSize или FontStyle, то в качестве BaseValueSource будет возвращено DefaultStyle, показывающее, что значение присвоено установщиком стиля темы. (Стили, заданные в темах, иногда называют *стилями по умолчанию*. Триггеру стиля темы соответствует элемент перечисления DefaultStyleTrigger.)

Не используйте этот метод в промышленном коде! В будущих версиях WPF допущения, сделанные вами относительно вычисления значения, могут оказаться ложными. Кроме того, различная обработка значения свойства в зависимости от его источника нарушает принципы проектирования WPF-приложений.

КОПНЕМ ГЛУБЖЕ

Очистка локального значения

Выше, в разделе «Уведомление об изменении», был продемонстрирован процедурный код, который изменяет цвет фона кнопки на синий в ответ на событие MouseEnter и восстанавливает черный цвет фона в ответ на событие MouseLeave. Проблема в том, что в обработчике MouseLeave черный цвет устанавливается как локальное значение, тогда как в начальном состоянии Button цвет фона поступает от установщика стиля из темы. Если впоследствии будет выбрана другая тема и установщик попробует изменить значение свойства Foreground (или же самое попробует сделать поставщик с более высоким приоритетом), то попытка закончится неудачно, так как черный цвет установлен локально.

На самом деле надо было бы *очистить* локальное значение и дать WPF возможность установить его заново, получив значение от применимого поставщика с самым высоким приоритетом. К счастью, в классе DependencyObject имеется как раз такой механизм: метод ClearValue. В C# его можно вызвать от имени объекта Button b:

```
b.ClearValue(Button.ForegroundProperty);
```

(Button.ForegroundProperty — статическое поле класса DependencyProperty.) После вызова ClearValue WPF пересчитывает базовое значение, просто не принимая во внимание локальное.

Отметим, что триггер для свойства IsMouseOver, показанный в разделе «Уведомление об изменении», не подвержен этой проблеме. Триггер либо активен, либо нет, а неактивные триггеры при вычислении значения свойства игнорируются.

Шаг 2: вычисление

Если значение, полученное на шаге 1, представляет собой *выражение* (объект класса, производного от `System.Windows.Expression`), то WPF выполняет специальный шаг вычисления для преобразования выражения в конкретное значение. Выражения чаще всего появляются в результате привязки к данным (это тема главы 13).

Шаг 3: применение анимаций

Если работает одна или несколько анимаций, то любая из них способна изменить текущее значение свойства (получив на входе значение, вычисленное на шаге 2) или вообще подменить его. Таким образом, анимации (тема главы 17) могут отменить решения всех прочих поставщиков значений – даже локальных! Начинающие изучать WPF часто попадают в эту ловушку.

Шаг 4: приведение

После того как все поставщики значения свойства сказали свое слово, WPF передает почти окончательное значение делегату `CoerceValueCallback`, если таковой был указан при регистрации свойства зависимости. Этот делегат должен вернуть новое значение, применяя соответствующую случаю логику. Например, встроенный в WPF элемент управления `ProgressBar` с помощью подобного делегата приводит значение свойства зависимости `Value` к диапазону от `Minimum` до `Maximum`, то есть возвращает `Minimum`, если входное значение меньше `Minimum`, и `Maximum` – если оно больше `Maximum`. Если логика приведения изменяется во время работы программы, то можно вызвать метод `CoerceValue` и заставить WPF заново выполнить шаги приведения и проверки.

Шаг 5: проверка

Наконец приведенное значение передается делегату `ValidateValueCallback`, если таковой был указан при регистрации свойства зависимости. Он должен вернуть `true`, если входное значение допустимо, и `false` в противном случае. Если возвращается `false`, то WPF возбуждает исключение, отменяя все проделанные вычисления.

СОВЕТ

В версии WPF 4 в класс `DependencyObject` добавлен новый метод `SetCurrentValue`. Он напрямую обновляет текущее значение, не изменяя его источник. (Приведение типа и проверка по-прежнему производятся.) Этот метод предназначен для элементов управления, которые устанавливают значения в ответ на действия пользователя. Например, элемент `RadioButton` изменяет значение свойства `IsChecked` других элементов `RadioButton` в той же группе, когда пользователь выбирает новый вариант переключателя. В прежних версиях WPF в таких случаях устанавливалось локальное значение, а значит, игнорировались все прочие источники значений. Это могло привести, например, к некорректной работе механизма привязки к данным. В WPF 4 класс `RadioButton` модифицирован и теперь пользуется методом `SetCurrentValue`.

Присоединенные свойства

Присоединенное свойство – это частный случай свойства зависимости, которое можно *присоединять* к произвольным объектам. Поначалу это может вас озадачить, однако у этого механизма есть несколько применений в WPF.

Предположим, что в примере диалогового окна About свойства `FontSize` и `FontStyle` заданы не для всего окна `Window` (как в листинге 3.4), а только для внутренней панели `StackPanel`, чтобы они наследовались лишь двумя кнопками `Button`. Однако перенос атрибутов свойств во внутренний элемент `StackPanel` работать не будет, потому что в классе `StackPanel` нет никаких свойств, относящихся к шрифту! Поэтому необходимо использовать присоединенные свойства `FontSize` и `FontStyle`, определенные в классе `TextElement`. В листинге 3.5 продемонстрирован синтаксис XAML, предназначенный для задания присоединенных свойств. В результате мы получаем желаемое наследование значений свойств (рис. 3.7).



Рис. 3.7. *Диалоговое окно About, в котором свойства FontSize и FontStyle в обеих кнопках унаследованы от внутренней панели StackPanel*

Листинг 3.5. Диалоговое окно About, в котором свойства шрифта перенесены во внутреннюю панель StackPanel

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4 Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4 Unleashed
    </Label>
    <Label>© 2010 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
```

```
    <StackPanel TextElement.FontSize="30" TextElement.FontStyle="Italic"
      Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

В элементе `StackPanel` необходимо записать `TextElement.FontSize` и `TextElement.FontStyle` (а не просто `FontSize` и `FontStyle`), потому что в классе `StackPanel` таких свойств нет. Когда анализатор или компилятор XAML встречает такой синтаксис, он предполагает, что в классе `TextElement` (который иногда называется *поставщиком присоединенных свойств*) имеются статические методы `SetFontSize` и `SetFontStyle`, которые умеют устанавливать соответствующие свойства. Поэтому приведенное в листинге 3.5 объявление `StackPanel` эквивалентно следующему коду на C#:

```
StackPanel panel = new StackPanel();
TextElement.SetFontSize(panel, 30);
TextElement.SetFontStyle(panel, FontStyles.Italic);
panel.Orientation = Orientation.Horizontal;
panel.HorizontalAlignment = HorizontalAlignment.Center;
Button helpButton = new Button();
helpButton.MinWidth = 75;
helpButton.Margin = new Thickness(10);
helpButton.Content = "Help";
Button okButton = new Button();
okButton.MinWidth = 75;
okButton.Margin = new Thickness(10);
okButton.Content = "OK";
panel.Children.Add(helpButton);
panel.Children.Add(okButton);
```

Отметим, что элементы перечисления, например `FontStyles.Italic`, `Orientation.Horizontal` и `HorizontalAlignment.Center`, в XAML-коде записывались просто как `Italic`, `Horizontal` и `Center`. Это стало возможно благодаря конвертеру типа `EnumConverter` из каркаса .NET Framework, который может преобразовывать любые строки без учета регистра букв.

Хотя в XAML-коде в листинге 3.5 логическое присоединение свойств `FontSize` и `FontStyle` к `StackPanel` выглядит очень изящно, код на C# показывает, что никаких хитростей тут нет, а есть просто вызов метода, который ассоциирует с элементом постороннее свойство. Одной из любопытных особенностей абстракции присоединенных свойств является тот факт, что никакие свойства .NET в ней на самом деле не участвуют!

На внутреннем уровне методы типа `SetFontSize` просто обращаются к тому же методу `DependencyObject.SetValue`, который вызывает аксессор обычного свойства зависимости, но от имени не текущего экземпляра, а переданного объекта `DependencyObject`:

```
public static void SetFontSize(DependencyObject element, double value)
{
  element.SetValue(TextElement.FontSizeProperty, value);
}
```

А если в присоединенном свойстве определен статический метод GetXXX (где XXX – имя свойства), то будет вызываться уже знакомый нам метод DependencyObject.GetValue:

```
public static double GetFontSize(DependencyObject element)
{
  return (double)element.GetValue(TextElement.FontSizeProperty);
}
```

Как и в случае обертывающих свойств для обычных свойств зависимости, методы GetXXX и SetXXX не должны делать ничего, кроме вызова методов GetValue и SetValue соответственно.

КОПНЕМ ГЛУБЖЕ

О поставщиках присоединенных свойств

Самым странным в работе с присоединенными свойствами FontSize и FontStyle в листинге 3.5 является тот факт, что они определены не в классе Button и даже не в базовом классе Control, где определены обычные свойства зависимости FontSize и FontStyle, а в, казалось бы, совершенно не относящемся к делу классе TextElement (а также в классе TextBlock, которым можно было бы воспользоваться вместо TextElement)!

Но как это может работать, если поле TextElement.FontSizeProperty никак не связано с полем Control.FontSizeProperty (а TextElement.FontStyleProperty – с полем Control.FontStyleProperty)? Ключ к решению загадки – способ внутренней регистрации этих свойств зависимости. В исходном коде класса TextElement имеются такие строки:

```
TextElement.FontSizeProperty = DependencyProperty.RegisterAttached(
  "FontSize", typeof(double), typeof(TextElement),new FrameworkPropertyMetadata(
  SystemFonts.MessageFontSize, FrameworkPropertyMetadataOptions.Inherits |
  FrameworkPropertyMetadataOptions.AffectsRender |
  FrameworkPropertyMetadataOptions.AffectsMeasure),
  new ValidateValueCallback(TextElement.IsValidFontSize));
```

Это похоже на приведенный выше пример регистрации свойства зависимости IsDefault в классе Button с тем отличием, что метод RegisterAttached оптимизирует обработку метаданных свойства так, чтобы его можно было использовать в качестве присоединенного.

Напротив, в классе Control свойство зависимости FontSize не зарегистрировано как присоединенное! Вместо этого вызывается метод AddOwner для уже зарегистрированного в TextElement свойства, а этот метод возвращает ссылку на уже имеющийся экземпляр:

```
Control.FontSizeProperty = TextElement.FontSizeProperty.AddOwner(
  typeof(Control), new FrameworkPropertyMetadata(SystemFonts.MessageFontSize,
  FrameworkPropertyMetadataOptions.Inherits));
```

Поэтому FontSize, FontStyle и все прочие относящиеся к шрифтам свойства зависимости, наследуемые всеми элементами управления, – это *те же самые* свойства, что раскрывает класс TextElement!

К счастью, в большинстве случаев класс, раскрывающий некоторое присоединенное свойство (методы GetXXX и SetXXX), – это тот же класс, в котором определено обычное свойство зависимости, так что путаницы не возникает.

КОПНЕМ ГЛУБЖЕ

Присоединенные свойства и механизм расширяемости

Как и в предшествующих технологиях, например Windows Forms, во многих классах WPF определено свойство Tag (типа System.Object), которое предназначено для хранения произвольных данных, ассоциированных с экземпляром. Однако присоединенные свойства – более мощный и гибкий механизм присоединения данных к объектам классов, производных от DependencyObject. Часто забывают, что присоединенные свойства позволяют присоединять произвольные данные даже к экземплярам запечатанных классов (а таких в WPF хватает)!

И еще одно замечание по поводу присоединенных свойств: хотя их установка в XAML-коде опирается на наличие статического метода SetXXX, при написании процедурного кода этот метод можно обойти и вызывать метод DependencyObject.SetValue напрямую. Это означает, что в процедурном коде *любое* свойство зависимости можно использовать как присоединенное. Например, в следующем фрагменте к объекту класса Button присоединяется свойство IsTextSearchEnabled, определенное в классе ItemsControl, после чего ему присваивается значение:

```
// Присоединить к Button постороннее свойство
// и присвоить ему значение true
okButton.SetValue(ItemsControl.IsTextSearchEnabledProperty, true);
```

Хотя особого смысла в этом на первый взгляд нет и никакой новой функциональности в классе Button волшебным образом не появится, тем не менее к значению этого свойства можно будет обращаться из приложения или компонента, и иногда это бывает полезно.

Есть и более интересные способы расширять элементы подобным образом. Например, свойство Tag в классе FrameworkElement является свойством зависимости, поэтому его можно присоединить к экземпляру класса GeometryModel3D (этот класс, к которому мы еще вернемся в главе 16, является запечатанным и *не* имеет свойства Tag):

```
GeometryModel3D model = new GeometryModel3D();
model.SetValue(FrameworkElement.TagProperty, "my custom data");
```

И это лишь один из многих способов расширения WPF без применения традиционного наследования.

Хотя в примере диалогового окна About присоединенные свойства использовались как специфический механизм наследования значений свойств, чаще они применяются для компоновки элементов в макете пользовательского интерфейса. (На самом деле присоединенные свойства первоначально и были предназначены именно для системы компоновки в WPF.) В различных производных от Panel классах определены присоединенные свойства, которые рассчитаны на присоединение к потомкам с целью задания способа компоновки. Поэтому любая панель Panel может применять свое поведение к произвольным дочерним элементам, не требуя, чтобы в каждом из них были определены соответствующие свойства. Заодно это позволяет без особого труда расширять систему компоновки и ей подобные, поскольку любой человек может написать новый подкласс Panel с нестандартными присоединенными свойствами. В главах 5 «Компоновка с помощью панелей» и 21 «Компоновка с помощью нестандартных панелей» эта тема рассматривается более подробно.

Резюме

В этой и двух предыдущих главах мы узнали, как на фундаменте каркаса .NET Framework возводится здание WPF. Разработчики WPF могли бы раскрыть ее механизмы с помощью типичных для .NET API, как в Windows Forms, и при этом все равно получилась бы интересная технология. Но они решили поступить по-другому и добавили несколько принципиально новых концепций, позволяющих раскрыть богатейший набор возможностей таким способом, который существенно повышает продуктивность работы программистов и дизайнеров.

Действительно, пристальное изучение новых концепций в этой главе показывает, что общая картина стала не такой простой, как раньше: появились новые типы свойств, различные деревья и разные способы достижения одного и того же результата (декларативный или процедурный код)! Надеюсь, что вы сумеете по достоинству оценить хотя бы часть этих новых механизмов. Далее в этой книге мы не будем подробно обсуждать рассмотренные концепции, поскольку основное внимание сосредоточим на решении конкретных задач разработки приложений.

II

Создание WPF-приложения

Глава 4 «Задание размера, положения и преобразований элементов»

Глава 5 «Компоновка с помощью панелей»

Глава 6 «События ввода: клавиатура, мышь, стилус и мультисенсорные устройства»

Глава 7 «Структурирование и развертывание приложения»

Глава 8 «Особенности Windows 7»

- Управление размером
- Управление положением
- Применение преобразований

4

Задание размера, положения и преобразований элементов

При создании WPF-приложения одной из первых встает задача о размещении многочисленных элементов управления на поверхности окна. Процедура задания размеров и положений элементов управления (и других элементов) называется *компоновкой*, или *верстkой макета*.

В WPF имеется богатая инфраструктура компоновки. В ее основе лежит механизм взаимодействия между элементами-родителями и их потомками. Совместно они договариваются об окончательных размерах и положении. Хотя в конечном итоге именно родитель говорит своим детям, где они должны рисовать себя и сколько места им отведено, но действует он не как диктатор, а как сотрудник; родитель спрашивает своих детей, сколько места они хотели бы получить, и только потом принимает окончательное решение.

Родительские элементы, поддерживающие компоновку нескольких детей, называются *панелями*, они наследуют абстрактному классу `System.Windows.Controls.Panel`. Все элементы, участвующие в процессе компоновки (как родители, так и потомки), наследуют классу `System.Windows.UIElement`.

Поскольку тема компоновки в WPF является обширной и важной, то в этой книге ей посвящено три главы:

- Глава 4 «Задание размера, положения и преобразований элементов»
- Глава 5 «Компоновка с помощью панелей»
- Глава 21 «Компоновка с помощью нестандартных панелей»

В этой главе мы сосредоточимся на дочерних элементах и рассмотрим вопрос о том, как управлять компоновкой на уровне отдельных дочерних элементов. Эти аспекты контролируются несколькими свойствами, большая часть которых перечислена на рис. 4.1 на примере произвольного элемента, находящегося внутри произвольной панели. Свойства, относящиеся к заданию размера,

показаны синим цветом, а относящиеся к положению, – красным. Дополнительно к элементам могут применяться преобразования (показаны зеленым цветом), влияющие как на их размер, так и на положение.

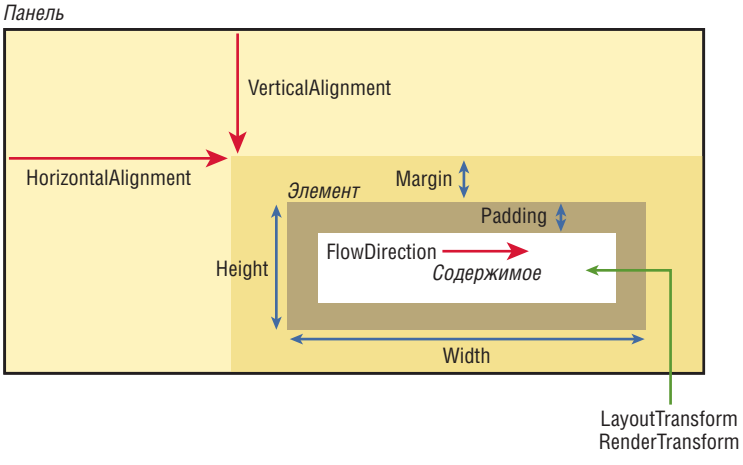


Рис. 4.1. *Основные свойства, управляющие компоновкой дочерних элементов, которые рассматриваются в этой главе*

В следующей главе мы продолжим рассказ о компоновке и обсудим все многообразие встроенных в WPF панелей, каждая из которых компонует дочерние элементы по-своему. Создание нестандартных панелей – отдельная непростая тема, которой посвящена последняя часть этой книги.

Управление размером

Всякий раз, когда требуется произвести компоновку (например, после изменения размера окна), дочерние элементы сообщают родительской панели свой предпочтительный размер. Обычно элементы WPF стремятся *подстроиться под размер своего содержимого*, то есть выбрать для себя размер, достаточный для размещения всего содержимого, но не больше. (Так поступает даже элемент `Window`, но только если явно задано свойство `SizeToContent`, как было в примерах из предыдущей главы.) Отдельные элементы могут влиять на выбор этого размера с помощью нескольких простых свойств.

Свойства Height и Width

Во всех классах, производных от `FrameworkElement`, есть свойства `Height` (высота) и `Width` (ширина) (типа `double`), а также `MinHeight`, `MaxHeight`, `MinWidth` и `MaxWidth`, которыми можно пользоваться для задания допустимых диапазонов значений. Все эти свойства можно задавать в любой комбинации как в процедурном коде, так и в XAML.

Обычно элемент стремится принять минимально возможный размер, поэтому если задано свойство MinHeight или MinWidth, то при визуализации выбирается именно такая высота или ширина при условии, что содержимое не вынуждает увеличить размер. Но увеличение можно ограничить с помощью свойств MaxHeight и MaxWidth (при условии, что эти значения больше соответствующих минимальных). Если одновременно с минимальными и максимальными значениями заданы свойства Height и Width, то последние имеют приоритет при условии, что попадают внутрь диапазона между Min и Max. По умолчанию MinHeight и MinWidth равны 0, а MaxHeight и MaxWidth – величине Double.PositiveInfinity (которая в XAML записывается просто как "Infinity").

ПРЕДУПРЕЖДЕНИЕ

Избегайте явного задания размеров!

Если явно задавать размеры элементов управления, особенно производных от класса ContentControl, например Button и Label, то возникает риск отсечения текста в случае, когда пользователь изменяет системный шрифт или текст переводится на другие языки. Поэтому лучше не задавать размеры явно, если без этого можно обойтись. К счастью, благодаря наличию панелей необходимость явно задавать размеры возникает редко.

КОПНЕМ ГЛУБЖЕ

Специальное значение длины "Auto"

Свойства Height и Width класса FrameworkElement по умолчанию принимают значение Double.NaN (NaN означает *not a number* – «не число»), которое означает, что размер элемента подстраивается под размер содержимого. Это значение можно и явно задать в XAML-коде в виде строки "NaN" (чувствительной к регистру) или более предпочтительной строки "Auto" (нечувствительной к регистру), что обеспечивает конвертер типа LengthConverter, ассоциированный с этими свойствами. Чтобы проверить, выбирается ли размер элемента автоматически, можно воспользоваться статическим методом Double.IsNaN.

Ситуация осложняется тем, что в классе FrameworkElement есть еще несколько свойств, относящихся к размеру:

- DesiredSize (наследуется от UIElement)
- RenderSize (наследуется от UIElement)
- ActualHeight и ActualWidth

В отличие от остальных шести свойств, они являются не *входными* данными для процедуры компоновки, а *выходными* – представляющими результат компоновки, и потому доступны только для чтения. Свойство элемента DesiredSize вычисляется в процессе компоновки на основе значений других свойств

(в том числе вышеупомянутых Width, Height, Min*XXX* и Max*XXX*) и места, которое родитель готов выделить. Оно используется панелями для внутренних целей.

Свойство RenderSize представляет окончательный размер элемента по завершении компоновки, а ActualHeight и ActualWidth – в точности то же самое, что RenderSize.Height и RenderSize.Width. Запомните: каким бы способом ни был задан размер элемента – явно, с помощью допустимых диапазонов значений или не задан вовсе, – родитель вправе изменить окончательный размер элемента на экране. Поэтому три упомянутых свойства бывают полезны в тех редких случаях, когда поведение программы зависит от размера элемента. С другой стороны, значения остальных свойств, относящихся к размеру, не очень интересны для формулирования алгоритма. Например, если свойства Height и Width не заданы явно, то они будут иметь значение Double.NaN, каким бы ни оказался истинный размер элемента.

В главе 21 использование всех этих свойств демонстрируется в контексте.

ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны использовании в коде свойств ActualHeight и ActualWidth (или RenderSize)!

При каждой компоновке значение RenderSize (а значит, также ActualHeight и ActualWidth) обновляется. Однако компоновка производится асинхронно, поэтому нельзя рассчитывать, что эти значения в любой момент правильны. Обращаться к ним безопасно только внутри обработчика события LayoutUpdated, определенного в классе UIElement.

Есть и другой способ – в классе UIElement имеется метод UpdateLayout, который синхронно производит все отложенные обновления макета, но лучше его не применять. Мало того что частые обращения к UpdateLayout могут негативно сказаться на производительности, так еще и нет гарантии, что используемые элементы корректно обрабатывают реентерабельность в методах, относящихся к компоновке.

Свойства Margin и Padding

Очень похожие свойства Margin и Padding тоже связаны с размером элемента. Свойство Margin определено для всех объектов, производных от FrameworkElement, а свойство Padding – во всех элементах управления, производных от класса Control (а также в классе Border). Различие в том, что Margin задает *внешнее* поле вокруг элемента, а Padding – *внутренний* отступ между содержимым элемента и его границами.

Оба свойства имеют тип System.Windows.Thickness; это любопытный класс, который может представлять одно, два или четыре значения типа double. Интерпретация этих значений демонстрируется в листинге 4.1, где для меток Label задаются различные отступы Padding и поля Margin. Второй набор меток заключен в рамки Border, поскольку иначе поля были бы не видны. На рис. 4.2

показано, как выглядит результат в случае, когда каждая метка помещена в отдельный элемент Canvas (это панель, рассматриваемая в следующей главе). Хотя на рисунке это и не показано, свойство Margin (но не Padding) может принимать отрицательные значения.

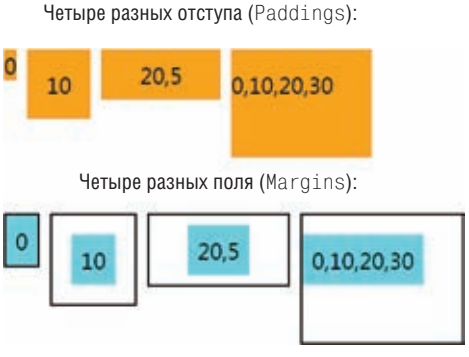


Рис. 4.2. *Результат установки свойств Margin и Padding*

Листинг 4.1. Задание свойств Margin и Padding с одним, двумя и четырьмя числовыми значениями

```
<!-- Отступы: -->

<!-- 1 значение: один и тот же отступ со всех четырех сторон: -->
<Label Padding="0" Background="Orange">0</Label>
<Label Padding="10" Background="Orange">10</Label>

<!-- 2 значения: первое значение относится к левому и правому
                 отступам, второе - к верхнему и нижнему: -->
<Label Padding="20,5" Background="Orange">20,5</Label>

<!-- 4 значения: левый, верхний, правый, нижний: -->
<Label Padding="0,10,20,30" Background="Orange">0,10,20,30</Label>

<!-- Поля: -->

<Border BorderBrush="Black" BorderThickness="1">
  <!-- Поле отсутствует: -->
  <Label Background="Aqua">0</Label>
</Border>

<Border BorderBrush="Black" BorderThickness="1">
  <!-- 1 значение: одно и то же поле со всех четырех сторон: -->
  <Label Margin="10" Background="Aqua">10</Label>
</Border>

<Border BorderBrush="Black" BorderThickness="1">
```

```
    <!-- 2 значения: первое значение относится к левому и правому
                     полям, второе - к верхнему и нижнему: -->
    <Label Margin="20,5" Background="Aqua">20,5</Label>
  </Border>

  <Border BorderBrush="Black" BorderThickness="1">
    <!-- 4 значения: левое, верхнее, правое, нижнее: -->
    <Label Margin="0,10,20,30" Background="Aqua">0,10,20,30</Label>
  </Border>
```

Для элемента `Label` свойство `Padding` по умолчанию равно `5`, но его можно заменить любым другим допустимым значением. Именно поэтому в листинге 4.1 свойство `Padding` для первой метки имеет значение `0`. В противном случае эта метка выглядела бы точно так же, как пятая (та, в которой демонстрируется неявно заданное поле `Margin`, равное `0`), и визуальное сравнение с другими значениями `Padding` было бы неочевидно.

КОПНЕМ ГЛУБЖЕ

Синтаксис задания значений типа Thickness

Синтаксис задания значений через запятую, поддерживаемый свойствами `Margin` и `Padding`, обеспечивает – догадайтесь, кто – конвертер типа `System.Windows.ThicknessConverter`, который конструирует объект типа `Thickness` из строки. В классе `Thickness` определены два конструктора: первый принимает одно значение типа `double`, второй – четыре. Следовательно, в программе на языке C# допустимы следующие формы:

```
myLabel.Margin = new Thickness(10);         // То же, что Margin="10" в XAML
myLabel.Margin = new Thickness(20,5,20,5);  // То же, что Margin="20,5" в XAML
myLabel.Margin = new Thickness(0,10,20,30); // То же, что Margin="0,10,20,30" в XAML
```

Обратите внимание, что удобный синтаксис с двумя числами доступен только через конвертер типа!

FAQ

Какие единицы измерения применяются в WPF?

Конвертер `LengthConverter`, ассоциируемый с различными свойствами, касающимися длин, поддерживает явное задание единиц измерения `cm`, `pt`, `in` и `px` (по умолчанию).

По умолчанию все абсолютные измерения (в частности, величины свойств, обсуждаемых в этом разделе) выражаются в *независимых от устройства пикселах*. Такие «логические пикселы» представляют 1/96 дюйма независимо от разрешения экрана, выраженного в точках на дюйм (DPI). Отметим, что количество независимых от устройства пикселов всегда задается в виде значения с двойной точностью, то есть может быть дробным.

Точная величина – 1/96 дюйма – не так существенна, а выбрана она была потому, что на типичном экране с разрешением 96 DPI один независимый от устройства пиксел в точности совпадает с одним физическим пикселом. Разумеется, понятие «истинного» дюйма зависит от физического устройства отображения. Если приложение нарисует отрезок длиной 1 дюйм на экране моего ноутбука, то при выводе изображения на проектор длина этого отрезка, конечно, окажется больше!

Важно лишь, что все измерения не зависят от разрешающей способности. Но само по себе это не мешает изображению уменьшаться при увеличении разрешения экрана. Чтобы размер изображения не зависел от разрешения, необходим механизм автоматического масштабирования, обсуждаемый в следующей главе.

Свойство Visibility

Может показаться странным, что мы обсуждаем свойство `Visibility` (определенное в классе `UIElement`) в контексте компоновки, однако оно действительно относится именно к этой теме. Тип свойства элемента `Visibility` – не `Boolean`, а перечисление `System.Windows.Visibility` с тремя состояниями, то есть оно может принимать три значения:

- `Visible` – элемент виден и участвует в компоновке.
- `Collapsed` – элемент не виден и не участвует в компоновке.
- `Hidden` – элемент не виден, но *тем не менее участвует в компоновке*.

Свернутый (`Collapsed`) элемент, по существу, имеет нулевой размер, тогда как скрытый (`Hidden`) элемент сохраняет свой первоначальный размер. (Так, значения его свойств `ActualHeight` и `ActualWidth` не изменяются.) Разница между состояниями `Collapsed` и `Hidden` продемонстрирована на рис. 4.3, где панель `StackPanel` со свернутой кнопкой:

```
<StackPanel Height="100" Background="Aqua">
  <Button Visibility="Collapsed">Collapsed Button</Button>
  <Button>Below a Collapsed Button</Button>
</StackPanel>
```

сравнивается с панелью `StackPanel` со скрытой кнопкой:

```
<StackPanel Height="100" Background="Aqua">
  <Button Visibility="Hidden">Hidden Button</Button>
  <Button>Below a Hidden Button</Button>
</StackPanel>
```

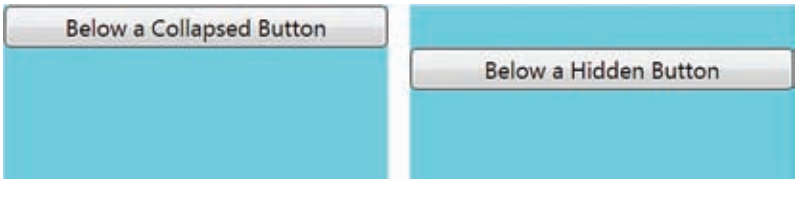


Рис. 4.3. *Скрытая кнопка занимает место на экране, а свернутая – нет*

Управление положением

В этом разделе мы не обсуждаем позиционирование элементов с помощью задания координат (X,Y). Родительские панели определяют собственные механизмы (для каждой панели свой), позволяющие дочерним элементам позиционировать себя (либо посредством присоединенных свойств, либо просто в том порядке, в котором дочерние элементы добавлялись на панель). Однако есть несколько механизмов, общих для всех дочерних элементов типа `FrameworkElement`, и именно их мы сейчас и рассмотрим. Все они касаются выравнивания, а сама концепция носит название «*направление потока*».

Выравнивание

С помощью свойств `HorizontalAlignment` и `VerticalAlignment` элемент может управлять распределением избыточного пространства, выделенного ему родителем. Значениями свойств являются одноименные перечисления, которые определены в пространстве имен `System.Windows`:

- `HorizontalAlignment` — `Left`, `Center`, `Right`, `Stretch`
- `VerticalAlignment` — `Top`, `Center`, `Bottom`, `Stretch`

По умолчанию оба свойства принимают значение `Stretch`, хотя в стилях тем для различных элементов управления оно может быть переопределено. Чтобы посмотреть, как свойство `HorizontalAlignment` влияет на компоновку, достаточно поместить несколько кнопок `Button` на панель `StackPanel` и задать для них разные значения перечисления:

```
<StackPanel>
  <Button HorizontalAlignment="Left"
          Background="Red">Left</Button>
  <Button HorizontalAlignment="Center"
          Background="Orange">Center</Button>
  <Button HorizontalAlignment="Right"
          Background="Yellow">Right</Button>
  <Button HorizontalAlignment="Stretch"
          Background="Lime">Stretch</Button>
</StackPanel>
```

Результат показан на рис. 4.4.



Рис. 4.4. *Влияние HorizontalAlignment на размещение кнопок на панели StackPanel*

Эти два свойства полезны только в случае, когда родительская панель выделяет дочернему элементу больше места, чем тому необходимо. Так, задание свойства VerticalAlignment для элементов на панели StackPanel, изображенной на рис. 4.4, ничего не изменит, потому что каждому элементу уже выделена ровно такая высота, какая ему требуется, – не больше и не меньше.

КОПНЕМ ГЛУБЖЕ

Взаимодействие между типом выравнивания Stretch и явным заданием размера элемента

Даже если для элемента в качестве выравнивания задано растяжение (`Stretch`) по горизонтали или по вертикали, приоритет все равно отдается явно заданной высоте `Height` или ширине `Width`. Свойства `MaxHeight` и `MaxWidth` также более приоритетны, но только в том случае, когда их значения меньше размера, получившегося после растяжения. Аналогично свойствам `MinHeight` и `MinWidth` приоритет отдается лишь тогда, когда их значения *больше* размера, получившегося после растяжения. Если свойство `Stretch` используется в контексте, где на размер элемента налагаются ограничения, то оно действует как тип выравнивания `Center` (или `Left`, если элемент слишком велик и не может быть отцентрирован внутри своего родителя).

Выравнивание содержимого

Помимо свойств `HorizontalAlignment` и `VerticalAlignment`, в классе `Control` имеются свойства `HorizontalContentAlignment` и `VerticalContentAlignment`. Они определяют порядок размещения содержимого *внутри* элемента управления. (То есть соотношение между выравниванием и выравниванием содержимого примерно такое же, как между полями и отступами.)

Свойства, управляющие выравниванием содержимого, принадлежат тем же типам перечисления, что и соответствующие свойства выравнивания, следовательно, и возможности у них точно такие же. Однако по умолчанию свойство `HorizontalContentAlignment` равно `Left`, а `VerticalContentAlignment` равно `Top`. Для показанных выше кнопок мы этого не наблюдали, потому что значения переопределены в стиле темы. (Вспомните порядок приоритетов различных поставщиков значений свойств зависимости, описанный в предыдущей главе. У значений по умолчанию приоритет самый низкий, поэтому они замещаются значениями, заданными в стилях.)

На рис. 4.5 показано, как выглядят кнопки при различных значениях `HorizontalContentAlignment`. Для этого мы просто модифицировали предыдущий фрагмент XAML-кода:

```
<StackPanel>
  <Button HorizontalContentAlignment="Left"
          Background="Red">Left</Button>
  <Button HorizontalContentAlignment="Center"
          Background="Orange">Center</Button>
```

```
    <Button HorizontalContentAlignment="Right"
            Background="Yellow">Right</Button>
    <Button HorizontalContentAlignment="Stretch"
            Background="Lime">Stretch</Button>
</StackPanel>
```

Кнопка `Button` на рис. 4.5, для которой `HorizontalContentAlignment="Stretch"`, выглядит несколько неожиданно. Ее внутренний элемент `TextBlock` действительно растянулся, однако класс `TextBlock` не является подклассом `Control` (он наследует непосредственно `FrameworkElement`), поэтому к заключенному внутри него тексту понятие растяжения неприменимо.



***Рис. 4.5.** Влияние HorizontalContentAlignment на размещение кнопок на панели StackPanel*

Свойство FlowDirection

Свойство `FlowDirection`, определенное в классе `FrameworkElement` (и еще нескольких), позволяет изменить направление визуализации внутреннего содержимого элемента. Оно применимо к некоторым панелям, где влияет на размещение дочерних элементов, а также модифицирует способ выравнивания содержимого внутри дочерних элементов. Тип этого свойства – перечисление `System.Windows.FlowDirection`, принимающее два значения: `LeftToRight` (по умолчанию в классе `FrameworkElement`) и `RightToLeft`.

Идея `FlowDirection` заключается в том, что для языка, записываемого справа налево, должно быть задано направление `RightToLeft`. Тем самым меняются местами понятия «левый» и «правый» для таких свойств, как выравнивание содержимого. В следующем фрагменте XAML присутствуют две кнопки, для которых задано одно и то же выравнивание содержимого `Top` и `Left`, но направление `FlowDirection` разное:

```
<StackPanel>
  <Button FlowDirection="LeftToRight"
          HorizontalContentAlignment="Left" VerticalContentAlignment="Top"
          Height="40" Background="Red">LeftToRight</Button>
  <Button FlowDirection="RightToLeft"
          HorizontalContentAlignment="Left" VerticalContentAlignment="Top"
          Height="40" Background="Orange">RightToLeft</Button>
</StackPanel>
```

Результат представлен на рис. 4.6.



Рис. 4.6. *Влияние FlowDirection на кнопки с выравниванием содержимого Top и Left*

Отметим, что свойство `FlowDirection` не оказывает влияния на направление записи букв в надписи внутри кнопок. Английские буквы всегда записываются слева направо, арабские – справа налево. Но понятия левого и правого во всех остальных частях интерфейса, которые обычно должны соответствовать направлению записи букв, меняются местами.

Свойство `FlowDirection` необходимо задавать явно в соответствии с текущей культурой (это достаточно сделать для элемента самого верхнего уровня). Это должно стать частью процедуры локализации.

Применение преобразований

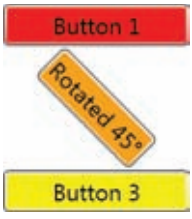
В WPF имеется целый ряд встроенных классов двумерных геометрических преобразований (производных от `System.Windows.Media.Transform`), которые позволяют изменять размер и положение элементов независимо от ранее рассмотренных свойств. Некоторые преобразования изменяют элементы и более экзотическими способами, например поворачивают или наклоняют их.

Во всех подклассах `FrameworkElement` имеется два свойства типа `Transform`, позволяющих применять преобразования:

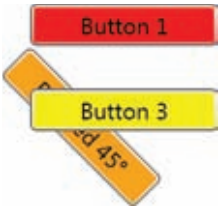
- `LayoutTransform` – применяется *до* компоновки элемента
- `RenderTransform` (унаследовано от `UIElement`) – применяется *после* завершения компоновки (непосредственно перед визуализацией элемента)

На рис. 4.7 показана разница между применением преобразования поворота `RotateTransform` в режиме `LayoutTransform` и в режиме `RenderTransform`. В обоих случаях преобразование применяется ко второй из трех кнопок. Но если оно применено как `LayoutTransform`, то третья кнопка сдвигается вниз, а если как `RenderTransform`, то третья кнопка размещается так, будто вторая не поворачивалась вовсе.

В классе `UIElement` имеется также полезное свойство `RenderTransformOrigin`, представляющее начальную точку преобразования (которая остается неподвижной). Для преобразования `RotateTransform` на рис. 4.7 началом является левый верхний угол кнопки, вокруг которого кнопка поворачивается. Для преобразований, применяемых в режиме `LayoutTransform`, понятие начальной точки не определено, потому что положение преобразованного элемента диктуется правилами компоновки, применяемыми родителем.



Поворот, примененный в режиме `LayoutTransform`



Поворот, примененный в режиме `RenderTransform`

Рис. 4.7. *Разница между применением преобразований LayoutTransform и RenderTransform к средней из трех кнопок на панели StackPanel*

Свойство `RenderTransformOrigin` имеет тип `System.Windows.Point` и по умолчанию равно (0,0). Этой точке соответствует левый верхний угол элемента, как показано на рис. 4.7. Точка (0,1) представляет левый нижний угол, (1,0) – правый верхний угол, а (1,1) – правый нижний угол. Можно задавать и числа, большие 1, – тогда начальная точка окажется вне границ элемента. Дробные значения также допустимы. В частности, точка (0.5,0.5) представляет середину объекта. На рис. 4.8 показано пять начальных точек, обычно задаваемых в качестве значения свойства `RenderTransformOrigin`.

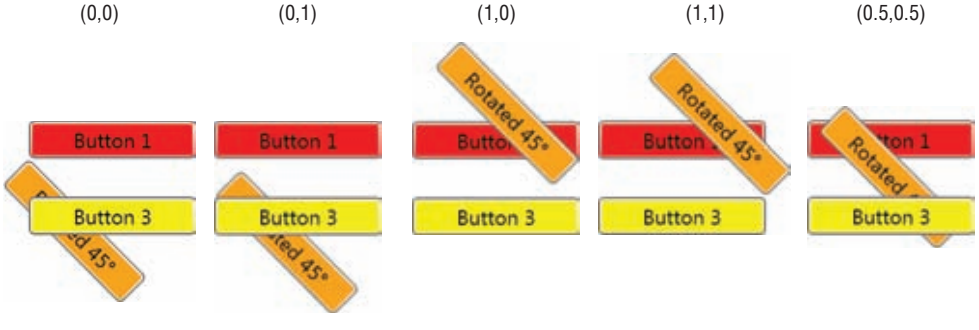


Рис. 4.8. *Пять типичных значений RenderTransformOrigin, применяемых для поворота кнопки, которая изображена на рис. 4.7*

Благодаря конвертеру `System.Windows.PointConverter` значение `RenderTransformOrigin` можно задавать в XAML в виде двух чисел, разделенных запятой (без скобок). Например, в следующем фрагменте создается кнопка `Button`, повернутая относительно своего центра (самая правая кнопка на рис. 4.8):

```
<Button RenderTransformOrigin="0.5,0.5" Background="Orange">
<Button.RenderTransform>
  <RotateTransform Angle="45"/>
</Button.RenderTransform>
  Rotated 45°
</Button>
```

Возможно, вам непонятно, зачем вообще в приложении может понадобиться повернутая кнопка. Да, для стандартных элементов управления со стилями по умолчанию подобное преобразование выглядит нелепо. Часто это имеет смысл в приложениях с сильно перегруженными темами, но даже для стилизованных по умолчанию элементов преобразования в сочетании с анимацией могут создать интересные эффекты.

В этом разделе мы рассмотрим все пять встроенных двумерных преобразований, определенных в пространстве имен `System.Windows.Media`:

- `RotateTransform`
- `ScaleTransform`
- `SkewTransform`
- `TranslateTransform`
- `MatrixTransform`

Преобразование RotateTransform

Преобразование `RotateTransform`, продемонстрированное в предыдущем разделе, поворачивает элемент в соответствии со следующими тремя свойствами типа `double`:

- `Angle` – угол поворота в градусах (по умолчанию `0`)
- `CenterX` – абсцисса центра поворота (по умолчанию `0`)
- `CenterY` – ордината центра поворота (по умолчанию `0`)

Точка (`CenterX`,`CenterY`), равная по умолчанию (0,0), соответствует левому верхнему углу. Свойства `CenterX` и `CenterY` принимаются во внимание, только если преобразование применяется в режиме `RenderTransform`, потому что для преобразования в режиме `LayoutTransform` положение центра поворота определяется родительской панелью.

FAQ

В чем разница между использованием свойств CenterX и CenterY для преобразований вида RotateTransform и свойством RenderTransformOrigin элемента типа UIElement?

Складывается впечатление, что, когда преобразование применяется к элементу типа `UIElement`, свойства `CenterX` и `CenterY` избыточны, так как уже имеется свойство `RenderTransformOrigin`. Ведь оба механизма определяют начальную точку преобразования и оба работают, только если преобразование применяется в режиме `RenderTransform`.

Однако же `CenterX` и `CenterY` задают абсолютное положение начальной точки, тогда как `RenderTransformOrigin` – относительное. Значения задаются в независимых от устройства пикселах, так что для правого верхнего угла элемента с шириной `Width`, равной `20`, свойство `CenterX` будет равно `20`, а `CenterY` – `0`, а не (1,0),

как для RenderTransformOrigin. Кроме того, при комбинировании нескольких преобразований RenderTransform (см. следующую главу) задание CenterX и CenterY для отдельных преобразований обеспечивает более точный контроль. Наконец, раздельные значения CenterX и CenterY типа double проще использовать для привязки к данным, чем одно значение RenderTransformOrigin типа Point.

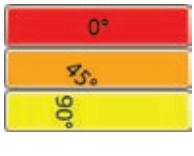
Тем не менее свойство RenderTransformOrigin, вообще говоря, более полезно, чем CenterX и CenterY. В типичном случае, когда элемент поворачивается относительно своей середины, относительные координаты (0.5,0.5), задаваемые с помощью RenderTransformOrigin, проще записать в XAML. Для достижения того же эффекта с помощью CenterX и CenterY пришлось бы писать процедурный код, вычисляющий абсолютные смещения.

Отметим, что задавать RenderTransformOrigin для элемента можно одновременно с установкой значений CenterX и CenterY для его преобразования. В этом случае пары значений X и Y комбинируются для вычисления окончательной начальной точки.

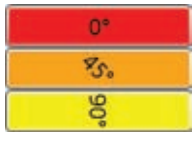
На рис. 4.7 и 4.8 показаны результаты поворота кнопки, а на рис. 4.9 демонстрируется, что происходит, когда преобразование RotateTransform применяется в режиме RenderTransform к *внутреннему содержимому* кнопок с двумя различными значениями RenderTransformOrigin. Для этого простая строка в составе элемента Button заменяется следующим явным элементом TextBlock:

```
<Button Background="Orange">
  <TextBlock RenderTransformOrigin="0.5,0.5">
  <TextBlock.RenderTransform>
    <RotateTransform Angle="45"/>
  </TextBlock.RenderTransform>
    45°
  </TextBlock>
</Button>
```

Может показаться, что элементы TextBlock внутри Button в левой части рис. 4.9 повернуты не вокруг левого верхнего угла кнопки, но это потому, что сам элемент TextBlock чуть больше содержащегося внутри него текста. Если закрасить текстовые блоки бирюзовым фоном, то результат поворота будет более наглядным (рис. 4.10).



Поворот текста вокруг левого верхнего угла



Поворот текста вокруг средней точки

Рис. 4.9. *Применение RotateTransform к содержимому кнопок на панели StackPanel*



Рис. 4.10. *Для внутренних элементов TextBlock, повернутых вокруг левого верхнего угла, явно задан цвет фона*

`RotateTransform` имеет параметризованные конструкторы, которые принимают значения угла или угла и центра, для удобства выполнения преобразования из процедурного кода.

Преобразование ScaleTransform

Преобразование `ScaleTransform` увеличивает или уменьшает элемент по горизонтали, по вертикали или в обоих направлениях. У него есть четыре свойства типа `double`:

- `ScaleX` – коэффициент изменения ширины элемента (по умолчанию 1)
- `ScaleY` – коэффициент изменения высоты элемента (по умолчанию 1)
- `CenterX` – начальная точка для масштабирования по горизонтали (по умолчанию 0)
- `CenterY` – начальная точка для масштабирования по вертикали (по умолчанию 0)

Если `ScaleX` равно `0.5`, то ширина рисуемого элемента уменьшается вдвое, а если `ScaleX` равно `2`, то вдвое увеличивается. Смысл свойств `CenterX` и `CenterY` такой же, как для преобразования `RotateTransform`.

В листинге 4.2 преобразование `ScaleTransform` применяется к трем кнопкам `Button` на панели `StackPanel`, демонстрируя возможность независимого изменения высоты и ширины. На рис. 4.11 представлен результат.

Листинг 4.2. Применение ScaleTransform к кнопкам на панели StackPanel

```
<StackPanel Width="100">
  <Button Background="Red">No Scaling</Button>
  <Button Background="Orange">
  <Button.RenderTransform>
    <ScaleTransform ScaleX="2"/>
  </Button.RenderTransform>
    X</Button>
  <Button Background="Yellow">
  <Button.RenderTransform>
    <ScaleTransform ScaleX="2" ScaleY="2"/>
  </Button.RenderTransform>
    X + Y</Button>
  <Button Background="Lime">
  <Button.RenderTransform>
    <ScaleTransform ScaleY="2"/>
```

```
    </Button.RenderTransform>
      Y</Button>
  </StackPanel>
```

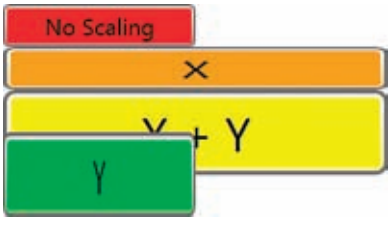


Рис. 4.11. *Результат масштабирования кнопок, произведенного в листинге 4.2*

На рис. 4.12 изображены кнопки из листинга 4.2, для которых дополнительно явно заданы свойства `CenterX` и `CenterY`. В качестве текста каждой кнопки указаны координаты начальной точки. Обратите внимание, что зеленая кнопка не сдвинута влево, как оранжевая, хотя `CenterX` в обоих случаях равно `70`. Дело в том, что `CenterX` принимается во внимание, только если значение `ScaleX` отлично от `1`, а `CenterY` – если `ScaleY` не равно `1`.

Как и во всех остальных классах преобразований, в `ScaleTransform` определено несколько конструкторов для удобства создания объектов в процедурном коде.

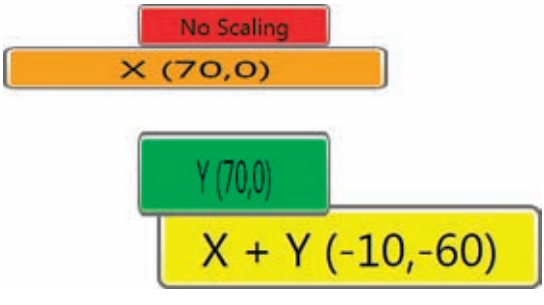


Рис. 4.12. *Кнопки из листинга 4.2, но с явно заданными центрами масштабирования*

КОПНЕМ ГЛУБЖЕ

Взаимодействие между ScaleTransform и выравниванием типа Stretch

Если преобразование `ScaleTransform` применяется в режиме `LayoutTransform` к элементу, который уже растянут в направлении масштабирования, то оно принимается во внимание только в случае, когда размер после масштабирования больше размера, получившегося в результате растяжения.

FAQ

Как преобразования, подобные ScaleTransform, влияют на свойства ActualHeight и ActualWidth элемента типа FrameworkElement и на свойство RenderSize элемента типа UIElement?

Применение преобразования к элементу типа `FrameworkElement` никогда не изменяет значений этих свойств. Это справедливо вне зависимости от того применяется преобразование в режиме `RenderTransform` или `LayoutTransform`. Поэтому из-за преобразований эти свойства могут сообщать «ложный» размер элемента на экране. Например, у всех кнопок на рис. 4.11 и 4.12 величины `ActualHeight`, `ActualWidth` и `RenderSize` одинаковы.

Быть может, вас удивляет такая «ложь», но это правильное решение. Во-первых, не вполне понятно, как следует выражать эти значения для некоторых преобразований. Важнее, однако, то, что цель преобразования – изменить внешний вид элемента так, чтобы сам элемент об этом ничего не знал. Создавая у элемента иллюзию того, что он визуализируется нормально, мы можем единообразно преобразовывать произвольные элементы.

FAQ

Как преобразование ScaleTransform влияет на свойства Margin и Padding?

Свойство `Padding` масштабируется вместе со всем содержимым (поскольку отступ находится внутри элемента), а свойство `Margin` не масштабируется вовсе. Как и в случае `ActualHeight` и `ActualWidth`, числовое значение свойства `Padding` не изменяется, несмотря на визуальный эффект масштабирования.

Преобразование SkewTransform

Преобразование `SkewTransform` наклоняет элемент в соответствии со значениями четырех свойств типа `double`:

- `AngleX` – угол наклона по горизонтали (по умолчанию 0)
- `AngleY` – угол наклона по вертикали (по умолчанию 0)
- `CenterX` – начальная точка для наклона по горизонтали (по умолчанию 0)
- `CenterY` – начальная точка для наклона по вертикали (по умолчанию 0)

Эти свойства по своему поведению очень похожи на свойства рассмотренных выше преобразований. На рис. 4.13 показаны результаты применения `SkewTransform` в качестве `RenderTransform` к нескольким кнопкам; центром наклона является левый верхний угол.

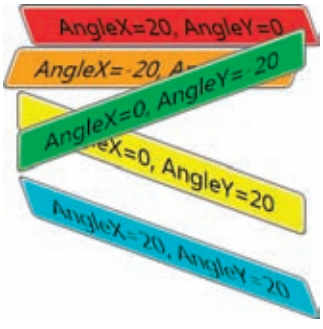


Рис. 4.13. *Применение преобразования SkewTransform к кнопкам на панели StackPanel*

Преобразование TranslateTransform

Преобразование `TranslateTransform` просто параллельно переносит элемент в соответствии со значениями двух свойств типа `double`:

- `X` – величина смещения по горизонтали (по умолчанию 0)
- `Y` – величина смещения по вертикали (по умолчанию 0)

`TranslateTransform` не дает никакого эффекта, когда применяется в режиме `LayoutTransform`, но применение его в режиме `RenderTransform` – удобный способ «подвинуть» элементы. Чаще всего это делается динамически в ответ на действия пользователя (и, быть может, в составе анимации). Маловероятно, что при работе с панелями, описанными в следующей главе, вы захотите использовать это преобразование для компоновки статического пользовательского интерфейса.

Преобразование MatrixTransform

Преобразование `MatrixTransform` представляет собой низкоуровневый механизм описания произвольного двумерного преобразования. У него есть единственное свойство `Matrix` (типа `System.Windows.Media.Matrix`), представляющее матрицу аффинного преобразования размером 3×3. На случай если вы не знакомы с линейной алгеброй, сообщу, что все рассмотренные выше преобразования (и их комбинации) также можно осуществить с помощью `MatrixTransform`.

Матрица устроена следующим образом:

$$\begin{bmatrix} M11 & M12 & 0 \\ M21 & M22 & 0 \\ OffsetX & OffsetY & 1 \end{bmatrix}$$

Значения в последнем столбце фиксированы, а остальные шесть значений можно задать в виде свойств объекта `Matrix` (с показанными на рисунке именами)

или в конструкторе, который принимает шесть чисел в порядке следования строк матрицы.

КОПНЕМ ГЛУБЖЕ

Конвертер типа MatrixTransform

`MatrixTransform` – единственное преобразование, конвертер типа которого позволяет описывать его в XAML с помощью простой строки. (Класс конвертера называется `TransformConverter` и, хотя он ассоциирован с абстрактным классом `Transform`, в реальности поддерживает только тип `MatrixTransform`.) Например, чтобы переместить кнопку на 10 единиц вправо и на 20 единиц вниз, нужно написать такой код:

```
<Button RenderTransform="1,0,0,1,10,20" />
```

Через запятую указаны элементы матрицы в следующем порядке: `M11`, `M12`, `M21`, `M22`, `OffsetX`, `OffsetY`. Последовательность `1`, `0`, `0`, `1`, `0`, `0` соответствует тождественной матрице (то есть отсутствию преобразования), поэтому задание преобразования `TranslateTransform` в виде `MatrixTransform`, по существу, означает, что мы начинаем с тождественной матрицы, а потом используем величины `OffsetX` и `OffsetY` в качестве свойств `X` и `Y` преобразования `TranslateTransform`. Масштабирование можно выразить, интерпретируя первое и четвертое значения как свойства `ScaleX` и `ScaleY` соответственно, а остальные оставив такими, как в тождественной матрице. Поворот и наклон записываются несколько сложнее, потому что необходимо привлекать функции `sin` и `cos` и углы, выраженные в радианах.

Если вы уверенно владеете матричной нотацией, то сможете сэкономить время при написании XAML-кода вручную, поскольку такая запись короче (но и менее понятна).

Комбинирование преобразований

Комбинировать преобразования можно различными способами, например повернуть элемент с одновременным масштабированием. Можно совместно применять преобразования в режимах `LayoutTransform` и `RenderTransform`. Или вычислить матрицу преобразования `MatrixTransform`, которое даст желаемый эффект. Однако, скорее всего, вы предпочтете воспользоваться классом `TransformGroup`.

Класс `TransformGroup` также наследует классу `Transform` (и потому может использоваться всюду, где разрешено применять описанные выше классы), а его задача – скомбинировать несколько дочерних объектов типа `Transform`. В процедурном коде объекты отдельных преобразований добавляются в коллекцию `Children`, в XAML это делается следующим образом:

```
<Button>
<Button.RenderTransform>
  <TransformGroup>
    <RotateTransform Angle="45"/>
```

```
      <ScaleTransform ScaleX="5" ScaleY="1"/>
      <SkewTransform AngleX="30"/>
    </TransformGroup>
  </Button.RenderTransform>
    OK
  </Button>
```

На рис. 4.14 показан результат применения всех трех преобразований к кнопке.



Рис. 4.14. *Кнопка, подвергнутая «пыткам» преобразований поворота, масштабирования и наклона*

Для повышения производительности WPF сначала вычисляет комбинированное преобразование на основе потомков объекта `TransformGroup`, а затем применяет одно результирующее преобразование (как если бы с самого начала применялось преобразование `MatrixTransform`). Отметим, что в составе группы `TransformGroup` может несколько раз встречаться одно и то же преобразование. Например, два поворота `RotateTransform` на 45°, эквивалентные одному повороту на 90°.

Резюме

На этом мы завершаем обзор свойств компоновки, с помощью которых дочерние элементы могут повлиять на способ своего размещения на экране. В этой главе вы также получили представление о таких вещах, которых не было ни в Win32, ни в Windows Forms: повернутых и наклоненных элементах управления!

Но самой важной частью механизма компоновки являются родительские панели. В этой главе мы для простоты пользовались только панелью `StackPanel`, а в следующей формально изучим как ее, так и все остальные панели.

ПРЕДУПРЕЖДЕНИЕ

Не все элементы типа FrameworkElement поддерживают преобразования!

Элементы, содержимое которых не является «родным» для WPF, не поддерживают преобразования, хотя и наследуют свойства `LayoutTransform` и `RenderTransform`. Например, к их числу относится элемент `HwndHost`, выступающий в роли владельца GDI-содержимого (обсуждается в главе 19 «Интероперабельность с другими технологиями»). Элемент управления `Frame`, который, в принципе, может содержать HTML-разметку (описывается в главе 9 «Однодетные элементы управления»), поддерживает преобразования в полном объеме, только если в нем нет HTML. В противном случае преобразование `ScaleTransform` можно применять для изменения размера, но содержимое при этом не масштабируется.

На рис. 4.15 показано, что происходит, когда на панели `StackPanel` размещено несколько кнопок и фрейм `Frame`, содержащий веб-страницу (с ограничением размера 100×100). Когда вся панель поворачивается и масштабируется, фрейм честно пытается выполнить масштабирование, но не поворачивается. В результате большая часть повернутых кнопок оказывается перекрыта.



Панель `StackPanel` до преобразования



`StackPanel` после масштабирования и поворота

Рис. 4.15. *Элемент Frame, содержащий HTML, частично реагирует на преобразование ScaleTransform, но игнорирует все остальные преобразования*

- Панель Canvas
- Панель StackPanel
- Панель WrapPanel
- Панель DockPanel
- Панель Grid
- Примитивные панели
- Обработка переполнения содержимого
- Все вместе: создание сворачиваемой, стыкуемой, изменяющей размер панели

5

Компоновка с помощью панелей

Компоновка – важнейший фактор, обеспечивающий удобство работы с приложением на различных устройствах, но без качественной поддержки со стороны платформы реализовать этот механизм исключительно трудно. Можно, конечно, размещать элементы пользовательского интерфейса статически, задавая координаты и размеры в пикселах, и где-то это даже будет работать, однако такое решение «посыпется» под воздействием самых разных факторов: разная разрешающая способность и линейные размеры экранов, заданные пользователем настройки, к примеру размер шрифта, содержимое, изменяющееся непредсказуемым образом (например, после перевода текста на другие языки). Добавим еще, что приложение, не позволяющее пользователю изменять размеры своих частей (и разумно распоряжаться имеющимся местом), многих раздражает.

Мой нетбук оснащен экраном с разрешающей способностью 1024×600. Outlook 2010 адаптируется к нему отлично, но многие программы, в том числе Visual Studio 2010, испытывают трудности. Если я переведу экран в портретный режим (600×1024), то Outlook 2010 прекрасно распорядится имеющимся местом, тогда как другие программы (та же Visual Studio 2010) справляются с этим куда хуже. (Ирония этой ситуации в том, что Visual Studio, по крайней мере частично, написана на WPF, а Outlook вообще не использует WPF. Впрочем, описанное явление – результат не столько применения тех или иных технологий, сколько недостатка внимания, которое разработчики уделяли адаптации к экранам небольшого или необычного размера.)

WPF содержит встроенные панели, которые помогают избежать неприятностей с компоновкой. В начале этой главы мы рассмотрим пять основных встроенных панелей (все в пространстве имен `System.Windows.Controls`) в порядке возрастания сложности (и полезности):

- `Canvas`
- `StackPanel`

- `WrapPanel`
- `DockPanel`
- `Grid`

Для полноты картины в этой главе рассматриваются также некоторые редко используемые «примитивные панели». Затем мы поговорим о переполнении содержимого (это происходит, когда родительская панель не может договориться со своими дочерними элементами об использовании имеющегося пространства) и закончим главу большим и содержательным примером. В нем мы применим различные способы компоновки, чтобы получить довольно развитый пользовательский интерфейс, аналогичный имеющимся в таких программах, как Visual Studio. Без средств компоновки, предоставляемых WPF, решить эту задачу было бы трудно.

Панель Canvas

`Canvas` (холст) – самая простая панель. Настолько простая, что использовать ее для организации пользовательского интерфейса вообще не стоит. `Canvas` поддерживает только «классическое» позиционирование элементов путем явного задания координат; впрочем, координаты хотя бы задаются в независимых от устройства пикселах, в отличие от прежних систем конструирования пользовательских интерфейсов. Панель `Canvas` позволяет задавать координаты относительно *любого*, а не только левого верхнего угла.

Позиционирование элемента на холсте осуществляется с помощью присоединенных свойств: `Left`, `Top`, `Right` и `Bottom`. Задавая значение `Left` или `Right`, вы определяете, что ближайшая сторона элемента должна всегда отстоять на фиксированное расстояние от соответствующей стороны холста. То же самое относится к свойствам `Top` и `Bottom`. По сути дела, вы указываете угол, к которому «примыкает» каждый элемент, а значения присоединенных свойств выступают в роли полей (к которым добавляются значения самого свойства `Margin` элемента). Если для некоторого элемента не задано ни одно присоединенное свойство (то есть все они имеют значение по умолчанию `Double.NaN`), то он помещается в левый верхний угол (что эквивалентно установке для `Left` и `Top` значения `0`). Использование панели `Canvas` демонстрируется в листинге 5.1, а результат показан на рис. 5.1.

Листинг 5.1. Расположение кнопок на панели Canvas

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Title="Buttons in a Canvas">
  <Canvas>
    <Button Background="Red">Left=0, Top=0</Button>
    <Button Canvas.Left="18" Canvas.Top="18"
            Background="Orange">Left=18, Top=18</Button>
    <Button Canvas.Right="18" Canvas.Bottom="18"
            Background="Yellow">Right=18, Bottom=18</Button>
    <Button Canvas.Right="0" Canvas.Bottom="0"
            Background="Lime">Right=0, Bottom=0</Button>
```

```
    <Button Canvas.Right="0" Canvas.Top="0"
            Background="Aqua">Right=0, Top=0</Button>
    <Button Canvas.Left="0" Canvas.Bottom="0"
            Background="Magenta">Left=0, Bottom=0</Button>
  </Canvas>
</Window>
```



Рис. 5.1. *Кнопки на панели Canvas из листинга 5.1*

ПРЕДУПРЕЖДЕНИЕ

Для элемента нельзя задавать более двух присоединенных свойств Canvas!

При попытке одновременно установить свойства `Canvas.Left` и `Canvas.Right` последнее будет проигнорировано. А при попытке одновременно установить свойства `Canvas.Top` и `Canvas.Bottom` будет проигнорировано `Canvas.Bottom`. Таким образом, невозможно пристыковать элемент более чем к одному углу холста.

В табл. 5.1 показано, как некоторые из обсуждавшихся в предыдущей главе свойств компоновки дочерних элементов применяются к элементам, расположенным на панели `Canvas`.

Таблица 5.1. Взаимодействие Canvas со свойствами компоновки дочерних элементов

Свойство	Допустимо ли внутри Canvas
`Margin`	Частично. Для двух сторон, использованных для позиционирования элемента (по умолчанию `Top` и `Left`), к значениям присоединенных свойств прибавляются соответствующие значения двух из четырех полей
`HorizontalAlignment` и `VerticalAlignment`	Нет. Элементам назначается в точности та величина вертикального выравнивания, которая им необходима
`LayoutTransform`	Да. Отличается от `RenderTransform` тем, что при использовании `LayoutTransform` элементы всегда отстоят на заданное расстояние от выбранного угла `Canvas`

СОВЕТ

Z-порядок по умолчанию (задающий, какие элементы располагаются «поверх» других) определяется порядком добавления дочерних элементов к родителю. В XAML это определяется порядком следования дочерних элементов в файле. Элементы, добавленные позже, располагаются поверх элементов, добавленных раньше. Так, на рис. 5.1 оранжевая кнопка располагается поверх красной, а зеленая – поверх желтой. Это проявляется не только для встроенных панелей, допускающих перекрытие элементов (в частности, `Canvas`), но и в случае, когда к перекрытию приводит применение преобразования `RenderTransform` (как на рис. 4.7, 4.8, 4.11, 4.12 и 4.13 в предыдущей главе).

Однако *Z*-порядок любого элемента можно задать явно, указав для него присоединенное свойство `ZIndex`, определенное в классе `Panel` (и наследуемое всеми панелями). `ZIndex` – это целое число, по умолчанию равное `0`; оно может принимать любое целое значение (положительное или отрицательное). Элементы с большим значением `ZIndex` рисуются поверх элементов с меньшим значением, то есть элемент с наименьшим значением `ZIndex` оказывается позади всех остальных, а элемент с наибольшим значением – впереди. В следующем примере свойство `ZIndex` задано так, что красная кнопка располагается поверх оранжевой несмотря на то, что в списке дочерних элементов `Canvas` она встречается раньше:

```
<Canvas>
  <Button Canvas.ZIndex="1" Background="Red">On Top!</Button>
  <Button Background="Orange">On Bottom with a Default ZIndex=0</Button>
</Canvas>
```

Если для нескольких элементов задано одно и то же значение `ZIndex`, то их взаимное расположение определяется порядком следования в коллекции `Children` панели, как в случае по умолчанию.

Таким образом, для манипулирования *Z*-порядком из программы достаточно изменить значение `ZIndex`. Чтобы поместить красную кнопку под оранжевую, можно присвоить присоединенному свойству любое значение, меньшее или равное нулю. В программе на C# это делается следующим образом (в предположении, что красная кнопка называется `redButton`):

```
Panel.SetZIndex(redButton, 0);
```

Хотя панель `Canvas` слишком примитивная для создания гибких пользовательских интерфейсов, она работает быстрее всех других панелей. Имейте это в виду, когда захотите максимально точно контролировать размещение элементов и при этом добиться наивысшей производительности. Например, `Canvas` очень удобна для точного позиционирования примитивных фигур в векторных рисунках (см. главу 15 «Двумерная графика»).

Панель StackPanel

Панель StackPanel популярна из-за своей простоты и удобства. Как следует из названия, она последовательно размещает своих потомков в виде стопки. В примерах из предыдущей главы мы пользовались панелью StackPanel, потому что она не требует задавать присоединенные свойства для получения приемлемого пользовательского интерфейса. На самом деле StackPanel – одна из немногих панелей, в которых вообще не определены собственные присоединенные свойства!

КОПНЕМ ГЛУБЖЕ

StackPanel и размещение справа налево

Если свойство FlowDirection равно RightToLeft, то для панели StackPanel с горизонтальной ориентацией сборка стопки производится справа налево, а не слева направо, как в случае по умолчанию.

В отсутствие присоединенных свойств единственный способ организовать дочерние элементы – воспользоваться свойством панели Orientation (типа System.Windows.Controls.Orientation), которое может принимать значение Horizontal или Vertical. По умолчанию подразумевается ориентация Vertical. На рис. 5.2 показано несколько кнопок, для которых заданы только свойства Background и Content, скомпонованные с помощью двух панелей StackPanel с разной ориентацией.



Рис. 5.2. Кнопки на панели StackPanel с разной ориентацией

В табл. 5.2 показано, как некоторые из обсуждавшихся в предыдущей главе свойств компоновки дочерних элементов применяются к элементам, расположенным на панели StackPanel.

Таблица 5.2. Взаимодействие StackPanel со свойствами компоновки дочерних элементов

Свойство	Допустимо ли внутри StackPanel
Margin	Да. Свойство Margin управляет промежутком между элементом и краями StackPanel, а также промежутком между краями соседних элементов
HorizontalAlignment и VerticalAlignment	Частично, поскольку выравнивание игнорируется в направлении сборки стопки (так как дочерним элементам отводится ровно столько места, сколько им необходимо). Если Orientation="Vertical", то игнорируется значение VerticalAlignment. Если Orientation="Horizontal", то игнорируется значение HorizontalAlignment
LayoutTransform	Да. Отличается от RenderTransform тем, что при использовании LayoutTransform оставшиеся в стопке элементы сдвигаются вниз, чтобы освободить место. При комбинировании компоновки Stretch с преобразованием RotateTransform или SkewTransform, применяемым в режиме LayoutTransform, растяжение происходит, только если угол кратен 90°

Последняя фраза в комментариях к свойству LayoutTransform в табл. 5.2 нуждается в пояснении. На рис. 5.3 видно, что при повороте элемента, который в нормальных условиях был бы растянут, растяжение производится лишь в случае, когда края элемента параллельны или перпендикулярны направлению растяжения. Это поведение не является особенностью StackPanel, а присутствует всюду, где элемент растягивается только в одном направлении. Эта странность проявляется, только когда преобразование применяется в режиме LayoutTransform; к RenderTransform она не относится.

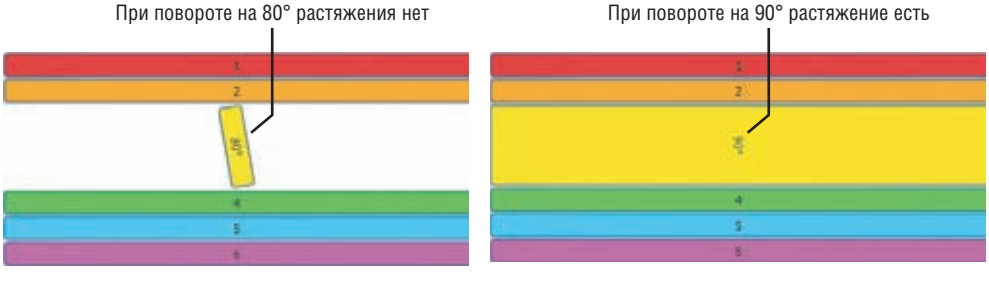


***Рис. 5.3.** Желтая кнопка повернута сначала на 80°, а затем на 90° с применением преобразования LayoutTransform*

КОПНЕМ ГЛУБЖЕ

Виртуализирующие панели

Важной деталью реализации нескольких элементов управления являются панели, производные от абстрактного класса `System.Windows.Controls.VirtualizingPanel`. Наиболее интересная из них – панель `VirtualizingStackPanel`, которая работает, как `StackPanel`, но для повышения производительности временно игнорирует все элементы, не видные на экране (только во время привязки к данным). Поэтому `VirtualizingStackPanel` является оптимальной панелью, когда требуется привязать к данным *по-настоящему* много дочерних элементов, и класс `ListBox` использует ее по умолчанию. Эту панель можно использовать также в `TreeView`, о чем пойдет речь в главе 10 «Многодетные элементы управления». Еще две виртуализирующие панели – `DataGridCellsPanel` и `DataGridRowsPresenter`, они используются в классе `DataGrid` и ассоциированных с ним (см. главу 11 «Изображения, текст и другие элементы управления»).

Панель WrapPanel

Панель `WrapPanel` похожа на `StackPanel`. Но помимо организации дочерних элементов в стопку она создает новые строки или столбцы, когда для одной стопки не хватает места. Это полезно для отображения заранее неизвестного числа элементов, когда компоновка должна отличаться от простого списка, – как, например, в Проводнике Windows.

Как и `StackPanel`, панель `WrapPanel` не обладает присоединенными свойствами для управления положением элементов. В классе `WrapPanel` определены три свойства, контролирующие его поведение:

- `Orientation` – аналогично одноименному свойству `StackPanel` с тем отличием, что по умолчанию подразумевается значение `Horizontal`. Панель с горизонтальной ориентацией выглядит, как вид Эскизы страниц в Проводнике Windows: элементы располагаются один за другим слева направо, а когда место кончается, переходят на следующую строку. Панель с вертикальной ориентацией выглядит, как вид Список в Проводнике Windows: элементы располагаются один под другим, а когда место кончается, начинается новый столбец.
- `ItemHeight` – единая высота для всех дочерних элементов. Каким образом каждый потомок распоряжается этой высотой, зависит от значений его свойств `VerticalAlignment`, `Height` и пр. Элементы, ширина которых превышает `ItemHeight`, отсекаются.
- `ItemWidth` – единая ширина для всех дочерних элементов. Каким образом каждый потомок распоряжается этой шириной, зависит от значений его свойств `HorizontalAlignment`, `Width` и пр. Элементы, высота которых превышает `ItemWidth`, отсекаются.

По умолчанию свойства `ItemHeight` и `ItemWidth` не установлены (точнее, имеют значение `Double.NaN`). В этом случае панель `WrapPanel` с вертикальной ориента-

цией отводит каждому столбцу ширину, равную ширине самого широкого элемента в нем, а панель с горизонтальной ориентацией отводит каждой строке высоту, равную высоте самого высокого элемента в ней. Поэтому по умолчанию ни в строках, ни в столбцах отсечение не производится.

СОВЕТ

Можно заставить панель `WrapPanel` располагать элементы в одну строку или в один столбец. Для этого следует присвоить свойству `Width` (в случае горизонтальной ориентации) или свойству `Height` (в случае вертикальной ориентации) значение `Double.MaxValue` либо `Double.PositiveInfinity`. В XAML это достигается с помощью расширения разметки `x:Static`, поскольку ни то ни другое значение не поддерживается конвертером типа `System.Double`.

На рис. 5.4 показано четыре динамических снимка экрана для панели `WrapPanel` с горизонтальной ориентацией, полученных в результате изменения размера объемлющего ее окна `Window`. На рис. 5.5 то же самое показано для панели с вертикальной ориентацией. Если для панели `WrapPanel` отведено достаточно места и свойства `ItemHeight/ItemWidth` не установлены, то она ведет себя, как `StackPanel`.

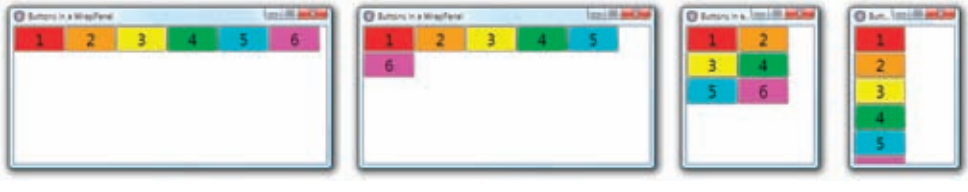


Рис. 5.4. *Изменение положения кнопок на панели WrapPanel с принимаемой по умолчанию горизонтальной ориентацией по мере сужения окна*



Рис. 5.5. *Изменение положения кнопок на панели WrapPanel с вертикальной ориентацией по мере сужения окна*

В табл. 5.3 показано, как некоторые из свойств компоновки дочерних элементов применяются к элементам, расположенным на панели `WrapPanel`.

КОПНЕМ ГЛУБЖЕ

WrapPanel и размещение справа налево

Если свойство `FlowDirection` равно `RightToLeft`, то для панели `WrapPanel` с вертикальной ориентацией новый столбец создается слева от заполненного, а для панели с горизонтальной ориентацией заполнение строки производится справа налево.

Таблица 5.3. Взаимодействие WrapPanel со свойствами компоновки дочерних элементов

Свойство	Допустимо ли внутри WrapPanel
`Margin`	Да. Поля учитываются, когда `WrapPanel` вычисляет размеры элементов, чтобы определить подразумеваемую по умолчанию ширину или высоту стопки
`HorizontalAlignment` и `VerticalAlignment`	Частично. Выравнивание можно задавать в направлении, противоположном направлению роста стопки, как и в случае `StackPanel`. Но выравнивание может быть полезно и в направлении роста стопки, если значение `ItemHeight` или `ItemWidth` таково, что в элементе имеется дополнительное пространство для выравнивания
`LayoutTransform`	Да. Отличается от `RenderTransform` тем, что при использовании `LayoutTransform` оставшиеся элементы сдвигаются, чтобы освободить место, но только если не установлено свойство `ItemHeight` или `ItemWidth` (в зависимости от ориентации). При комбинировании компоновки `Stretch` с преобразованием `RotateTransform` или `SkewTransform`, применяемым в режиме `LayoutTransform`, растяжение происходит, только если угол кратен 90°, как и в случае `StackPanel`

Панель `WrapPanel` обычно используется не для компоновки элементов управления внутри окна `Window`, а для компоновки внутри *вложенных* элементов управления. Как это делается, объяснено в главе 10.

Панель DockPanel

Панель `DockPanel` дает простой способ пристыковки элемента к одной из сторон, растягивая его на всю имеющуюся ширину или высоту. (Отличие от `Canvas` заключается в том, что элементы пристыковываются не к одному углу, а ко всей стороне.) Кроме того, `DockPanel` позволяет расположить один элемент, так чтобы он занял все место, свободное от пристыкованных элементов.

В классе `DockPanel` определено присоединенное свойство `Dock` (типа `System.Windows.Controls.Dock`), с помощью которого дочерние элементы могут управлять своим положением. Оно может принимать четыре значения: `Left` (подразуме-

вается по умолчанию, если свойство Dock не задано явно), Top, Right и Bottom. Отметим, что у свойства Dock нет значения Fill, означающего, что нужно заполнить оставшееся место. Вместо этого действует соглашение о том, что все оставшееся место отдается последнему дочернему элементу, добавленному в DockPanel, если только свойство LastChildFill не равно false. Если LastChildFill равно true (по умолчанию), то значение свойства Dock, заданное для последнего добавленного элемента, игнорируется. Если же оно равно false, то последний элемент можно пристыковать к любой стороне (по умолчанию к левой, Left).

На рис. 5.6 показано пять кнопок на панели DockPanel (LastChildFill оставлено равным true), скомпонованных следующим образом:

```
<DockPanel>
  <Button DockPanel.Dock="Top" Background="Red">1 (Top)</Button>
  <Button DockPanel.Dock="Left" Background="Orange">2 (Left)</Button>
  <Button DockPanel.Dock="Right" Background="Yellow">3 (Right)</Button>
  <Button DockPanel.Dock="Bottom" Background="Lime">4 (Bottom)</Button>
  <Button Background="Aqua">5</Button>
</DockPanel>
```

Порядок добавления кнопок на панель обозначен числом (и цветом).

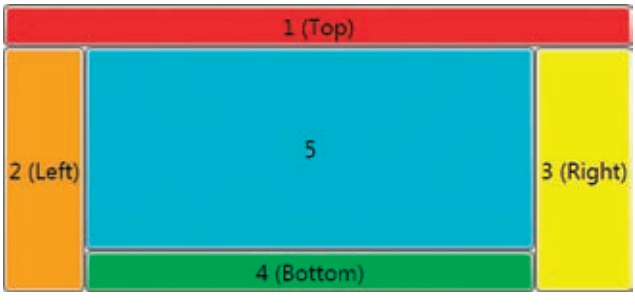


***Рис. 5.6.** Кнопки на панели DockPanel*

Как и в случае StackPanel, растяжение элементов определяется подразумеваемым по умолчанию значением свойства HorizontalAlignment или VerticalAlignment. Если элемент не будет занимать все пространство, выделенное ему панелью DockPanel, то можно задать другое выравнивание. На рис. 5.7 показано, как будет выглядеть панель, когда у всех кнопок, кроме одной, явно задано значение HorizontalAlignment или VerticalAlignment:

```
<DockPanel>
  <Button DockPanel.Dock="Top" HorizontalAlignment="Right"
    Background="Red">1 (Top, Align=Right)</Button>
  <Button DockPanel.Dock="Left" VerticalAlignment="Bottom"
    Background="Orange">2 (Left, Align=Bottom)</Button>
  <Button DockPanel.Dock="Right" VerticalAlignment="Bottom"
    Background="Yellow">3 (Right, Align=Bottom)</Button>
```

```
    <Button DockPanel.Dock="Bottom" HorizontalAlignment="Right"
      Background="Lime">4 (Bottom, Align=Right)</Button>
    <Button Background="Aqua">5</Button>
  </DockPanel>
```

Отметим, что, хотя четыре элемента отказались занимать все выделенное им место, нераспределенное пространство не отдается другим элементам.



Рис. 5.7. *Кнопки на панели DockPanel не занимают все выделенное им место*

Панель `DockPanel` полезна для организации верхнего уровня интерфейса внутри элемента `Window` или `Page`, когда пристыкованные элементы по большей части представляют собой другие панели, где и находится все самое важное. Так, обычно к верхней стороне пристыковывается меню (`Menu`), справа и слева находятся какие-то панели, а снизу – строка состояния (`StatusBar`). Центральную же часть занимают основные данные приложения.

Порядок добавления дочерних элементов на панель имеет значение, потому что каждому потомку выделяется все оставшееся место на стороне, к которой он пристыковывается. (Можно провести аналогию с эгоистом, который, первым сев в кресло в самолете или в аудитории, занимает оба подлокотника.)

На рис. 5.8 показаны те же пять кнопок, что и на рис. 5.6, но добавленные в другом порядке (обозначенном числом и цветом). Обратите внимание, что компоновка изменилась.

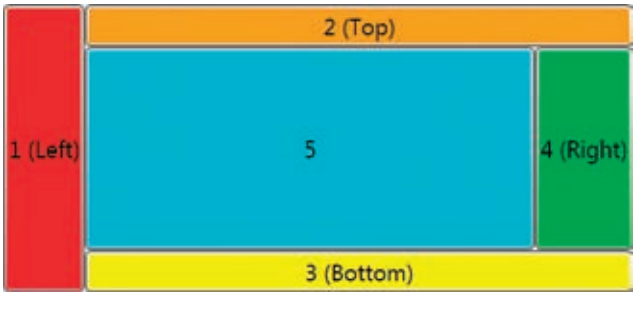


Рис. 5.8. *Кнопки на панели DockPanel, скомпонованные иначе, чем на рис. 5.6*

DockPanel поддерживает произвольное количество дочерних элементов, а не только пять. Если к одной стороне пристыковано несколько элементов, то они просто организуются в стопку соответствующего направления. На рис. 5.9 показана панель DockPanel с восьмью дочерними элементами – три слева, два сверху, два снизу и один заполняет оставшееся пространство.



Рис. 5.9. К каждой стороне пристыковано несколько элементов

Таким образом, DockPanel является обобщением StackPanel. Если свойство LastChildFill равно false, то DockPanel ведет себя, как горизонтальная панель StackPanel, когда все потомки пристыковываются к левой стороне, и как вертикальная – когда все потомки пристыковываются к верхней стороне.

В табл. 5.4 показано, как некоторые из свойств компоновки дочерних элементов применяются к элементам, расположенным на панели DockPanel.

Таблица 5.4. Взаимодействие DockPanel со свойствами компоновки дочерних элементов

Свойство	Допустимо ли внутри DockPanel
Margin	Да. Свойство Margin определяет, сколько места оставлять между элементом и стороной панели, а также промежуток между самими элементами
HorizontalAlignment и VerticalAlignment	Частично. Как и в случае StackPanel, выравнивание в направлении пристыковки игнорируется. Иначе говоря, если Dock равно Left или Right, то не имеет смысла задавать свойство HorizontalAlignment, а если Top или Bottom – то свойство VerticalAlignment. Однако для элемента, заполняющего оставшееся пространство, применимы оба свойства, HorizontalAlignment и VerticalAlignment
LayoutTransform	Да. Отличается от RenderTransform тем, что при использовании LayoutTransform оставшиеся элементы сдвигаются, чтобы освободить место. При комбинировании компоновки Stretch с преобразованием RotateTransform или SkewTransform, применяемым в режиме LayoutTransform, растяжение происходит, только если угол кратен 90°, за исключением элемента, заполняющего оставшееся пространство (потому что он может растягиваться в обоих направлениях)

Панель Grid

Grid (сетка) – самая гибкая из всех панелей и, пожалуй, наиболее употребительная. (В проектах Visual Studio и Expression Blend панель Grid используется по умолчанию.) Она позволяет расположить дочерние элементы в несколько строк и несколько столбцов, не полагаясь на режим автоматического переноса (как в WrapPanel). Кроме того, предоставляется ряд интересных способов управления строками и столбцами. Работа с панелью Grid очень напоминает использование элемента TABLE в HTML.

СОВЕТ

В WPF также имеется класс Table (в пространстве имен System.Windows.Documents), предоставляющий примерно такие же возможности, как Grid. Но Table не наследует классу Panel (и даже классу UIElement). Этот класс, производный от FrameworkContentElement, предназначен для отображения содержимого документов и рассматривается в главе 11.

Мы больше не будем демонстрировать разные способы компоновки на примере раскрашенных кнопок, а с помощью панели Grid сконструируем интерфейс, напоминающий начальную страницу старых версий Visual Studio. Код в листинге 5.2 определяет сетку размером 4×2, в ячейках которой находятся метка Label и четыре элемента GroupBox.

Листинг 5.2. Первая попытка построить интерфейс, подобный начальной странице Visual Studio

```
<Grid Background="LightBlue">

<!-- Определяем четыре строки: -->
<Grid.RowDefinitions>
  <RowDefinition/>
  <RowDefinition/>
  <RowDefinition/>
  <RowDefinition/>
</Grid.RowDefinitions>

<!-- Определяем два столбца: -->
<Grid.ColumnDefinitions>
  <ColumnDefinition/>
  <ColumnDefinition/>
</Grid.ColumnDefinitions>

<!-- Размещаем дочерние элементы: -->
  <Label    Grid.Row="0" Grid.Column="0" Background="Blue"Foreground="White"
            HorizontalContentAlignment="Center">Start Page</Label>
  <GroupBox Grid.Row="1" Grid.Column="0" Background="White"
            Header="Recent Projects">...</GroupBox>
```

```
    <GroupBox Grid.Row="2" Grid.Column="0" Background="White"
              Header="Getting Started">...</GroupBox>
    <GroupBox Grid.Row="3" Grid.Column="0" Background="White"
              Header="Headlines">...</GroupBox>
    <GroupBox Grid.Row="1" Grid.Column="1" Background="White"
              Header="Online Articles">
      <ListBox>
        <ListBoxItem>Article #1</ListBoxItem>
        <ListBoxItem>Article #2</ListBoxItem>
        <ListBoxItem>Article #3</ListBoxItem>
        <ListBoxItem>Article #4</ListBoxItem>
      </ListBox>
    </GroupBox>
  </Grid>
```

В простейшем случае мы определяем количество строк и столбцов, помещая нужное число элементов `RowDefinition` и `ColumnDefinition` внутрь элементов, соответствующих свойствам сетки `RowDefinitions` и `ColumnDefinitions`. (Немного длинно, но удобно, когда для отдельных строк и столбцов нужно задать разные размеры.) Затем позиционируем дочерние элементы с помощью присоединенных свойств `Row` и `Column`, принимающих целочисленные значения, начиная с 0. Если явно не описать строки и столбцы, то по умолчанию сетка будет состоять всего из одной ячейки. А если не указать для дочерних элементов свойства `Grid.Row` и `Grid.Column`, то по умолчанию подразумевается значение `0`.

Ячейки сетки можно оставлять пустыми, и, наоборот, в одной ячейке может находиться несколько элементов. В таком случае они просто располагаются один над другим в соответствии с заданным *Z*-порядком. Как и в случае панели `Canvas`, дочерние элементы, находящиеся в одной ячейке, не взаимодействуют между собой, а просто перекрывают друг друга.

На рис. 5.10 показан результат визуализации кода в листинге 5.2.

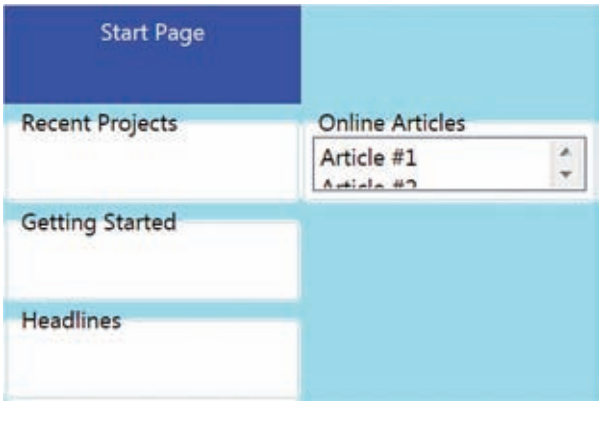


Рис. 5.10. *Первая попытка имитировать начальную страницу Visual Studio – не слишком удачная*

Сразу бросается в глаза недостаток: список онлайновых статей (Online articles) слишком мал. Кроме того, было бы лучше, если бы заголовок Start Page занимал всю ширину сетки. К счастью, обе проблемы легко решить с помощью дополнительных присоединенных свойств класса `Grid`: `RowSpan` и `ColumnSpan`.

По умолчанию свойства `RowSpan` и `ColumnSpan` равны `1`, но могут принимать любое значение, большее или равное `1`, — соответственно количество строк или столбцов, занимаемых данной ячейкой. (Если значение больше общего количества строк или столбцов, то ячейка занимает столько строк или столбцов, сколько возможно.) Таким образом, если добавить в последний элемент `GroupBox` в листинге 5.2 атрибут

```
Grid.RowSpan="3"
```

а в элемент `Label` — атрибут

```
Grid.ColumnSpan="2"
```

то получится куда более симпатичный результат, показанный на рис. 5.11.

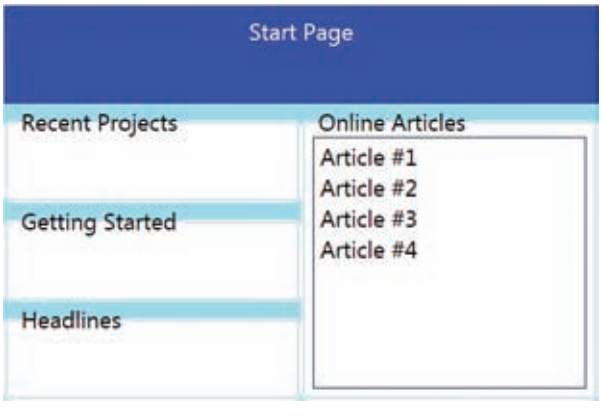


Рис. 5.11. *Использование свойств RowSpan и ColumnSpan позволяет точнее имитировать начальную страницу Visual Studio*

Сетка на рис. 5.11 все еще выглядит не вполне удовлетворительно, потому что по умолчанию высоты всех строк и ширины всех столбцов равны. В идеале нужно было бы оставить больше места для списка онлайновых статей, а заголовок вверху занимает слишком много пространства. Но это легко исправить, указав, что размеры первой строки и первого столбца должны соответствовать содержимому. Такой *автоматический выбор размера* достигается путем присваивания свойствам `Height` и `Width` соответственно в элементах `RowDefinition` и `ColumnDefinition` специального значения `Auto`, нечувствительного к регистру букв. После следующей модификации определений в листинге 5.2:

```
<!-- Определяем четыре строки: -->
<Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
```

```
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>

  <!-- Определяем два столбца: -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
```

получаем результат, показанный на рис. 5.12.

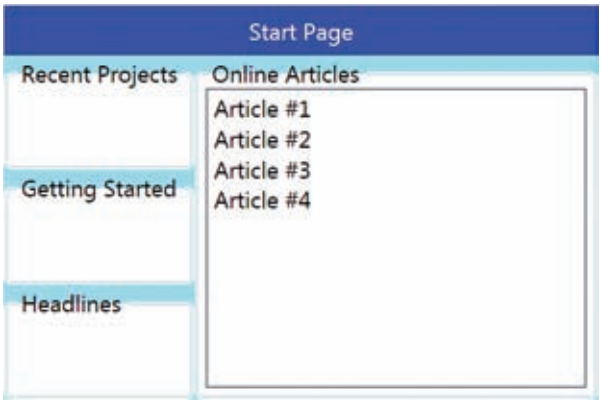


Рис. 5.12. *Окончательный вид страницы после автоматического выбора размера первой строки первого столбца*

FAQ

Как задать для ячеек сетки Grid цвет фона, отступ и рамку по аналогии с ячейками HTML-таблицы?

Не существует готового механизма, позволяющего задать эти свойства для ячейки сетки, но его можно легко смоделировать, воспользовавшись тем, что в одной ячейке может быть несколько элементов. Чтобы придать ячейке цвет фона, достаточно поместить в нее прямоугольник `Rectangle`, задав для него такое значение свойства `Fill`, при котором он займет всю ячейку. Чтобы получить отступ, можно задать режим автоматического выбора размера и установить поле `Margin` для соответствующего дочернего элемента. Чтобы получить рамку, мы снова воспользуемся прямоугольником, но явно зададим обводку (свойство `Stroke`) нужного цвета или поместим внутрь элемент `Border`.

Только не забудьте, что элементы `Rectangle` и `Border` нужно добавлять в сетку раньше всех остальных дочерних элементов (или явно задать для них присоединенное свойство `ZIndex`), чтобы *Z*-порядок обеспечил их размещение под основным содержимым.

СОВЕТ

У панели `Grid` имеется простое свойство `ShowGridLines`; если оно равно `true`, то ячейки будут разграничены желто-синими пунктирными линиями. В промышленном приложении это ни к чему, зато полезно для «отладки» компоновки. На рис. 5.13 показан результат установки свойства `ShowGridLines="True"` для сетки, изображенной на рис. 5.12.

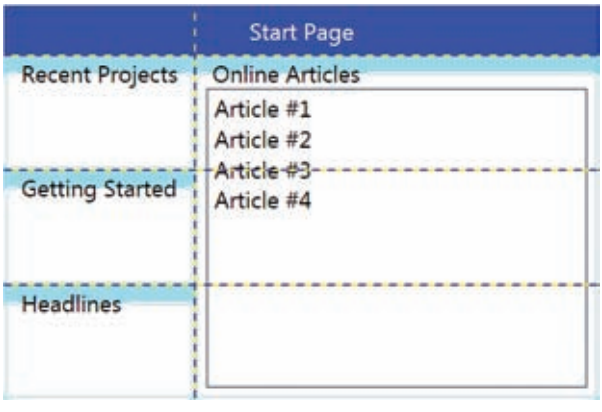


Рис. 5.13. *Результат задания свойства ShowGridLines для сетки Grid*

Задание размеров строк и столбцов

В отличие от элементов типа `FrameworkElement`, свойства `Height` и `Width` элементов `RowDefinition` и `ColumnDefinition` по умолчанию не равны `Auto` (или `Double.NaN`). И, в отличие от всех прочих свойств `Height` и `Width` в WPF, они имеют тип `System.Windows.GridLength`, а не `double`. Поэтому панель `Grid` поддерживает три способа задания размера в элементах `RowDefinition` и `ColumnDefinition`:

- **Абсолютный размер** – числовое значение `Height` или `Width` означает, что размер задан в независимых от устройства пикселах (как и все прочие свойства `Height` и `Width` в WPF). В отличие от других способов задания размера, абсолютные значения не позволяют строкам и столбцам увеличиваться или сжиматься при изменении размера самой сетки `Grid` или находящихся внутри нее элементов.
- **Автоматический выбор размера** – если `Height` или `Width` равно `Auto` (мы уже экспериментировали с этой установкой выше), то дочерним элементам выделяется столько места, сколько необходимо, но не больше (для свойств `Height` и `Width` во всех остальных классах WPF это режим по умолчанию). Для строки эта величина равна высоте самого высокого элемента, а для столбца – ширине самого широкого элемента. Когда речь идет о тексте, этот режим лучше задания абсолютных размеров, так как можно не опасаться отсечения из-за выбора другого шрифта или локализации.

- **Пропорциональное изменение размера** (иногда называется размером «звездочка») – предусмотрен специальный синтаксис задания свойств `Height` и `Width`, позволяющий распределить имеющееся пространство поровну или в соответствии с заданными пропорциями. Если задано пропорциональное изменение размера, строка и столбец увеличиваются или сжимаются при изменении размера сетки.

С абсолютным и автоматическим заданием размера все понятно, но вот пропорциональное измерение требует пояснений. Звездочка работает следующим образом:

- Если высота строки или ширина столбца равна `*`, то соответствующему структурному элементу выделяется все оставшееся место.
- Если размер `*` задан для нескольких строк или столбцов, то все оставшееся место делится между ними поровну.
- Перед символом `*` можно указывать коэффициент (например, `2*` или `5.5*`), тогда соответствующей строке или столбцу будет выделено пропорционально больше места, чем остальным строкам или столбцам, в размере которых присутствует символ `*`. Столбец шириной `2*` всегда в два раза шире столбца шириной `*` (это означает в точности то же самое, что `1*`) *в той же самой* сетке. Столбец шириной `5.5*` в два раза шире столбца шириной `2.75*` *в той же самой* сетке.

Под словами «оставшееся пространство» понимается высота или ширина сетки за вычетом всех строк или столбцов, для которых задан абсолютный размер или его автоматический выбор. На рис. 5.14 показано три разных случая задания размеров простых столбцов в сетке.

По умолчанию высота любой строки и ширина любого столбца сетки равны `*`. Именно поэтому на рис. 5.10 и 5.11 строки и столбцы распределены равномерно.

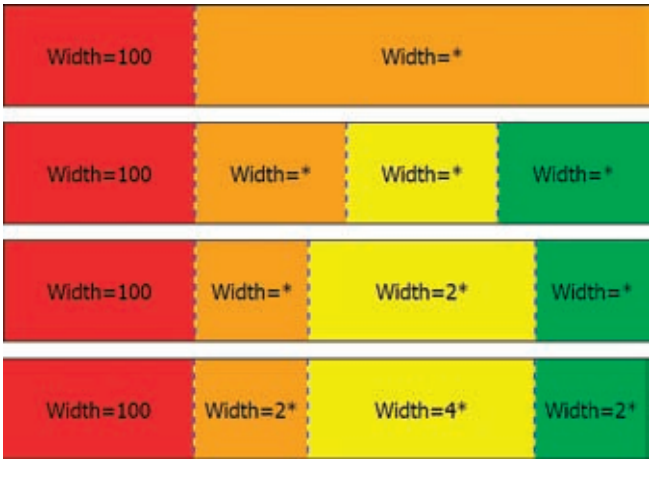


Рис. 5.14. *Пропорционально измеренные столбцы сетки*

FAQ

Почему в WPF не встроена поддержка процентного задания размеров, как в HTML?

Самый распространенный способ использования процентов в HTML – задание ширины или высоты элемента 100% – в большинстве панелей можно реализовать с помощью присваивания свойству `HorizontalAlignment` или `VerticalAlignment` значения `Stretch`. В более сложных случаях пропорциональное измерение в `Grid` эквивалентно заданию размера в процентах, только нужно привыкнуть к синтаксису. Например, если требуется, чтобы столбец занимал 25% ширины сетки, задайте для него размер `*` и проследите, чтобы совокупная ширина остальных столбцов составляла `3*`.

Разработчики WPF выбрали такой синтаксис, поскольку программисту не нужно будет следить за тем, чтобы сумма процентов оставалась равной 100 при динамическом добавлении или удалении столбцов. К тому же тот факт, что пропорциональное деление применяется к оставшемуся пространству (а не ко всей сетке), делает поведение более понятным, чем в случае HTML-таблицы с ее смесью пропорционального и абсолютного задания размеров строк и столбцов.

КОПНЕМ ГЛУБЖЕ

Использование свойства GridLength в процедурном коде

Конвертер типа `System.Windows.GridLengthConverter` преобразует строки вида `"100"`, `"auto"` или `"2*"` в структуры `GridLength`. В C# для создания объекта `GridLength` можно воспользоваться одним из двух конструкторов. Ключом является перечисление `GridUnitType`, в котором определены все три вида значений.

Для задания абсолютного размера можно воспользоваться конструктором, принимающим один аргумент типа `double` (например, `100`):

```
GridLength length = new GridLength(100);
```

или конструктором, принимающим дополнительный аргумент типа `GridUnitType`:

```
GridLength length = new GridLength(100, GridUnitType.Pixel);
```

В обоих случаях длина `100` выражается в независимых от устройства пикселах.

Конструкторы `GridLength` не поддерживают значение `Double.NaN`, поэтому для автоматического задания размера следует указывать значение `GridUnitType.Auto`:

```
GridLength length = new GridLength(0, GridUnitType.Auto);
```

Число, передаваемое в качестве первого параметра, игнорируется. Впрочем, рекомендуется просто пользоваться статическим свойством `GridLength.Auto`, которое возвращает готовый экземпляр `GridLength`, в точности такой, как созданный в показанной выше строке. Для задания пропорционального изменения размера можно передать число вместе со значением `GridUnitType.Star`:

```
GridLength length = new GridLength(2, GridUnitType.Star);
```

Этот код эквивалентен записи `2*` в XAML. Чтобы получить эквивалент `*`, нужно в качестве первого аргумента передать 1, а в качестве второго – `GridUnitType.Star`.

Интерактивное задание размера с помощью GridSplitter

Еще одна привлекательная особенность панели `Grid` – поддержка интерактивного изменения размера строк и столбцов мышью или клавишами (или стилусом, или пальцем – в зависимости от имеющегося оборудования). Достигается это с помощью класса `GridSplitter` в том же самом пространстве имен. В сетку `Grid` можно добавить произвольное число дочерних элементов `GridSplitter`, указав для них присоединенные свойства `Grid.Row`, `Grid.Column`, `Grid.RowSpan` и/или `Grid.ColumnSpan`, как для любых других потомков. Буксировка `GridSplitter` изменяет размер по меньшей мере одной ячейки. Что происходит с остальными – изменение размера или просто перемещение – зависит от заданного способа изменения размера: пропорционально или как-то иначе.

По умолчанию ячейки, на которых изменение размера отражается непосредственно, определяются свойствами выравнивания `GridSplitter`. В табл. 5.5 описано соответствующее поведение и цветом показано, как выглядит `GridSplitter` при различных параметрах (если представлять себе ячейки таблицы как ячейки сетки).

СОВЕТ

Хотя `GridSplitter` по умолчанию располагается в одной ячейке, его действие всегда распространяется на весь столбец (при буксировке по горизонтали) или на всю строку (при буксировке по вертикали). Поэтому лучше задавать для него свойство `ColumnSpan` или `RowSpan`, так чтобы он пересекал всю сетку.

Таблица 5.5. Ячейки, изменяемые непосредственно при буксировке GridSplitter с различным выравниванием

VerticalAlignment \ HorizontalAlignment	Left	Right	Center	Stretch
Top	Текущая ячейка и ячейка слева от нее	Текущая ячейка и ячейка справа от нее	Ячейки слева и справа	Текущая ячейка и ячейка сверху от нее
Bottom	Текущая ячейка и ячейка слева от нее	Текущая ячейка и ячейка справа от нее	Ячейки слева и справа	Текущая ячейка и ячейка снизу от нее
Center	Текущая ячейка и ячейка слева от нее	Текущая ячейка и ячейка справа от нее	Ячейки слева и справа	**Ячейки сверху и снизу**
Stretch	Текущая ячейка и ячейка слева от нее	Текущая ячейка и ячейка справа от нее	Ячейки слева и справа	**Ячейки слева и справа, если высота `GridSplitter` больше ширины, или ячейки сверху и снизу, если ширина `GridSplitter` больше высоты**

В классе GridSplitter свойство HorizontalAlignment по умолчанию равно Right, а свойство VerticalAlignment — Stretch, поэтому по умолчанию он примыкает к правой стороне указанной ячейки. Чтобы применение GridSplitter приносило нужный эффект, следует задавать режим выравнивания Stretch хотя бы в одном направлении. В противном случае может получиться маленькая точка, как видно в табл. 5.5.

Если для всех строк или столбцов задано пропорциональное изменение размера, то буксировка GridSplitter изменяет коэффициенты для двух строк или столбцов. Если же для всех строк или столбцов задан абсолютный размер, то буксировка изменяет только размер верхней или нижней из двух соседних ячеек (в зависимости от направления буксировки). Оставшиеся ячейки смещаются вниз или вправо, освобождая место. Буксировка происходит точно так же, когда строки или столбцы изменяются автоматически, но для той строки или столбца, размер которых меняется, устанавливаются абсолютные значения.

Все аспекты изменения размера можно, конечно, задать с помощью свойств выравнивания GridSplitter, однако в этом классе есть еще два свойства, позволяющих управлять поведением явно и независимо: ResizeDirection (типа GridResizeDirection) и ResizeBehavior (типа GridResizeBehavior). По умолчанию ResizeDirection (направление изменения размера) равно Auto, но может также принимать значение Rows или Columns, правда, они принимаются во внимание, лишь если GridSplitter растягивается в обоих направлениях (правая нижняя ячейка в табл. 5.5). ResizeBehavior (поведение при изменении размера) по умолчанию равно BasedOnAlignment, при этом обеспечивается поведение, описанное в табл. 5.5. Но возможны также значения PreviousAndCurrent, CurrentAndNext или PreviousAndNext, которые управляют тем, на какие две строки или столбца изменение размера воздействует непосредственно.

СОВЕТ

Лучше всего поместить GridSplitter в отдельную строку или столбец с автоматическим выбором размера. В таком случае он не будет перекрывать содержимое соседних ячеек. Если вы все же решите поместить GridSplitter в одну ячейку с другими элементами, то хотя бы добавляйте его последним (или задавайте свойство ZIndex), чтобы *Z*-порядок для него был наибольшим!

Задание общего размера для строк и столбцов

В классах RowDefinitions и ColumnDefinitions имеется свойство SharedSizeGroup, позволяющее задать режим, при котором линейные размеры нескольких строк и/или столбцов будут оставаться одинаковыми даже в случае, когда размер любой из них изменяется в процессе выполнения программы (например, с помощью GridSplitter). Свойству SharedSizeGroup можно присвоить произвольное строковое значение (чувствительное к регистру); оно интерпрети-

руется как имя группы. Размеры всех строк или столбцов, находящихся в одной группе, изменяются синхронно.

В качестве простого примера рассмотрим сетку с тремя столбцами, показанную на рис. 5.15; в ней свойство SharedSizeGroup не используется:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <Label Grid.Column="0" Background="Red"
         HorizontalContentAlignment="Center"
         VerticalContentAlignment="Center">1
  </Label>
  <GridSplitter Grid.Column="0" Width="5"/>
  <Label Grid.Column="1" Background="Orange"
         HorizontalContentAlignment="Center"
         VerticalContentAlignment="Center">2
  </Label>
  <Label Grid.Column="2" Background="Yellow"
         HorizontalContentAlignment="Center"
         VerticalContentAlignment="Center">3
  </Label>
</Grid>
```

Размер первого столбца выбирается автоматически, в нем находится метка Label и элемент GridSplitter. Остальные два столбца изменяются в размерах пропорционально и содержат только метку. Когда ширина первого столбца увеличивается, оставшиеся два делят поровну между собой уменьшившееся пространство.



Рис. 5.15. *Простая сетка без использования свойства SharedSizeGroup*

СОВЕТ

Чтобы элемент GridSplitter был виден и доступен для использования, его ширина Width (или высота Height – в зависимости от ориентации) должна быть задана явно.

На рис. 5.16 показано, что происходит с той же сеткой, когда для первого и последнего столбцов задано одинаковое значение SharedSizeGroup. Вначале для всех членов группы устанавливается наибольший автоматически изменяемый или абсолютный размер. После увеличения ширины первого столбца последний изменяется соответственно. Средний столбец теперь оказался единственным изменяемым пропорционально, поэтому он занимает все оставшееся место.

Ниже приведен XAML-код этой сетки.

```
<Grid IsSharedSizeScope="True">
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="Auto" SharedSizeGroup="myGroup"/>
  <ColumnDefinition/>
  <ColumnDefinition SharedSizeGroup="myGroup"/>
</Grid.ColumnDefinitions>

  <Label Grid.Column="0" Background="Red"
         HorizontalContentAlignment="Center"
         VerticalContentAlignment="Center">1
  </Label>
  <GridSplitter Grid.Column="0" Width="5"/>
  <Label Grid.Column="1" Background="Orange"
         HorizontalContentAlignment="Center"
         VerticalContentAlignment="Center">2
  </Label>
  <Label Grid.Column="2" Background="Yellow"
         HorizontalContentAlignment="Center"
         VerticalContentAlignment="Center">3
  </Label>
</Grid>
```



Компоновка по умолчанию

Компоновка после буксировки GridSplitter вправо

***Рис. 5.16.** Та же сетка, что на рис. 5.15, но для первого и последнего столбцов установлено свойство SharedSizeGroup*

Свойство IsSharedSizeScope следует установить потому, что группы размеров могут применяться сразу к *нескольким сеткам*! Чтобы избежать потенциального конфликта имен (и сократить расходы на необходимый в этом случае обход логических деревьев), все сетки, к которым применяется одно и то же значение свойства SharedSizeGroup, должны находиться под общим родителем, а свойство IsSharedSizeScope для них должно быть равно true. Это не просто

свойство зависимости в классе Grid, но еще и присоединенное свойство, которое можно задавать для родителей, не являющихся сетками, например:

```
<StackPanel Grid.IsSharedSizeScope="True">
  <Grid>...можно использовать SharedSizeGroup...</Grid>
  <Grid>...можно использовать SharedSizeGroup...</Grid>
  <WrapPanel>
    <Grid>...можно использовать SharedSizeGroup...</Grid>
  </WrapPanel>
</StackPanel>
```

В разделе «А теперь все вместе: создание сворачиваемой, стыкуемой, изменяющей размер панели, как в Visual Studio» в конце главы мы воспользуемся возможностью задавать SharedSizeGroup для нескольких сеток, чтобы создать удобный пользовательский интерфейс.

Сравнение Grid с другими панелями

Панель Grid – лучший выбор для особо сложной компоновки, потому что она умеет делать все, на что способны другие панели, а также многое другое. Единственное, чего ей не хватает, так это умения динамически генерировать новые строки и столбцы, как WrapPanel. С помощью сетки можно верстать такие макеты, для которых иначе потребовалось бы использовать несколько панелей. Например, начальную страницу, изображенную на рис. 5.12, можно было бы создать, комбинируя DockPanel и StackPanel. В этом случае DockPanel была бы внешним элементом, к ее верхней стороне была бы пристыкована метка Label, а к левой – панель StackPanel (и в ней находились бы первые три элемента GroupBox). Последний элемент GroupBox заполнил бы все оставшееся место.

Чтобы убедиться в том, что панель Grid действительно часто является оптимальным решением, посмотрим, как можно с ее помощью смоделировать поведение других панелей, памятуя о том, что в нашем распоряжении есть еще и дополнительные возможности Grid.

Моделирование Canvas с помощью Grid

Если взять сетку с одной строкой и одним столбцом и для всех дочерних элементом задавать любое значение HorizontalAlignment и VerticalAlignment, кроме Stretch, то добавляемые в единственную ячейку элементы будут вести себя так же, как при добавлении на панель Canvas. Задание HorizontalAlignment равным Left и VerticalAlignment равным Top эквивалентно установке для Canvas.Left и Canvas.Top значения 0. Задание HorizontalAlignment равным Right и VerticalAlignment равным Bottom эквивалентно установке для Canvas.Right и Canvas.Bottom значения 0. Кроме того, задание для каждого элемента поля Margin может дать такой же эффект, как присвоение таких же значений присоединенным свойствам Canvas. Именно так поступает конструктор Visual Studio, когда пользователь помещает или передвигает элементы на поверхности конструктора.

Моделирование StackPanel с помощью Grid

Сетка с одним столбцом и автоматически измеряемыми строками выглядит так же, как вертикальная стопка StackPanel, если каждый элемент вручную помещать в последовательно нумеруемые строки. Аналогично сетка с одной строкой и автоматически изменяемыми столбцами выглядит так же, как горизонтальная стопка, если каждый элемент вручную помещать в последовательно нумеруемые столбцы.

Моделирование DockPanel с помощью Grid

С помощью свойств RowSpan и ColumnSpan легко сделать так, чтобы внешние элементы пристыковывались к сторонам сетки и автоматически растягивались – как на панели DockPanel. На рис. 5.12 метка Label, по сути, пристыкована к верхней стороне.

В табл. 5.6 показано, как некоторые из свойств компоновки дочерних элементов применяются к элементам, расположенным на панели Grid.

Таблица 5.6. Взаимодействие Grid со свойствами компоновки дочерних элементов

Свойство	Допустимо ли внутри DockPanel
Margin	Да. Свойство Margin определяет, сколько места оставлять между элементом и сторонами объемлющей его ячейки
HorizontalAlignment и VerticalAlignment	Да. В отличие от остальных панелей, можно в полной мере использовать оба направления, если только не окажется, что ячейка с автоматическим изменением размеров вообще не оставила элементу дополнительного места. Поэтому по умолчанию большинство элементов растягиваются, заполняя свои ячейки целиком
LayoutTransform	Да. Отличается от RenderTransform тем, что при использовании LayoutTransform элементы остаются внутри ячеек (если это возможно) и учитывается величина поля Margin. В отличие от RenderTransform, элемент, вышедший в результате масштабирования за пределы ячейки, отсекается

СОВЕТ

Хотя складывается впечатление, что панель Grid может практически все, в случае, когда количество дочерних элементов заранее неизвестно, лучше все же использовать StackPanel или WrapPanel (обычно при компоновке дискретных элементов управления, описываемых в главе 10). Кроме того, DockPanel со сложными подпанелями иногда предпочтительнее Grid, потому что изоляция, которую обеспечивают подпанели, удобнее в ситуации, когда пользовательский интерфейс изменяется. Если для этой цели использовать единственную сетку, то при добавлении строк и столбцов для сохранения иллюзии стыковки придется изменять свойства RowSpan и ColumnSpan.

Примитивные панели

Рассмотренные выше панели в общем случае полезны для компоновки на уровне как приложения, так и отдельных элементов управления. Но в состав WPF входит также несколько простых панелей, более удобных внутри элементов управления, – все равно, идет ли речь о стилизации встроенного элемента (см. главу 14 «Стили, шаблоны, обложки и темы») или о создании нового нестандартного элемента (см. главу 20 «Пользовательские и нестандартные элементы управления»). Они не настолько универсальны, как предыдущие панели, но все же заслуживают беглого знакомства. Все эти панели определены в пространстве имен `System.Windows.Controls.Primitives`, за исключением `ToolBarTray`, которая находится в пространстве имен `System.Windows.Controls`.

Панель TabPanel

`TabPanel` очень похожа на `WrapPanel`, но обладает некоторыми ограничениями, с одной стороны, и дополнительными возможностями, с другой. Как следует из ее названия, эта панель используется в подразумеваемом по умолчанию стиле элемента `TabControl`, где служит для организации вкладок. В отличие от `WrapPanel`, она поддерживает только горизонтальную ориентацию стопки и вертикальное оборачивание. Когда происходит оборачивание, элементы равномерно растягиваются, так что все строки занимают всю ширину панели. Элемент управления `TabControl` рассматривается в главе 10.

Панель ToolBarPanel

Панель `ToolBarPanel`, по умолчанию используемая в стиле элемента `ToolBar`, напоминает `StackPanel`. Однако она работает совместно с панелью переполнения (см. ниже) и организует элементы, не умещающиеся в ее пределах (главной области панели инструментов `ToolBar`). Элемент `ToolBar` рассматривается в главе 10.

Панель ToolBarOverflowPanel

Панель `ToolBarOverflowPanel` – это упрощенный вариант `WrapPanel`, поддерживающий только горизонтальную ориентацию стопки и вертикальное оборачивание. Она используется в подразумеваемом по умолчанию стиле `ToolBar` для отображения не помещающихся элементов в области переполнения. Помимо возможностей `WrapPanel` здесь имеется свойство `WrapWidth`, которое ведет себя, как свойство `Padding`. Однако нет убедительных причин, по которым эту панель следовало бы предпочесть `WrapPanel`.

Панель ToolBarTray

Панель `ToolBarTray` поддерживает только потомков типа `ToolBar` (и возбуждает исключение `InvalidOperationException` при попытке добавить дочерний элемент другого типа). Она компонует элементы `ToolBar` последовательно (по умолча-

нию горизонтально) и позволяет перетаскивать их, организуя дополнительные строки, или сворачивать и разворачивать соседние элементы ToolBar.

Панель UniformGrid

Панель UniformGrid интересна, но ее практическая полезность сомнительна. Это упрощенный вариант сетки Grid, в которой все строки и столбцы имеют размер *, и изменить это невозможно. Из-за этого в классе UniformGrid есть два простых свойства типа double, задающих количество строк и столбцов, вместо куда более многословных определений коллекций RowDefinitions и ColumnDefinitions. Кроме того, в нем нет никаких присоединенных свойств; дочерние элементы добавляются по строкам, и в каждой ячейке может быть только один потомок.

Наконец, если количество строк и столбцов явно не задано (или количество дочерних элементов превосходит явно заданное количество ячеек), то UniformGrid автоматически выбирает подходящие значения. Например, если количество элементов от 2 до 4, то они размещаются в сетке 2×2, если от 5 до 9 – то в сетке 3×3, если от 10 до 16 – то в сетке 4×4 и т. д. На рис. 5.17 показано, как по умолчанию выглядит панель UniformGrid, когда в нее добавлено восемь кнопок.



Рис. 5.17. *Восемь кнопок на панели UniformGrid*

Панель SelectiveScrollingGrid

SelectiveScrollingGrid – подкласс Grid, используемый в подразумеваемом по умолчанию стиле элемента управления DataGridRow. В дополнение к функциональности Grid он позволяет «замораживать» некоторые ячейки, не препятствуя прокрутке остальных. Этим поведением управляет свойство SelectiveScrollingOrientation, принимающее следующие значения:

- None – ячейки не могут прокручиваться ни в каком направлении
- Horizontal – ячейки могут прокручиваться только по горизонтали
- Vertical – ячейки могут прокручиваться только по вертикали
- Both – ячейки могут прокручиваться в любом направлении. Это значение по умолчанию

Обработка переполнения содержимого

Встроенные панели делают все возможное для того, чтобы удовлетворить потребности своих дочерних элементов в месте на экране. Но иногда они вынуждены выделять потомкам меньше места, чем требуется, и бывает, что потомки отказываются полностью рисовать себя, когда места недостаточно. Например, может случиться, что для элемента явно задана ширина, превышающая ширину объемлющей панели. Или список `ListBox` содержит так много элементов, что они не помещаются в объемлющее его окно `Window`. В таких случаях возникает проблема переполнения содержимого.

Для ее разрешения можно применять различные стратегии:

- отсечение
- прокрутку
- масштабирование
- оборачивание
- обрезку

В этом разделе рассматриваются первые три стратегии. Примеры оборачивания уже встречались при обсуждении панели `WrapPanel` (а также `TabPanel` и `ToolBarOverflowPanel`). Это единственный способ реализовать оборачивание для нетекстового содержимого (компоновка текста рассматривается в главе 11).

Обрезка – это более интеллектуальная форма отсечения. Она поддерживается только для текста элементами `TextBlock` и `AccessText`, в которых есть свойство `TextTrimming` (типа `System.Windows.TextTrimming`), принимающее значения `None` (по умолчанию), `CharacterEllipsis` или `WordEllipsis`. В последних двух случаях отброшенный текст заменяется многоточием (...), а не просто прерывается в произвольном месте.

Отсечение

Отсечение дочерних элементов – это тот способ, который панели применяют по умолчанию, когда потомков становится слишком много. Отсечение может происходить на краях панели или внутри нее (например, на краях ячейки сетки или в заполняемой области `DockPanel`). Впрочем, этим поведением можно в какой-то степени управлять.

Во всех классах, производных от `UIElement`, есть булевское свойство `ClipToBounds`, которое управляет тем, можно ли рисовать дочерние элементы вне границ родителя. Однако, если внешний край элемента совпадает с внешним краем `Window` или `Page`, отсечение все равно производится. Этот механизм *не* является средством рисовать вне границ окна `Window`. (Непрямоугольные окна обсуждаются в главе 7 «Структурирование и развертывание приложения».)

Несмотря на то, что все панели наследуют свойство `ClipToBounds`, большая их часть автоматически отсекает потомков вне зависимости от значения этого свойства. Но панели `Canvas` и `UniformGrid` по умолчанию *не* отсекают свои дочерние элементы, и обе поддерживают установку для свойства `ClipToBounds` значения `true`, чтобы принудительно включить режим отсечения.

На рис. 5.18 показано, как свойство ClipToBounds влияет на изображение кнопки, которая целиком не помещается на родительской панели Canvas (светло-коричневого цвета).



ClipToBounds="False" ClipToBounds="True"

***Рис. 5.18.** Свойство ClipToBounds определяет, будут ли дочерние элементы рисоваться за пределами панели*

Такое поведение означает, что в случае, когда ClipToBounds не равно true, размер Canvas неважен; можно задать Height и Width равными 0, а содержимое все равно будет рисоваться, как будто Canvas занимает весь экран!

Элементы, производные от Control, также могут управлять отсечением своего содержимого с помощью свойства ClipToBounds. Например, в классе Button ClipToBounds по умолчанию равно false. На рис. 5.19 показано, что происходит, когда для этого свойства установлено значение true, а внутренний текст масштабируется с помощью преобразования ScaleTransform (примененного в режиме RenderTransform).



ClipToBounds="False" ClipToBounds="True"

***Рис. 5.19.** Свойство ClipToBounds можно использовать в элементах управления (например, Button) для управления рисованием внутреннего содержимого*

СОВЕТ

Панель Canvas можно вставлять в качестве промежуточного элемента для предотвращения отсечения в других панелях. Например, отсечения большой кнопки на границе Grid можно избежать, если вставить в соответствующую ячейку панель Canvas (которая займет ее целиком), а уже внутрь Canvas поместить кнопку Button. Разумеется, если вы хотите, чтобы поведение кнопки при растяжении было таким же, как если бы она была прямым потомком Grid, то придется написать код.

Тот же подход применим для предотвращения отсечения на границах *внутренних* ячеек сетки, но если нужно, чтобы элемент «просачивался» в соседние ячейки, то обычно лучше увеличить значения его свойств RowSpan и/или ColumnSpan.

ПРЕДУПРЕЖДЕНИЕ

Отсечение производится до применения преобразований в режиме RenderTransform!

Если элемент увеличивается путем применения преобразования ScaleTransform в режиме RenderTransform, то он может выйти за границы родительской панели, не подвергаясь отсечению (при условии, что он не достигнет края Window или Page). *Уменьшение* элемента с помощью ScaleTransform в режиме RenderTransform – вещь более тонкая. Если бы немасштабированный элемент был подвергнут отсечению из-за того, что вышел за границы родителя, то и масштабированный элемент отсекается точно таким же образом, пусть он даже целиком помещается на родительской панели! Объясняется это тем, что отсечение – часть процесса компоновки и к моменту применения RenderTransform оно уже полностью определено. Если вы хотите уменьшить большой элемент с помощью преобразования ScaleTransform, то попробуйте применить его в режиме LayoutTransform – возможно, так получится лучше.

Прокрутка

Во многих приложениях критически важна возможность прокручивать содержимое, которое из-за его размера нельзя увидеть целиком. С WPF это просто – стоит поместить элемент внутрь элемента управления System.Windows.Controls.ScrollViewer, как он сразу же становится прокручиваемым. ScrollViewer применяет элементы управления ScrollBar, которые автоматически присоединяются к содержимому, когда в этом возникает необходимость.

В классе ScrollViewer имеется свойство Content, значением которого может быть один-единственный элемент, обычно в этом качестве выступает некая панель. Поскольку Content – свойство содержимого с точки зрения XAML, то объект, нуждающийся в прокрутке, можно описать в разметке как дочерний элемент:

```
<Window Title="Using ScrollViewer"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ScrollViewer>
    <StackPanel>
    ...
    </StackPanel>
  </ScrollViewer>
</Window>
```

На рис. 5.20 показано окно Window, содержащее простую панель StackPanel, – с применением ScrollViewer и без него.

Элементы ScrollBar отвечают на различные события ввода, в том числе на нажатия клавиш со стрелками для перемещения на небольшое расстояние, клавиш Page Up и Page Down – на большее расстояние и сочетаний клавиш Ctrl+Home и Ctrl+End для перемещения в начало и конец соответственно.



Без прокрутки С прокруткой

***Рис. 5.20.** ScrollViewer позволяет прокручивать элемент, который не умещается в отведенном ему пространстве*

В классе `ScrollViewer` имеется еще ряд свойств и методов для манипулирования из программы, но самыми важными являются свойства `VerticalScrollBarVisibility` и `HorizontalScrollBarVisibility`. Оба они имеют тип перечисления `ScrollBarVisibility`, которое определяет четыре состояния полосы прокрутки:

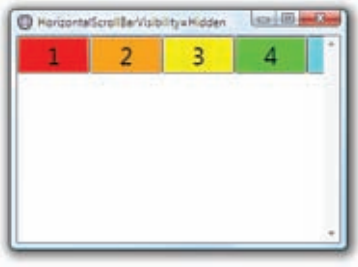
- `Visible` – полоса прокрутки всегда присутствует, даже если она не нужна. Если необходимости в ней нет, то она выглядит неактивной и не реагирует на события ввода. (Однако это не то же самое, что значение `Disabled` свойства `ScrollBarVisibility`.)
- `Auto` – полоса прокрутки видна, если содержимое нуждается в прокрутке в данном направлении. В противном случае полоса прокрутки отсутствует.
- `Hidden` – полоса прокрутки всегда невидима, но логически существует, то есть содержимое можно прокручивать клавишами со стрелками. Поэтому содержимое полностью доступно в данном направлении.
- `Disabled` – полоса прокрутки не только невидима, но и вообще не существует, то есть прокрутка невозможна ни с помощью клавиатуры, ни посредством мыши. В таком случае доступна только та часть содержимого, которая видна в пределах родителя.

По умолчанию свойство `VerticalScrollBarVisibility` равно `Visible`, а свойство `HorizontalScrollBarVisibility` равно `Auto`, поскольку именно это характерно для большинства приложений.

В зависимости от содержимого внутри `ScrollViewer` тонкое различие между `Hidden` и `Disabled` может оказаться вовсе не таким тонким. Например, на рис. 5.21 показаны два окна `Window`, содержащие `ScrollViewer`, внутри которого находится одна и та же панель `WrapPanel`. Единственное различие заключается в том, что в первом окне свойство `HorizontalScrollBarVisibility` равно `Hidden`, а во втором – `Disabled`.

В случае `Hidden` панель `WrapPanel` получает столько места, сколько ей требуется (как если бы `HorizontalScrollBarVisibility` было равно `Visible` или `Auto`), поэтому использует его целиком и компонует дочерние элементы в одну строку. В случае `Disabled` для панели выделяется только ширина, равная ширине

родительского окна Window, поэтому производится оборачивание, как если бы элемента ScrollViewer не существовало.





HorizontalScrollBarVisibility="Hidden" HorizontalScrollBarVisibility="Disabled"

Рис. 5.21. *Хотя горизонтальная полоса прокрутки в обоих случаях невидима, различная установка свойства HorizontalScrollBarVisibility радикально изменяет внешний вид панели WrapPanel*

СОВЕТ

В главе 3 «Основные принципы WPF» было объяснено, что подразумеваемое по умолчанию визуальное дерево элемента ListBox содержит ScrollViewer. Воспользовавшись синтаксисом присоединенных свойств, его свойствам VerticalScrollBarVisibility и HorizontalScrollBarVisibility можно присвоить значения и тем самым повлиять на поведение скрытого ScrollViewer:

```
<ListBox ScrollViewer.HorizontalScrollBarVisibility="Disabled">
  ...
</ListBox>
```

Масштабирование

Прокрутка – популярный и привычный способ работы с чрезмерно большим содержимым, но в некоторых случаях больше подходит динамическое увеличение или уменьшение содержимого с целью подогнать его под размер имеющейся области. Представьте, к примеру, что вы пишете программу для игры в карты. Очень вероятно, что вы захотите пропорционально масштабировать карты с учетом размера окна.

На рис. 5.22 показаны некоторые фигуры, составляющие графическое представление игральной карты (вместе с исходным кодом на XAML они приведены в главе 20). Эти фигуры помещены на панель Canvas, находящуюся внутри Window. Поскольку размеры заданы явно, то фигуры не масштабируются при изменении размера окна (даже если их поместить в сетку Grid, а не в Canvas). Очевидно, что в данном случае карта слишком велика.



***Рис. 5.22.** Фигуры, из которых состоит изображение игральной карты, не масштабируются вместе с окном*

Преобразование `ScaleTransform` может масштабировать элементы *относительно их собственного размера* (и поможет изменить размер игральной карты), но не дает механизма для масштабирования элементов *относительно имеющегося пространства* без написания дополнительного кода. К счастью, класс `System.Windows.Controls.Viewbox` предлагает простой механизм масштабирования произвольного содержимого в указанном пространстве.

Класс `Viewbox` относится к так называемым *декораторам*, то есть панелеподобным классам, у которых может быть только один дочерний элемент. Как и `Border`, он наследует классу `System.Windows.Controls.Decorator`. По умолчанию `Viewbox` (как и большинство элементов управления) растягивается в обоих направлениях, занимая все выделенное ему место. Но у него также имеется свойство `Stretch`, позволяющее указать, как в занимаемой области должен масштабироваться единственный дочерний элемент. Это свойство имеет тип перечисления `System.Windows.Media.Stretch` и может принимать следующие значения (продемонстрированы на рис. 5.23, где внутрь `Viewbox` помещена панель `Canvas`):

- `None` – масштабирование не производится вовсе. Результат такой же, как если бы `Viewbox` вообще не было.
- `Fill` – размеры дочернего элемента устанавливаются такими же, как размеры самого `Viewbox`. Поэтому отношение сторон дочернего элемента может не сохраняться.
- `Uniform` – дочерний элемент масштабируется так, чтобы он целиком поместился внутри `Viewbox` с сохранением отношения сторон. Поэтому, если отношения сторон `Viewbox` и дочернего элемента не совпадают, то в одном направлении останется пустое место. Этот вариант подразумевается по умолчанию.
- `UniformToFill` – дочерний элемент масштабируется так, чтобы он целиком заполнял `Viewbox` с сохранением отношения сторон. Поэтому, если отношения сторон `Viewbox` и дочернего элемента не совпадают, то в одном направлении содержимое будет отсечено.

Маловероятно, что в карточной игре понадобится, чтобы карта занимала всю площадь окна, но та же техника позволяет расположить карты в некоторой

подобласти окна. На рис. 5.23 Viewbox является дочерним элементом Window, но в реальном приложении вы, наверное, поместите его в ячейку подходящим образом выбранной сетки.

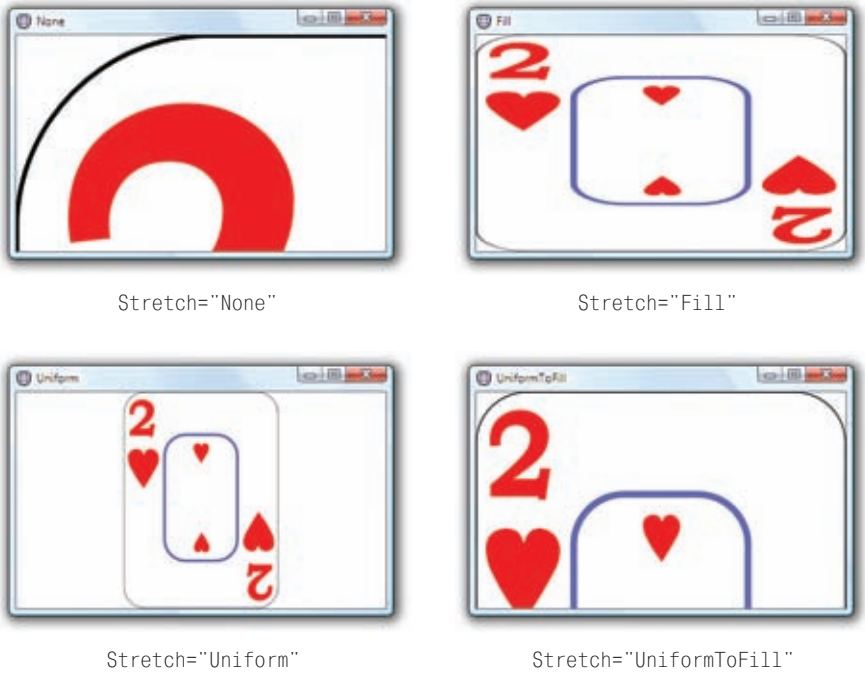


Рис. 5.23. *Влияние четырех возможных значений свойства Stretch элемента Viewbox на масштабирование игральной карты*

Второе свойство Viewbox позволяет указать, какие операции разрешены: только уменьшение содержимого, только увеличение или и то и другое. Оно называется StretchDirection, имеет тип перечисления System.Windows.Controls.StretchDirection и может принимать следующие значения:

- UpOnly – увеличивает содержимое, если необходимо. Если содержимое уже слишком велико, то Viewbox оставляет текущий размер без изменения.
- DownOnly – уменьшает содержимое, если необходимо. Если содержимое уже достаточно мало, то Viewbox оставляет текущий размер без изменения.
- Both – увеличивает или уменьшает содержимое в соответствии с заданным значением описанного выше свойства Stretch. Этот вариант подразумевается по умолчанию.

Поразительно, как просто задать стратегию прокрутки и масштабирования для работы с большим содержимым. Взгляните на код элемента Window, изображенного на рис. 5.20:

```
<Window Title="Using ScrollViewer"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ScrollViewer>
    <StackPanel>
      ...
    </StackPanel>
  </ScrollViewer>
</Window>
```

Достаточно заменить `ScrollViewer` на `Viewbox` (и изменить заголовок окна), чтобы получить результат, показанный на рис. 5.24:

```
<Window Title="Using Viewbox"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Viewbox>
    <StackPanel>
      ...
    </StackPanel>
   </Viewbox>
</Window>
```

И теперь мы видим все восемь кнопок, каким бы ни был размер окна!



Рис. 5.24. *Та же панель StackPanel, что на рис. 5.20, только помещенная внутрь Viewbox, а не ScrollViewer*

ПРЕДУПРЕЖДЕНИЕ

Viewbox отключает оборачивание!

Элемент `Viewbox` удобен во многих ситуациях, но не подходит в случае, когда требуется оборачивание, например при переносе текста на другую строку или компоновке содержимого с помощью `WrapPanel`. Дело в том, что содержимому сначала выделяется столько места в обоих направлениях, сколько оно запрашивает, а уже потом производится масштабирование (если необходимо). На рис. 5.25 этот эффект продемонстрирован на примере той же панели `WrapPanel`, что и на рис. 5.21, но с заменой `ScrollViewer` на `Viewbox`.

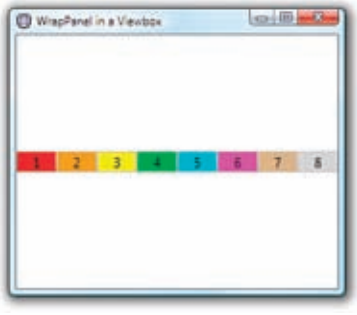


Рис. 5.25. *Панель WrapPanel, изображенная на рис. 5.21, не нуждается в оборачивании, если помещена не в ScrollViewer, а в Viewbox*

В результате все содержимое располагается в одной строке, которая может получиться гораздо меньше, чем вам хотелось бы. Установка для свойства `StretchDirection` значения `UpOnly` вместо подразумеваемого по умолчанию `Both` не поможет. Компоновка содержимого `Viewbox` производится до потенциального масштабирования. Поэтому значение `UpOnly` предотвращает уменьшение кнопок, но они по-прежнему располагаются в одной строке, как показано на рис. 5.26.

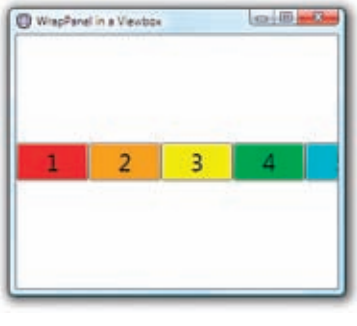


Рис. 5.26. *Установка для Viewbox, показанного на рис. 5.25, режима StretchDirection="UpOnly" предотвращает уменьшение кнопок, но не влияет на внутреннюю компоновку WrapPanel*

Получившийся результат аналогичен установке режима `HorizontalScrollBarVisibility="Hidden"` на рис. 5.21 с тем отличием, что прокрутить содержимое и увидеть то, что скрыто, невозможно даже с помощью клавиатуры.

Все вместе: создание сворачиваемой, стыкуемой, изменяющей размер панели

Давайте протестируем имеющиеся в WPF средства компоновки, попытавшись сконструировать более сложный пользовательский интерфейс. В этом разделе мы создадим панели по типу применяемых в Visual Studio – панели, умеющие пристыковываться к основному содержимому или сворачиваться в кнопку вдоль края окна. Если панель свернута, то при задержке указателя мыши над кнопкой показывается панель, но не в пристыкованном виде, а поверх основного содержимого. Вне зависимости от того, пристыкована панель или нет, ее размер можно изменить с помощью разделителя. На рис. 5.27–5.33 изображено несколько последовательных состояний пользовательского интерфейса в процессе работы с ним.

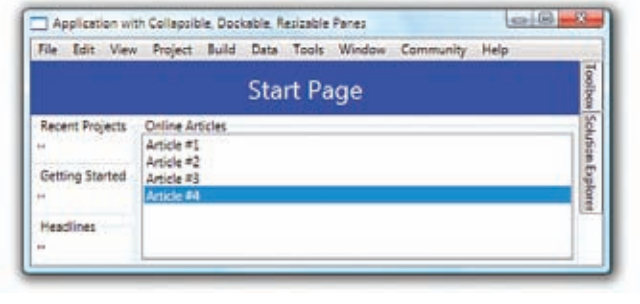


Рис. 5.27. *Сначала обе панели скрыты и видны только кнопки, пристыкованные к правому краю*

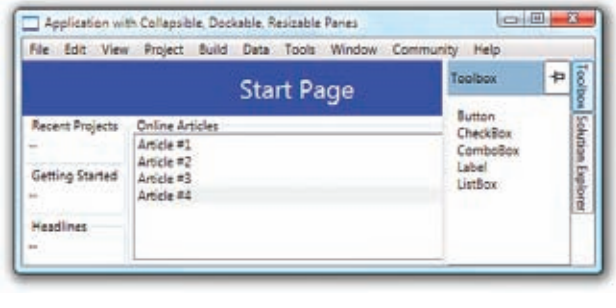


Рис. 5.28. *Если задержать указатель мыши над кнопкой Toolbox, то появится непристыкованная панель Toolbox, которая останется открыта, пока пользователь не переместит указатель на основное содержимое или на кнопку другой панели*

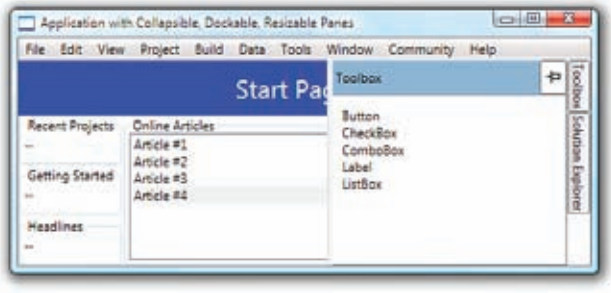


Рис. 5.29. *Размер непристыкованной панели можно изменять, но она все равно будет перекрывать основное содержимое*

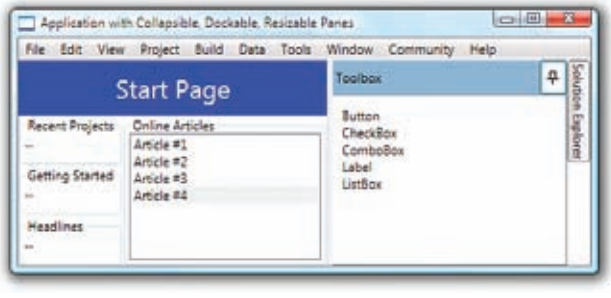


Рис. 5.30. *Если пристыковать панель Toolbox, щелкнув по значку канцелярской кнопки, то размер области основного содержимого уменьшится в соответствии с имеющимся пространством, а кнопка Toolbox пропадет*

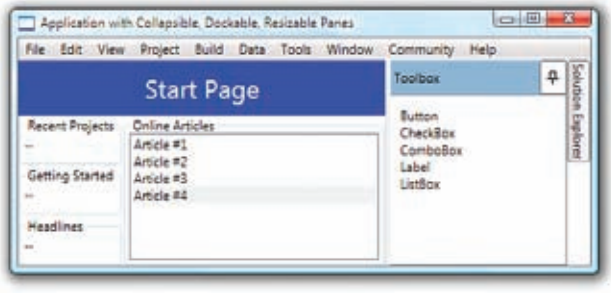


Рис. 5.31. *Размер пристыкованной панели по-прежнему можно изменять с помощью элемента GridSplitter, но на этот раз синхронно увеличивается или уменьшается область основного содержимого*

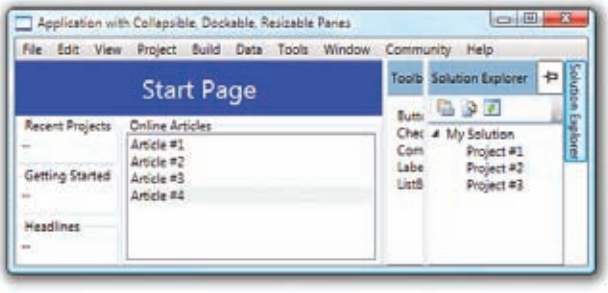


Рис. 5.32. *Если задержать указатель мыши над кнопкой Solution Explorer, то появится непристыкованная панель Solution Explorer, перекрывающая все остальное содержимое (включая пристыкованную панель Toolbox). Размер этой непристыкованной панели можно изменять независимо, увеличивая или уменьшая перекрытую область*

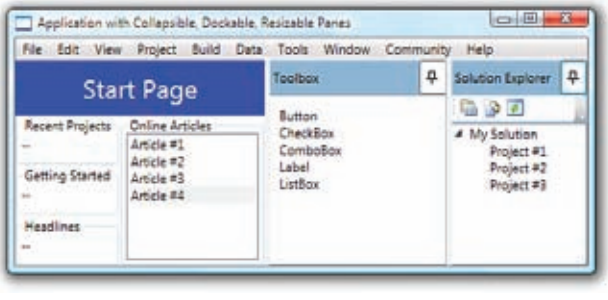


Рис. 5.33. *Панель Solution Explorer можно пристыковать, щелкнув по значку канцелярской кнопки. При этом панель Toolbox отодвигается, а правая полоса с кнопками исчезает, потому что кнопок, представляющих непристыкованные панели, больше не осталось*

Когда обе панели не пристыкованы, их размеры изменяются независимо от основного содержимого и друг от друга. Если же обе панели пристыкованы (как на рис. 5.33), то интерфейс ведет себя, как одна сетка `Grid` с тремя ячейками, которые могут меняться в размерах, но никогда не перекрываются.

Ну и как подойти к реализации подобного интерфейса? Поскольку для интерактивного изменения размера нужны разделители, кажется естественным взять за основу панель `Grid` с разделителями `GridSplitter`. Никакая другая встроенная панель не предоставляет интерактивных разделителей. Но поскольку непристыкованные панели должны перекрываться и независимо изменять размеры, то одной сетки недостаточно. Мы воспользуемся тремя независимыми сетками – по одной для основного содержимого и двух панелей – и расположим их поверх друг друга. А чтобы обеспечить синхронизацию трех независимых сеток, когда в этом есть необходимость (то есть когда панели

пристыкованы), мы применим свойство `SharedSizeGroup`. На рис. 5.34 показано, как эти сетки устроены и связаны между собой.

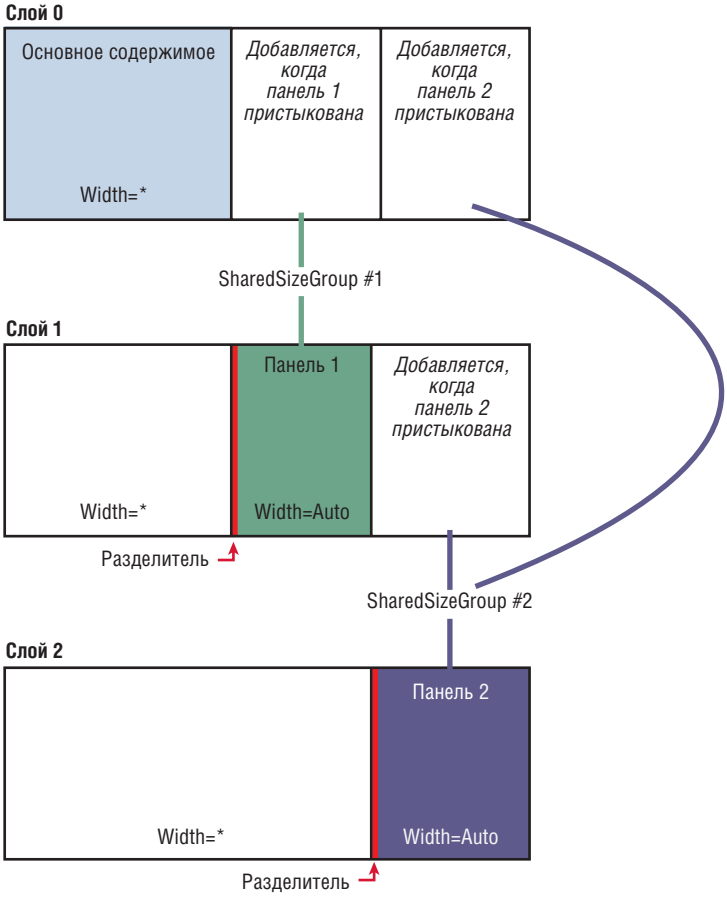


Рис. 5.34. *Три независимые сетки Grid используются для реализации двух сворачиваемых, стыкуемых, изменяющих размер панелей*

В нижнем слое (слое 0) расположено основное содержимое, которое, растягиваясь, заполняет сетку, когда обе панели свернуты. Задержка указателя мыши над любой кнопкой изменяет видимость соответствующей панели в слое 1 или 2 с `Collapsed` на `Visible`. Разделитель любой панели позволяет изменить соотношение места, занимаемого ею и столбцом слева от нее (он пуст, поэтому сквозь него видно содержимое, находящееся ниже, в слое 0).

Самое интересное происходит, когда наступает время пристыковать панель. При пристыковке панели 1 основное содержимое необходимое уменьшить до ширины пустого столбца 0 в слое 1. Поэтому в слой 0 динамически добавляется

пустой столбец такой же ширины, как панель 1. Поскольку мы не определяем жестко ширину в коде, а используем свойство SharedSizeGroup, при работе с разделителем в слое 1 нижний слой остается синхронизированным.

Та же техника применяется при стыковке панели 2, только теперь фиктивный столбец необходимо добавить во все нижележащие слои (0 и 1). В результате обе панели видны одновременно и не перекрываются, а размер основного содержимого в слое 0 остается правильным в присутствии одной или двух пристыкованных панелей, а также при полном их отсутствии. Отметим, что порядок следования панелей в случае, когда они обе пристыкованы, фиксирован.

Все три сетки помещены (куда бы вы думали?) в сетку с одной строкой и одним столбцом, чтобы они могли перекрывать друг друга и вместе с тем растягиваться, занимая все отведенное им место. *Z*-порядок слоя 0 всегда наименьший, но *Z*-порядок двух остальных слоев может меняться так, чтобы текущая непристыкованная панель всегда была наверху.

В листинге 5.3 приведен XAML-код приложения, изображенного на рис. 5.27–5.33, некоторые несущественные части для краткости опущены. Весь проект целиком имеется в исходном коде, прилагаемом к книге (по адресу *http://informit.com/title/9780672331190*).

Листинг 5.3. VisualStudioLikePanes.xaml – XAML-часть реализации приложения, изображенного на рис. 5.27–5.33

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Application with Collapsible, Dockable, Resizable Panes">
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      ...
    </Menu>

    <!-- Полоса с кнопками, пристыкованная к правому краю: -->
    <StackPanel Name="buttonBar" Orientation="Horizontal" DockPanel.Dock="Right">
    <StackPanel.LayoutTransform>
      <RotateTransform Angle="90"/>
    </StackPanel.LayoutTransform>
      <Button Name="pane1Button" MouseEnter="pane1Button_MouseEnter">
        Toolbox
      </Button>
      <Button Name="pane2Button" MouseEnter="pane2Button_MouseEnter">
        Solution Explorer
      </Button>
    </StackPanel>

    <!-- Сетка, содержащая три дочерних сетки, заполняет DockPanel: -->
    <Grid Name="parentGrid" Grid.IsSharedSizeScope="True">

      <!-- Слой 0: -->
      <Grid Name="layer0" MouseEnter="layer0_MouseEnter">
```

```
  ... (содержимое сетки такое же, как в листинге 5.2)
</Grid>

<!-- Слой 1: -->
<Grid Name="layer1" Visibility="Collapsed">
<Grid.ColumnDefinitions>
  <ColumnDefinition/>
  <ColumnDefinition SharedSizeGroup="column1" Width="auto"/>
</Grid.ColumnDefinitions>
  <!-- Столбец 0 пуст, но столбец 1 содержит элементы
       Grid и GridSplitter: -->
  <Grid Grid.Column="1" MouseEnter="pane1_MouseEnter"
     Background="{DynamicResource
                  {x:Static SystemColors.ActiveCaptionBrushKey}}">
  <Grid.RowDefinitions>
     <RowDefinition Height="auto"/>
     <RowDefinition/>
  </Grid.RowDefinitions>
      <!-- Строка 0 содержит заголовок,
           а строка 1 - содержимое конкретной панели: -->
    <DockPanel Grid.Row="0">
      <Button Name="pane1Pin" Width="26" DockPanel.Dock="Right"
              Click="pane1Pin_Click" Background="White">
       <Image Name="pane1PinImage" Source="pinHorizontal.gif"/>
     </Button>
     <TextBlock Padding="8" TextTrimming="CharacterEllipsis"
                Foreground="{DynamicResource
                           {x:Static SystemColors.ActiveCaptionTextBrushKey}}"
                DockPanel.Dock="Left">Toolbox</TextBlock>
    </DockPanel>
    ... (содержимое панели находится в строке 1)
  </Grid>
  <GridSplitter Width="5" Grid.Column="1" HorizontalAlignment="Left"/>
</Grid>

<!-- Слой 2: -->
<Grid Name="layer2" Visibility="Collapsed">
<Grid.ColumnDefinitions>
  <ColumnDefinition/>
  <ColumnDefinition SharedSizeGroup="column2" Width="auto"/>
</Grid.ColumnDefinitions>
    <!-- Столбец 0 пуст, но столбец 1 содержит элементы
         Grid и GridSplitter: -->
  <Grid Grid.Column="1" MouseEnter="pane2_MouseEnter"
        Background="{DynamicResource
                     {x:Static SystemColors.ActiveCaptionBrushKey}}">
  <Grid.RowDefinitions>
    <RowDefinition Height="auto"/>
    <RowDefinition Height="auto"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
```

```
        <!-- Строка 0 содержит заголовок,
             а строки 1 и 2 - содержимое конкретной панели: -->
        <DockPanel Grid.Row="0">
          <Button Name="pane2Pin" Width="26" DockPanel.Dock="Right"
                  Click="pane2Pin_Click" Background="White">
            <Image Name="pane2PinImage" Source="pinHorizontal.gif"/>
          </Button>
          <TextBlock Padding="8" TextTrimming="CharacterEllipsis"
                     Foreground="{DynamicResource
                              {x:Static SystemColors.ActiveCaptionTextBrushKey}}"
                     DockPanel.Dock="Left">Solution Explorer
          </TextBlock>
        </DockPanel>
        ... (содержимое панели находится в строке 1)
      </Grid>
      <GridSplitter Width="5" Grid.Column="1" HorizontalAlignment="Left"/>
    </Grid>
  </Grid>
</DockPanel>
</Window>
```

На верхнем уровне окна `Window` расположена панель `DockPanel`, которая организует меню, панель `StackPanel`, содержащую полосу с кнопками (повернутую на 90° с помощью преобразования `RotateTransform`), и сетку из одной ячейки, где находятся три сетки, определяющие «слои». Отметим, что меню `Menu` добавляется в `DockPanel` раньше, чем `StackPanel`, чтобы оно растянулось на всю ширину вдоль верхнего края окна.

В каждой сетке-слое имеется всего один столбец для хранения содержимого, и во всех трех случаях содержимое заключено в сетку. Каждый разделитель `GridSplitter` пристыкован к левой стороне столбца с содержимым, поэтому он не перекрывает содержимое из других слоев. Отметим одну тонкость – заголовок каждой панели помещен не в метку `Label`, а в элемент `TextBlock`, чтобы можно было установить свойство `TextTrimming="CharacterEllipsis"`, тогда при уменьшении размера панели система будет не просто отсекать текст заголовка, а заменять отброшенную часть многоточием. Это выглядит более профессионально.

В листинге 5.4 приведен застраничный код на C#, дополняющий код в листинге 5.3.

Листинг 5.4. VisualStudioLikePanes.xaml.cs – C#-часть реализации приложения, изображенного на рис. 5.27–5.33

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Imaging;

public partial class MainWindow : Window
{
  // Фиктивные столбцы для слоев 0 и 1:
  ColumnDefinition column1CloneForLayer0;
```

```
ColumnDefinition column2CloneForLayer0;
ColumnDefinition column2CloneForLayer1;

public MainWindow()
{
  InitializeComponent();

  // Инициализировать фиктивные столбцы, используемые,
  // когда панели пристыкованы
  column1CloneForLayer0 = new ColumnDefinition();
  column1CloneForLayer0.SharedSizeGroup = "column1";
  column2CloneForLayer0 = new ColumnDefinition();
  column2CloneForLayer0.SharedSizeGroup = "column2";
  column2CloneForLayer1 = new ColumnDefinition();
  column2CloneForLayer1.SharedSizeGroup = "column2";
}

// Переключаем состояние: пристыкована/не пристыкована (панель 1)
public void pane1Pin_Click(object sender, RoutedEventArgs e)
{
  if (pane1Button.Visibility == Visibility.Collapsed)
    UndockPane(1);
  else
    DockPane(1);
}

// Переключаем состояние: пристыкована/не пристыкована (панель 2)
public void pane2Pin_Click(object sender, RoutedEventArgs e)
{
  if (pane2Button.Visibility == Visibility.Collapsed)
    UndockPane(2);
  else
    DockPane(2);
}

// Показываем панель 1, когда указатель мыши находится над ее кнопкой
public void pane1Button_MouseEnter(object sender, RoutedEventArgs e)
{
  layer1.Visibility = Visibility.Visible;

  // Корректируем Z-порядок, чтобы панель всегда была сверху:
  Grid.SetZIndex(layer1, 1);
  Grid.SetZIndex(layer2, 0);

  // Скрываем вторую панель, если она не пристыкована
  if (pane2Button.Visibility == Visibility.Visible)
     layer2.Visibility = Visibility.Collapsed;
}

// Показываем панель 2, когда указатель мыши находится над ее кнопкой
public void pane2Button_MouseEnter(object sender, RoutedEventArgs e)
```

```
{
  layer2.Visibility = Visibility.Visible;

  // Корректируем Z-порядок, чтобы панель всегда была сверху:
  Grid.SetZIndex(layer2, 1);
  Grid.SetZIndex(layer1, 0);

  // Скрываем вторую панель, если она не пристыкована
  if (pane1Button.Visibility == Visibility.Visible)
     layer1.Visibility = Visibility.Collapsed;
}

// Скрываем все непристыкованные панели, когда указатель мыши
// перемещается в слой 0
public void layer0_MouseEnter(object sender, RoutedEventArgs e)
{
  if (pane1Button.Visibility == Visibility.Visible)
    layer1.Visibility = Visibility.Collapsed;
  if (pane2Button.Visibility == Visibility.Visible)
    layer2.Visibility = Visibility.Collapsed;
}

// Скрываем вторую панель, если она не пристыкована, когда указатель
// мыши перемещается на панель 1
public void pane1_MouseEnter(object sender, RoutedEventArgs e)
{
  // Скрываем вторую панель, если она не пристыкована
  if (pane2Button.Visibility == Visibility.Visible)
    layer2.Visibility = Visibility.Collapsed;
}

// Скрываем вторую панель, если она не пристыкована, когда указатель
// мыши перемещается на панель 2
public void pane2_MouseEnter(object sender, RoutedEventArgs e)
{
  // Скрываем вторую панель, если она не пристыкована
  if (pane1Button.Visibility == Visibility.Visible)
    layer1.Visibility = Visibility.Collapsed;
}

// Пристыковываем панель, при этом скрывается
// соответствующая ей кнопка
public void DockPane(int paneNumber)
{
  if (paneNumber == 1)
  {
    pane1Button.Visibility = Visibility.Collapsed;
    pane1PinImage.Source = new BitmapImage(new Uri("pin.gif",
              UriKind.Relative));

    // Добавляем клонированный столбец в слой 0:
```

```
    layer0.ColumnDefinitions.Add(column1CloneForLayer0);

    // Добавляем клонированный столбец в слой 1, но только если
    // панель 2 пристыкована:
    if (pane2Button.Visibility == Visibility.Collapsed)
      layer1.ColumnDefinitions.Add(column2CloneForLayer1);
  }
  else if (paneNumber == 2)
  {
    pane2Button.Visibility = Visibility.Collapsed;
    pane2PinImage.Source = new BitmapImage(new Uri("pin.gif",
              UriKind.Relative));

    // Добавляем клонированный столбец в слой 0:
    layer0.ColumnDefinitions.Add(column2CloneForLayer0);

    // Добавляем клонированный столбец в слой 1, но только если
    // панель 1 пристыкована:
    if (pane1Button.Visibility == Visibility.Collapsed)
      layer1.ColumnDefinitions.Add(column2CloneForLayer1);
  }
}

// Отстыковываем панель, при этом становится видна
// соответствующая ей кнопка
public void UndockPane(int paneNumber)
{
  if (paneNumber == 1)
  {
    layer1.Visibility = Visibility.Visible;
    pane1Button.Visibility = Visibility.Visible;
    pane1PinImage.Source = new BitmapImage
            (new Uri("pinHorizontal.gif", UriKind.Relative));

    // Удаляем клонированные столбцы из слоев 0 и 1:
    layer0.ColumnDefinitions.Remove(column1CloneForLayer0);

    // Этот столбец присутствует не всегда, но метод Remove
    // молча игнорирует попытку удалить несуществующий столбец:
    layer1.ColumnDefinitions.Remove(column2CloneForLayer1);
  }
  else if (paneNumber == 2)
  {
    layer2.Visibility = Visibility.Visible;
    pane2Button.Visibility = Visibility.Visible;
    pane2PinImage.Source = new BitmapImage
            (new Uri("pinHorizontal.gif", UriKind.Relative));

    // Удаляем клонированные столбцы из слоев 0 и 1:
    layer0.ColumnDefinitions.Remove(column2CloneForLayer0);
```

```
            // Этот столбец присутствует не всегда, но метод Remove
            // молча игнорирует попытку удалить несуществующий столбец:
            layer1.ColumnDefinitions.Remove(column2CloneForLayer1);
        }
    }
}
```

Этот код на C# умеет работать ровно с двумя панелями. Наверное, вы захотите его обобщить до уровня нестандартного элемента управления, но с точки зрения компоновки идея не изменится.

Отметим, что нет кода, который скрывал бы «полосу с кнопками», когда все панели пристыкованы, и показывал ее, если хотя бы одна панель отстыкована. Это происходит автоматически, потому что `StackPanel` по умолчанию адаптируется под размер своего содержимого, поэтому сворачивание обеих кнопок приводит к сворачиванию всей панели.

Кода в листинге 5.4 не так уж много (и он не сложен), но задачу конструирования довольно хитроумного пользовательского интерфейса он тем не менее решает.

Резюме

Средства, описанные в этой и предыдущей главе, позволяют управлять компоновкой различными интересными способами. Это вам не старые добрые времена, когда чуть ли не единственным способом было задание размера и координат точки на экране.

Встроенные панели – и прежде всего `Grid` – ключ к применению WPF в качестве инструмента быстрой разработки. Но одним из самых замечательных аспектов компоновки в WPF является тот факт, что родительские панели сами могут быть потомками других панелей. В этой главе мы рассматривали панели по отдельности, но с помощью вложенных панелей можно добиваться поистине впечатляющих результатов.

- Маршрутизируемые события
- События клавиатуры
- События мыши
- События стилуса
- Мультисенсорные события
- Команды

6

События ввода: клавиатура, мышь, стилус и мультисенсорные устройства

Теперь, когда мы знаем, как скомпоновать пользовательский интерфейс в WPF, настало время сделать его интерактивным. В этой главе рассматриваются две важных составных части инфраструктуры WPF – маршрутизируемые события и команды. Заодно обсуждаются события для каждой категории устройств ввода: клавиатуры, мыши, стилуса и мультисенсорных устройств.

Маршрутизируемые события

В главе 3 «Основные принципы WPF» показано, как с помощью свойств зависимости WPF реализует дополнительную инфраструктуру поверх хорошо известной идеи свойств .NET. Но этим дело не ограничивается – WPF еще и дополняет понятие события. *Маршрутизируемые события* предназначены для работы с деревьями элементов. Сгенерированное маршрутизируемое событие может распространяться вверх или вниз по визуальному и логическому дереву, достигая каждого элемента простым и естественным образом без написания дополнительного кода.

Маршрутизация событий позволяет большинству приложений вообще не задумываться о наличии визуального дерева (что очень удобно для стилизации) и является основой механизма композиции элементов в WPF. Например, кнопка `Button` генерирует событие `Click` в результате обработки низкоуровневых событий `MouseLeftButtonDown` и `KeyDown`. Но когда пользователь нажимает левую кнопку мыши, наведя ее указатель на стандартную кнопку, он в реальности взаимодействует с визуальным дочерним элементом `ButtonChrome` или `TextBlock`. Однако поскольку событие распространяется вверх по *визуальному* дереву, то элемент `Button` в конечном итоге получит это событие и сможет его обработать. Аналогично в случае кнопки Stop, подобной кнопке медиаплеера (см. главу 2 «Все тайны XAML»), пользователь может нажать левую кнопку

мыши, поместив указатель над логическим потомком Rectangle. Поскольку событие распространяется вверх по *логическому* дереву, элемент Button все равно увидит событие и сможет на него отреагировать. (Впрочем, если вы захотите различать события элемента Rectangle и объемлющей его кнопки Button, ничто не помешает вам это сделать.)

Таким образом, внутрь любого элемента, к примеру Button, можно поместить сколь угодно сложное содержимое (применяя технику, описанную в главе 14 «Стили, шаблоны, обложки и темы»), но щелчок левой кнопкой мыши при наведении указателя на любой внутренний элемент все равно приведет к возникновению события Click для родительской кнопки. Не будь маршрутизируемых событий, авторы внутреннего содержимого или клиенты кнопки должны были бы писать дополнительный код для связывания всего воедино.

Реализация и поведение маршрутизируемых событий имеют много общего со свойствами зависимости. Как и при обсуждении свойств зависимости, мы сначала посмотрим, как реализуется простое маршрутизируемое событие, чтобы сделать обсуждение более конкретным. Затем рассмотрим некоторые особенности маршрутизируемых событий и применим это к диалоговому окну About из главы 3.

Реализация маршрутизируемого события

В большинстве случаев маршрутизируемые события внешне мало чем отличаются от обычных событий .NET. Как и в случае со свойствами зависимости, .NET-совместимые языки (кроме XAML) ничего не знают о том, что такое маршрутизация. Дополнительную поддержку предоставляют лишь различные классы WPF.

В листинге 6.1 показана схема реализации маршрутизируемого события Click в классе Button. (На самом деле событие Click реализовано в базовом классе Button, но сейчас это несущественно.)

Напомним, что свойства зависимости представлены открытыми статическими полями типа DependencyProperty с принимаемым по соглашению суффиксом Property. Точно так же маршрутизируемые события представлены открытыми статическими полями типа RoutedEvent с принимаемым по соглашению суффиксом Event. Так же, как свойство зависимости, маршрутизируемое событие регистрируется в статическом конструкторе, и дополнительно определяется обычное событие .NET – *обертывающее событие*, – чтобы было проще писать процедурный код и добавлять обработчик события в XAML-коде с помощью стандартного синтаксиса атрибутов событий. Как и обертывающее свойство, обертывающее событие не должно делать в аксессорах ничего, кроме вызова методов AddHandler и RemoveHandler.

Листинг 6.1. Стандартная реализация маршрутизируемого события

```
public class Button : ButtonBase
{
  // Маршрутизируемое событие
  public static readonly RoutedEvent ClickEvent;
```

```
  static Button()
  {
    // Зарегистрировать событие
    Button.ClickEvent = EventManager.RegisterRoutedEvent("Click",
      RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(Button));
    ...
  }

  // Обертывающее событие .NET (необязательное)
  public event RoutedEventHandler Click
  {
    add { AddHandler(Button.ClickEvent, value); }
    remove { RemoveHandler(Button.ClickEvent, value); }
  }

  protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
  {
    ...
    // Сгенерировать событие
    RaiseEvent(new RoutedEventArgs(Button.ClickEvent, this));
    ...
  }
  ...
}
```

Методы `AddHandler` и `RemoveHandler` наследуются не от класса `DependencyObject`, а от `UIElement`. Они соответственно присоединяют и отсоединяют делегат от маршрутизируемого события. Внутри метода `OnMouseLeftButtonDown` вызывается метод `RaiseEvent` (также определенный в базовом классе `UIElement`), которому передается поле типа `RoutedEvent`, соответствующее генерации события `Click`. В качестве источника события передается текущий экземпляр `Button` (`this`). В листинге это не показано, но на самом деле событие `Click` кнопки генерируется также в ответ на событие `KeyDown`, то есть поддерживается нажатие кнопки посредством клавиши пробела или Enter.

Стратегии маршрутизации и обработчики событий

В момент регистрации маршрутизируемого события задается одна из трех *стратегий маршрутизации* – вариантов распространения события по дереву элементов. Стратегии описываются перечислением `RoutingStrategy`:

- `Tunneling` – событие сначала возникает в корне дерева, а потом опускается вниз по дереву, заново возникая в каждом элементе на пути к источнику, включая его самого (если туннелирование не будет прервано по дороге в результате пометки события как обработанного).
- `Bubbling` – событие сначала возникает в элементе-источнике, а затем поднимается вверх по дереву, заново возникая в каждом элементе на пути к корню, включая сам корень (если всплытие не будет прервано по дороге в результате пометки события как обработанного).

- `Direct` – событие возникает только в элементе-источнике. Точно так же ведут себя обычные события .NET; различие лишь в том, что к маршрутизируемому событию применяются и другие механизмы, в частности триггеры событий.

Сигнатуры обработчиков маршрутизируемых событий устроены так же, как сигнатуры всех обработчиков событий в .NET: первый параметр – объект типа `System.Object`, который обычно называют `sender`, второй (обычно называемый `e`) – экземпляр класса, производного от `System.EventArgs`. Передаваемый обработчику параметр `sender` – это всегда элемент, к которому присоединен данный обработчик. Параметр `e` является объектом класса `RoutedEventArgs` (или производного от него) – подкласса `EventArgs`, обладающего следующими полезными свойствами:

- `Source` – элемент логического дерева, первоначально сгенерировавший событие.
- `OriginalSource` – элемент визуального дерева, первоначально сгенерировавший событие (например, в случае стандартной кнопки `Button` это будет дочерний элемент `TextBlock` или `ButtonChrome`).
- `Handled` – булевский флаг, которому можно присвоить значение `true` и тем самым пометить, что событие обработано. Именно таким способом прерывается туннелирование и всплытие.
- `RoutedEvent` – сам объект маршрутизированного события (например, `Button.ClickEvent`), который может быть полезен для различения событий в случае, когда один и тот же обработчик используется для обработки разных событий.

Наличие свойств `Source` и `OriginalSource` позволяет работать как с высокоуровневым логическим, так и с низкоуровневым визуальным деревом. Впрочем, это различие существенно только для физических событий, таких как события мыши. Более абстрактные события могут и не иметь прямой связи с элементом визуального дерева (например, событие `Click` – вследствие поддержки клавиатуры), и тогда в качестве `Source` и `OriginalSource` выступает один и тот же объект.

Маршрутизируемые события в действии

В классе `UIElement` определено много маршрутизируемых событий для клавиатуры, мыши, мультисенсорных устройств и стилуса. Большая часть из них всплывающие, но для многих есть и парные туннелируемые. Туннелируемые события легко распознать, потому что по принятому соглашению их имена начинаются со слова `Preview`. Такое событие – также по соглашению – генерируется непосредственно перед парным ему всплывающим. Например, туннелируемое событие `PreviewMouseMove` генерируется перед всплывающим событием `MouseMove`.

Идея, стоящая за такими парами событий, заключается в том, чтобы дать элементам возможность отменить или иным способом модифицировать событие, которое еще только произойдет. По соглашению встроенные в WPF элементы

предпринимают действия только в ответ на всплывающее событие (в случае если определена пара событий – всплывающее и туннелируемое), гарантируя тем самым, что туннелируемые события отвечают своему названию (preview означает «предварительный просмотр»). Представим, к примеру, что требуется реализовать элемент TextBox, который позволяет вводить только строки, отвечающие некоторому образцу или регулярному выражению (например, номера телефонов и почтовые индексы). Если обрабатывать в нем событие KeyDown, то лучшее, что можно сделать, – удалить текст, который уже отображен в поле TextBox. Если же обрабатывать событие PreviewKeyDown, то можно пометить его как «обработанное» и тем самым не только прервать туннелирование, но и воспрепятствовать генерации всплывающего события KeyDown. В таком случае TextBox вообще не получит уведомления о событии KeyDown и введенный символ не появится в поле.

Чтобы продемонстрировать работу со всплывающими событиями, в листинге 6.2 приведена модификация диалогового окна About из главы 3 – к окну Window присоединен обработчик события MouseRightButtonDown. В листинге 6.3 показана реализация этого обработчика в застраничном коде на C#.

Листинг 6.2. Диалоговое окно About с обработчиком события в корневом элементе

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AboutDialog"
        MouseRightButtonDown="AboutDialog_MouseRightButtonDown"
        Title="About WPF 4 Unleashed" SizeToContent="WidthAndHeight"
        Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4 Unleashed
    </Label>
    <Label>© 2010 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

Листинг 6.3. Застраничный код для разметки в листинге 6.2

```
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Controls;
```

```
public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
  }

  void AboutDialog_MouseRightButtonDown(object sender, MouseButtonEventArgs e)
  {
    // Вывести информацию о событии
    this.Title = "Source = " + e.Source.GetType().Name + ", OriginalSource = " +
    e.OriginalSource.GetType().Name + " @ " + e.Timestamp;

    // В этом примере все возможные источники наследуют Control
    Control source = e.Source as Control;

    // Показать или скрыть рамку вокруг элемента-источника
    if (source.BorderThickness != new Thickness(5))
    {
      source.BorderThickness = new Thickness(5);
      source.BorderBrush = Brushes.Black;
    }
    else
      source.BorderThickness = new Thickness(0);
  }
}
```

Когда событие `MouseRightButtonDown` всплывает до элемента `Window`, его обработчик выполняет два действия: выводит информацию о событии в строке заголовка окна и рисует (а впоследствии стирает) толстую черную рамку вокруг элемента логического дерева, над которым произошел щелчок правой кнопкой мыши. Результат изображен на рис. 6.1. Отметим, что при щелчке по метке `Label` свойство `Source` предоставляет ссылку на эту метку, а `OriginalSource` — ссылку на ее визуальный дочерний элемент `TextBlock`.

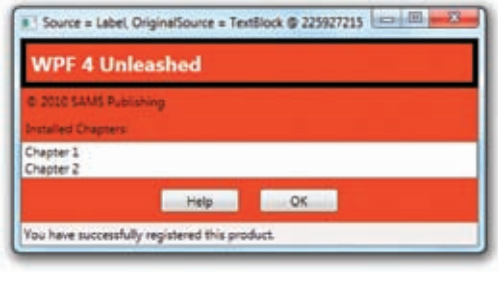


Рис. 6.1. *Модифицированное окно About после щелчка правой кнопкой мыши по первой метке*

Если запустить эту программу и последовательно щелкнуть правой кнопкой мыши по всем элементам, то обнаружатся два любопытных эффекта:

- `Window` не получает событие `MouseRightButtonDown`, если щелкнуть по любому элементу списка `ListBoxItem`. Дело в том, что `ListBoxItem` сам обрабатывает это событие, равно как и `MouseLeftButtonDown` (и прерывает всплытие), – это нужно ему для реализации выбора элементов.
- `Window` получает событие `MouseRightButtonDown` при щелчке по кнопке `Button`, но никаких изменений во внешнем виде рамки не происходит. Это объясняется структурой стандартного визуального дерева `Button`, которая была показана на рис. 3.3. В отличие от элементов `Window`, `Label`, `ListBox`, `ListBoxItem` и `StatusBar`, в визуальном дереве `Button` нет элемента `Border`.

КОПНЕМ ГЛУБЖЕ

Прерывание маршрутизации события – иллюзия!

Хотя присваивание значения `true` свойству `Handled` объекта `RoutedEventArgs` в обработчике маршрутизируемого события должно приводить к прерыванию туннелирования или всплытия, обработчики, присоединенные к элементам, находящимся выше или ниже в дереве, все равно могут запросить получение событий! Сделать это можно только в процедурном коде с помощью перегруженного варианта метода `AddHandler`, который принимает дополнительный булевский параметр `handledEventsToo`.

Например, в листинге 6.2 можно удалить атрибут события и заменить его таким обращением к `AddHandler` в конструкторе `AboutDialog`:

```
public AboutDialog()
{
  InitializeComponent();
  this.AddHandler(Window.MouseRightButtonDownEvent,
     new MouseButtonEventHandler(AboutDialog_MouseRightButtonDown), true);
}
```

Поскольку в третьем параметре передано значение `true`, то обработчик `AboutDialog_MouseRightButtonDown` получает событие щелчка правой кнопкой мыши по `ListBoxItem` и рисует черную рамку!

Однако лучше не прибегать к этому приему, потому что для пометки события как обработанного, очевидно, была какая-то причина. Более правильно было бы присоединить обработчик `Preview`-версии того же события.

Но в целом мы хотели подчеркнуть, что прерывание туннелирования и всплытия – на самом деле иллюзия. Распространение события все равно продолжается, но по умолчанию обработчики видят только те события, которые не помечены как уже обработанные.

Присоединенные события

Туннелирование и всплытие некоторого маршрутизируемого события происходит естественно, когда это событие определено в каждом элементе дерева. Но WPF поддерживает туннелирование и всплытие маршрутизируемых событий даже для элементов, в которых данное событие не определено! Возможно это благодаря понятию *присоединенного события*.

Присоединенные события работают примерно так же, как присоединенные свойства (а их использование в механизме туннелирования и всплытия напоминает использование присоединенных свойств совместно с механизмом наследования значений свойств). В листинге 6.4 мы снова изменили окно About, добавив прямо в корень окна `Window` обработку всплывающего события `SelectionChanged`, генерируемого списком `ListBox`, и всплывающего события `Click`, генерируемого обеими кнопками `Button`. Поскольку в классе `Window` не определены события `SelectionChanged` и `Click`, то имена атрибутов событий необходимо снабдить префиксами, содержащими имя класса, в котором соответствующее событие определено. В листинге 6.5 представлен застраничный код, содержащий определения обоих обработчиков событий. Обработка сводится к выводу в окне `MessageBox` информации о том, что произошло.

Листинг 6.4. Диалоговое окно About с двумя обработчиками присоединенных событий в корневом окне

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AboutDialog"
        ListBox.SelectionChanged="ListBox_SelectionChanged"
        Button.Click="Button_Click"
        Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
        Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4 Unleashed
    </Label>
    <Label>© 2010 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

Листинг 6.5. Застраничный код для разметки в листинге 6.4

```
using System.Windows;
using System.Windows.Controls;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
  }

  void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
  {
    if (e.AddedItems.Count > 0)
      MessageBox.Show("You just selected " + e.AddedItems[0]);
  }

  void Button_Click(object sender, RoutedEventArgs e)
  {
    MessageBox.Show("You just clicked " + e.Source);
  }
}
```

Любое маршрутизируемое событие можно использовать как присоединенное. Синтаксис присоединенных событий, показанный в листинге 6.4, допустим потому, что компилятор XAML видит событие .NET `SelectionChanged`, определенное в классе `ListBox`, и событие .NET `Click`, определенное в классе `Button`. Однако во время выполнения вызывается метод `AddHandler`, который присоединяет оба события к элементу `Window`. Поэтому эти два атрибута события эквивалентны следующему коду в конструкторе `Window`:

```
public AboutDialog()
{
  InitializeComponent();
  this.AddHandler(ListBox.SelectionChangedEvent,
    new SelectionChangedEventHandler(ListBox_SelectionChanged));
  this.AddHandler(Button.ClickEvent, new RoutedEventHandler(Button_Click));
}
```

КОПНЕМ ГЛУБЖЕ

Консолидация обработчиков маршрутизируемых событий

Поскольку вместе с маршрутизируемым событием передается достаточно много информации, при желании есть возможность обработать все туннелируемые и всплывающие события в одном «мегаобработчике» на верхнем уровне. Он мог бы исследовать объект `RoutedEvent`, определить, какое событие сгенерировано, привести параметр `RoutedEventArgs` к типу соответствующего подкласса (например, `KeyEventArgs`, `MouseButtonEventArgs` и т. д.), а потом предпринять соответствующие действия.

Например, код в листинге 6.5 можно было бы переписать, поместив обработчики событий ListBox.SelectionChanged и Button.Click в один метод GenericHandler:

```
void GenericHandler(object sender, RoutedEventArgs e)
{
  if (e.RoutedEvent == Button.ClickEvent)
  {
    MessageBox.Show("You just clicked " + e.Source);
  }
  else if (e.RoutedEvent == ListBox.SelectionChangedEvent)
  {
    SelectionChangedEventArgs sce = (SelectionChangedEventArgs)e;
    if (sce.AddedItems.Count > 0)
      MessageBox.Show("You just selected " + sce.AddedItems[0]);
  }
}
```

Это возможно благодаря встроенному в каркас .NET Framework механизму контравариантности делегатов, позволяющему использовать делегат с методом, в сигнатуре которого указан базовый класс ожидаемого параметра (например, RoutedEventArgs вместо SelectionChangedEventArgs). Метод GenericHandler просто приводит параметр RoutedEventArgs к нужному типу, когда ему необходимо получить дополнительную информацию, специфичную для события SelectionChanged.

События клавиатуры

Основные события клавиатуры, поддерживаемые всеми подклассами UIElement, – это всплывающие события KeyDown и KeyUp и парные им туннелируемые события PreviewKeyDown и PreviewKeyUp. Обработчикам событий клавиатуры передается аргумент типа KeyEventArgs, содержащий целый ряд свойств, в том числе:

- Key, ImeProcessedKey, DeadCharProcessedKey, SystemKey – четыре свойства, принадлежащие типу перечисления Key, в котором определены все возможные клавиши. Свойство Key определяет, какая клавиша вызвала генерацию события. Если клавиша обрабатывается или будет обрабатываться редактором метода ввода (Input Method Editor – IME), то можно проверить значение свойства ImeProcessedKey. Если клавиша является слепой в последовательности, то свойство Key будет равно DeadCharProcessed, тогда как реальную клавишу можно получить из свойства DeadCharProcessedKey. Если нажата системная клавиша, например Alt, то Key будет равно System, а сама клавиша берется из свойства SystemKey.
- IsUp, IsDown, IsToggled – булевские свойства, сообщающие дополнительную информацию о событии клавиатуры, хотя в некоторых случаях она избыточна. (Раз уж вы обрабатываете событие KeyDown, то точно знаете, что клавиша нажата!) Свойство IsToggled относится к клавишам с фиксируемым переключением состояния, таким как Caps Lock и Scroll Lock.

СОВЕТ

Для получения информации о состоянии клавиатуры в любой момент времени, а не только внутри обработчика события от нее можно воспользоваться статическим классом `System.Windows.Input` и его свойством `PrimaryDevice` (типа `KeyboardDevice`).

- `KeyStates` – свойство типа `KeyStates`, битового перечисления, состоящего из произвольной комбинации битов `None`, `Down` и `Toggled`. Эти значения отображаются на свойства `IsUp`, `IsDown` и `IsToggled` соответственно. Поскольку `Toggled` иногда комбинируется с `Down`, остерегайтесь определять значение `KeyStates` с помощью простой проверки на равенство. Лучше всего пользоваться свойствами `IsXXX`.
- `IsRepeat` – булевское свойство, равное `true`, когда нажатие клавиши повторяется. Например, так происходит, когда вы удерживаете нажатой пробельную клавишу и получаете лавину событий `KeyDown`. Свойство `IsRepeat` будет содержать `true` для всех событий `KeyDown`, кроме самого первого.
- `KeyboardDevice` – свойство типа `KeyboardDevice`, которое позволяет работать с клавиатурой на более низком уровне, например узнать, какие клавиши сейчас нажаты, или потребовать передать фокус конкретному элементу.

Одна из причин для обращения к классу `KeyboardDevice` – получение его свойства `Modifiers` типа `ModifierKeys` (еще одно перечисление). Оно показывает, какие клавиши нажаты одновременно с основной. Возможны следующие значения: `None`, `Alt`, `Control`, `Shift` и `Windows`. Это битовое перечисление, поэтому не следует проводить проверку на равенство, если только вы не заинтересованы в точной комбинации модификаторов. Например, в следующем коде проверяется, что нажаты клавиши Alt и A, но при этом не исключается нажатие комбинаций Alt+Shift+A, Alt+Ctrl+A и т. д.:

```
protected override void OnKeyDown(KeyEventArgs e)
{
  if ((e.KeyboardDevice.Modifiers & ModifierKeys.Alt) ==
       ModifierKeys.Alt && (e.Key == Key.A || e.SystemKey == Key.A))
  {
    // Нажато сочетание Alt+A, возможно, с Ctrl, Shift или Windows
  }
  base.OnKeyDown(e);
}
```

А в следующем коде проверяется, что нажата в точности комбинация Alt+A без каких-либо дополнительных модификаторов:

```
protected override void OnKeyDown(KeyEventArgs e)
{
  if (e.KeyboardDevice.Modifiers == ModifierKeys.Alt
      && (e.Key == Key.A || e.SystemKey == Key.A))
  {
```

```
      // Нажато сочетание Alt+A и только Alt+A
    }
    base.OnKeyDown(e);
  }
```

FAQ

Как узнать, какая из клавиш Alt, Ctrl и Shift нажата: левая или правая?

В перечислении `Key` имеются следующие значения: `LeftAlt` и `RightAlt`, `LeftCtrl` и `RightCtrl`, `LeftShift` и `RightShift`. Но поскольку клавиша Alt обычно считается системной, то она может определяться как `System`, маскируя тем самым, какая из двух клавиш Alt в действительности нажата. К счастью, можно воспользоваться методом `IsKeyDown` класса `KeyboardDevice` (или `IsKeyUp` либо `IsKeyToggled`), чтобы узнать о состоянии конкретной клавиши, например левой или правой Alt. В следующем коде проверяется нажатие комбинации [левая]Alt+A:

```
protected override void OnKeyDown(KeyEventArgs e)
{
  if (e.KeyboardDevice.Modifiers == ModifierKeys.Alt
      && (e.Key == Key.A || e.SystemKey == Key.A)
      && e.KeyboardDevice.IsKeyDown(Key.LeftAlt))
  {
    // Нажато LeftAlt+A
  }
  base.OnKeyDown(e);
}
```

Иногда в этих событиях можно запутаться, но настоящие трудности при работе с клавиатурой обычно возникают, когда речь заходит о *фокусе* ввода. (Проблема еще больше осложняется при взаимодействии с технологиями, отличными от WPF, но это уже тема главы 19 «Интероперабельность с другими технологиями».) Элемент `UIElement` получает события клавиатуры, только если владеет фокусом. Указать, может ли некоторый элемент получать фокус, позволяет булевское свойство `Focusable`, по умолчанию равное `true`. При изменении значения этого свойства возникает событие `FocusableChanged`.

В классе `UIElement` определено еще много свойств и событий, относящихся к фокусу клавиатуры. Отметим из них свойство `IsKeyboardFocused`, которое сообщает, принадлежит ли фокус текущему элементу, и свойство `IsKeyboardFocusWithin`, сообщающее то же самое, но в отношении не только текущего элемента, но и его потомков. (Эти свойства доступны только для чтения; чтобы передать фокус клавиатуры, пользуйтесь методами `Focus` или `MoveFocus`.) Об изменении этих свойств уведомляют события `IsKeyboardFocusedChanged`, `IsKeyboardFocusWithinChanged`, `GotKeyboardFocus`, `LostKeyboardFocus`, `PreviewGotKeyboardFocus` и `PreviewLostKeyboardFocus`.

События мыши

Все подклассы UIElement поддерживают следующие основные события мыши:

- MouseEnter и MouseLeave
- MouseMove и PreviewMouseMove
- MouseLeftButtonDown, MouseRightButtonDown, MouseLeftButtonUp, MouseRightButtonUp и более общие MouseDown и MouseUp, а также Preview-версии всех шести событий
- MouseWheel и PreviewMouseWheel

События MouseEnter и MouseLeave можно использовать для создания эффекта ролловера, хотя более предпочтительно использовать триггер со свойством IsMouseOver.

В подклассах UIElement имеется также свойство IsMouseDirectlyOver (и соответствующее ему событие IsMouseDirectlyOverChanged), которое позволяет исключить дочерние элементы. Оно используется в тех редких случаях, когда вы точно знаете, с каким визуальным деревом работаете.

FAQ

А где же событие для обработки нажатия средней кнопки мыши?

Эту информацию можно получить с помощью обобщенных событий MouseDown и MouseUp (или их Preview-версий). Объект EventArgs, передаваемый их обработчиком, содержит свойства, показывающие, какая их следующих кнопок была нажата или отпущена: LeftButton, RightButton, MiddleButton, XButton1 или XButton2.

СОВЕТ

Если вы не хотите, чтобы элемент генерировал события мыши (или блокировал события мыши, генерируемые лежащими под ним элементами), то можете присвоить значение false его свойству IsHitTestVisible.

ПРЕДУПРЕЖДЕНИЕ

Прозрачные области генерируют события мыши, но null-области – нет!

Хотя и можно рассчитывать на то, что установка для свойства IsHitTestVisible значения false подавит события мыши, но условия, при которых эти события вообще генерируются, довольно запутанны. Если свойство Visibility элемента равно Collapsed, то события мыши подавляются, но установка для свойства Opacity значения 0 не влияет на генерацию событий. Еще одна тонкость касается областей, для которых любое из свойств Background, Fill или Stroke равно null.

В таких областях события мыши не генерируются. Однако, если явно присвоить любому из свойств Background, Fill или Stroke значение Transparent (или любой другой цвет), то в такой области события мыши *будут* генерироваться. (null-кисть внешне неотличима от прозрачной (Transparent) кисти, но с точки зрения проверки положения указателя мыши ведет себя иначе.)

Класс MouseEventArgs

Обработчикам всех вышеупомянутых событий мыши (кроме IsMouseDirectlyOverChanged) передается объект класса MouseEventArgs. В нем есть пять свойств типа MouseButtonState, содержащих информацию обо всех потенциально возможных нажатиях кнопок мыши: LeftButton, RightButton, MiddleButton, XButton1 и XButton2. MouseButtonState – это перечисление с двумя значениями: Pressed и Released. В классе MouseEventArgs определен также метод GetPosition, который возвращает структуру Point со свойствами X и Y, отражающими точные координаты указателя мыши.

GetPosition – это метод, а не просто свойство, поскольку он позволяет получить позицию указателя мыши несколькими способами: относительно левого верхнего угла экрана или левого верхнего угла произвольного нарисованного элемента UIElement. Чтобы узнать координаты относительно экрана, передайте в качестве единственного параметра null. А для получения координат относительно элемента передайте в качестве параметра интересующий вас элемент.

Обработчикам событий MouseWheel и PreviewMouseWheel передается объект класса Mouse**Wheel**EventArgs, производного от MouseEventArgs. Этот класс добавляет целочисленное свойство Delta, показывающее, на какой угол колесико мыши повернулось с момента последнего события. Обработчикам всех 12 событий семейства MouseUp/MouseDown передается объект класса Mouse**Button**EventArgs, еще одного подкласса MouseEventArgs. Этот класс добавляет свойство ChangedButton, которое сообщает, какая кнопка изменила состояние (значение принадлежит перечислению MouseButton); свойство ButtonState, которое информирует, нажата кнопка или отпущена; и свойство ClickCount.

Свойство ClickCount показывает, сколько раз подряд была нажата кнопка мыши, причем ведется подсчет нажатий, промежуток времени между которыми не превышает системного параметра, описывающего скорость выполнения двойного щелчка (задается на Панели управления). Класс Button генерирует событие Click, обрабатывая низкоуровневое событие MouseLeftButtonDown, а его базовый класс Control генерирует событие MouseDoubleClick, сравнивая значение ClickCount с 2 в обработчике MouseLeftButtonDown, и событие PreviewMouseDoubleClick, делая то же самое в обработчике PreviewMouseLeftButtonDown. Имея такую поддержку, вы легко сможете реагировать и на другие действия пользователя, например на тройное нажатие, двойное нажатие средней кнопки и т. д.

ПРЕДУПРЕЖДЕНИЕ

Панель Canvas генерирует свои собственные события мыши только в области, определяемой ее свойствами Width и Height!

Легко забыть о том, что по умолчанию ширина `Width` и высота `Height` панели `Canvas` равны `0`, так как ее дочерние элементы рисуются вне границ холста. Но события мыши самой панели `Canvas` (кроме событий, всплывающих от потомков) генерируются только в области, занятой прямоугольником размером `Width`×`Height` (и только при условии, что свойство `Background` не равно `null`). Поэтому по умолчанию события мыши уровня `Canvas` генерируются только ее дочерними элементами.

Перетаскивание

Во всех подклассах `UIElement` определены события для работы с перетаскиванием:

- `DragEnter`, `DragOver`, `DragLeave`, а также `PreviewDragEnter`, `PreviewDragOver` и `PreviewDragLeave`
- `Drop` и `PreviewDrop`
- `QueryContinueDrag` и `PreviewQueryContinueDrag`

Это перетаскивание элемента в буфер обмена и бросание содержимого буфера на элемент в стиле Win32, а *не* перетаскивание и бросание самих элементов. Элемент может принять участие в перетаскивании, установив значение `true` для свойства `AllowDrop`.

Обработчикам событий из первых двух наборов передается объект типа `DragEventArgs`, содержащий следующие свойства и методы:

- `GetPosition` – такой же метод, как в классе `MouseEventArgs`
- `Data` – свойство типа `IDataObject`, представляющее перетаскиваемый или бросаемый объект буфера обмена Win32
- `Effects` и `AllowedEffects` – битовое перечисление `DragDropEffects`, допускающее произвольную комбинацию флагов `Copy`, `Move`, `Link`, `Scroll`, `All` и `None`
- `KeyStates` – еще одно битовое перечисление (`DragDropKeyStates`), показывающее, какие кнопки мыши или клавиши-модификаторы были нажаты во время перетаскивания или бросания: `LeftMouseButton`, `RightMouseButton`, `MiddleMouseButton`, `ShiftKey`, `ControlKey`, `AltKey` или `None`

События `QueryContinueDrag` и `PreviewQueryContinueDrag` генерируются, если во время перетаскивания изменяется состояние клавиатуры или какой-нибудь кнопки мыши. Это позволяет обработчику без труда отменить всю операцию. Обработчикам этих событий передается объект класса `QueryContinueDragEventArgs`, имеющий следующие свойства:

- `KeyStates` – аналогично одноименному свойству класса `DragEventArgs`

- `EscapePressed` – отдельное булевское свойство, показывающее, была ли нажата клавиша Esc
- `Action` – свойство, которое обработчик может установить, чтобы определить судьбу операции перетаскивания; принадлежит перечислению `DragAction` и принимает значение `Continue`, `Drop` или `Cancel`

СОВЕТ

Для получения информации о состоянии мыши *почти* в любой момент времени, а не только внутри обработчиков ее событий можно воспользоваться статическим классом `System.Windows.Input.Mouse`. Чего нельзя сделать – так это получить достоверную позицию указателя мыши от статического метода `Mouse.GetPosition` во время перетаскивания. Вместо этого приходится либо вызывать метод `GetPosition` объекта `DragEventArgs`, переданного обработчику события, либо, минуя обработчики событий, с помощью технологии PInvoke вызвать функцию Win32 API `GetCursorPos`, которая даст правильные координаты.

Захват мыши

Предположим, что нужно поддержать перетаскивание и бросание самих элементов `UIElement`, а не объектов буфера обмена. Легко представить себе, как это можно реализовать с помощью событий `MouseLeftButtonDown`, `MouseMove` и `MouseLeftButtonUp`. В начале операции перетаскивания можно установить некую булевскую переменную в обработчике события `MouseLeftButtonDown` элемента, потом в обработчике `MouseMove` перемещать элемент, так чтобы он оставался под указателем мыши, если эта переменная равна `true`, а в обработчике `MouseLeftButtonUp` сбросить переменную, обозначив конец перетаскивания. Однако выясняется, что эта схема не так уж хороша, потому что пользователь может двигать мышь слишком быстро или ее указатель может оказаться под другим элементом, в результате чего указатель потеряет связь с элементом, который вы пытаетесь перетащить.

К счастью, WPF позволяет любому элементу `UIElement` в любой момент *захватить* или *освободить* мышь. Когда элемент захватил мышь, он получает все события мыши, даже если указатель оказывается вне занимаемой им области. После освобождения мыши поведение событий возвращается в нормальное русло. Для захвата и освобождения мыши предназначены два метода класса `UIElement`: `CaptureMouse` и `ReleaseMouseCapture`. (И, разумеется, есть ряд свойств и событий, сообщающих о состоянии захвата мыши, точнее, свойства `IsMouseCaptured` и `IsMouseCaptureWithin` и события `GotMouseCapture`, `LostMouseCapture`, `IsMouseCaptureChanged` и `IsMouseCaptureWithinChanged`.)

Поэтому для реализации перетаскивания необходимо захватить мышь в обработчике `MouseLeftButtonDown` и освободить ее в обработчике `MouseLeftButtonUp`. Единственная сложность – придумать оптимальный способ фактического

перемещения элемента в обработчике MouseMove. Выбор зависит от компоновки приложения, но, скорее всего, вы будете применять к перетаскиваемому элементу преобразование в режиме RenderTransform или LayoutTransform.

События стилуса

В WPF имеется специальная поддержка для цифрового пера, или стилуса, применяемого в таких устройствах, как Tablet PC. (Иногда ее называют поддержкой рукописного ввода.) Если приложение специально не поддерживает стилус, то он интерпретируется как обычная мышь и генерирует все относящиеся к мыши события, в частности MouseDown, MouseMove и MouseUp. Такое поведение позволяет использовать стилус в программах, которые не были разработаны специально для Tablet PC.

Но если вы хотите, чтобы пользователю было удобно работать именно со стилусом, то можете организовать взаимодействие с экземпляром класса System.Windows.Input.StylusDevice. Получить его можно тремя способами.

- Воспользоваться свойством StylusDevice объекта MouseEventArgs для доступа к объекту в обработчиках событий мыши. (Если стилуса нет, то это свойство равно null.)
- Воспользоваться статическим классом System.Windows.Input.Stylus и его свойством CurrentStylusDevice – так можно получить доступ к стилусу в любой момент. (Если стилуса нет, то это свойство тоже равно null.)
- Обрабатывать события, специфичные для стилуса.

Все эти средства применимы не только к перьевым, но и к сенсорным дигитайзерам.

FAQ



Я и так могу получить данные стилуса, если буду работать с ним, как с мышью. Так зачем еще какая-то дополнительная информация?

Перьевой и сенсорный дигитайзеры поддерживают два аспекта, отсутствующие у обычной мыши (сейчас мы не говорим о мультисенсорном вводе, это тема следующего раздела): чувствительность к силе нажатия и более высокую разрешающую способность.

Для приложения с рукописным вводом или предназначенного для рисования и то и другое может сделать процесс ввода данных более естественным, чем с помощью мыши. Стилус также позволяет выполнять недоступные для мыши действия, что обеспечивает набор свойств и событий, обсуждаемых ниже в этом разделе. Кроме того, поскольку система может одновременно распознавать несколько стилусов, открывается возможность писать код, ориентированный на мультисенсорный ввод, – при работе в Windows 7 с установленным пакетом WPF 3.5 SP1.

Класс StylusDevice

Класс StylusDevice содержит ряд свойств, в том числе:

- Inverted – булевское значение, показывающее, что стилус используется как ластик (то есть экрана касается его обратный конец).
- InAir – булевское значение, показывающее, касается ли стилус экрана. Это важно, потому что некоторые устройства регистрируют его перемещение даже без касания при условии, что стилус находится достаточно близко к экрану.
- StylusButtons – коллекция объектов типа StylusButton. В отличие от мыши, у стилуса нет фиксированного списка кнопок. В каждом объекте StylusButton имеется строковое свойство Name и идентификатор Guid, а также свойство StylusButtonState, принимающее одно из значений: Up или Down.
- TabletDevice – свойство типа System.Windows.Input.TabletDevice, предоставляющее детальную информацию о текущем оборудовании и возможностях стилуса (в частности, чувствительность к силе нажатия или поддержка перемещений без касания). Свойство Type равно Stylus для перьевого и Touch – для сенсорного дигитайзера.

В классе StylusDevice имеется метод GetPosition, работающий так же, как его аналог для мыши. Но дополнительно есть более подробный метод GetStylusPoints, который возвращает коллекцию объектов StylusPoint. В каждом объекте StylusPoint имеются следующие свойства:

- X – абсцисса точки касания стилуса относительно элемента, на котором он находится.
- Y – ордината точки касания стилуса относительно элемента, на котором он находится.
- PressureFactor – значение от 0 до 1, показывающее давление, приложенное к стилусу в момент регистрации точки. Чем больше значение, тем сильнее нажатие (если оборудование вообще поддерживает чувствительность к силе нажатия). Если чувствительность к силе нажатия не поддерживается, то PressureFactor будет равно 0.5.

Метод GetStylusPoints возвращает именно коллекцию точек (и уровней давления), а не одно значение, из-за высокой разрешающей способности стилуса. Это означает, что между двумя событиями MouseMove может быть обнаружено и зарегистрировано много отдельных перемещений.

События

К стилусу относятся следующие события:

- StylusEnter и StylusLeave
- StylusMove и PreviewStylusMove
- StylusInAirMove и PreviewStylusInAirMove
- StylusDown, StylusUp, PreviewStylusDown и PreviewStylusUp

- StylusButtonDown, StylusButtonUp, PreviewStylusButtonDown и PreviewStylusButtonUp
- StylusSystemGesture и PreviewStylusSystemGesture
- StylusInRange, StylusOutOfRange, PreviewStylusInRange и PreviewStylusOutOfRange
- GotStylusCapture и LostStylusCapture

Обработчикам этих событий передается объект класса StylusEventArgs, свойство StylusDevice которого дает доступ к объекту StylusDevice. Для удобства в нем определены также члены InAir, Inverted, GetPosition и GetStylusPoints, обертывающие одноименные члены класса StylusDevice.

Некоторым обработчикам в качестве аргумента передается объект одного из подклассов StylusEventArgs:

- **StylusDownEventArgs** – передается обработчикам событий StylusDown и PreviewStylusDown; добавляет целочисленное свойство TapCount, аналогичное свойству ClickCount в событиях мыши.
- **StylusButtonEventArgs** – передается обработчикам событий StylusButtonDown, StylusButtonUp и их Preview-версий; добавляет свойство StylusButton, описывающее нажатую кнопку.
- **StylusSystemGestureEventArgs** – передается обработчикам событий StylusSystemGesture и PreviewStylusSystemGesture; добавляет свойство SystemGesture, принадлежащее типу перечисления SystemGesture и принимающее следующие значения: Tap, RightTap, TwoFingerTap, Drag, RightDrag, Flick, HoldEnter, HoldLeave, HoverEnter, HoverLeave, None.

СОВЕТ

В WPF определен объект Stroke (росчерк), с помощью которого можно визуально представить информацию, хранящуюся в коллекции StylusPoints, и элемент InkPresenter, содержащий коллекцию объектов Stroke. Во многих сценариях рисования и рукописного ввода можно также использовать элемент InkCanvas, описанный в главе 11 «Изображения, текст и другие элементы управления», который основан на использовании InkPresenter. В InkCanvas встроена возможность работы со стилусом, если таковой имеется, а также средства для сбора и отображения росчерков. При использовании этого элемента вам вообще не придется обрабатывать события стилуса самостоятельно!

Мультисенсорные события

При работе в ОС Windows 7 или более поздней, оснащенной оборудованием для мультисенсорного ввода, можно воспользоваться новыми событиями, добавленными в WPF 4. Их можно разбить на две категории: простые события касания и события манипулирования более высокого уровня.

Хотя такие мультисенсорные события, как события стилуса, можно представить в виде событий мыши, обратное неверно. Нельзя получить от мыши со-

бытие касания в одной точке, как если бы речь шла о касании экрана пальцем, не проделав дополнительную работу по эмуляции сенсорного устройства.

СОВЕТ

Если вы хотите эмулировать мультисенсорный (или даже простой сенсорный) ввод на «обычном» компьютере, то можете воспользоваться комплектом MultiPoint Mouse SDK (*http://microsoft.com/multipoint/mouse-sdk*), который позволяет одновременно подключить к компьютеру до 25 мышей! Однако этого недостаточно. Еще предстоит раскрыть функциональность MultiPoint в виде специализированного *сенсорного устройства*; соответствующая техника описана в статье по адресу *http://blogs.msdn.com/ansont/archive/2010/01/30/custom-touch-devices.aspx*.

Простые события касания

Простые события касания во многом похожи на события мыши:

- `TouchEnter` и `TouchLeave`
- `TouchMove` и `PreviewTouchMove`
- `TouchDown`, `TouchUp`, `PreviewTouchDown` и `PreviewTouchUp`
- `GotTouchCapture` и `LostTouchCapture`

Когда экрана касаются несколько пальцев одновременно, генерируются независимые события для каждого пальца. Кроме того, благодаря описанной выше поддержке стилуса для первого пальца генерируются также события мыши.

Обработчикам событий касания передается объект класса `TouchEventArgs`, содержащий следующие члены:

- `GetTouchPoint` – метод, возвращающий объект `TouchPoint`. Этот объект представляет точку касания в системе координат, связанной с элементом, которому она принадлежит. Аналог метода `GetPosition` для событий мыши.
- `GetIntermediateTouchPoints` – метод, возвращающий коллекцию объектов `TouchPoint` в координатах элемента, собранных за время, прошедшее между текущим и предыдущим событиями касания. Аналог метода `GetStylusPoints` для событий стилуса.
- `TouchDevice` – свойство, возвращающее объект `TouchDevice`.

В классе `TouchPoint` имеется не только свойство `Position`, но и `Size`, показывающее, какая часть пальца находится в контакте с экраном, а также свойство `Bounds`, точно описывающее область контакта. Кроме того, он дает информацию, которая уже известна в контексте обработчика события, но может оказаться полезной в других контекстах, – устройство `TouchDevice` и совершенное действие `Action`, которое может принимать следующие значения: `Down`, `Move`, `Up` (из перечисления `TouchAction`).

С каждым нажатием пальцем ассоциирован отдельный объект `TouchDevice`, идентифицируемый целочисленным свойством `Id`. Этот идентификатор (или

сам объект TouchDevice) можно использовать для отслеживания пальцев во время обработки событий.

В листинге 6.6 события TouchDown, TouchMove и TouchUp используются для создания картинок с изображением пальцев (но не самих отпечатков пальцев!) в местах их соприкосновения с экраном. Это застраничный код для следующего простого окна, содержащего элемент Canvas с именем canvas:

```
<Window x:Class="TouchEvents.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Touch Events">
  <Canvas Name="canvas">
    <Canvas.Background>
      <LinearGradientBrush>
        <GradientStop Color="Black"/>
        <GradientStop Color="Red" Offset="1"/>
      </LinearGradientBrush>
    </Canvas.Background>
  </Canvas>
</Window>
```

Результат показан на рис. 6.2.

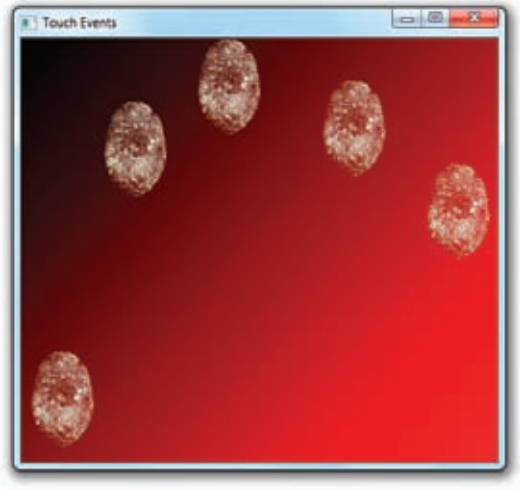


***Рис. 6.2.** При касании экрана пятью пальцами мы видим пять картинок с изображением пальца в местах касания*

Листинг 6.6. MainWindow.xaml.cs – обработка событий TouchDown, TouchMove и TouchUp

```
using System;
using System.Collections.Generic;
using System.Windows;
```

```
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace TouchEvents
{
  public partial class MainWindow : Window
  {
    // Сопоставляем изображения с объектами TouchDevice
    Dictionary<TouchDevice, Image> fingerprints =
        new Dictionary<TouchDevice, Image>();

    public MainWindow()
    {
        InitializeComponent();
    }

    protected override void OnTouchDown(TouchEventArgs e)
    {
      base.OnTouchDown(e);

      // Захватываем данное сенсорное устройство
      canvas.CaptureTouch(e.TouchDevice);

      // Создаем новое изображение для этого касания
      Image fingerprint = new Image { Source = new BitmapImage(
          new Uri("pack://application:,,,/fingerprint.png")) };

      // Перемещаем изображение в точку касания
      TouchPoint point = e.GetTouchPoint(canvas);
      fingerprint.RenderTransform = new TranslateTransform(
        point.Position.X, point.Position.Y);

      // Запоминаем изображение и помещаем его на холст
      fingerprints[e.TouchDevice] = fingerprint;
      canvas.Children.Add(fingerprint);
    }

    protected override void OnTouchMove(TouchEventArgs e)
    {
      base.OnTouchMove(e);

      if (e.TouchDevice.Captured == canvas)
      {
        // Находим нужное изображение
        Image fingerprint = fingerprints[e.TouchDevice];
        TranslateTransform transform =
          fingerprint.RenderTransform as TranslateTransform;

        // Перемещаем его в новое место
        TouchPoint point = e.GetTouchPoint(canvas);
        transform.X = point.Position.X;
```

```
            transform.Y = point.Position.Y;
          }
        }

        protected override void OnTouchUp(TouchEventArgs e)
        {
          base.OnTouchUp(e);

          // Освобождаем захваченное устройство
          canvas.ReleaseTouchCapture(e.TouchDevice);

          // Удаляем изображение с холста и из словаря
          canvas.Children.Remove(fingerprints[e.TouchDevice]);
          fingerprints.Remove(e.TouchDevice);
        }
      }
    }
```

Эта программа работает по принципу схемы перетаскивания и бросания элементов, описанной в разделе «События мыши», только элемент создается по событию `TouchDown`, а удаляется по событию `TouchUp`. И мы решили не присоединять обработчики всех трех событий, а переопределить соответствующие методы `On`*`XXX`* класса `Window`.

В методе `OnTouchDown` программа захватывает сенсорное устройство, чтобы операция перетаскивания работала надежно. Но, в отличие от клавиатуры, мыши и стилуса, один элемент может захватить сразу несколько сенсорных устройств. В данном случае холст `Canvas` захватывает все такие устройства. Изображение `Image` создается из внедренного ресурса с помощью синтаксиса, который мы рассмотрим в главе 12 «Ресурсы», позиционируется посредством преобразования `TranslateTransform`, после чего помещается на холст и добавляется в словарь, к которому обращаются и другие обработчики. Ключом словаря является сам объект `TouchDevice`.

Метод `OnTouchMove` находит изображение, соответствующее текущему объекту `TouchDevice`, и перемещает его в текущую точку `TouchPoint`. Он проверяет, что событие принадлежит одному из устройств `TouchDevice`, захваченных холстом. Метод `OnTouchUp` освобождает захваченные сенсорные устройства, после чего удаляет объект `Image` с холста и из словаря.

СОВЕТ

В версии Silverlight 4 событий касания нет. Если вы хотите написать код, который поддерживал бы мультисенсорный ввод и работал как в WPF, так и в Silverlight, то можете воспользоваться низкоуровневым событием `FrameReported`, которое присутствует в обеих системах. Событие `FrameReported` определено в статическом классе `System.Windows.Input.Touch` и сообщает о точках касания `TouchPoint` для всего приложения в целом. Это не маршрутизируемое событие; выяснять, где произошло касание, придется самостоятельно.

Качество работы этой программы зависит от имеющегося у вас оборудования. Мой мультисенсорный нетбук поддерживает только два одновременных касания, поэтому я не могу получить больше двух отпечатков пальцев за раз.

События манипулирования, описывающие сдвиг, поворот и масштабирование

Мультисенсорный ввод обычно применяется пользователями для сдвига, поворота и масштабирования элементов. Тут все просто, так как эти действия точно отображаются на преобразования `TranslateTransform`, `RotateTransform` и `ScaleTransform` соответственно. А вот определить, *когда* эти преобразования следует применять и *с какими параметрами*, куда сложнее.

Скользящее движение одного пальца, обычно обозначающее сдвиг, распознать относительно просто, но определить, что пользователь произвел вращательное движение двумя пальцами или жест, обозначающий масштабирование, с помощью описанных выше событий очень трудно. К тому же несогласованность алгоритмов, применяемых разными разработчиками для распознавания жестов, только раздражала бы пользователей.

На наше счастье, WPF предоставляет высокоуровневые события манипулирования, позволяющие без труда поддержать сдвиг, поворот и масштабирование. Вот перечень основных событий такого рода:

- `ManipulationStarting` и `ManipulationStarted`
- `ManipulationDelta`
- `ManipulationCompleted`

В этих событиях передается информация, собранная с независимых, одновременно обновляемых сенсорных устройств и представленная в удобном для работы виде. Чтобы элемент мог получать такие события, свойству `IsManipulationEnabled` его самого или его родителя нужно присвоить значение `true` и не обрабатывать низкоуровневые события касания.

Использование событий манипулирования

Когда для первого пальца генерируется событие `TouchDown`, возникает сначала событие `ManipulationStarting`, а затем `ManipulationStarted`. Для каждого события `TouchMove` генерируется событие `ManipulationDelta`, а после того, как все пальцы подняты, – событие `ManipulationCompleted`. События `ManipulationStarting` и `ManipulationStarted` дают возможность настроить различные аспекты манипулирования, ограничить допустимые манипуляции или вообще отменить операцию.

Событие `ManipulationDelta` сообщает подробную информацию об ожидаемом параллельном переносе, повороте или масштабировании элемента; ее можно напрямую использовать в соответствующем геометрическом преобразовании. Информация передается в следующих свойствах класса `ManipulationDelta`:

- `Translation` – свойство типа `Vector`, содержащее значения `X` и `Y`
- `Scale` – еще одно свойство типа `Vector`

- `Rotation` – свойство типа `double`, определяющее угол поворота в градусах
- `Expansion` – свойство типа `Vector`, которое при наличии `Scale` можно считать избыточным; сообщает разницу в размерах, выраженную в абсолютных независимых от устройства пикселах, а не в терминах коэффициентов масштабирования

Заметим еще, что объект `ManipulationDeltaEventArgs`, передаваемый обработчику события `ManipulationDelta`, содержит *два* свойства типа `ManipulationDelta`: `DeltaManipulation` (сообщает об изменениях, произошедших с момента последней генерации этого события) и `CumulativeManipulation` (сообщает об изменениях, произошедших с момента генерации события `ManipulationStarted`). Так что, какой бы способ использования данных вы ни выбрали, система найдет, чем вас порадовать!

В листинге 6.7 приведен застраничный код для показанного ниже окна `Window`. Он позволяет двигать, поворачивать и масштабировать фотографию с помощью стандартных жестов: движение одним пальцем по прямой, движение по кругу и движение двумя пальцами в разные стороны.

```
<Window x:Class="ManipulationEvents.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Manipulation Events">
  <Canvas Name="canvas" IsManipulationEnabled="True">
    <Image Name="photo" Source="photo.jpg">
      <Image.RenderTransform>
        <MatrixTransform/>
      </Image.RenderTransform>
    </Image>
  </Canvas>
</Window>
```

Результат показан на рис. 6.3.



Рис. 6.3. *Сдвиг, поворот и масштабирование изображения с помощью обработки события ManipulationDelta*

Листинг 6.7. MainWindow.xaml.cs – работа с классом ManipulationDelta для сдвига, поворота и масштабирования

```
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace ManipulationEvents
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
      canvas.ManipulationDelta += Canvas_ManipulationDelta;
    }

    void Canvas_ManipulationDelta(object sender,
      ManipulationDeltaEventArgs e)
    {
      MatrixTransform transform = photo.RenderTransform as MatrixTransform;
      if (transform != null)
      {
        // Применить дельты к матрице, затем воспользоваться
        // созданной матрицей в преобразовании MatrixTransform
        Matrix matrix = transform.Matrix;
        matrix.Translate(e.DeltaManipulation.Translation.X,
                         e.DeltaManipulation.Translation.Y);
        matrix.RotateAt(e.DeltaManipulation.Rotation,
                        e.ManipulationOrigin.X, e.ManipulationOrigin.Y);
        matrix.ScaleAt(e.DeltaManipulation.Scale.X,
                       e.DeltaManipulation.Scale.Y,
                       e.ManipulationOrigin.X, e.ManipulationOrigin.Y);
        transform.Matrix = matrix;
        e.Handled = true;
      }
    }
  }
}
```

Для объекта `Image` с именем `photo` в разметке уже определено преобразование `MatrixTransform`, применяемое в режиме `RenderTransform`, поэтому коду в обработчике `ManipulationDelta` остается только записать в матрицу `Matrix` данные, взятые из объекта `ManipulationDeltaEventArgs`. Методы `RotateAt` и `ScaleAt` используются для задания центра поворота и масштабирования (`e.ManipulationOrigin`).

Манипуляции всегда производятся относительно *контейнера манипулирования*. По умолчанию это элемент, для которого свойство `IsManipulationEnabled=True`; именно поэтому в XAML-разметке для данного примера оно установлено в элементе `Canvas`, а не в `Image`. Назначить контейнером манипулирования

можно любой элемент, для этого достаточно обработать его событие ManipulationStarting и записать в свойство ManipulationStartingEventArgs.ManipulationContainer ссылку на данный элемент.

Добавление инерции

События манипулирования поддерживают также придание объектам инерции, благодаря которой их движение будет постепенно замедляться, а не сразу останавливаться по завершении жеста. При этом ощущение от жеста получается более реалистичным и открывается возможность поддержать «щелчки», так чтобы расстояние, на которое перемещен объект, зависело от скорости щелчка.

Чтобы включить инерцию, следует обработать событие ManipulationInertiaStarting — в дополнение к другим событиям манипулирования. Именно ManipulationInertiaStarting — а не ManipulationCompleted — первое событие манипулирования, которое генерируется после убирания всех пальцев с экрана. В обработчике ManipulationInertiaStarting вы можете решить, что именно поддерживать, для этого следует установить какие-то из свойств ManipulationInertiaStartingEventArgs.TranslationBehavior, ManipulationInertiaStartingEventArgs.RotationBehavior и ManipulationInertiaStartingEventArgs.ExpansionBehavior. В результате система продолжит генерировать события ManipulationDelta (в которых свойство ManipulationDeltaEventArgs.IsInertial будет равно true) до тех пор, пока «трение» не заставит объект остановиться, а в этот момент будет сгенерировано событие ManipulationCompleted. (Если в обработчике события ManipulationInertiaStarting ничего не делать, то событие ManipulationCompleted генерируется сразу после него.)

Ниже приведен перечень свойств, которые можно установить для настройки инерции при сдвиге, повороте или масштабировании:

- `TranslationBehavior` — DesiredDisplacement, DesiredDeceleration, InitialVelocity
- `RotationBehavior` — DesiredRotation, DesiredDeceleration, InitialVelocity
- `ExpansionBehavior` — DesiredExpansion, DesiredDeceleration, InitialRadius, InitialVelocity

Обычно необходимо задавать только DesiredDeceleration (желательное замедление) или специфические для конкретного поведения свойства DesiredDisplacement (желательный сдвиг), DesiredRotation (желательный угол поворота) либо DesiredExpansion (желательный коэффициент масштабирования). Последние три свойства нужны для того, чтобы элемент не переместился слишком далеко. По умолчанию InitialVelocity и InitialRadius инициализируются текущими значениями, что обеспечивает плавный переход. Можно получить различные скорости в момент возникновения события ManipulationInertiaStarting, опросив объект ManipulationInertiaStartingEventArgs.InitialVelocities, в котором есть свойства LinearVelocity, AngularVelocity и ExpansionVelocity.

В листинге 6.8 код из листинга 6.7 модифицирован, чтобы поддержать инерцию.

Листинг 6.8. MainWindow.xaml.cs – работа с классами ManipulationDelta и ManipulationInertiaStarting для сдвига, поворота и масштабирования с инерцией

```
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace ManipulationEvents
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
      canvas.ManipulationDelta += Canvas_ManipulationDelta;
      canvas.ManipulationInertiaStarting += Canvas_ManipulationInertiaStarting;
    }

    void Canvas_ManipulationInertiaStarting(object sender,
           ManipulationInertiaStartingEventArgs e)
    {
      e.TranslationBehavior.DesiredDeceleration = 0.01;
      e.RotationBehavior.DesiredDeceleration = 0.01;
      e.ExpansionBehavior.DesiredDeceleration = 0.01;
    }

    void Canvas_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
    {
      MatrixTransform transform = photo.RenderTransform as MatrixTransform;
      if (transform != null)
      {
        // Применить дельты к матрице, затем воспользоваться
        // созданной матрицей в преобразовании MatrixTransform
        Matrix matrix = transform.Matrix;
        matrix.Translate(e.DeltaManipulation.Translation.X,
                         e.DeltaManipulation.Translation.Y);
        matrix.RotateAt(e.DeltaManipulation.Rotation,
                        e.ManipulationOrigin.X, e.ManipulationOrigin.Y);
        matrix.ScaleAt(e.DeltaManipulation.Scale.X,
                       e.DeltaManipulation.Scale.Y,
                       e.ManipulationOrigin.X, e.ManipulationOrigin.Y);
        transform.Matrix = matrix;
        e.Handled = true;
      }
    }
  }
}
```

Необходимо следить, чтобы элемент не ушел полностью за пределы экрана, особенно если включена инерция. Можно воспользоваться событием Manipu-

lationBoundaryFeedback, чтобы получать уведомления о том, что элемент достиг границы контейнера манипулирования, и воспрепятствовать его перемещению.

СОВЕТ

WPF предлагает простой способ заставить окно колебаться, когда что-то проходит через его границу, – как в эффекте прокрутки за конец списка, который сделался популярным благодаря iPhone. Чтобы этого добиться, нужно в обработчике события ManipulationDelta вызвать метод ReportBoundaryFeedback полученного объекта ManipulationDeltaEventArgs. Тогда будет сгенерировано событие ManipulationBoundaryFeedback, которое будет обработано элементом Window, и результатом станет желаемый эффект.

FAQ

В классе ManipulationDeltaEventArgs есть методы Complete и Cancel. В чем между ними разница?

Метод Complete останавливает манипуляцию (как прямую, так и инерционную). Метод Cancel тоже останавливает манипуляцию, но передает данные о касании событиям мыши, так что поведение может быть частично продолжено для элементов, умеющих работать с мышью, но не с сенсорными устройствами.

В листинге 6.9 инерция вращения используется для реализации «колеса фортуны», показанного на рис. 6.4. Это застраничный код для следующего окна Window:

```
<Window x:Class="SpinThePrizeWheel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Spin the Prize Wheel">
  <Window.Background>
    <LinearGradientBrush>
      <GradientStop Color="White"/>
      <GradientStop Color="Orange" Offset="1"/>
    </LinearGradientBrush>
  </Window.Background>
  <Grid Name="grid" IsManipulationEnabled="True">
    <Image Name="prizeWheel" RenderTransformOrigin="0.5,0.5"
           Source="prizeWheel.png" Margin="0 30 0 0">
      <Image.RenderTransform>
        <RotateTransform/>
      </Image.RenderTransform>
    </Image>
    <Image Source="arrow.png" VerticalAlignment="Top" Stretch="None"/>
  </Grid>
</Window>
```



Рис. 6.4. *Из-за инерции вращения колесо продолжает крутиться и после завершения жеста, как в некоторых играх*

Листинг 6.9. MainWindow.xaml.cs – реализация колеса фортуны с инерцией

```
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace SpinThePrizeWheel
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
      grid.ManipulationStarting        += Grid_ManipulationStarting;
      grid.ManipulationDelta           += Grid_ManipulationDelta;
      grid.ManipulationInertiaStarting += Grid_ManipulationInertiaStarting;
      grid.ManipulationCompleted       += Grid_ManipulationCompleted;
    }

    void Grid_ManipulationStarting(object sender, ManipulationStartingEventArgs e)
    {
      // Включаем только для поворотов
      e.Mode = ManipulationModes.Rotate;
    }

    void Grid_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
    {
      (prizeWheel.RenderTransform as RotateTransform).Angle +=
        e.DeltaManipulation.Rotation;
    }
```

```
    void Grid_ManipulationInertiaStarting(object sender,
                                          ManipulationInertiaStartingEventArgs e)
    {
      e.RotationBehavior.DesiredDeceleration = 0.001;
    }

    void Grid_ManipulationCompleted(object sender,
                                    ManipulationCompletedEventArgs e)
    {
      // Колесо остановилось. Пора сообщить пользователю,
      // что он выиграл!
    }
  }
}
```

В листинге 6.9 обработчик события `ManipulationStarting` сообщает, что его интересуют только повороты. Это необязательно, потому что обработчик события `ManipulationDelta` только на данные об угле поворота и обращает внимание, но является признаком хорошего тона (да и для производительности полезно). Обработчик `ManipulationDelta` изменяет параметры преобразования `RotateTransform`, увеличивая угол поворота `Angle` на величину `e.DeltaManipulation.Rotation`. Можно было бы вместо этого просто записать в `Angle` значение `e.CumulativeManipulation.Rotation`, но тогда при последующем запуске колесо начало бы вращение с угла 0°, что будет резать глаз и выглядит неестественно.

В обработчике `ManipulationInertiaStarting` колесу придается очень небольшое замедление, так что после прекращения контакта оно будет вращаться довольно долго. Наконец, обработчик события `ManipulationCompleted` – самое подходящее место для определения конечного положения колеса и награждения пользователя.

СОВЕТ

Можно воспользоваться встроенной в элемент `ScrollViewer` поддержкой сдвигов и присвоить свойству `PanningMode` одно из значений `HorizontalOnly`, `VerticalOnly`, `HorizontalFirst`, `VerticalFirst` или `Both`. В классе `ScrollViewer` имеются также свойства `PanningDeceleration` и `PanningRatio`. Последнее используется как коэффициент при вычислении расстояния для реализующего манипуляцию преобразования `TranslateTransform`.

По умолчанию свойство `PanningMode` равно `None`, но некоторые элементы управления WPF задают для своего внутреннего `ScrollViewer` другое значение, более подходящее для стандартного стиля и позволяющее работать с мультисенсорными устройствами без явных действий со стороны программиста.

СОВЕТ

В доступном для скачивания наборе инструментов Surface Toolkit for Windows Touch есть немало превосходных элементов управления WPF для Microsoft Surface, которые оптимизированы для работы с мультисенсорными устройствами. В их число входят как варианты большинства стандартных элементов управления для сенсорного рабочего стола (например, `SurfaceButton` и `SurfaceCheckBox`), так и совершенно новые элементы (в частности, `ScatterView` и `LibraryStack`).

Команды

Хотя эта глава посвящена в основном событиям, важно иметь представление о встроенной в WPF поддержке *команд*, более абстрактной и слабо связанной версии событий. Если события жестко связаны с деталями конкретных действий пользователя (например, нажатие кнопки или выбор элемента `ListBoxItem` из списка), то команды представляют действия независимо от того, как они выглядят в пользовательском интерфейсе. Каноническими примерами служат команды Cut (Вырезать), Copy (Копировать) и Paste (Вставить). В приложениях эти действия часто представляются сразу несколькими способами: пункты `MenuItem` меню `Menu`, пункты `MenuItem` меню `ContextMenu`, кнопки `Button` на панели инструментов `ToolBar`, сочетания клавиш и т. д.

Наличие нескольких представлений таких команд, как Cut, Copy и Paste, можно сравнительно неплохо обработать с помощью событий. Например, можно определить обобщенные обработчики для каждого действия и присоединить их к соответствующим событиям элементов интерфейса (событию `Click` кнопки `Button`, событию `KeyDown` главного окна `Window` и т. д.). Кроме того, нужно будет активировать и деактивировать элементы управления, когда соответствующие им действия недопустимы (например, деактивировать операцию вставки в интерфейсе, когда в буфере обмена ничего нет). Но реализация такой двусторонней связи довольно быстро становится утомительным делом, особенно если вы не хотите жестко зашивать в код список элементов управления, нуждающихся в обновлении.

К счастью, поддержка команд в WPF спроектирована так, чтобы максимально упростить работу в подобных ситуациях. Предлагаемый механизм уменьшает объем написанного вами кода (а иногда позволяет вообще не писать процедурный код) и дает вам возможность более гибко изменять пользовательский интерфейс, не нарушая стоящую за ним логику. Команды – не новинка, появившаяся только в WPF; в прежних технологиях, в частности в библиотеке классов Microsoft Foundation Class Library (MFC), тоже существовал подобный механизм. Разумеется, даже если вы знакомы с MFC, изучить уникальные особенности команд в WPF все равно придется.

Мощь механизма команд основывается на трех основных особенностях:

- В WPF определено много встроенных команд.

- В команды встроена автоматическая поддержка жестов ввода (например, сочетаний клавиш).
- Встроенное поведение некоторых элементов управления WPF уже ориентировано на те или иные команды.

Встроенные команды

Командой называется любой объект, реализующий интерфейс ICommand (из пространства имен System.Windows.Input), в котором объявлены три простых члена:

- Execute – метод, который выполняет характерную для команды логику
- CanExecute – метод, который возвращает true, если команда активирована, и false, если она деактивирована
- CanExecuteChanged – событие, которое генерируется при изменении значения CanExecute

Чтобы создать команды Cut, Copy и Paste, нужно было бы сделать следующее: написать три класса, реализующих интерфейс ICommand; решить, куда поместить их экземпляры (быть может, в статических полях класса главного окна); вызывать метод Execute из соответствующих обработчиков событий (когда CanExecute возвращает true) и в обработчике события CanExecuteChanged переключать свойство IsEnabled соответствующих элементов пользовательского интерфейса. Получается немногим лучше, чем работа с самими событиями.

К счастью, в такие элементы управления, как Button, CheckBox и MenuItem, уже встроена логика, позволяющая им взаимодействовать с любой командой от вашего имени. В этих элементах имеется простое свойство Command (типа ICommand). Если оно установлено, то элемент автоматически вызывает метод команды Execute (если CanExecute возвращает true) всякий раз, как генерирует событие Click. Кроме того, свойство IsEnabled автоматически синхронизируется со значением, возвращаемым методом CanExecute, – для этого используется событие CanExecuteChanged. Поскольку вся эта функциональность становится доступна в результате присваивания простому свойству, то к ней можно обращаться из кода на XAML.

Но и это еще не все. В WPF уже определен целый ряд команд, поэтому вам не придется писать реализующие ICommand классы для таких команд, как Cut, Copy и Paste, и думать о том, где хранить соответствующие объекты. Встроенные в WPF команды доступны в виде статических свойств пяти разных классов:

- ApplicationCommands – Close, Copy, Cut, Delete, Find, Help, New, Open, Paste, Print, PrintPreview, Properties, Redo, Replace, Save, SaveAs, SelectAll, Stop, Undo и др.
- ComponentCommands – MoveDown, MoveLeft, MoveRight, MoveUp, ScrollByLine, ScrollPageDown, ScrollPageLeft, ScrollPageRight, ScrollPageUp, SelectToEnd, SelectToHome, SelectToPageDown, SelectToPageUp и др.

- **MediaCommands** — ChannelDown, ChannelUp, DecreaseVolume, FastForward, IncreaseVolume, MuteVolume, NextTrack, Pause, Play, PreviousTrack, Record, Rewind, Select, Stop и др.
- **NavigationCommands** — BrowseBack, BrowseForward, BrowseHome, BrowseStop, Favorites, FirstPage, GoToPage, LastPage, NextPage, PreviousPage, Refresh, Search, Zoom и др.
- **EditingCommands** — AlignCenter, AlignJustify, AlignLeft, AlignRight, CorrectSpellingError, DecreaseFontSize, DecreaseIndentation, EnterLineBreak, EnterParagraphBreak, IgnoreSpellingError, IncreaseFontSize, IncreaseIndentation, MoveDownByLine, MoveDownByPage, MoveDownByParagraph, MoveLeftByCharacter, MoveLeftByWord, MoveRightByCharacter, MoveRightByWord и др.

Каждое из этих свойств возвращает не какой-то уникальный тип, реализующий интерфейс ICommand, а объект одного и того же класса RoutedUICommand, который не только реализует ICommand, но и поддерживает всплытие как маршрутизируемые события.

В диалоговом окне About, к которому мы уже возвращались в начале этой главы, имеется кнопка Help (Справка). В настоящий момент она не делает ничего, поэтому воспользуемся ею для демонстрации работы встроенных команд — присоединим некоторую логику с помощью команды Help, определенной в классе ApplicationCommands. В предположении, что эта кнопка называется helpButton, ассоциирование ее с командой Help в C# производится следующим образом:

```
helpButton.Command = ApplicationCommands.Help;
```

Во всех объектах RoutedUICommand определено свойство Text, которое содержит имя команды для показа в пользовательском интерфейсе. (Только наличием этого свойства класс RoutedUICommand отличается от своего базового класса RoutedCommand.) Например, для команды Help свойство Text будет содержать строку Help (что и неудивительно). Теперь вместо того, чтобы зашивать в код значение свойства кнопки Content, мы можем написать:

```
helpButton.Content = ApplicationCommands.Help.Text;
```

СОВЕТ

Строка Text во всех командах RoutedUICommand автоматически локализуется при использовании любого языка, поддерживаемого WPF! Это означает, что кнопка, свойству Content которой присвоено значение ApplicationCommands.Help.Text, автоматически будет называться «Справка», если в текущей культуре пользовательского интерфейса задан русский язык. Даже в контексте, где предполагается использование изображений, а не текста (скажем, на панели инструментов), эту строку можно использовать, например, в виде всплывающей подсказки.

Разумеется, ответственность за локализацию других строк в пользовательском интерфейсе по-прежнему ложится на вас. Использование свойства Text в командах лишь позволяет уменьшить количество нуждающихся в переводе терминов.

Если вы запустите окно About после этого изменения, то увидите, что кнопка неактивна. Объясняется это тем, что встроенные команды не могут знать, ни когда им следует активироваться и деактивироваться, ни какое действие они должны выполнять. Эта логика делегируется клиенту команды.

Для подключения своего кода необходимо добавить объект CommandBinding к самому элементу, который будет выполнять команду, или к любому его родителю (благодаря всплытию маршрутизируемых команд). Во всех классах, производных от UIElement (и ContentElement), имеется коллекция CommandBindings, в которой хранятся объекты типа CommandBinding. Поэтому объект CommandBinding для кнопки Help можно добавить прямо в корневой элемент Window окна About. В застраничном файле это делается так:

```
this.CommandBindings.Add(new CommandBinding(ApplicationCommands.Help,
  HelpExecuted, HelpCanExecute));
```

Здесь предполагается, что определены методы HelpExecuted и HelpCanExecute. Именно их будет вызывать каркас, когда возникнет необходимость обратиться к реализациям методов CanExecute и Execute команды Help.

В листингах 6.10 и 6.11 приведена очередная версия диалогового окна About, в которой к кнопке Help привязана команда Help, причем сделано это целиком на XAML (хотя оба обработчика все-таки нужно определять в застраничном файле).

Листинг 6.10. Окно About с поддержкой команды Help

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AboutDialog"
        Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
        Background="OrangeRed">
  <Window.CommandBindings>
    <CommandBinding Command="Help"
      CanExecute="HelpCanExecute" Executed="HelpExecuted"/>
  </Window.CommandBindings>
    <StackPanel>
      <Label FontWeight="Bold" FontSize="20" Foreground="White">
        WPF 4 Unleashed
      </Label>
      <Label>© 2010 SAMS Publishing</Label>
      <Label>Installed Chapters:</Label>
      <ListBox>
        <ListBoxItem>Chapter 1</ListBoxItem>
        <ListBoxItem>Chapter 2</ListBoxItem>
      </ListBox>
      <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <Button MinWidth="75" Margin="10" Command="Help" Content=
          "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
        <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
```

```
    </StackPanel>
</Window>
```

Листинг 6.11. Застраничный код для разметки в листинге 6.10

```
using System.Windows;
using System.Windows.Input;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
  }

  void HelpCanExecute(object sender, CanExecuteRoutedEventArgs e)
  {
    e.CanExecute = true;
  }

  void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
  {
    System.Diagnostics.Process.Start("http://www.adamnathan.net/wpf");
  }
}
```

Элемент `CommandBinding` для `Window` можно задавать в XAML, потому что в нем определен конструктор по умолчанию, а внутренним полям можно присваивать значения с помощью свойств. Свойству `Content` кнопки `Button` можно даже присвоить значение свойства `Text` выбранной команды (все в XAML) благодаря широко применяемой технике привязки к данным, которую мы будем рассматривать в главе 13. Обратите еще внимание, как упрощается задание команды Help из-за наличия конвертера типа. Класс `CommandConverter` знает все о встроенных командах, поэтому в обоих местах свойству `Command` можно просто присвоить значение `Help`, не прибегая к более громоздкому синтаксису `{x:Static ApplicationCommands.Help}`. (Для написанных вами команд компилятор не окажет подобную любезность.) Метод `HelpCanExecute` в застраничном файле написан так, что команда все время активна, а метод `HelpExecuted` запускает веб-браузер, передавая ему URL-адрес страницы со справкой.

Выполнение команд с помощью жестов ввода

Применение команды Help в простом окне About может показаться перебором – ведь было бы достаточно простого обработчика события `Click`. Но у команды, помимо локализованного текста, есть и еще одно достоинство: автоматическая привязка к комбинации клавиш.

Обычно справка вызывается, когда пользователь нажимает клавишу F1. И надо же – в окне About из листинга 6.10 при нажатии F1 автоматически вызывается команда Help, как если бы вы нажали кнопку Help! А все потому, что для встроенных команд типа Help определены подразумеваемые по умолчанию

жесты ввода, которые приводят к выполнению команды. Можно связать с командой и свой жест ввода, добавив в коллекцию InputBindings подходящий объект KeyBinding и/или MouseBinding. (Поддержка для стилуса или сенсорного ввода не предусмотрена.) Например, чтобы назначить клавишу F2 в качестве активатора команды Help, можно добавить следующее предложение в конструктор класса AboutDialog:

```
this.InputBindings.Add(
  new KeyBinding(ApplicationCommands.Help, new KeyGesture(Key.F2)));
```

Но при этом активировать команду Help будут *обе* клавиши: F1 и F2. Чтобы подавить подразумеваемую по умолчанию клавишу F1, нужно связать с ней специальную команду NotACommand:

```
this.InputBindings.Add(
  new KeyBinding(ApplicationCommands.NotACommand, new KeyGesture(Key.F1)));
```

Оба предложения можно представить и в XAML-разметке следующим образом:

```
<Window.InputBindings>
  <KeyBinding Command="Help" Key="F2"/>
  <KeyBinding Command="NotACommand" Key="F1"/>
</Window.InputBindings>
```

Элементы управления со встроенными привязками к командам

Некоторые элементы управления WPF содержат собственные привязки к командам – первый раз встретившись с этим явлением, воспринимаешь его, как чудо. Простейший пример – элемент TextBox, в который встроены привязки к командам Cut, Copy и Paste для взаимодействия с буфером обмена, а также к командам Undo и Redo. Это означает не только то, что TextBox реагирует на стандартные комбинации Ctrl+X, Ctrl+C, Ctrl+V, Ctrl+Z и Ctrl+Y, но и что в этих действиях могут принимать участие дополнительные элементы.

Механизм встроенных привязок к командам демонстрируется в следующей автономной XAML-разметке:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            Orientation="Horizontal" Height="25">
  <Button Command="Cut" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Copy" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Paste" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Undo" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Redo" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
```

```
    <TextBox x:Name="textBox" Width="200"/>
  </StackPanel> 6
```

Этот фрагмент можно скопировать в любую программу просмотра XAML или сохранить в файле с расширением *.xaml* и просмотреть в Internet Explorer – никакой процедурный код не нужен. Каждая из пяти кнопок ассоциирована с некоторой командой и записывает в свое свойство `Content` строку, возвращаемую свойством `Text` команды. Новым здесь является присваивание свойству `CommandTarget` кнопки ссылки на объект `TextBox` (с помощью механизма привязки к данным, а не атрибута `x:Reference`, чтобы это работало во всех версиях WPF). В результате команда выполняется от имени `TextBox`, а не `Button`, это необходимо, чтобы `TextBox` реагировал на команды.

Приведенный выше XAML-код дает результат, показанный на рис. 6.5. Первые две кнопки автоматически сделаны неактивными, потому что в поле `TextBox` не выделен никакой текст, а когда выделение имеется, они – также автоматически – становятся активными. Аналогично кнопка Paste автоматически становится активной, если в буфере обмена имеется текстовое содержимое, в противном случае она неактивна.

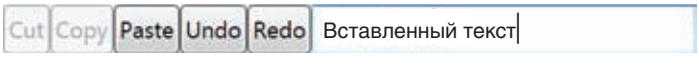


Рис. 6.5. *Все пять кнопок работают, как ожидается, безо всякого процедурного кода – благодаря встроенным в TextBox привязкам*

Элементы `Button` и `TextBox` ничего не знают друг о друге, но с помощью команд могут взаимодействовать между собой. Именно поэтому длинный список встроенных в WPF команд так важен. Чем шире используются встроенные команды в сторонних элементах управления, тем органичнее (и при этом декларативно) разные элементы смогут взаимодействовать, ничего не зная друг о друге.

Резюме

Механизм событий ввода в WPF позволяет создавать интерактивное содержимое, в полной мере задействующее возможности устройства ввода. Хотя маршрутизируемые события и команды сложнее обычных событий .NET, зато они несут с собой богатую функциональность и существенно упрощают решение довольно сложных задач.

В этой главе мы говорили об элементах, производных от класса `UIElement`, но те же самые события ввода применимы и к элементам `ContentElement` (см. главу 11) и `UIElement3D` (см. главу 16 «Трехмерная графика»).

- Стандартные приложения Windows
- Приложения Windows с навигацией
- Приложения-гаджеты
- XAML-приложения для браузера
- Автономные XAML-страницы

7

Структурирование и развертывание приложения

Выше мы рассмотрели основы построения пользовательского интерфейса на базе WPF и связывания его с программной логикой. Теперь настало время поговорить о пакетировании приложений. Не существует какого-то одного канонического способа структурировать WPF-приложения. WPF поддерживает и стандартные приложения Windows, которым в полном объеме доступны ресурсы локального компьютера, и веб-приложения, обладающие расширенными возможностями несмотря на то, что работают они в зоне безопасности Интернета с ограниченным правами, и массу других вариантов.

Чтобы помочь вам по-настоящему разобраться в различиях между разными типами приложений (а не просто прочитать о них), в исходном коде, прилагаемом к книге, имеется набор примеров Photo Gallery (Фотоальбом), построенных по образцу программы Windows Live Photo Gallery. Варианты этих примеров соответствуют типам приложений, рассматриваемым в данной главе.

Стандартные приложения Windows

Стандартное приложение Windows работает на локальном компьютере. Его интерфейс представляет собой одно или несколько окон. На рис. 7.1 показан «стандартный» вариант приложения Photo Gallery.

При создании нового проекта `WPF Application` в Visual Studio автоматически генерируется несколько файлов. Большая их часть знакома разработчикам .NET, например *AssemblyInfo.**, *Resources.**, *Settings.**. А вся специфика WPF сосредоточена в файлах *App.xaml* и *MainWindow.xaml* (и соответствующих застраничных файлах). Именно там находятся объекты `Application` и `Window`, составляющие основу приложений такого типа. (В предыдущих версиях Visual Studio файл *MainWindow.xaml* назывался *Window1.xaml*.)



Рис. 7.1. *Приложение Photo Gallery для локального просмотра фотографий*

Класс Window

Объект `Window` – основной элемент традиционных приложений, в нем отображается их основное содержимое. В WPF за объектом `Window` скрывается обычное окно Win32. Операционная система не различает окна с WPF- и Win32-содержимым: обрамление рисуется одинаково, на панели задачи они неотличимы друг от друга и т. д. (*Обрамление* (Chrome) – это просто другое название неклиентской области, которая в числе прочего содержит кнопки Minimize (Свернуть), Maximize (Развернуть) и Close (Закрыть).)

Следовательно, `Window` – это прямолинейная абстракция окна Win32 (так же, как класс `Form` в Windows Forms), содержащая ряд простых методов и свойств. Создав объект `Window`, вы сможете показать его на экране с помощью метода `Show`, скрыть – методом `Hide` (это то же самое, что присвоить свойству `Visibility` значение `Hidden` или `Collapsed`) и закрыть навсегда – методом `Close`. Несмотря на то, что класс `Window` наследует `Control`, его зависимость от Win32 означает, что некоторые операции, например преобразования, к нему неприменимы.

Внешним видом `Window` можно управлять с помощью таких свойств, как `Icon`, `Title` (интерпретируется как заголовок окна) и `Windowstyle`. Для управления положением на экране служат свойства `Left` и `Top`. Более осмысленного поведения можно добиться, присваивая свойству `WindowstartupLocation` значение `CenterScreen` или `CenterOwner`. Короче говоря, с помощью установки свойств можно делать почти все, что принято ожидать от окна. Скажем, если установить для `Topmost` значение `true`, то окно будет всегда отображаться поверх других, а если присвоить `ShowInTaskbar` значение `false`, то значок окна не будет показываться на панели задач.

Объект `Window` может создавать произвольное число дополнительных окон. Для этого нужно лишь создать объект класса, производного от `Window`, и вызвать его метод `Show`. Эти дополнительные окна при желании можно сделать дочерними. Дочернее окно ведет себя так же, как любое другое окно верхнего уровня, но автоматически закрывается, когда закрывается его родитель,

и минимизируется тоже вместе с родителем. Иногда такие окна называют *немодальными диалоговыми окнами*.

Если некое окно хочет сделать другое окно дочерним, оно должно записать в свойство `Owner` (типа `Window`) последнего ссылку на себя, но только после того, как родитель уже был показан на экране. Перебрать дочерние окна позволяет доступное только для чтения свойство `OwnedWindows`.

Всякий раз, как окно становится активным или неактивным (например, из-за того, что пользователь переключается между разными окнами), возникает событие `Activated` или `Deactivated` объекта `Window`. Можно также принудительно сделать окно активным, вызвав его метод `Activate` (который ведет себя так же, как функция `SetForegroundWindow` из Win32 API). Можно *предотвратить* автоматическую активацию окна при первом показе, присвоив свойству `ShowActivated` значение `false`.

В листинге 7.1 приведена часть окна класса `MainWindow` из приложения Photo Gallery.

Листинг 7.1. Фрагменты файла MainWindow.xaml.cs, относящиеся к управлению окном

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  protected override void OnClosing(CancelEventArgs e)
  {
    base.OnClosing(e);
    if (MessageBox.Show("Are you sure you want to close Photo Gallery?",
      "Annoying Prompt", MessageBoxButton.YesNo, MessageBoxImage.Question)
      == MessageBoxResult.No)
        e.Cancel = true;
  }

  protected override void OnClosed(EventArgs e)
  {
    base.OnClosed(e);

    // Сохранить список избранного
    ...
  }

  protected override void OnInitialized(EventArgs e)
  {
    base.OnInitialized(e);

    // Восстановить сохраненный список избранного
    ...
```

```
    }
...
    void exitMenu_Click(object sender, RoutedEventArgs e)
    {
      this.Close();
    }
      ...
}
```

Конструктор `MainWindow` вызывает метод `InitializeComponent`, чтобы инициализировать ту часть `Window`, которая определена в XAML-разметке. Далее мы видим обработку событий `Closing`, `Closed` и `Initialized`. Но делается это путем переопределения методов `On`*`EventName`*, а не присоединения обработчика к каждому событию. По принятому соглашению управляемые классы содержат защищенные методы `On`*`EventName`*, по одному для каждого события, и классы WPF – не исключение. Конечный результат не зависит от того, присоединили вы обработчик или переопределили метод, но последний способ работает чуть быстрее. Разработчики .NET Framework полагают также, что переопределение метода – более естественный способ обработки событий базового класса в его подклассе.

ПРЕДУПРЕЖДЕНИЕ

Не забывайте вызывать InitializeComponent!

Мы уже говорили об этом в главе 2 «Все тайны XAML» но нелишне будет повторить: если не вызвать метод `InitializeComponent` в конструкторе любого класса, для которого имеется откомпилированный XAML-код, то объект будет сконструирован неправильно. Дело в том, что именно в этом методе производится обработка XAML-кода на этапе выполнения. К счастью, Visual Studio автоматически генерирует вызовы `InitializeComponent`, поэтому случайно забыть про них довольно трудно.

Событие `Closing` возникает, когда производится попытка закрыть окно – из программы или в результате того, что пользователь нажал кнопку Close, комбинацию клавиш Alt+F4 и т. п. Однако обработчик события может запретить закрытие окна, присвоив значение `true` свойству `Cancel` переданного ему объекта `CancelEventArgs` (то же самое и для той же цели можно сделать в Windows Forms). В нашем случае метод `OnClosing` выводит диалоговое окно подтверждения и, если пользователь нажимает кнопку No, отменяет закрытие окна. В этом примере запрашивать подтверждение излишне, потому что нет никаких потенциально несохраненных данных. Обычно же в этом обработчике программа предлагает пользователю сохранить данные, если он этого еще не сделал. Если процесс закрытия не прерван, то окно закрывается и генерируется событие `Closed` (его отменить уже невозможно).

В листинге 7.1 обработчик события `Closed` сохраняет список папок, которые пользователь определил как избранные. Кроме того, в классе `MainWindow` обрабатывается событие `Initialized` – мы считываем сохраненный список и соот-

ветственно изменяем пользовательский интерфейс. (В разделе «Сохранение и восстановление состояния приложения» будет показано, как это делается.) В самом конце листинга приведен обработчик события выбора из меню File пункта Exit, которое закрывает окно.

Класс Application

Теперь осталось только реализовать точку входа в приложение, где можно будет создать и показать окно. Быть может, вы думаете, что для этого следует написать в классе `MainWindow`, показанном в листинге 7.1, следующий метод `Main`:



```
public static void Main()
{
  MainWindow window = new MainWindow();
  window.Show();
}
```

Однако это неправильно по двум причинам. Во-первых, главный поток WPF-приложения должен работать в однопоточном подразделении (STA). Значит, метод `Main` следует снабдить атрибутом `STAThread`. Но важнее другое – вызов метода `Show` неблокирующий, то есть он показывает окно (вызывая функцию `ShowWindow` из Win32 API) и сразу же возвращает управление. Но ведь обращение к `Show` – последняя строка `Main`, следовательно, приложение на этом завершится. В результате окно `MainWindow` на мгновение промелькнет на экране и тут же исчезнет!

FAQ



Неужели я только что прочел слова «однопоточное подразделение»?! Это же старый-престарый COM!

Да, подразделения – это механизм COM. Но, как и все предшествующие каркасы для построения пользовательских интерфейсов на основе Win32 (в том числе и Windows Forms), WPF требует, чтобы главный поток работал в однопоточном подразделении. Связано это прежде всего с необходимостью обеспечить интероперабельность с технологиями, отличными от WPF (см. главу 19 «Интероперабельность с другими технологиями»). Но даже и без требования интероперабельности модель STA – в которой разработчику не нужно задумываться о корректной обработке вызовов из других потоков – позволяет существенно упростить программирование в WPF. Если объект создан в STA-потоке, то к нему можно обращаться только из этого же потока.

WPF следит за тем, чтобы различные методы (классов, производных от `DispatcherObject`) вызывались из правильного потока. Если это не так, возбуждается исключение. Таким образом, невозможно случайно обратиться к методу из другого потока, что могло бы привести к нерегулярным ошибкам (отлаживать которые невероятно трудно). В то же время WPF предоставляет простой механизм взаимодействия произвольных потоков с потоком пользовательского интерфейса, мы поговорим о нем во врезке ниже.

> Если вы ничего не знаете о COM и не хотите заниматься многопоточной обработкой, пожалуйста. Просто пометьте метод `Main` атрибутом `STAThread` и забудьте об этих правилах!

Чтобы метод `Main` не завершился сразу после показа `MainWindow`, необходимо попросить приложение начать цикл диспетчеризации сообщений, которые операционная система передает окну `MainWindow`, и оставаться в нем, пока окно не будет закрыто. Это те же самые сообщения, на базе которых строятся все приложения Win32: `WM_PAINT`, `WM_MOUSEMOVE` и т. д. WPF обязана обрабатывать эти сообщения внутренне, ведь она работает на платформе Windows. В Win32 следовало бы написать *цикл обработки сообщений*, который обрабатывает все поступающие сообщения и посылает их соответствующей оконной процедуре. В WPF ту же задачу проще всего решить с помощью класса `System.Windows.Application`.

Метод Application.Run

В классе `Application` определен метод `Run`, который не дает приложению завершиться и диспетчеризует сообщения. Поэтому правильная реализация `Main` выглядит следующим образом:

```
[STAThread]
public static void Main()
{
  Application app = new Application();
  MainWindow window = new MainWindow();
  window.Show();
  app.Run(window);
}
```

Кроме того, в классе `Application` имеется свойство `StartupUri`, с помощью которого можно показать первое окно приложения другим способом, а именно:

```
[STAThread]
public static void Main()
{
  Application app = new Application();
  app.StartupUri = new Uri("MainWindow.xaml", UriKind.Relative);
  app.Run();
}
```

Эта реализация `Main` эквивалентна предыдущей, только теперь создание экземпляра `MainWindow` и обращение к методу `Show` неявно производит объект `Application`. Отметим две особенности: во-первых, `MainWindow` идентифицируется только именем исходного XAML-файла в виде универсального идентификатора ресурса (URI), а во-вторых, вызывается другой перегруженный вариант метода `Run`, которому не передается экземпляр `Window`. Использование URI-адресов в WPF мы рассмотрим в главе 12 «Ресурсы».

Свойство StartupUri предназначено прежде всего для переноса этой стандартной инициализации в XAML. На самом деле имеющийся в Visual Studio шаблон для проектов WPF Application определяет производный от Application класс App в XAML-файле и присваивает его свойству StartupUri ссылку на главное окно Window. В приложении Photo Gallery файл *App.xaml* выглядит следующим образом:

```
<Application x:Class="PhotoGallery.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml"/>
```

В StartupUri можно записывать обычную строку благодаря наличию конвертера типа для класса Uri.

В соответствующем застраничном файле – *App.xaml.cs* – просто вызывается метод InitializeComponent:

```
using System.Windows;

namespace PhotoGallery
{
  public partial class App : Application
  {
    public App()
    {
      InitializeComponent();
    }
  }
}
```

Это наиболее распространенный подход к структурированию стандартного WPF-приложения и показу его главного окна. Отметим, однако, что если вы не собираетесь ничего добавлять в застраничный файл Application, то можете его вообще опустить.

FAQ



А где же в моем WPF-приложении метод Main?

В файлах, которые Visual Studio генерирует для проекта WPF Application, отсутствует метод Main, и тем не менее приложение работает! Более того, если вы попробуете сами добавить метод Main, то компилятор выдаст ошибку, сообщая, что такой метод уже определен.

Компиляция файла *App.xaml* оказывается особым случаем, потому что Visual Studio сопоставляет ему действие при построении ApplicationDefinition. В результате автоматически генерируется метод Main. В приложении Photo Gallery точка входа находится в файле *App.g.cs*:

```
[System.STAThreadAttribute()]
public static void Main() {
  PhotoGallery.App app = new PhotoGallery.App();
  app.InitializeComponent();
  app.Run();
}
```

По умолчанию Visual Studio скрывает файл *App.g.cs*. Чтобы увидеть его в обозревателе решения (Solution Explorer), следует установить режим показа всех файлов (Show All Files).

FAQ

Как в WPF-приложении получить аргументы командной строки?

Обычно доступ к аргументам командной строки производится с помощью массива строк, переданного методу `Main`, но при стандартной реализации WPF-приложения метод `Main` вы сами не пишете. Решить эту проблему можно двумя способами. Во-первых, отказаться от определения производного от `Application` класса в XAML-файле, тогда можно будет вручную написать метод `Main` и добраться до интересующего нас массива строк. Но проще обратиться в любом месте приложения к свойству `System.Environment.GetCommandLineArgs`, которое вернет тот же самый массив строк, что был передан в `Main`.

Еще один вариант реализовать нестандартную логику инициализации (будь то обработка командной строки, вывод заставки или что-то еще) состоит в том, чтобы в качестве действия при построении для вашего производного от `Application` класса использовать не `ApplicationDefinition`, а `Page`. Тогда вы сможете самостоятельно написать метод `Main`. Реализовав в нем все, что было намечено, создайте экземпляр класса `Application` и вызовите его метод `Run` – то есть сами добавьте те три строки, которые обычно помещаются в автоматически генерируемый файл *App.g.cs*.

Другие применения класса Application

Класс `Application` – это не просто точка входа и диспетчер сообщений. В нем определено множество событий, свойств и методов для выполнения различных действий на уровне приложения. К числу событий, которые в производных от `Application` классах (например, классе `App`, генерируемом Visual Studio) обычно обрабатываются путем переопределения метода `On`*`EventName`*, относятся `Startup` и `Exit`, `Activated` и `Deactivated` (они ведут себя, как одноименные события `Window`, но генерируются для всех без исключения окон, создаваемых в приложении), а также `SessionEnding` – допускающее отмену события, которое возникает, когда пользователь завершает свой сеанс или выключает компьютер. Точная причина указана в передаваемых вместе с этим событием данных – в виде свойства, принадлежащего типу перечисления `ReasonSessionEnding`.

Поскольку в приложении часто бывает несколько окон, в классе Application определена доступная только для чтения коллекция Windows, дающая доступ ко всем открытым окнам. Начальное окно имеет особый статус, его можно получить с помощью свойства MainWindow. Однако это свойство допускает как чтение, так и запись, поэтому придать такой статус можно любому окну в любой момент.

ПРЕДУПРЕЖДЕНИЕ

Не полагайтесь на фиксированный индекс в коллекции Windows!

Объекты Window добавляются в коллекцию Application.Windows в том порядке, в котором были в первый раз показаны, а удаляются в момент закрытия. Поэтому на протяжении жизни приложения индекс одного и того же объекта в этой коллекции может изменяться. Так, не следует предполагать, что элемент Windows[2] всегда ссылается на одно и то же окно!

По умолчанию приложение завершается (то есть метод Run класса Application возвращает управление), когда закрыты все окна. Но это поведение можно изменить путем присваивания свойству ShutdownMode различных значений из перечисления ShutdownMode. Например, можно заставить приложение завершаться, когда закрывается главное окно (на которое указывает свойство MainWindow), вне зависимости от состояния других окон. Или заставить приложение продолжать работу, пока не будет явно вызван метод Shutdown, даже если все окна закрыты. Последний режим удобен для приложений, которые сворачиваются в значок в области уведомлений Windows (впрочем, эта мода выходит из употребления благодаря усовершенствованиям панели задач Windows).

В классе Application есть также очень полезное свойство-коллекция Properties. Как и состояние приложения или сеанса в ASP.NET, это словарь, предназначенный для удобного хранения произвольных данных (в виде пар ключ/значение), общих для различных окон или других объектов. Вместо того чтобы определять открытые поля или свойства в своем производном от Application классе, вы можете просто поместить нужные данные в коллекцию Properties. Например, в приложении Photo Gallery так хранится имя файла текущей выбранной фотографии:

```
myApplication.Properties["CurrentPhotoFilename"] = filename;
```

А получить к нему доступ можно следующим образом:

```
string filename = myApplication.Properties["CurrentPhotoFilename"] as string;
```

Отметим, что и ключ, и значение имеют тип Object, то есть необязательно должны быть строками.

СОВЕТ

Задачи уровня приложения обычно выполняются в коде внутри объектов `Window`, а значит, разным окнам приложения нужно получать ссылку на текущий экземпляр `Application`. К счастью, это легко можно сделать с помощью статического свойства `Application.Current`. Так, переменную `myApplication` в предыдущих фрагментах можно заменить на `Application.Current`:

```
Application.Current.Properties["CurrentPhotoFilename"] = filename;
```

FAQ

Как в WPF создать приложение с многодокументным интерфейсом (MDI)?

В WPF нет встроенной поддержки для создания MDI-интерфейса, но она есть в Windows Forms. Поэтому для создания такого интерфейса в WPF-приложении, в принципе, можно воспользоваться механизмом интероперабельности (см. главу 19). Однако не делайте этого! MDI-интерфейс не способен в полной мере воспользоваться такими средствами для работы с несколькими мониторами или управления окнами, как технология Aero Snap в Windows 7 или Flip 3D в Windows Vista. Если вы хотите избежать создания нескольких окон, подумайте о реализации интерфейса с вкладками (современная вариация на тему MDI), для которого в WPF имеется поддержка.

FAQ

Как в WPF создать приложение, которое может существовать в единственном экземпляре?

К WPF-приложениям применим классический подход к решению этой задачи: именованный (то есть единственный во всей операционной системе) мьютекс. Ниже показано, как это сделать на C#:

```
bool mutexIsNew;
using (System.Threading.Mutex m =
  new System.Threading.Mutex(true, uniqueName, out mutexIsNew))
{
  if (mutexIsNew)
    // Это первый экземпляр, запускаем приложение.
  else
    // Экземпляр уже работает. Выходим!
}
```

Только нужно гарантировать, что имя *uniqueName* не выбрано ни в каком другом приложении! Обычно с этой целью генерируют глобально уникальный идентификатор (GUID) на этапе разработки и потом им пользуются. Разумеется, ничто не может помешать злонамеренному приложению создать мьютекс с таким же именем и тем самым не дать вашему приложению запуститься!

Часто бывает нужно не просто выйти, а передать работающему экземпляру приложения аргументы командной строки. В .NET Framework для этого можно воспользоваться классом `Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase`, который, несмотря на свое название, доступен из приложения на любом языке, в том числе из WPF-приложений. Или поступить так: первый экземпляр открывает RPC-канал, а все остальные пытаются подключиться к нему и передать информацию.

КОПНЕМ ГЛУБЖЕ

Создание приложения без объекта Application

Хотя применение объекта `Application` – рекомендуемый способ структурирования WPF-приложения, это отнюдь не непререкаемое требование. Окна легко показать и без `Application`, но необходимо по меньшей мере организовать диспетчеризацию сообщений, чтобы не столкнуться с проблемой «мгновенного выхода», описанной в начале раздела.

Для этого можно было бы обратиться к функциям Win32, но в WPF также определен низкоуровневый класс `Dispatcher` в пространстве имен `System.Windows.Threading`, который позволяет организовать диспетчеризацию, не прибегая к вызовам Win32 API.

Например, в методе `Main` после показа главного окна можно вызвать не `Application.Run`, а `Dispatcher.Run`. (На самом деле метод `Application.Run` сам вызывает `Dispatcher.Run` для диспетчеризации сообщений!) Но такому приложению все-таки будет недоставать важной функциональности. Так, метод `Dispatcher.Run` не возвращает управление, пока откуда-нибудь явно не будет вызван метод `Dispatcher.ExitAllFrames` (например, из обработчика события окна `Closed`).

КОПНЕМ ГЛУБЖЕ

Многопоточные приложения

В типичном WPF-приложении имеется один поток пользовательского интерфейса и один поток визуализации. (Поток визуализации – это деталь реализации, которая разработчикам напрямую недоступна. Он работает в фоновом режиме и занимается разными низкоуровневыми операциями, в частности композицией.) Вы можете запускать и дополнительные фоновые потоки, но они не должны напрямую обращаться к любым производным от `DispatcherObject` объектам, созданным в потоке пользовательского интерфейса. (Из этого правила есть несколько исключений, например замороженный объект типа `Freezable`.)

К счастью, в WPF имеется простой механизм, позволяющий любому потоку запланировать выполнение некоторого кода в потоке пользовательского интерфейса. В классе `DispatcherObject` определено свойство `Dispatcher` (типа `Dispatcher`). Возвращаемый им объект содержит несколько перегруженных вариантов методов `Invoke` (синхронный вызов) и `BeginInvoke` (асинхронный вызов). Эти методы позволяют передать делегат, который должен быть вызван в соответствующем

диспетчеру потоке пользовательского интерфейса. Всем вариантам методов `Invoke` и `BeginInvoke` в обязательном порядке передается значение перечисления `DispatcherPriority`, в котором определено 10 приоритетов, начиная от высшего `Send` (то есть выполнить немедленно) до низшего `SystemIdle` (выполнить, когда в очереди диспетчера больше ничего нет).

Можно даже создать в приложении несколько потоков пользовательского интерфейса, если вызвать метод `Dispatcher.Run` в запущенном вами потоке. Таким образом, если у приложения более одного окна верхнего уровня, то каждое такое окно может работать в своем потоке. Это редко бывает необходимо, но в случае, когда одно окно может начать операцию, потребляющую все ресурсы потока, такая схема способна улучшить время отклика приложения. Правда, это негативно отражается на абстракции `Application`, потому что она подразумевает наличие единственного диспетчера. Например, коллекция `Application.Windows` содержит только окна, созданные в том же потоке, что и `Application`.

Показ заставки

В идеале необходимости в заставке вообще не должно возникать, но иногда от момента запуска приложения до показа главного окна проходит заметное время – особенно при первом запуске в сеансе данного пользователя (эта задержка называется *временем холодного запуска*). Поэтому в WPF включены специальные средства для добавления заставки.

Заставка представляет собой изображение, которое появляется сразу после запуска приложения и исчезает в момент появления главного окна. Чтобы получить эффект непрямоугольного окна, можно использовать PNG-файл с прозрачными областями, но анимированное содержимое (например, анимированный GIF-файл) не допускается. Нельзя использовать ни динамическое содержимое, ни элементы WPF, поскольку заставка показывается еще до того, как закончилась загрузка WPF. (В противном случае для показа заставки могло бы потребоваться столько же времени, сколько для показа главного окна!) Поэтому не получится создать занимательные заставки в духе Office 2010, где есть и анимации, и обновляемая информация о состоянии. Зато вы имеете полезный эффект, почти не прикладывая усилий.

Чтобы воспользоваться этой поддержкой в Visual Studio 2010, достаточно выбрать в диалоговом окне Add New Item (Добавить новый элемент) пункт Splash Screen (WPF). (Для Visual Studio 2008 SP1 такой шаблон можно скачать с сайта *http://codeplex.com*.) В результате в проект будет добавлено изображение, для которого определено действие при построении `SplashScreen`; это изображение вы можете изменить по своему усмотрению. Вот и все! На рис. 7.2 показана заставка для приложения Photo Gallery.

Другой способ добиться того же эффекта – просто добавить нужное изображение в проект и задать для него действие при построении `SplashScreen`. В версии Visual Studio 2008 SP1 это самый простой вариант, поскольку не требуется ничего скачивать. Или, если нужен чуть больший контроль над заставкой,

например динамический выбор изображения либо задание максимального времени присутствия заставки на экране, можно воспользоваться классом `System.Windows.SplashScreen`. Он содержит несколько простых методов для создания, показа и убирания заставки.

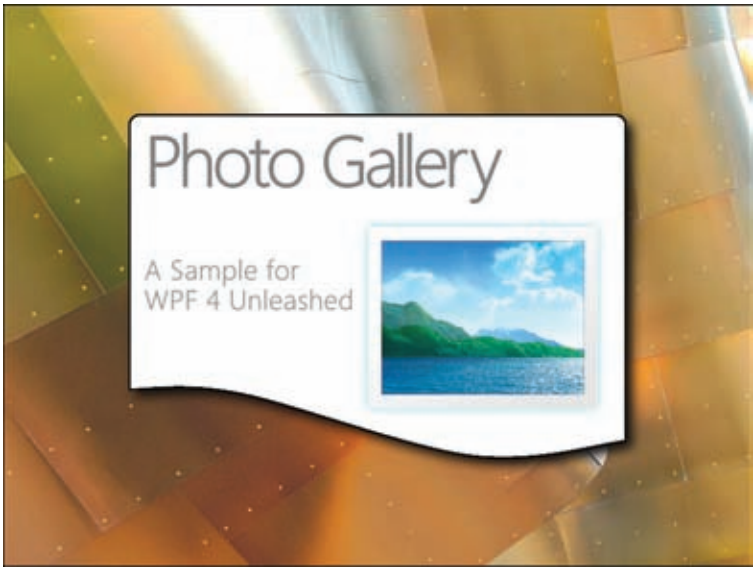


Рис. 7.2. *В заставке для Photo Gallery используется частично прозрачное PNG-изображение*

Создание и показ диалоговых окон

В ОС Windows имеется набор стандартных *диалоговых окон* (модальных окон), позволяющих выполнять такие типичные операции, как открытие или сохранение файлов, обзор папок, выбор шрифта или цвета и печать. Вы и сами можете создавать диалоговые окна с таким же модальным поведением. (Иными словами, это окна, которые не позволяют взаимодействовать с текущим окном, пока вы их явно не закроете.)

Стандартные диалоговые окна

В WPF встроены средства доступа к нескольким стандартным диалоговым окнам с помощью классов, раскрывающих их функциональность в виде простых свойств и методов. Отметим, что WPF не рисует эти диалоговые окна самостоятельно, а обращается к функциям из Win32 API. Но это и хорошо, потому что внешний вид диалогового окна оказывается согласован с версией операционной системы, в которой запущено приложение.

Для использования стандартного диалогового окна достаточно создать объект соответствующего ему класса, вызвать его метод `ShowDialog`, а затем обра-

ботать результат. Например, в приложении Photo Gallery диалоговое окно PrintDialog для печати фотографий используется следующим образом:

```
void printMenu_Click(object sender, RoutedEventArgs e)
{
  string filename = (pictureBox.SelectedItem as ListBoxItem).Tag as string;
  Image image = new Image();
  image.Source = new BitmapImage(new Uri(filename, UriKind.RelativeOrAbsolute));

  PrintDialog pd = new PrintDialog();
  if (pd.ShowDialog() == true) // результат может быть равен true, false или null
    pd.PrintVisual(image, Path.GetFileName(filename) + " from Photo Gallery");
}
```

Даже не думайте о том, чтобы написать собственное диалоговое окно взамен стандартного, предоставляемого Windows. Мало того что в этом случае внешний вид вашей программы будет отличаться от большинства приложений Windows, так еще вы, без сомнения, забудете реализовать какие-то функции, нужные некоторым пользователям, и при выходе каждой новой версии Windows придется вносить изменения. Взгляните только, сколько функций предоставляет встроенное в Windows 7 диалоговое окно открытия файла: поиск; особая поддержка для избранного, библиотек и домашней группы (HomeGroup); несколько видов с большим выбором столбцов, по которым можно сортировать и фильтровать; панель предварительного просмотра и многое другое. Имеются также функции, которые сразу не видны, например запоминание открываемых файлов с целью сформировать списки недавно и часто открывавшихся файлов в таких местах, как списки переходов в Windows 7.

СОВЕТ

И в Windows Forms, и в WPF имеются управляемые классы, обертывающие стандартные диалоговые окна Windows. Однако в текущей версии WPF не для всех диалоговых окон есть соответствующие классы. (В Windows Forms есть классы ColorDialog, FontDialog и FolderBrowser, для которых в WPF до сих пор нет аналогов.) Поэтому, чтобы воспользоваться не включенными диалоговыми окнами, проще всего добавить ссылку на сборку System.Windows.Forms.dll и работать с классами, определенными в Windows Forms.

Нестандартные диалоговые окна

Хотя мысль о том, чтобы написать *собственное* стандартное диалоговое окно, должна быть без колебаний отвергнута, очень часто есть все основания для показа *нестандартных* диалоговых окон, например простого окна для переименования фотографии (Rename Photo) в программе Photo Gallery (рис. 7.3).

В WPF такие диалоговые окна создаются и используются почти так же, как объекты Window. На самом деле *это и есть* объекты Window, только с небольшим дополнением для возврата так называемого *результата диалогового окна*.

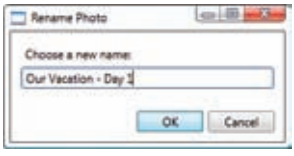


Рис. 7.3. *Нестандартное диалоговое окно для переименования фотографии*

Чтобы показать окно `Window` в виде модального (в отличие от немодального) диалогового окна, достаточно вызвать метод `ShowDialog` вместо `Show`. В отличие от `Show`, метод `ShowDialog` блокирует выполнение программы (то есть не возвращает управление, пока окно не будет закрыто) и возвращает допускающее `null` булевское значение (тип `bool?` в C#). Вот как в приложении Photo Gallery используется нестандартное диалоговое окно `RenameDialog`:

```
void renameMenu_Click(object sender, RoutedEventArgs e)
{
  string filename = (pictureBox.SelectedItem as ListBoxItem).Tag as string;
  RenameDialog dialog = new RenameDialog(
    Path.GetFileNameWithoutExtension(filename));
  if (dialog.ShowDialog() == true) // результат может быть равен true,
                                   // false или null
  {
    // Пытаемся переименовать файл
    try
    {
      File.Move(filename, Path.Combine(Path.GetDirectoryName(filename),
        dialog.NewFilename) + Path.GetExtension(filename));
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message, "Cannot Rename File", MessageBoxButton.OK,
        MessageBoxImage.Error);
    }
  }
}
```

При разработке окна, которое заведомо будет использоваться в качестве диалогового (как в случае `RenameDialog`), обычно желательно, чтобы метод `ShowDialog` возвращал `true`, если действие, ради которого диалоговое окно написано, выполнено успешно, и `false` – в случае ошибки или отмены. Чтобы задать возвращаемое этим методом значение, достаточно присвоить его свойству `DialogResult` (типа `bool?`). Побочным результатом установки `DialogResult` является закрытие окна. Следовательно, обработчик нажатия кнопки `OK` в диалоговом окне `RenameDialog` мог бы выглядеть следующим образом:

```
void okButton_Click(object sender, RoutedEventArgs e)
{
   this.DialogResult = true;
}
```

А можно просто присвоить свойству `IsDefault` значение `true`, что позволяет достичь того же эффекта без написания процедурного кода.

КОПНЕМ ГЛУБЖЕ

Еще одно применение метода ShowDialog

Чтобы обеспечить блокировку доступа к родительскому окну, не прерывая процесса диспетчеризации сообщений, метод `ShowDialog` в классе `Window`, по сути дела, вызывает `Dispatcher.Run` – точно так же, как это делает метод `Application.Run`. Поэтому для того, чтобы корректно запустить WPF-приложение без использования класса `Application`, можно прибегнуть к следующему приему:

```
[STAThread]
public static void Main()
{
  MainWindow window = new MainWindow();
  window.ShowDialog();
}
```

Сохранение и восстановление состояния приложения

Стандартное приложение Windows может обращаться ко всем ресурсам компьютера (в пределах параметров безопасности работающего с ним пользователя), поэтому есть несколько вариантов сохранения данных, например в реестре или в локальной файловой системе. Но у этих классических способов есть и интересная альтернатива: *изолированное хранилище* .NET Framework. Мало того что с ним просто работать, так еще эта техника применима ко всем средам, в которых может исполняться управляемый код, в частности в Silverlight- или XAML-приложении для браузера (рассматривается ниже в этой главе).

В программе Photo Gallery для сохранения избранных данных пользователя в изолированном хранилище и последующего восстановления из него применяется код, показанный в листинге 7.2.

Листинг 7.2. Часть файла MainWindow.xaml.cs, относящаяся к изолированному хранилищу

```
protected override void OnClosed(EventArgs e)
{
  base.OnClosed(e);

  // Перед тем как закрыть приложение, записываем избранные элементы
  IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForAssembly();
  using (IsolatedStorageFileStream stream =
    new IsolatedStorageFileStream("myFile", FileMode.Create, f))
  using (StreamWriter writer = new StreamWriter(stream))
  {
    foreach (TreeViewItem item in favoritesItem.Items)
```

```
        writer.WriteLine(item.Tag as string);
    }
  }

  protected override void OnInitialized(EventArgs e)
  {
    base.OnInitialized(e);

    // В момент инициализации приложения считываются все избранные элементы
    IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForAssembly();
    using (IsolatedStorageFileStream stream =
      new IsolatedStorageFileStream("myFile", FileMode.OpenOrCreate, f))
    using (StreamReader reader = new StreamReader(stream))
    {
      string line = reader.ReadLine();
      while (line != null)
      {
        AddFavorite(line);
        line = reader.ReadLine();
      }
    }
    ...
  }
```

Классы `IsolatedStorageFile` и `IsolatedStorageFileStream` находятся в пространстве имен `System.IO.IsolatedStorage`. Все данные, помещенные в изолированное хранилище, физически находятся в скрытой папке внутри папки Documents (Документы) текущего пользователя.

СОВЕТ

Если вы хотите еще больше упростить сохранение и восстановление настроек приложения, ознакомьтесь с генерируемым Visual Studio классом `Settings` (в разделе `Properties\Settings.settings`). Этот механизм позволяет сохранять данные в конфигурационном файле приложения и обеспечивает строго типизированный доступ к ним.

Развертывание: технология ClickOnce и установщик Windows

Когда речь заходит о развертывании стандартного приложения Windows, вы, вероятно, сразу представляете себе программу установки, которая помещает файлы в каталог *Program Files* (или в каталог, указанный пользователем), регистрирует необходимые компоненты, добавляет себя в список установленных программ на Панели управления и, быть может, в меню Пуск и на рабочий стол. Все это можно сделать и для WPF-приложения, воспользовавшись

установщиком Windows. И в Visual Studio для этой цели имеется несколько типов проекта Setup and Deployment (Установка и развертывание).

С другой стороны, технология ClickOnce проще и появилась сравнительно недавно (в версии .NET Framework 2.0). Это привлекательная альтернатива для приложений, которым не нужна вся мощь установщика Windows. В Visual Studio доступ к функциональности ClickOnce открывает Мастер, вызваемый командами меню Build→Publish (Построение→Публикация). Если у вас нет Visual Studio, то можете воспользоваться Windows SDK, где имеется два инструмента для работы с ClickOnce: командная утилита mage.exe и графическая программа mageUI.exe.

Короче говоря, установщик Windows имеет следующие преимущества по сравнению с ClickOnce:

- Поддерживает настраиваемый пользовательский интерфейс, например показ лицензионного соглашения
- Позволяет задавать местоположение файлов при установке
- Поддерживает (с помощью настраиваемых действий) написание произвольного кода, исполняемого на этапе установки
- Поддерживает установку общих сборок в глобальный кэш сборок
- Поддерживает регистрацию COM-компонентов и сопоставлений файлов
- Поддерживает установку для компьютера в целом (то есть таким образом, что программа становится доступна всем пользователям)
- Поддерживает автономную установку с CD/DVD

С другой стороны, технология ClickOnce имеет следующие преимущества по сравнению с инсталлятором Windows:

- Включает встроенную поддержку автоматического обновления и отката к предыдущей версии.
- Предлагает две модели установки: веб-модель, когда приложение идентифицируется посредством URL-адресов в браузере и после закрытия как бы «исчезает» (хотя на самом деле сохраняется в кэше), и традиционную модель, в которой у приложения может быть ярлык в меню Пуск и оно присутствует в списке установленных программ на Панели управления.
- Гарантирует, что устанавливаемое приложение не окажет никакого влияния на другие приложения, потому что все файлы помещаются в изолированную область и не производится никакой регистрации.
- Гарантирует полную деинсталляцию, так как во время установки никакой написанный пользователем код не выполняется (правда, приложения с полным доверием все-таки могут оставлять после себя какие-то следы на компьютере).
- Интегрируется с имеющимся в .NET механизмом разграничения доступа кода, который позволяет пользователю запускать даже те приложения, которым он не доверяет на все сто процентов.

СОВЕТ

Многие не понимают, что технологию ClickOnce можно применять даже тогда, когда приложение содержит неуправляемый код, при условии, правда, что главный исполняемый файл не является полностью неуправляемым. Однако, чтобы это заработало, возможно, придется внести некоторые изменения в неуправляемый код. Например, вместо регистрации COM-объектов придется прибегнуть к технологии COM без регистрации.

Приложения Windows с навигацией

Хотя понятие навигации обычно ассоциируется с веб-браузером, такая же схема применяется и во многих других приложениях Windows, например в Проводнике Windows, Windows Media Player и, конечно же, в программе Windows Live Photo Gallery, послужившей прототипом для нашего приложения Photo.

В первой версии Photo Gallery, представленной на рис. 7.1, применялся примитивный подход к навигации для перемещения по фотографиям и возврата в главное окно альбома. Однако в WPF встроена развитая инфраструктура для включения в программы средств навигации с минимумом усилий. Она позволяет безо всякого труда написать приложение, которое умеет осуществлять навигацию так же, как браузер.

Хотя заголовок этого раздела наводит на мысль, будто включение навигации определяет всю структуру приложения, на самом деле ее поддержку можно интегрировать и в приложение с традиционной структурой, причем глубина интеграции оставлена на ваше усмотрение. И даже если вы не хотите строить интерфейс по типу браузера, все равно навигацию можно использовать для того, чтобы приложение выглядело похоже на веб-сайт. Например, различные части интерфейса можно организовать в виде отдельных страниц, адресуемых с помощью URI, и для переходов между ними использовать гиперссылки. Или использовать навигацию только в небольшой части приложения, скажем, в мастере.

В этом разделе мы рассмотрим все эти возможности и продемонстрируем некоторые изменения в «стандартной» версии Photo Gallery, позволяющие ими воспользоваться. Добавление навигации не отменяет все сказанное в предыдущем разделе по поводу развертывания, сохранения данных и т. д. Просто мы ознакомимся с рядом дополнительных элементов, в частности `NavigationWindow` и `Page`.

Страницы и их навигационные контейнеры

При использовании навигации в WPF содержимое обычно организуется в виде элементов `Page`. (`Page` – это, по существу, упрощенная версия класса `Window`.) Элементы `Page` могут располагаться в одном из двух встроенных навигационных контейнеров: `NavigationWindow` или `Frame`. Они предоставляют средства для

перехода от одной страницы к другой, «журнал», в котором хранится история навигации, и ряд относящихся к навигации событий.

FAQ

В чем разница между NavigationWindow и Frame?

Функционально эти классы почти одинаковы за одним исключением: `NavigationWindow` ведет себя скорее, как окно браузера верхнего уровня, а `Frame` – как HTML-элемент `FRAME` или `IFRAME`. Если `NavigationWindow` – окно верхнего уровня, то `Frame` может заполнять произвольную (но прямоугольную) область внутри своего родительского элемента. `Frame` может быть вложен в `NavigationWindow` или в другой `Frame`. По умолчанию вдоль верхнего края `NavigationWindow` располагается панель с кнопками Назад/Вперед, а во фрейме ее нет, но в обоих случаях панель можно добавить или убрать с помощью свойства `ShowsNavigationUI`, содержащегося внутри страницы `Page`. Кроме того, в классе `NavigationWindow` имеется свойство `ShowsNavigationUI`, а в классе `Frame` – свойство `NavigationUIVisibility`, оба они позволяют показать или скрыть эту панель вне зависимости от настроек `Page`.

В версии приложения Photo Gallery с навигацией свойство `StartupUri` объекта `Application` указывает на следующий объект `NavigationWindow`:

```
<NavigationWindow x:Class="PhotoGallery.Container"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Photo Gallery" Source="MainPage.xaml">
</NavigationWindow>
```

Корневым элементом в файле *MainPage.xaml*, на который ссылается этот элемент `NavigationWindow`, является элемент `Page`, который содержит все, что раньше находилось в файле *MainWindow.xaml*:

```
<Page x:Class="PhotoGallery.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Photo Gallery" Loaded="Page_Loaded">
  ... Содержимое конкретного приложения...
</Page>
```

Точно так же застраничный код в файле *MainPage.xaml.cs* соответствует застраничному коду, ранее находившемуся в файле *MainWindow.xaml.cs*. Основное отличие от *MainPage.xaml.cs* заключается в том, что логика, реализованная в методах `OnClosing` и `OnClosed`, перенесена на уровень окна, потому что в классе `Page` таких методов нет (было бы неправильно вызывать их всякий раз при смене страницы).

На рис. 7.4 видно, что добавление `NavigationWindow` и `Page` в программу Photo Gallery мало что изменило – просто в окне появилась верхняя панель с кнопками Назад и Вперед (неактивными). Тем не менее, как мы вскоре увидим, при-

ложение получило возможность переходить к другому содержимому в том же самом контейнере.

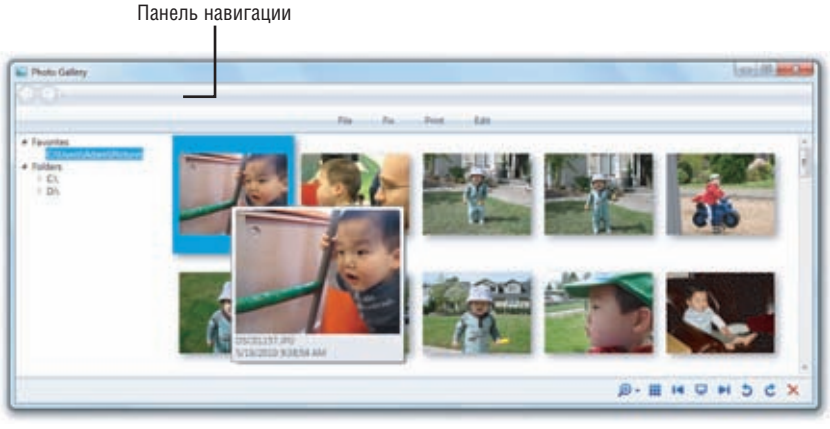


***Рис. 7.4.** Если фотоальбом помещен внутрь NavigationWindow, то сверху появляется дополнительная панель*

Конечно, наличие дополнительной верхней панели в данном приложении вызывает легкое недоумение. В программе Photo Gallery было бы уместнее реализовать собственные кнопки Назад и Вперед, интегрированные с механизмом навигации, встроенным в класс `NavigationWindow`. Например, обработчик события `Click` кнопки Назад мог бы вызывать метод `NavigationWindow.GoBack`, обработчик события `Click` кнопки Вперед – метод `NavigationWindow.GoForward`.

СОВЕТ

Навигационные контейнеры в WPF могут содержать не только элементы `Page`, но также HTML-файлы (находящиеся как в локальной файловой системе, так и в Интернете)! Можно даже переходить от WPF-содержимого к HTML-содержимому и обратно. Как это делается, будет описано в следующем разделе.

Элемент `Page` может взаимодействовать со своим навигационным контейнером с помощью класса `NavigationService`, который предлагает одну и ту же функциональность вне зависимости от того, является ли контейнер объектом `NavigationWindow` или `Frame`. Чтобы получить экземпляр `NavigationService`, следует вызвать статический метод `NavigationService.GetNavigationService`, передав ему экземпляр `Page`. Но можно поступить и проще, обратившись к свойству `NavigationService` объекта `Page`. Например, следующий код устанавливает заголовок страницы, который будет показан в раскрывающемся меню, ассоциированном с кнопками Назад и Вперед:

```
this.NavigationService.Title = "Main Photo Gallery Page";
```

А чтобы обновить текущую страницу, нужно написать:

```
this.NavigationService.Refresh();
```

Но в классе Page имеется и несколько собственных свойств, управляющих поведением родительского контейнера, например WindowHeight, WindowWidth и WindowTitle. Они особенно удобны, потому что могут быть установлены в XAML-разметке элемента Page.

Переходы между страницами

Смысл навигации в том, чтобы переходить от одной страницы к другой – линейно (как в простом мастере), вдоль пути, определяемого пользователем (как на большинстве веб-сайтов), или вдоль динамически генерируемого пути.

Есть три основных способа навигации:

- Посредством метода Navigate
- С помощью гиперссылок (объектов Hyperlink)
- С помощью журнала

Вызов метода Navigate

Навигационные контейнеры поддерживают метод Navigate, позволяющий изменять текущую страницу. При его вызове можно указать либо объект, представляющий целевую страницу, либо ее URI:

```
// Перейти к экземпляру страницы
PhotoPage nextPage = new PhotoPage();
this.NavigationService.Navigate(nextPage);
// Или перейти на страницу с заданным URI
this.NavigationService.Navigate(new Uri("PhotoPage.xaml", UriKind.Relative));
```

Страница Page, заданная с помощью URI, может быть автономным XAML-файлом или откомпилированным ресурсом. (О том, как работают URI в WPF, рассказано в главе 12.) Корневым элементом XAML-файла должен быть элемент Page.

КОПНЕМ ГЛУБЖЕ

Представление метода Navigate в виде двух свойств

В навигационных контейнерах есть два свойства, которые могут служить эквивалентами двух вариантов метода Navigate. Чтобы перейти к объекту Page, достаточно установить свойство Content:

```
this.NavigationService.Content = nextPage;
```

а перейти по URI можно, установив свойство Source:

```
this.NavigationService.Source = new Uri("PhotoPage.xaml", UriKind.Relative);
```

Если не считать удобства для декларативной разметки, никаких других причин использовать эти свойства вместо метода Navigate не существует.

Чтобы перейти к HTML-странице, необходимо воспользоваться перегруженным вариантом метода Navigate, которому передается URI. Например:

```
this.NavigationService.Navigate(new Uri("http://www.adamnathan.net/wpf"));
```

Использование элемента Hyperlink

Для простых схем навигации в WPF имеется элемент Hyperlink, который ведет себя во многом аналогично гиперссылкам в HTML. Если элемент Hyperlink вложен в TextBlock то, как и в случае HTML-тега A, содержимое визуализируется в виде гиперссылки, после щелчка по которой выполняется переход от текущей страницы к целевой. Целевая страница определяется свойством NavigateUri элемента Hyperlink (аналог атрибута href в HTML). Например, следующая XAML-разметка визуализируется, как показано на рис. 7.5:

```
<TextBlock>
Click <Hyperlink NavigateUri="PhotoPage.xaml">here</Hyperlink> to view the photo.
</TextBlock>
```

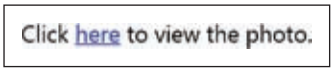


Рис. 7.5. *Результат визуализации Hyperlink выглядит так же, как гиперссылка в HTML*

Таким образом, Hyperlink – это просто более длинная форма записи HTML-тега A. С этими объектами можно работать и из программы, как с любым другим элементом WPF, но основное их назначение – описывать простые HTML-подобные ссылки, когда целевая страница известна заранее.

СОВЕТ

Если вам нужна гибкость программной навигации в сочетании с удобными средствами автоматического форматирования текста, предоставляемыми классом Hyperlink, то можете указать в Hyperlink фиктивное значение свойства NavigateUri, а потом в обработчике события Click этого элемента вызвать метод Navigate, задав нужный адрес перехода.

СОВЕТ

Класс Hyperlink поддерживает и дополнительные возможности – как и гиперссылки в HTML. Например, для перехода в конкретный фрейм Frame в случае, когда фреймов несколько, следует присвоить свойству TargetName элемента Hyperlink имя нужного фрейма. Чтобы перейти к определенному месту внутри страницы Page (как в HTML-якорях, обозначаемых символом #), достаточно дописать в конец URI символ # и имя любого элемента на целевой странице.

FAQ

Как задать в HTML-странице ссылку, ведущую на элемент WPF Page?

В HTML гиперссылки работают автоматически, но не существует никакого способа задать в атрибуте `HREF` значение, указывающее на откомпилированный объект WPF `Page`. Вместо этого для осуществления навигации из HTML в WPF можно воспользоваться приемом, похожим на описанный выше: в качестве значения `HREF` задать какое-нибудь фиктивное значение и написать обработчик события `Navigating`, в котором динамически изменить цель, вызвав метод `Navigate` самостоятельно (`Navigating` и другие события рассматриваются в следующем разделе). В зависимости от характера желаемого взаимодействия между HTML и WPF можно также рассмотреть возможность создания приложения XAML для браузера или автономной XAML-страницы (или вообще подумать, не стоит ли прибегнуть к Silverlight). Эти варианты обсуждаются в конце главы.

Использование журнала

С обоими навигационными контейнерами ассоциирован *журнал*, в котором хранится история навигации, – точно так же, как в браузере. С помощью журнала реализуется поведение кнопок Вперед и Назад, показанных на рис. 7.4. На внутреннем уровне система ведет два стека – обратных и прямых переходов – и использует их, как показано в табл. 7.1.

Таблица 7.1. Отражение навигации в журнале

Действие	Результат
Назад	Помещает текущую страницу в стек прямых переходов, извлекает страницу из стека обратных переходов и переходит на нее
Вперед	Помещает текущую страницу в стек обратных переходов, извлекает страницу из стека прямых переходов и переходит на нее
Любая другая навигация	Помещает текущую страницу в стек обратных переходов и опустошает стек прямых переходов

Действия перехода назад и вперед могут быть инициированы как пользователем, так и программой – путем вызова методов `GoBack` и `GoForward` навигационного контейнера (предварительно вызвав соответственно метод `CanGoBack` или `CanGoForward`, чтобы избежать исключения в результате попытки извлечения из пустого стека).

В объекте `NavigationWindow` журнал есть всегда, тогда как в объекте `Frame` собственного журнала может и не быть; это зависит от его свойства `JournalOwnership`, которое может принимать следующие значения:

- `OwnsJournal` – у фрейма есть свой журнал.
- `UsesParentJournal` – история хранится в журнале родительского контейнера или не хранится вовсе, если у родителя нет журнала.

- `Automatic` — эквивалентно `UsesParentJournal`, если фрейм содержится в любом из двух навигационных контейнеров (`NavigationWindow` или `Frame`), в противном случае эквивалентно `OwnsJournal`. Это значение по умолчанию.

Если у фрейма есть собственный журнал, он получает также встроенные кнопки навигации. Если они вам не нужны, присвойте свойству `NavigationUIVisibility` значение `Hidden`.

СОВЕТ

В случае перехода на страницу `Page` с помощью URI (неважно, путем вызова метода `Navigate` или посредством гиперссылки `Hyperlink`) создается новый экземпляр `Page`, даже если вы уже посещали эту страницу. Поэтому, если требуется, чтобы страница «запоминала» свои данные, необходимо хранить состояния отдельно (например, в статических переменных или в словаре `Application.Properties`). (При вызове варианта `Navigate`, принимающего экземпляр `Page`, вы, разумеется, вольны сами решать, передать ли ему новый или старый объект.)

Однако в случае навигации по журналу можно установить режим принудительного использования одного и того же объекта `Page`, присвоив его присоединенному свойству `JournalEntry.KeepAlive` значение `true`.

СОВЕТ

Объект `Page` может потребовать не заносить себя в журнал, присвоив своему свойству `RemoveFromJournal` значение `true`. Это имеет смысл для страниц, являющихся частью некоторой последовательности шагов, которые нельзя открывать в произвольном порядке после завершения операции.

FAQ

Действия Вперед и Назад обрабатываются журналом, а как реализовать аналоги действий браузера Остановить и Обновить?

Для кнопок `Остановить` и `Обновить` нет встроенной поддержки в пользовательском интерфейсе, но навигационные контейнеры вполне способны выполнять соответствующие действия.

Чтобы в любой момент остановить еще не законченную операцию перехода, вызовите метод контейнера `StopLoading`.

А для обновления страницы достаточно вызвать метод контейнера `Refresh` без параметров. Это все равно, что вызвать метод `Navigate`, передав ему URI или экземпляр текущей страницы, только обработчику события `Navigating` в качестве данных передается значение `NavigationMode.Refresh` на случай, если тому потребуется модифицировать свое поведение в подобной ситуации.

КОПНЕМ ГЛУБЖЕ

Использование журнала для других целей

В журнал можно добавлять свои записи, не имеющие никакого отношения к встроенной навигации. Например, на основе журнала можно построить специализированную схему отмены и повтора операций, причем большую часть функциональности вы получите задаром.

Для этого вызовите метод контейнера `AddBackEntry`, передав ему объект типа `CustomContentState`. `CustomContentState` – абстрактный класс, поэтому необходимо создать его подкласс, в котором реализован метод `Replay`. Этот метод вызывается, когда в результате перехода вперед или назад данный объект становится текущим. Можно также переопределить еще и свойство `JournalEntryName`, которое возвращает метку данного объекта в раскрывающемся списке.

В приложении Photo Gallery эта техника применяется для реализации поворота изображения, допускающего отмену:

```
[Serializable]
class RotateState : CustomContentState
{
  FrameworkElement element;
  double rotation;

  public RotateState(FrameworkElement element, double rotation)
  {
    this.element = element;
    this.rotation = rotation;
  }

  public override string JournalEntryName
  {
    get { return "Rotate " + rotation + "°"; }
  }

  public override void Replay(NavigationService navigationService,
                              NavigationMode mode)
  {
    // Повернуть элемент на указанный угол
    element.LayoutTransform = new RotateTransform(rotation);
  }
}
```

События навигации

Вне зависимости от того, как инициирована навигация – путем вызова метода `Navigate`, с помощью гиперссылок `Hyperlink` или по журналу, – она всегда производится асинхронно. В процессе навигации генерируется ряд событий, позволяющих сообщать пользователю подробную информацию или даже прервать навигацию.

На рис. 7.6 и 7.7 показана последовательность возникновения относящихся к навигации событий при загрузке первой страницы и при переходе от одной страницы к другой.

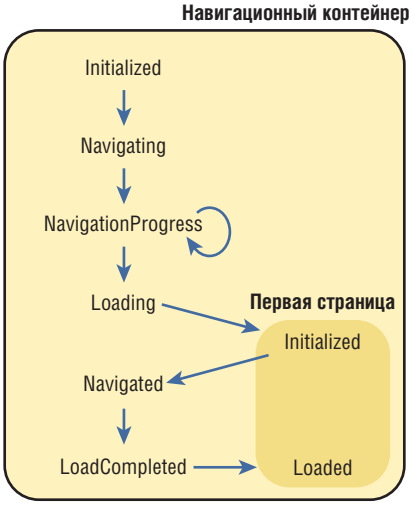


Рис. 7.6. *События навигации, возникающие при загрузке первой страницы*

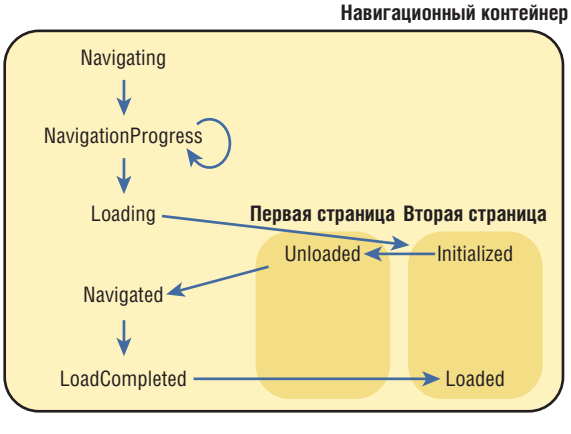


Рис. 7.7. *События навигации, возникающие при переходе от одной страницы к другой*

Событие `NavigationProgress` генерируется периодически вплоть до возникновения события `Navigated`. На рисунке не показано событие `NavigationStopped`. Оно генерируется вместо `LoadCompleted`, если навигация была отменена или произошла ошибка.

СОВЕТ

Показанные на рис. 7.6 и 7.7 события генерирует навигационный контейнер, когда они возникают внутри него (включая и дочерние контейнеры). Но те же самые события генерирует и объект `Application`, чтобы в одном месте можно было обработать события от всех навигационных контейнеров в приложении.

ПРЕДУПРЕЖДЕНИЕ

События навигации не генерируются при переходе между HTML-страницами!

События навигации WPF генерируются при переходе от одной страницы WPF `Page` к другой, от страницы WPF к странице HTML и от страницы HTML к странице WPF. Но при переходе от одной страницы HTML к другой странице HTML эти события не генерируются. Переходы от HTML к HTML не регистрируются и в журнале.

Передача данных между страницами

Если навигация используется не только для обзора документов, то, вероятно, при переходах между страницами необходимо передавать какие-то данные. В веб-приложениях на базе HTML данные можно кодировать в параметрах URL или использовать переменные на стороне сервера. В WPF для передачи и возврата данных применяются разнообразные способы.

Передача данных странице

WPF поддерживает схему, аналогичную параметрам URL, с помощью перегруженных вариантов метода `Navigate`, которые принимают дополнительный параметр типа `Object`. Такие варианты есть для метода, принимающего как экземпляр `Page`, так и `Uri`. В объекте-параметре можно передать произвольные данные (встроенный тип, массив, структуру данных и т. д.), которые получит целевая страница. Например:

```
int photoId = 10;
// Перейти к экземпляру Page
PhotoPage nextPage = new PhotoPage();
this.NavigationService.Navigate(nextPage, photoId);
// Или перейти к странице по URI
this.NavigationService.Navigate(
  new Uri("PhotoPage.xaml", UriKind.Relative), photoId);
```

Чтобы целевая страница могла получить данные, она должна обработать событие `LoadCompleted` навигационного контейнера и опросить свойство `ExtraData` аргумента события:

```
this.NavigationService.LoadCompleted += new
  LoadCompletedEventHandler(container_LoadCompleted);
```

```
...
void container_LoadCompleted(object sender, NavigationEventArgs e)
{
  if (e.ExtraData != null)
    LoadPhoto((int)e.ExtraData);
}
```

Но есть и более простая схема передачи данных – воспользоваться основным вариантом метода Navigate, который принимает экземпляр Page, и определить в классе целевой страницы конструктор, принимающий дополнительные данные (количество его аргументов может быть произвольным). В приложении Photo Gallery это выглядит следующим образом:

```
int photoId = 10;
// Перейти к экземпляру Page
PhotoPage nextPage = new PhotoPage(photoId);
this.NavigationService.Navigate(nextPage);
```

Чтобы этот пример заработал, в классе PhotoPage должен быть такой конструктор:

```
public PhotoPage(int id)
{
  LoadPhoto(id);
}
```

Достоинство такого подхода состоит в том, что параметры могут быть строго типизированы, поэтому объект PhotoPage может быть уверен, что получил именно целое число. Это гарантирует система типов!

Третий способ – организовать глобальное обобществление данных с помощью коллекции Properties объекта Application, которую мы обсуждали выше в этой главе. Например:

```
// Перейти к экземпляру Page или по URI
Application.Properties["PhotoId"] = 10;
this.NavigationService.Navigate(...);
```

Целевая страница затем может проверить значение в любом месте кода после вызова Navigate:

```
if (Application.Properties["PhotoId"] != null)
  LoadPhoto((int)Application.Properties["PhotoId"]);
```

Такой подход может оказаться удобным, когда нужно сделать данные доступными нескольким страницам (а не просто передать от одной страницы другой). Однако недостатком, как и в первом случае, является отсутствие строгой типизации.

Возврат данных от страницы с помощью PageFunction

Бывает так, что нужно позволить пользователю перейти к некоторой странице, предпринять какое-то действие, а затем автоматически вернуться к предыдущей странице и выполнить ту или иную операцию в зависимости от этого

действия (поэтому необходимо получить данные от страницы, где побывал пользователь). Классический пример – страница настроек. Такое поведение можно сымитировать, осуществив прямой переход к первой странице и передав ей данные одним из первых двух вышеупомянутых способов. Этот процесс показан на рис. 7.8.



Рис. 7.8. *Имитация возврата данных путем прямого перехода к странице, находящейся в стеке обратных переходов*

Но такой способ не всегда удобен. Если вы осуществляете переход по URI, то придется вручную восстановить состояние нового экземпляра `MainPage`, так чтобы оно совпадало с состоянием старого экземпляра. Кроме того, прямая навигация, примененная для имитации обратного перехода, вызывает нежелательные побочные эффекты в журнале.

Вместо этого можно было бы сохранить данные в глобальной коллекции `Application.Properties` и в целевой странице вызвать метод `GoBack` навигационного контейнера для возврата к предыдущей странице. Такой подход работоспособен, но несколько неряшлив, потому что в глобальной коллекции сохраняются данные (причем нетипизированные), имеющие отношение только к двум страницам, а не ко всему приложению.

Поэтому WPF предлагает еще один механизм «возврата» данных предыдущей странице безопасным относительно типов способом с автоматической навигацией обратно к ней. Он показан на рис. 7.9.



Рис. 7.9. *Рациональный поток навигации может быть реализован с помощью PageFunction*

Достигается это с помощью класса с забавным названием `PageFunction`. На самом деле он *является* классом `Page` (поскольку наследует ему), но механизм возврата данных делает его похожим скорее на функцию.

В Visual Studio есть шаблон для создания класса `PageFunction`, аналогичный шаблону для создания `Page`. Вот что вы получаете, выбрав в диалоговом окне Add New Item пункт Page Function (WPF):

```
<PageFunction
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    x:Class="MyProject.PageFunction1"
    x:TypeArguments="sys:String"
    Title="PageFunction1">
  <Grid>
  </Grid>
</PageFunction>
```

Обратите внимание на ключевое слово TypeArguments. PageFunction – в действительности универсальный класс (то есть имеет вид PageFunction<T>), где аргумент-тип представляет собой тип возвращаемого значения. В показанном выше элементе PageFunction возвращается строка. Из-за использования нотации универсальных классов определение PageFunction выглядит несколько запутанно, зато в награду мы получаем безопасность относительно типов, чем не могут похвастаться некоторые рассмотренные выше схемы.

Поскольку PageFunction – подкласс Page, то на элемент этого типа можно переходить точно так же, как на любую другую страницу:

```
PageFunction1 nextPage = new PageFunction1<string>();
this.NavigationService.Navigate(nextPage);
```

Для получения возвращенного значения исходная страница должна обработать событие Return объекта PageFunction:

```
nextPage.Return += new ReturnEventHandler<string>(nextPage_Return);
...
void nextPage_Return(object sender, ReturnEventArgs<string> e)
{
  string returnValue = e.Result;
}
```

Отметим, что в универсальных типах ReturnEventHandler и ReturnEventArgs применяется один и тот же аргумент-тип. Поэтому свойство Result аргумента обработчика события будет иметь тот же тип, что и данные, возвращенные страницей PageFunction (в нашем случае строка).

Страница, производная от PageFunction, может вернуть данные, обернув их типом ReturnEventArgs и вызвав метод OnReturn, унаследованный от базового класса PageFunction:

```
OnReturn(new ReturnEventArgs<string>("the data"));
```

Приложения-гаджеты

WPF существенно упрощает создание непрямоугольных окон верхнего уровня. Имея такую поддержку, вы можете придать стандартному во всех прочих отношениях приложению прихотливое обрамление. Или создать приложение

в виде небольшого гаджета, который выглядит как «плавающий» на рабочем столе объект.

Чтобы воспользоваться этими средствами, нужно сделать следующее:

1. Присвоить свойству `AllowsTransparency` объекта `Window` значение `true`. Если вы делаете это из программы, не устанавливайте его до показа окна. В противном случае будет возбуждено исключение `InvalidOperationException`.)
2. Присвоить свойству `Windowstyle` объекта `Window` значение `None`, чтобы полностью удалить обрамление. (Попытка установить любое другое значение в сочетании с `AllowsTransparency="True"` заканчивается исключением `InvalidOperationException`.)
3. Присвоить свойству `Background` объекта `Window` значение `Transparent`. В результате содержимое не будет окружено непрозрачным прямоугольником.
4. Решить, как пользователь будет перемещать окно по экрану, и в нужных местах вызывать для этой цели метод `DragMove` объекта `Window`. Технически это необязательно, но приложение, окно которого нельзя передвигать, не порадует пользователя.
5. Подумать о добавлении кнопки закрытия, чтобы пользователю не приходилось завершать приложение щелчком правой кнопки мыши по значку на панели задач. Это особенно важно, если свойству `ShowInTaskbar` присвоено значение `false`!

Ниже приведен XAML-файл для такого окна. В нем определяется полупрозрачный красный круг и кнопка закрытия `Close`:

```
<Window x:Class="GadgetWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Height="300" Width="300"
        AllowsTransparency="True" Windowstyle="None" Background="Transparent"
        MouseLeftButtonDown="Window_MouseLeftButtonDown">
  <Grid>
    <Ellipse Fill="Red" Opacity="0.5" Margin="20">
      <Ellipse.Effect>
        <DropShadowEffect/>
      </Ellipse.Effect>
    </Ellipse>
    <Button Margin="100" Click="Button_Click">Close</Button>
  </Grid>
</Window>
```

Эффект `DropShadowEffect` рассматривается в главе 15 «Двумерная графика»; он придает кругу чуть более изысканный вид. С этой разметкой ассоциирован такой застраничный файл:

```
using System.Windows;
using System.Windows.Input;

public partial class GadgetWindow : Window
{
```

```
    public GadgetWindow()
    {
      InitializeComponent();
    }

    void Window_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
    {
      this.DragMove();
    }

    void Button_Click(object sender, RoutedEventArgs e)
    {
      this.Close();
    }
  }
```

Чтобы окно можно было перемещать, обработчик события `MouseLeftButtonDown` просто вызывает метод `Window.DragMove`. Все остальное метод `DragMove` сделает сам. На рис. 7.10 показано, как выглядит это крохотное приложение.

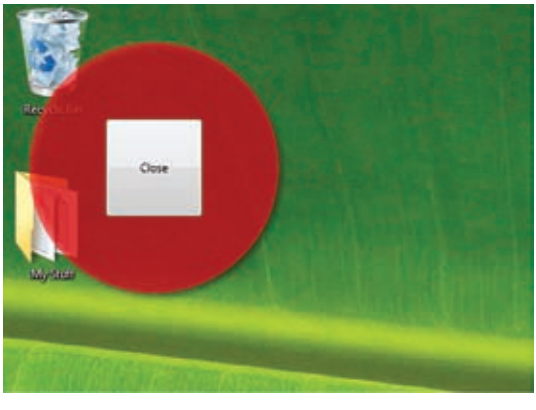


***Рис. 7.10.** Невидимое окно Window с непрямоугольным (и полупрозрачным) содержимым*

XAML-приложения для браузера

WPF поддерживает создание приложений, способных работать непосредственно в веб-браузере. Они называются XAML-приложениями для браузера (XAML Browser Applications – XBAPs), хотя правильнее было бы говорить «WPF-приложения для браузера». XBAP-приложения утрачивают свою привлекательность по мере того, как Silverlight по своим возможностям все больше приближается к WPF. Однако они по-прежнему решают задачу выполнения в браузере WPF-содержимого с частичным доверием без надоедливых вопросов.

FAQ

Работают ли XBAP-приложения в любой операционной системе и в любом браузере?

Нет. В отличие от приложений Silverlight, XBAP-приложениям необходима полная версия .NET Framework (3.0 или выше), поэтому они работают только в Windows и только в браузерах Internet Explorer (или любой программе, поддерживающей элемент управления ActiveX `WebBrowser`) и Firefox (при наличии версии .NET Framework 3.5 или более поздней). Для работы с .NET Framework 4.0 в Firefox необходимо скачать и установить дополнение. (Дополнение для версии 3.5 устанавливается автоматически.)

Создание XBAP-приложения мало чем отличается от создания стандартного приложения Windows при условии, что разработчик не выходит за рамки подмножества .NET, доступного для программы с частичным доверием. Перечислим основные различия:

- По умолчанию доступны не все средства WPF и .NET Framework.
- Навигация интегрирована в браузер.
- Развертывание осуществляется по-другому.

В этом разделе мы рассмотрим все три отличительные особенности XAML-приложений для браузера.

Итак, как же создается XBAP-приложение? В Visual Studio достаточно выполнить следующие шаги:

1. Создать новый проект, в Visual Studio его тип, как и положено, называется WPF Browser Application.
2. Сконструировать пользовательский интерфейс внутри элемента `Page` и написать застраничный код.
3. Откомпилировать и запустить проект.

Если у вас нет Visual Studio, то можно воспользоваться программой MSBuild, задав в проекте соответствующие настройки (см. врезку «КОПНЕМ ГЛУБЖЕ» ниже).

КОПНЕМ ГЛУБЖЕ

Как работают XAML-приложения для браузера

В файлах, которые генерирует Visual Studio, нет ничего специфического именно для XBAP-приложений. Важны лишь некоторые настройки в файле проекта, например:

```
<HostInBrowser>True</HostInBrowser>
<Install>False</Install>
<TargetZone>Internet</TargetZone>
```

В файле проекта есть также настройки, предписывающие отладчику запускать программу `PresentationHost.exe`, а не результат компиляции.

Стандартный исполняемый файл генерируется, но, если его запустить непосредственно, ничего не произойдет, поскольку инфраструктура прерывает выполнение, когда видит, что программа не работает в контексте браузера. Помимо EXE-файла генерируются еще два XML-файла:

- файл с расширением *.manifest* – манифест ClickOnce-приложения;
- файл с расширением *.xbap* – манифест развертывания ClickOnce-приложения (для приложений, отличных от XBAP, такие файлы обычно имеют расширение *.application*)

Ну вот, собственно, и все. XBAP-приложения – это, по существу, ClickOnce-приложения, способные работать только в онлайновом режиме, которые WPF обрабатывает особым образом для лучшей интеграции с браузером.

ПРЕДУПРЕЖДЕНИЕ

Остерегайтесь кэширования ClickOnce!

XBAP-приложения основаны на технологии ClickOnce, в которой имеется механизм кэширования, только мешающий на этапе разработки. Для достижения максимальной производительности ClickOnce-приложение при первом запуске сохраняется в кэше. Последующие запросы на запуск приложения удовлетворяются из кэша, если только не изменился номер версии приложения. (Как и изолированное хранилище, кэш ClickOnce реализован в виде скрытой папки, находящейся внутри папки Documents конкретного пользователя.)

Поэтому, изменив код приложения, перекомпилировав его и снова запустив, вы не увидите результата изменения, если одновременно не зададите другой номер версии! По умолчанию Visual Studio увеличивает номер версии при каждой перекомпиляции (из-за строки `AssemblyVersion("1.0.*")` в исходном файле *AssemblyInfo*), так что вы не столкнетесь с этой проблемой, если явно не присвоите приложению фиксированный номер версии.

Если вы считаете, что увеличение номера версии при каждой компиляции – неприемлемая практика, то можете в любой момент очистить кэш, воспользовавшись инструментом mage.exe из Windows SDK. Достаточно выполнить команды `mage -cc`. Если SDK не установлен, то подойдет также команда

```
rundll32 %windir%\system32\dfshim.dll CleanOnlineAppCache
```

Ограниченный набор возможностей

В случае простенького WPF-приложения достаточно изменить в проекте несколько настроек, перекомпилировать его и получить отлично работающее XAML-приложение для браузера. Но обычно WPF-приложения не настолько просты. Разработка XBAP-приложений осложняется тем фактом, что они работают в зоне Интернета с частичным доверием, а в этом контексте доступны

не все API. Например, если попытаться конвертировать стандартную версию приложения Photo Gallery в XBAP, то сразу обнаружится, что, например, следующий вызов приводит к исключению безопасности (весьма многословному):

```
// Опа! Коду с частичным доверием не разрешено обращаться к этим данным!
AddFavorite(Environment.GetFolderPath(Environment.SpecialFolder.MyPictures));
```

Встроенный в .NET Framework механизм разграничения доступа кода блокирует этот вызов, потому что для его выполнения требуется разрешение `FileIOPermission`, которое по умолчанию зоне Интернета не предоставляется. (Отметим, что пользователь в принципе может расширить набор разрешений, предоставляемых в зоне Интернета, но делать это не рекомендуется из соображений безопасности.)

Большинство разработчиков выясняют, что работает, а что не работает в зоне Интернета, методом проб и ошибок. Некоторые средства не работают, потому что по природе своей небезопасны – например, произвольный доступ к локальной файловой системе или реестру, интероперабельность с неуправляемым кодом или создание новых объектов `Window`. (Создавать элементы `Popup` можно, но они не смогут выйти за границы объемлющего элемента `Page`.) Причины, по которым в зоне Интернета запрещены другие средства, не всегда очевидны, так как ограничения являются результатом особенностей реализации. Некоторые средства могут быть запрещены в одном браузере и разрешены в другом. Например, WPF не разрешает использовать в XBAP-приложении элемент управления `WebBrowser`, *если это приложение работает в браузере Firefox*.

СОВЕТ

Если требуется использовать общий код в стандартном приложении с полным доверием и в XBAP-приложении с частичным доверием, то рекомендуется на этапе выполнения определять, в какой среде приложение работает, и соответственно модифицировать поведение программы. Сделать это можно с помощью статического булевского свойства `BrowserInteropHelper.IsBrowserHosted` из пространства имен `System.Windows.Interop`.

Несмотря на все ограничения, в зоне Интернета доступна весьма обширная функциональность. Можно отображать форматированный текст и мультимедийные данные, писать в изолированное хранилище и читать из него (до 512 Кб), открывать произвольные файлы на веб-сервере. Можно даже запустить стандартное диалоговое окно браузера с помощью команд меню Файл→Открыть и работать с локальными файлами (получив явное разрешение пользователя). Делается это посредством метода `Microsoft.Win32.OpenFileDialog`:

```
string fileContents = null;
OpenFileDialog ofd = new OpenFileDialog();
if (ofd.ShowDialog() == true) // результат может быть равен true, false или null
{
```

```
    using (Stream s = ofd.OpenFile())
    using (StreamReader sr = new StreamReader(s))
    {
      fileContents = sr.ReadToEnd();
    }
  }
```

СОВЕТ

Еще одно различие между XBAP и стандартным приложением Windows заключается в способе передачи параметров (да и вообще любых внешних данных). Проще всего передать параметры в URL-адресе HTML-страницы, содержащей XBAP-приложение, а для получения полного URL (вместе с параметрами) вызвать в самом приложении метод `BrowserInteropHelper.Source`. Другой подход – сохранить информацию в cookie браузера, а для получения этих данных вызвать метод `Application.GetCookie`.

FAQ

Как мне запустить свои собственные компоненты в зоне Интернета?

Используйте механизм, общий для всех компонентов .NET: если пометить сборку атрибутом `AllowPartiallyTrustedCallers` и установить ее в глобальный кэш сборок (а сделать это можно только, если пользователь доверяет вашему коду и готов выполнить его), то любое XBAP-приложение сможет обращаться к находящимся в этой сборке открытым API.

Отметим, что помечать сборку атрибутом `AllowPartiallyTrustedCallers` следует лишь после тщательного анализа. Любая ошибка проектирования или реализации, из-за которой компонент может оказаться непригодным для работы в зоне Интернета, открывает зияющую брешь в системе защиты. И если такое случится, пользователи, возможно, никогда больше не будут доверять вашему коду.

FAQ

Как создать XBAP-приложение с полным доверием?

Если вы хотите воспользоваться средствами, требующими более высокого уровня доверия, и тем не менее выполнять приложение в браузере, то можете сконфигурировать XBAP-приложение с полным доверием. Правда, для этого нужно выполнить два хитрых шага:

1. В манифесте ClickOnce-приложения (*app.manifest*) добавьте строку `Unrestricted="true"` в XML-элемент `PermissionSet`, как показано в следующем примере:

```
<PermissionSet class="System.Security.PermissionSet" version="1"
ID="Custom" SameSite="site" Unrestricted="true"/>
```

2. В файле проекта (с расширением *.csproj* или *.vbproj*) измените строку

```
<TargetZone>Internet</TargetZone>
```

на такую:

```
<TargetZone>Custom</TargetZone>
```

Эквивалентные действия можно проделать и в Visual Studio – в окне свойств проекта на вкладке Security (Безопасность).

После этого XBAP-приложение можно будет развернуть и запустить в зоне Локальный компьютер. Такое приложение с полным доверием можно запускать и в зоне Интернета, но только если пользователь явно включит вас (точнее, сертификат, использованный для подписи манифеста) в список доверенных издателей.

Интегрированная навигация

Все элементы `Page` в XBAP-приложении неявно вложены в элемент `NavigationWindow`. В Internet Explorer 6 и Firefox вы увидите типичную панель с кнопками Назад и Вперед. Обычно это нежелательно, так как немногие XBAP-приложения нуждаются в навигации. Но, даже если нуждаются, иметь отдельные кнопки Назад и Вперед прямо под точно такими же кнопками браузера неестественно. Чтобы убрать ненужную панель навигации, присвойте значение `false` свойству `ShowsNavigationUI` элемента `Page`.

К счастью, в версии Internet Explorer 7 и последующих журнал объекта `NavigationWindow` объединен с собственным журналом браузера, что делает работу гораздо более естественной. Отдельная панель навигации не показывается, а записи, добавленные в журнал WPF, автоматически появляются в списке прямых и обратных переходов, который показывает браузер, – наряду с веб-страницами.

СОВЕТ

Интеграция с журналом браузера в Internet Explorer 7 (и более поздних версиях) применима только к странице `Page` верхнего уровня. Если XBAP-приложение работает в HTML-фрейме `IFRAME`, то панель навигации будет видна, если только не сброшено в `false` свойство `ShowsNavigationUI` WPF-элемента `Page`.

Развертывание

Развернуть XBAP-приложение так же просто, как любое другое ClickOnce-приложение. Все сводится к использованию Мастера публикации в Visual Studio (или инструмента Mage из Windows SDK) и копированию файлов на веб-сервер либо в общую папку. (Веб-сервер следует также правильно сконфигурировать для обслуживания данного контента.)

Самое поразительное в XBAP-приложении – тот факт, что пользователь может установить и запустить его, просто перейдя по его URL-адресу, даже дополнений никаких не требуется (в случае с Internet Explorer). Кроме того, в отличие от других ClickOnce-приложений, браузер не выдает никаких предупреждений, касающихся безопасности, если, конечно, XBAP-приложение не требует каких-то нестандартных разрешений. (Поэтому, чтобы начать работу с таким приложением, даже щелчка мышью не потребуется!)

FAQ

При запуске XBAP не выдаются предупреждения, касающиеся безопасности. Разве это не гигантская брешь в защите?

Любая программа самим фактом своего запуска потенциально рискует открыть брешь в системе защиты. Но наличие нескольких уровней защиты в самой ОС Windows, в Internet Explorer и в каркасе .NET Framework вселяет в команду разработчиков WPF уверенность в том, что хакеры не смогут воспользоваться механизмом XBAP, чтобы обойти защиту. Например, .NET Framework организует «песочницу» (sandbox) поверх той, что уже активирована Internet Explorer. И хотя теоретически уже такой защиты должно быть достаточно, WPF идет дальше и исключает избыточные привилегии уровня операционной системы (например, возможность загружать драйверы устройств) из маркера безопасности объемлющего процесса – просто на тот невероятный случай, когда все остальные уровни защиты взломаны.

СОВЕТ

Наряду с Silverlight, технология XBAP – ключ к использованию WPF-содержимого в разных окружениях. Например, Windows Media Center и гаджеты рабочего стола Windows позволяют разработчику подключать HTML. Стоит разместить XBAP-приложение в такой HTML-странице, как вы получаете приложение для WPF Media Center или WPF-гаджет рабочего стола!

Загрузка файлов по требованию

Технология ClickOnce поддерживает загрузку файлов по требованию приложения, так что можно спроектировать небольшое приложение, которое быстро загружается само, а затем по мере необходимости подгружает дополнительное содержимое, руководствуясь собственной логикой. Эта возможность – настоящее спасение для больших XBAP-приложений, которые в противном случае загружались бы слишком долго, но она применима и к приложениям других типов.

Чтобы воспользоваться ею, следует в проекте Visual Studio поместить несколько автономных файлов в *группу загрузки*. Это можно сделать на вкладке, открываемой командами меню Publish→Application Files (Публикация→Файлы приложения), на странице свойств проекта. Затем вы можете запросить загрузку

этих файлов из программы и получить уведомление по завершении загрузки. Для этой цели в пространстве имен `System.Deployment.Application` (сборка `System.Deployment.dll`) имеются соответствующие API.

В листинге 7.3 показано, как отобразить в пользовательском интерфейсе информацию о ходе загрузки основного содержимого приложения. Предполагается, что выполнение приложения начинается с загрузки страницы `Page1`, застраничный файл которой приведен в листинге 7.3. (Как именно выглядит определенный в XAML-файле пользовательский интерфейс, не столь существенно.) Класс `Page1` инициирует загрузку файлов, отнесенных к группе загрузке MyGroup, а по ее завершении переходит к странице `Page2` (которая предположительно нуждается в каких-то загруженных файлах).

Листинг 7.3. Использование встроенной в технологию ClickOnce возможности загрузки по требованию

```
using System;
using System.Windows.Controls;
using System.Windows.Threading;
using System.Deployment.Application;

public partial class Page1 : Page
{
  public Page1()
  {
    InitializeComponent();
  }

  protected override void OnInitialized(EventArgs e)
  {
    base.OnInitialized(e);

    if (ApplicationDeployment.IsNetworkDeployed)
    {
      // Обработать событие, генерируемое по завершении загрузки
      // всех файлов в группе MyGroup
      ApplicationDeployment.CurrentDeployment.DownloadFileGroupCompleted +=
      delegate {
        // Мы работаем в другом потоке, поэтому вызываем метод
        // GotoPage2 в потоке ГИП с помощью BeginInvoke
        Dispatcher.BeginInvoke(DispatcherPriority.Send,
          new DispatcherOperationCallback(GotoPage2), null);
      };
      ApplicationDeployment.CurrentDeployment.DownloadFileGroupAsync("MyGroup");
    }
    else
    {
      // Мы работаем не в контексте ClickOnce (быть может, просто
      // под отладчиком), поэтому сразу переходим к Page2
      GotoPage2(null);
    }
  }
```

```
    // Переходит к Page2 по завершении загрузки. Принимает и возвращает объект
    // только ради совместимости с сигнатурой метода DispatcherOperationCallback.
    private object GotoPage2(object o)
    {
      return NavigationService.Navigate(new Uri("Page2.xaml", UriKind.Relative));
    }
  }
```

Поддержка загрузки по требованию применяется, только когда приложение работает в сети (а не локально под отладчиком), поэтому мы сначала опрашиваем свойство `ApplicationDeployment.IsNetworkDeployed`, проверяя, можно ли на эту поддержку рассчитывать. Если приложение развернуто не в сети, то все файлы уже находятся в локальной файловой системе, поэтому мы сразу переходим к странице `Page2`. В противном случае инициируем загрузку, вызывая метод `DownloadFileGroupAsync`. Но предварительно присоединяем к событию `DownloadFileGroupCompleted` анонимный делегат, чтобы навигацию можно было продолжить сразу по завершении загрузки. В классе `ApplicationDeployment` определены и другие события на случай, если вы захотите отображать более детальную информацию о ходе загрузки.

Автономные XAML-страницы

Если установлена версия .NET Framework 3.0 или более поздняя, то Internet Explorer получает возможность непосредственно отображать с помощью WPF XAML-файлы точно так же, как обычные HTML-файлы. Поэтому при определенных условиях вместо HTML можно использовать XAML, обеспечивая улучшенную поддержку компоновки, текста, графики и т. д. Правда, есть и ограничения: в автономных XAML-файлах не должно быть процедурного кода и отображаться они могут только в Windows. Впрочем, поэкспериментировать с этой возможностью все равно интересно.

Даже несмотря на отсутствие процедурного кода, в автономных XAML-файлах можно создать довольно развитый динамический интерфейс – благодаря привязке к данным (см. главу 13 «Привязка к данным»). На рис. 7.11 показана версия приложения Photo Gallery в виде автономного XAML-файла. Она отображает статический набор изображений, хранящихся на веб-сервере, но для реализации качественного увеличения использует привязку к данным.

СОВЕТ

Если вы хотите, чтобы сайт мог воспользоваться всем богатством автономного XAML, но при этом был способен показывать обычную HTML-страницу пользователям, не имеющим возможности просматривать XAML, то можете поддерживать две версии контента и динамически выбирать подходящую. Для этого достаточно проверить, есть ли в строке агента пользователя подстрока вида `".NET CLR 3.0"`. Впрочем, я еще не встречал сайта, который применял бы такую уловку. Адаптивное добавление Silverlight решает эту задачу гораздо лучше.

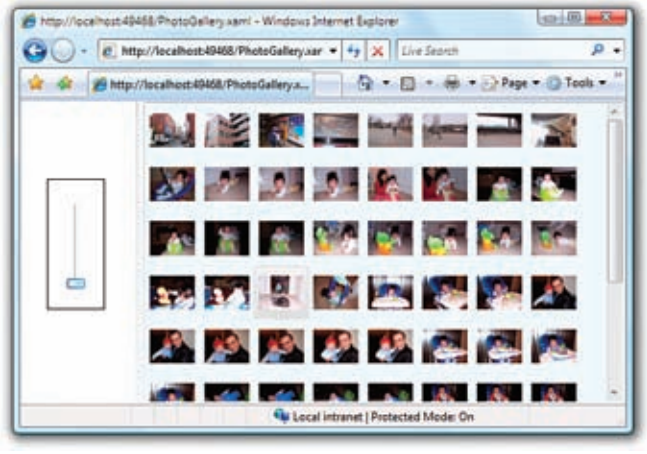


Рис. 7.11. *Приложение Photo Gallery в виде автономной XAML-страницы все равно обладает интересными возможностями*

СОВЕТ

Чтобы одновременно использовать контент в виде HTML и автономного XAML, достаточно поместить один или несколько XAML-файлов во фреймы `IFRAME` на HTML-странице.

Резюме

Средства WPF для создания приложения охватывают все, что необходимо для стандартных приложений Windows, а также позволяют осуществлять навигацию, как в браузере, и выполнять приложения в контексте браузера. В исходном коде приложения Photo Gallery, прилагаемом к книге (доступен по адресу *http://informit.com/title/9780672331190*), демонстрируется, что иногда одна и та же реализация пользовательского интерфейса применима как в традиционном приложении Windows, так и в насыщенной веб-странице без какого бы то ни было кода.

Во всех рассмотренных в этой главе случаях развертывание приложения производится легко и быстро. Единственная шероховатость – необходимость установить подходящую версию .NET Framework. К счастью, вместе с Windows Vista по умолчанию устанавливается WPF 3.0, а вместе с Windows 7 – WPF 3.5. В последующих версиях Windows, скорее всего, по умолчанию будет устанавливаться WPF 4 или более поздняя версия. Так что это требование не является обременительным, если только вам не нужна самая свежая версия .NET Framework.

- Списки переходов
- Настройка элементов на панели задач
- Функция Aero Glass
- Функция TaskDialog

8

Особенности Windows 7

В каждой версии Windows появляется много новой функциональности, интересной для разработчиков, и Windows 7 – не исключение. Как и в Windows Vista, в версии Windows 7 реализован целый ряд новых идей в области пользовательского интерфейса, и все они доступны приложениям. Обладая новыми возможностями, приложение обретет более современный облик и больше понравится пользователям.

В начале этой главы мы посмотрим, как сделать внешний вид WPF-приложения более соответствующим Windows 7 с помощью двух новых механизмов:

- Списки переходов
- Настройка элементов на панели задач

А затем продемонстрируем два средства, появившиеся еще в Windows Vista, но сохраняющих актуальность и в Windows 7:

- Функция Aero Glass
- Функция `TaskDialog`

Списки переходов

Одно из самых значительных нововведений в области пользовательского интерфейса Windows 7 – списки переходов для элементов на панели задач. Список переходов содержит удобные ярлыки, а чтобы посмотреть его, достаточно щелкнуть по элементу на панели задач правой кнопкой мыши или пальцем потянуть его вверх. На рис. 8.1 показан список переходов для Internet Explorer.

Даже если приложение ничего не делает для того, чтобы воспользоваться списком переходов, оно все равно получает такой список по умолчанию. На рис. 8.2 показаны два варианта подразумеваемого по умолчанию списка переходов для приложения Photo Gallery из предыдущей главы: когда оно открыто и когда закрыто. (Список переходов для закрытого приложения можно увидеть, только если это приложение закреплено на панели задач.)

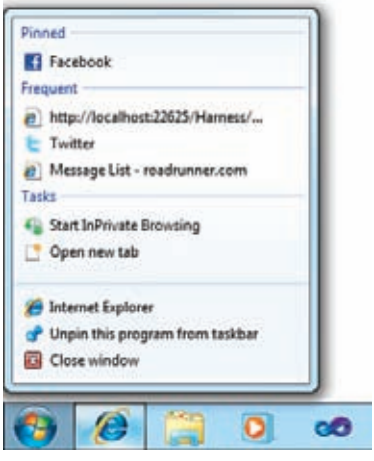


Рис. 8.1. *Список переходов для Internet Explorer может содержать элементы из разных категорий*

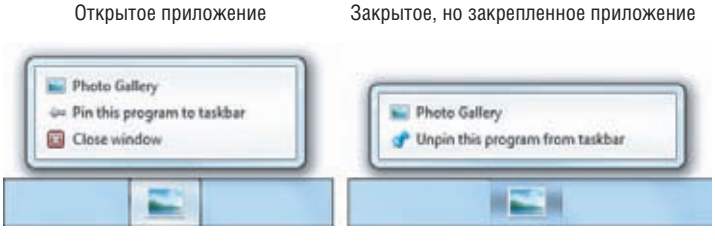


Рис. 8.2. *Список переходов, подразумеваемый по умолчанию для приложения Photo Gallery*

В WPF 4 имеется класс `System.Windows.Shell.JumpList`, который позволяет определить собственный список переходов для приложения с помощью несложного управляемого кода или даже целиком в XAML! Это не означает, что внутри списка переходов можно использовать визуальные элементы WPF, но имеющаяся функциональность раскрывается в виде управляемых объектов с простыми свойствами.

Чтобы связать с приложением список переходов, нужно установить присоединенное свойство с забавным названием `JumpList.JumpList` для экземпляра `Application`, записав в него ссылку на экземпляр класса `JumpList`. А в процедурном коде следует вызвать метод `JumpList.SetJumpList`. Если объект `JumpList` создается или модифицируется в процедурном коде, то для отправки информации об изменениях оболочке Windows следует вызвать метод `Apply` этого объекта.

В классе `JumpList` имеется также свойство содержимого `JumpItems`, которое может содержать элементы двух типов: `JumpTask` и `JumpPath`; оба они наследуют абстрактному классу `JumpItem`.

Элемент JumpTask

С точки зрения пользователя, элементы JumpTask представляют выполняемые действия, например Start InPrivate Browsing (Начать просмотр InPrivate) или Open new tab (Открыть новую вкладку) на рис. 8.1. С точки зрения разработчика, объект JumpTask представляет запускаемую программу (задачу операционной системы). Обычно они применяются для запуска программы-владельца списка с аргументами командной строки, определяющими, что она должна делать.

В листинге 8.1 демонстрируется использование нескольких элементов JumpTask в файле *App.xaml*, взятом из примера Photo Gallery из предыдущей главы и немного модифицированном. Получившийся список переходов показан на рис. 8.3. Отметим, что три нижних пункта (два, если приложение закреплено и закрыто) присутствуют всегда, поэтому наше определение списка переходов влияет лишь на то, что находится выше этих стандартных пунктов.

Листинг 8.1. App.xaml – создание списка переходов с простыми элементами JumpTask

```
<Application x:Class="PhotoGallery.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <JumpList.JumpList>
    <JumpList>
      <JumpTask Title="Launch another instance"
                Description="Launches another instance of this program."/>
      <JumpTask Title="Task #1" Arguments="-task1"
                Description="Performs task #1."/>
      <JumpTask Title="Task #2" Arguments="-task2"
                Description="Performs task #2."/>
    </JumpList>
  </JumpList.JumpList>
</Application>
```

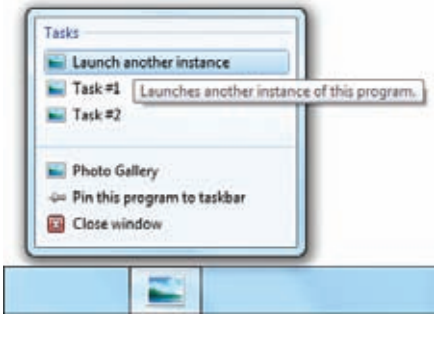


***Рис. 8.3.** Список переходов с тремя простыми элементами JumpTask*

У каждого элемента JumpTask имеется атрибут Title – строка, отображаемая в списке, – и необязательный атрибут Description, то есть всплывающая под-

сказка. Поскольку никаких других свойств не задано, первый элемент JumpTask просто перезапускает приложение Photo Gallery. Это дублирует действие стандартного элемента, расположенного в конце списка переходов, и в реальном приложении не имеет смысла. Но вот следующие два элемента JumpTask передают новому экземпляру Photo Gallery дополнительные аргументы командной строки, инструктируя приложение о необходимости предпринять какие-то другие действия. Photo Gallery может прочитать эти аргументы из свойства Environment.CommandLine и отреагировать соответствующим образом.

СОВЕТ

С точки зрения пользователя, типичная задача в списке переходов не запускает новый экземпляр программы, а приводит к выполнению каких-то действий в уже запущенном экземпляре. Чтобы добиться подобного поведения, можно написать приложение так, чтобы у него всегда было не более одного работающего экземпляра (эта тема обсуждалась в предыдущей главе), и передать этому экземпляру информацию, указанную в командной строке.

Если у приложения имеется нестандартный список переходов, то его элементы появляются также в меню Пуск, когда данное приложение становится текущим. На рис. 8.4 показано, как список переходов, описанный в листинге 8.1, автоматически добавляется в меню Пуск.

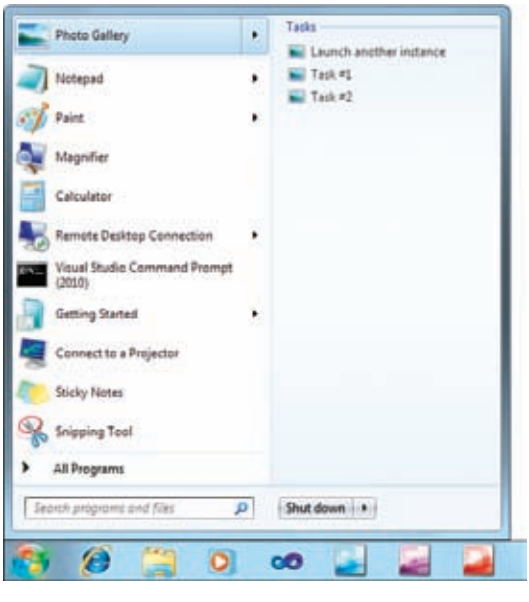


Рис. 8.4. *В меню Пуск автоматически появляется список переходов, показанный на рис. 8.3*

ПРЕДУПРЕЖДЕНИЕ

Отладчик Visual Studio взаимодействует со списками переходов!

При запуске под отладчиком в Visual Studio приложение представляется в виде файла *vshost32.exe*, как показано на рис. 8.5. Свой список переходов вы видите, но значки могут выглядеть иначе, а щелчок по ним не работает (потому что приводит к запуску *vshost32.exe*, а не вашей программы). Еще хуже обстоит дело с элементами `JumpPath`, описанными в следующем разделе, – они не появляются вовсе. Чтобы обойти эту проблему, можно сбросить флажок Enable the Visual Studio hosting process (Включить ведущий процесс Visual Studio) в разделе Debug (Отладка) на странице свойств проекта.

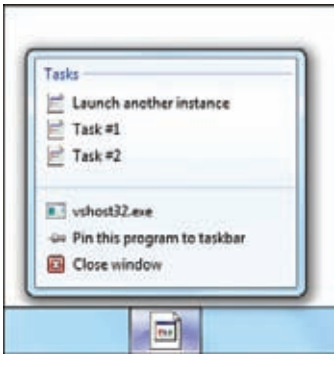


Рис. 8.5. *Ведущий процесс отладчика Visual Studio оказывает влияние на список переходов*

ПРЕДУПРЕЖДЕНИЕ

Списки переходов разделяются всеми экземплярами приложения!

Списки переходов ассоциированы с приложением, а не с его конкретным окном или работающим экземпляром. Все элементы, помещенные в список переходов, сохраняются, даже когда приложение не работает. Если будет запущен второй экземпляр приложения, который включит в список переходов другие элементы, то они заменят элементы, ранее помещенные первым экземпляром.

Настройка поведения JumpTask

У элемента `JumpTask` имеется ряд свойств для установки значков и запуска приложений, отличных от определенных владельцем списка. Эти свойства демонстрируются в листинге 8.2, а на рис. 8.6 показан результат.

Листинг 8.2. App.xaml – демонстрация дополнительных свойств элемента JumpTask

```
<Application x:Class="PhotoGallery.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <JumpList.JumpList>
    <JumpList>
      <JumpTask Title="Magnifier"
                Description="Open the Windows Magnifier."
                ApplicationPath="%WINDIR%\system32\magnify.exe"/>
      <JumpTask Title="Calculator"
                Description="Open the Windows Calculator."
                ApplicationPath="%WINDIR%\system32\calc.exe"
                IconResourcePath="%WINDIR%\system32\calc.exe"/>
      <JumpTask Title="Notepad"
                Description="Open Notepad."
                ApplicationPath="%WINDIR%\system32\notepad.exe"
                IconResourcePath="%WINDIR%\system32\notepad.exe"
                WorkingDirectory="%HOMEDRIVE%%HOMEPATH%"/>
      <JumpTask Title="Internet Explorer (No Add-Ons)"
                Description="Start without ActiveX controls or extensions."
                ApplicationPath="%PROGRAMFILES%\Internet Explorer\iexplore.exe"
                IconResourcePath="%PROGRAMFILES%\Internet Explorer\iexplore.exe"
                WorkingDirectory="%HOMEDRIVE%%HOMEPATH%"
                IconResourceIndex="6" Arguments="-extoff"/>
    </JumpList>
  </JumpList.JumpList>
</Application>
```

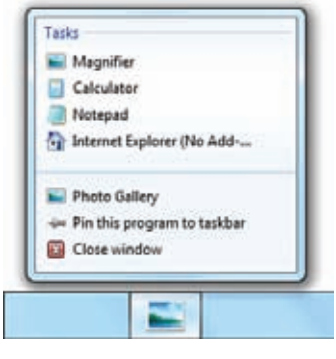


***Рис. 8.6.** Запуск других программ с помощью элементов JumpTask*

Каждый из показанных в листинге элементов `JumpTask` устанавливает какое-то дополнительное свойство, добавляющее очередную возможность. В первом элементе установлено свойство `ApplicationPath` – для запуска программы

magnify.exe. Отметим, что в ApplicationPath можно указывать переменные окружения, так что некоторые пути можно задавать в XAML, а не строить их в процедурном коде.

Во втором элементе JumpTask установлено свойство IconResourcePath, задающее путь к значку. Значок должен быть ресурсом Win32, внедренным в EXE- или DLL-файл. (Можно задать и независимый файл с расширением *.ico*, но тогда придется указать полный путь, не содержащий переменных окружения, так что делать это в XAML-разметке неразумно.) Прописав путь к EXE-файлу, вы можете без труда получить значок для программы по умолчанию. Если свойство IconResourcePath равно null, как в первом элементе JumpTask, то берется программа-владелец списка переходов. Именно поэтому для первого элемента JumpTask показывается значок Photo Gallery.

СОВЕТ

В файлах *%WINDIR%\System32\shell32.dll и %WINDIR%\System32\imageres.dll* имеется много готовых значков, которые вполне можно использовать в элементах JumpTask. Не гарантируется, что во всех версиях Windows они одинаковы, но польза все равно есть.

В третьем элементе JumpTask установлено свойство WorkingDirectory, которое влияет на способ запуска программы (в данном случае Блокнота). В нем, как и в свойствах ApplicationPath и IconResourcePath, можно использовать переменные окружения.

В последнем элементе JumpTask не только задается свойство Arguments, запускающее браузер Internet Explorer в режиме «без надстроек», но и с помощью свойства IconResourceIndex изменяется его значок. Именно поэтому на рис. 8.6 браузер представлен значком с изображением домика, а не синим логотипом «e». В EXE- или DLL-файл можно внедрить длинный список ресурсов значков. Если оставить значение IconResourceIndex по умолчанию, то есть 0, то будет взят самый первый значок (тот, что отображается в оболочке Windows). Если же в EXE- или DLL-файле имеются дополнительные значки, то ими можно воспользоваться, задав значение IconResourceIndex, большее нуля. Задав недопустимый индекс, вы получите стандартный значок, как на рис. 8.5.

СОВЕТ

Если вы не хотите показывать значок рядом с именем элемента JumpTask в списке переходов, присвойте свойству IconResourceIndex значение -1. Этот способ работает вне зависимости от того, задано свойство IconResourcePath явно или нет.

СОВЕТ

Чтобы поставить между элементами JumpTask горизонтальную линию, достаточно добавить в нужное место элемент JumpTask, не задавая для него никаких свойств. На рис. 8.7 показано, что получается, если добавить элементы <JumpTask/> между первыми двумя и последними двумя элементами в листинге 8.2.

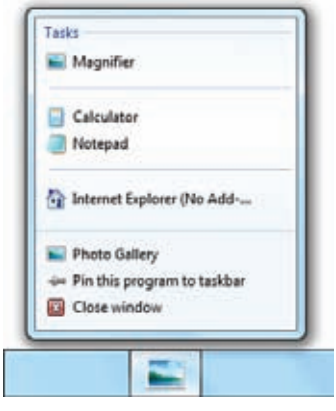


Рис. 8.7. *Добавление горизонтальных разделителей с помощью пустых элементов JumpTask*

Нестандартные категории

Рассматриваемое в этом разделе свойство CustomCategory определено не в классе JumpTask, а в его базовом классе JumpItem. Присвоив ему непустую строку, вы сможете поместить элемент в отдельную секцию с заголовком, отличным от стандартного Tasks (Задачи).

В листинге 8.3 мы поместили один элемент в категорию One и два элемента – в категорию Two. Результат изображен на рис. 8.8.

Листинг 8.3. App.xaml – использование свойства CustomCategory

```
<Application x:Class="PhotoGallery.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <JumpList.JumpList>
    <JumpList>
      <JumpTask Title="Magnifier" CustomCategory="One"
                Description="Open the Windows Magnifier."
                ApplicationPath="%WINDIR%\system32\magnify.exe"/>
      <JumpTask Title="Calculator" CustomCategory="Two"
                Description="Open the Windows Calculator."
```

```
                ApplicationPath="%WINDIR%\system32\calc.exe"
                IconResourcePath="%WINDIR%\system32\calc.exe"/>
        <JumpTask Title="Notepad" CustomCategory="Two"
                Description="Open Notepad."
                ApplicationPath="%WINDIR%\system32\notepad.exe"
                IconResourcePath="%WINDIR%\system32\notepad.exe"
                WorkingDirectory="%HOMEDRIVE%%HOMEPATH%"/>
        <JumpTask Title="Internet Explorer (No Add-Ons)"
                Description="Start without ActiveX controls or extensions."
                ApplicationPath="%PROGRAMFILES%\Internet Explorer\iexplore.exe"
                IconResourcePath="%PROGRAMFILES%\Internet Explorer\iexplore.exe"
                WorkingDirectory="%HOMEDRIVE%%HOMEPATH%"
                IconResourceIndex="6" Arguments="-extoff"/>
      </JumpList>
    </JumpList.JumpList>
  </Application>
```

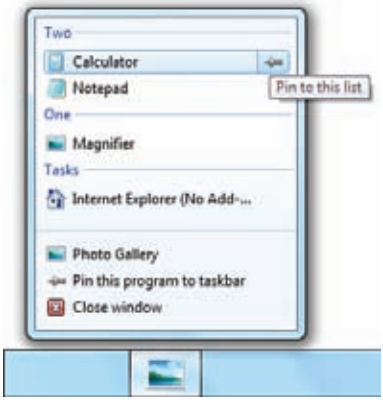


Рис. 8.8. *Задание нестандартных категорий в списке переходов*

ПРЕДУПРЕЖДЕНИЕ

Закрепление JumpTask не работает, если не задано свойство Arguments!

Из-за ошибки в Windows 7 задачи без аргументов закрепить невозможно. Кнопка закрепления присутствует, но при ее нажатии ничего не происходит. К счастью, у большинства задач есть хотя бы один аргумент. Если необходимо запустить программу, которой аргументы не нужны, а фиктивный аргумент передать невозможно, то можно написать промежуточную программу запуска, которая принимает и игнорирует аргумент.

Элементы, находящиеся в нестандартных категориях, автоматически поддерживают закрепление и удаление пользователем (последняя возможность доступна с помощью контекстного меню). Закрепленные элементы перемеща-

ются в категорию Pinned (Закреплено). Впоследствии пользователь может открепить элемент, как показано на рис. 8.9.

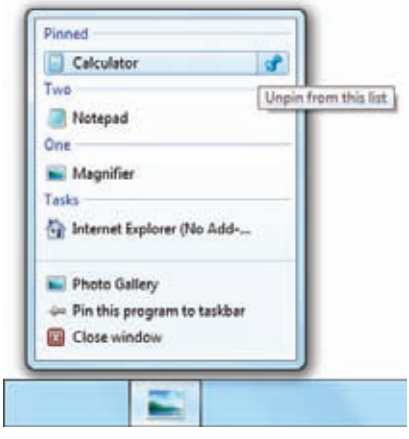


Рис. 8.9. *Закрепление элемента JumpTask из нестандартной категории*

ПРЕДУПРЕЖДЕНИЕ

Нестандартные категории отображаются в порядке снизу вверх!

И элементы `JumpTask`, и нестандартные категории отображаются в том порядке, в котором они хранятся в коллекции `JumpItems`. Но если список `JumpTask` растет сверху вниз, то список категорий – снизу вверх! Именно поэтому категория Two на рис. 8.8 и 8.9 находится над категорией One.

Элемент JumpPath

Если элемент `JumpTask` представляет программу, то `JumpPath` представляет файл, который открывается приложением-владельцем списка. В действительности приложение может использовать элементы `JumpPath`, только если оно зарегистрировано в Windows для обработки файлов с соответствующим расширением. Чтобы выполнить примеры из этого раздела, необходимо временно зарегистрировать приложение как обработчик JPG-файлов (в ходе экспериментов это можно сделать не программно, а в Проводнике Windows, выбрав из контекстного меню файла пункты Открыть с помощью→Выбрать программу (Open With→Choose Default Program).

В листинге 8.4 разметка из листинга 8.3 модифицирована – в коллекцию элементов `JumpTask` добавлены элементы `JumpPath` (и те и другие элементы можно смешивать, потому что они наследуют общему базовому классу `JumpItem`). Поскольку файл существует на диске C: и приложение зарегистрировано для обработки JPG-файлов, то список переходов теперь выглядит, как показано на

рис. 8.10. Если бы хотя бы одно из вышеперечисленных условий не выполнялось, то список переходов выглядел бы, как на рис. 8.8.

Листинг 8.4. App.xaml – добавление JumpPath в разметку из листинга 8.3

```
<Application x:Class="PhotoGallery.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <JumpList.JumpList>
    <JumpList>
      <JumpPath Path="C:\Users\Adam\Pictures\DSC06397.jpg"
                CustomCategory="Photos"/>
      <JumpTask Title="Magnifier" CustomCategory="One"
                Description="Open the Windows Magnifier."
                ApplicationPath="%WINDIR%\system32\magnify.exe"/>
      ...
    </JumpList>
  </JumpList.JumpList>
</Application>
```

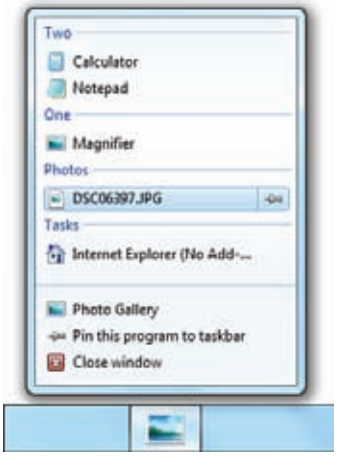


***Рис. 8.10.** В категорию Photos списка переходов добавлен элемент JumpPath*

По умолчанию элементы `JumpPath` помещаются в категорию Tasks, что выглядит странновато. Однако их можно поместить и в другие категории, задав свойство `CustomCategory` (унаследованное от `JumpItem`). Достоинство такого подхода в том, что каждый элемент автоматически становится доступным для закрепления.

Когда пользователь щелкает по элементу DSC06397.jpg, запускается новый экземпляр приложения-владельца, которому в качестве единственного аргумента командной строки передается путь `Path`. Поэтому, если не считать значка и контекстного меню, то элемент `JumpPath` в листинге 8.4 аналогичен следующему элементу `JumpTask`:

```
<JumpTask Title="DSC06397.jpg"
          Arguments="C:\Users\Adam\Pictures\DSC06397.jpg"
          Description="DSC06397 (C:\Users\Adam\My Pictures)"
          CustomCategory="Photos"/>
```

Обязанность учитывать переданный в командной строке аргумент и выполнять «открытие» файла, что бы это ни означало, возлагается на само приложение – точно так же, как в случае прочих элементов JumpTask.

ПРЕДУПРЕЖДЕНИЕ

В свойстве Path элемента JumpPath не поддерживаются переменные окружения!

Именно поэтому в листинге 8.4 путь к JPG-файлу зашит жестко. На практике, однако, это не должно составлять серьезную проблему. Обычно приложения добавляют пути JumpPath динамически в процедурном коде, а в этом случае логика формирования пути может быть произвольной (в том числе с применением переменных окружения).

Недавние и часто посещаемые пути JumpPath

В большинстве приложений – даже в зарегистрированных обработчиках определенных типов файлов – нет причин явно манипулировать элементами JumpPath. Дело в том, что списки переходов автоматически поддерживают две наиболее распространенных категории: *недавние элементы* и *часто посещаемые элементы*.

Чтобы та или другая категория появилась в списке переходов, достаточно установить для свойства ShowRecentCategory и/или ShowFrequentCategory элемента JumpList значение true. Тогда соответствующая категория появится и будет заполняться автоматически. Windows учитывает открытие файла, если оно было произведено посредством диалогового окна Открытие файла (File Open) или в результате использования зарегистрированной ассоциации с типом (например, после двойного щелчка по файлу в Проводнике Windows или щелчка по элементу JumpPath).

Если вы хотите принудительно поместить элемент в какой-либо из этих списков (например, потому, что приложение открывает файлы в обход вышеупомянутых механизмов), то можете вызвать метод JumpList.AddToRecentCategory. У него есть перегруженные варианты, принимающие путь в виде строки, объекта JumpPath и даже объекта JumpTask. Не существует метода AddToFrequentCategory; для того чтобы файл появился в списке часто посещаемых, необходимо достаточно много раз поместить его в категорию недавних.

После добавления обеих категорий в список переходов из листинга 8.4 мы получим результат, изображенный на рис. 8.11.

```
<JumpList ShowFrequentCategory="True" ShowRecentCategory="True">
  <JumpPath Path="C:\Users\Adam\Pictures\DSC06397.jpg"
```

```
            CustomCategory="Photos"/>
  <JumpTask Title="Magnifier" CustomCategory="One"
            Description="Open the Windows Magnifier."
            ApplicationPath="%WINDIR%\system32\magnify.exe"/>
  ...
</JumpList>
```

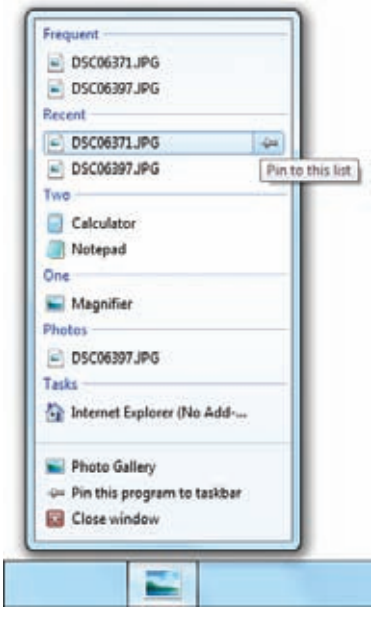


Рис. 8.11. *Использование категорий Recent и Frequent*

Разумеется, использовать обе категории одновременно не имеет особого смысла из-за того, что оба списка сильно перекрываются. Как показано на рис. 8.1, в программе Internet Explorer используется список часто посещаемых файлов (Frequent), тогда как в большинстве других программ – список недавних файлов (Recent). (Windows 7 автоматически предоставляет категорию Recent приложениям, которые не работают со списками переходов явно.)

Реакция на отказ от добавления или на удаление элемента из списка переходов

Если приложение не зарегистрировано как обработчик файлов определенного типа или файл не существует, то Windows отказывается добавлять путь `JumpPath` в коллекцию `JumpItems` списка переходов, а может и удалить уже присутствующий в списке путь. Чтобы предотвратить такое автоматическое удаление, следует обработать событие `JumpItemsRejected` объекта `JumpList`.

Событие `JumpItemsRejected` генерируется *один раз* при удалении одного или нескольких элементов, но не раньше очередного обновления объекта `JumpList`,

например при следующем запуске приложения. Чтобы обработать это событие для списка JumpList, определенного в XAML-разметке, следует присоединить обработчик к XAML. Если объект JumpList создан в процедурном коде, не забудьте присоединить обработчик до вызова метода Apply.

Объект типа JumpItemsRejectedEventArgs, передаваемый обработчику события, содержит список всех отвергнутых элементов JumpItem, а также список значений перечисления JumpItemRejectionReason, а именно:

- NoRegisteredHandler – приложение не зарегистрировано как обработчик файлов данного типа.
- InvalidItem – файл не существует (при работе в версии Windows, предшествующей Windows 7).
- RemovedByUser – элемент удален пользователем.
- None – причина отказа неизвестна.

Если вас интересуют только элементы, удаленные пользователем, то можно ограничиться обработкой события JumpItemsRemovedByUser, вместе с которым передается список удаленных элементов JumpItem. Это имеет смысл, например, для того, чтобы узнать, когда пользователь удалил какие-то из добавленных вами задач JumpTask. В этом случае следует прекратить добавлять такую задачу в список переходов при последующих запусках программы.

КОПНЕМ ГЛУБЖЕ

О моменте возникновения событий JumpItemsRejected и JumpItemsRemovedByUser

Тот факт, что эти события генерируются лишь при следующем вызове метода JumpList.Apply, вносит некоторую путаницу, но в этом плане WPF ограничена поведением используемых Shell Win32 API. Оболочка Windows Shell не предоставляет средств для опроса текущего содержимого списка переходов и не позволяет заранее узнать, будет ли добавление элемента в список принято или отвергнуто. Клиенты (в частности, WPF) должны атомарно обновлять всю категорию. Windows либо примет, либо отвергнет операцию; иногда при этом возвращается осмысленный код ошибки, а иногда – нет. Кроме того, Windows применяет эвристическое правило – отвергает элемент, если пользователь удалял его ранее, но только в случае, когда удаление было произведено в интервале между текущей и предыдущей попыткой обновить список.

Метод Apply существует в классе JumpList для того, чтобы воспрепятствовать добавлению объекта JumpTask или JumpPath, в котором установлена лишь часть обязательных свойств. Может оказаться, что объект с частично заданными свойствами недопустим или допустим, но перестает быть таковым после того, как заданы недостающие свойства. После вызова метода Apply содержимое объекта WPF JumpList соответствует тому, что оболочка считает принятым списком. События (одно или два) генерируются внутри Apply, потому что только там WPF может узнать, что сделал пользователь с момента последнего обновления списка переходов программой.

Настройка элементов на панели задач

Начиная с версии WPF 4 в классе `Window` имеется свойство `TaskbarItemInfo` (типа `System.Windows.Shell.TaskbarItemInfo`), которое позволяет настраивать значок приложения на панели задач или его эскиз. Например, чтобы добавить всплывающую подсказку для эскиза приложения на панели задач, достаточно установить свойство `Description` элемента `TaskbarItemInfo` следующим образом:

```
<Window ...>
  <Window.TaskbarItemInfo>
    <TaskbarItemInfo Description="Custom tooltip"/>
  </Window.TaskbarItemInfo>
  ...
</Window>
```

Или в программе на C#:

```
public MainWindow()
{
  ...
  this.TaskbarItemInfo = new TaskbarItemInfo();
  this.TaskbarItemInfo.Description = "Custom tooltip";
}
```

На рис. 8.12 показан результат этой операции. Разумеется, `TaskbarItemInfo` позволяет задавать не только всплывающие подсказки.

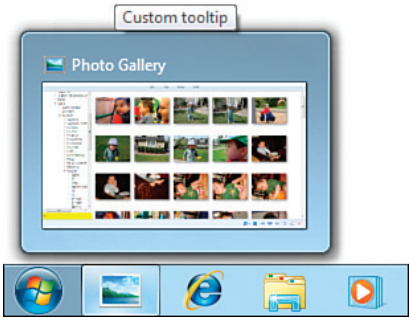


Рис. 8.12. *Всплывающая подсказка, заданная с помощью TaskbarItemInfo.Description*

Индикатор выполнения для элемента на панели задач

Элементы на панели задач поддерживают встроенный индикатор выполнения. Он бывает полезен, когда нужно отобразить состояние долго работающей задачи, не отвлекая внимание пользователя. Эта возможность используется и в Проводнике Windows, и в Internet Explorer. Очень удобно – работаешь в одной программе и время от времени поглядываешь, что делается в другой.

Чтобы показать индикатор выполнения, достаточно задать два свойства объекта `TaskbarItemInfo`: `ProgressValue` и `ProgressState`. `ProgressValue` может принимать значение типа `double` от `0` (0%) до `1` (100%), показывающее процент заполненности полосы индикатора. `ProgressState` может принимать следующие значения, принадлежащие перечислению `TaskbarItemProgressState`:

- `Normal` – показывать зеленый индикатор.
- `Paused` – показывать желтый индикатор.
- `Error` – показывать красный индикатор.
- `Indeterminate` – показывать зеленый анимированный индикатор, а не стандартную частично заполненную полосу, отражающую значение `ProgressValue`.
- `None` – не показывать индикатор. Это значение по умолчанию.

Первые три значения приводят к отображению «обычной» полосы индикатора, различия только в цвете. Желтый означает, что программа приостановлена, а красный свидетельствует об ошибке. Впрочем, интерпретация цветов целиком в ваших руках. Так, ничто не мешает сообщать о продолжении работы, даже если свойство `ProgressState` равно `Paused`.

Значение `Indeterminate` свойства `ProgressState` подходит для случая, когда вы не знаете, как далеко продвинулось выполнение. В этом состоянии значение `ProgressValue` игнорируется, а вместо него показывается стандартная анимация.

Изменять свойства `ProgressState` и `ProgressValue` можно в любой момент, и результат сразу же отражается на индикаторе выполнения. На рис. 8.13 показаны все пять значений `ProgressState` в предположении, что `ProgressValue` равно `.85`.

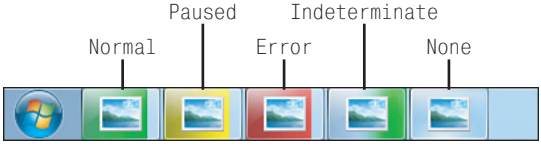


Рис. 8.13. *Пять значений ProgressState, которые поддерживает индикатор выполнения, связанный с элементом на панели задач*

Наложения для элементов на панели задач

Помимо индикаторов выполнения, элементы на панели задач поддерживают наложение маленьких изображений поверх основного значка для передачи дополнительной информации о состоянии. В классе `TaskbarItemInfo` эта возможность реализуется с помощью свойства `Overlay` типа `ImageSource` (этот класс рассматривается в последующих главах).

На рис. 8.14 показано, что происходит после следующего задания наложения:

```
<Window ...>
  <Window.TaskbarItemInfo>
    <TaskbarItemInfo Overlay="overlay.png"/>
  </Window.TaskbarItemInfo>
...
</Window>
```

overlay.png

Результат наложения



***Рис. 8.14.** Наложенное изображение и его представление на панели задач*

Если установлен режим показа мелких значков на панели задач, то наложение не поддерживается, то есть установка этого свойства ничего не дает. Кроме того, попытка воспользоваться *любой* функциональностью класса `TaskbarItemInfo` ни к чему не приводит, если приложение работает в версии Windows, предшествующей Windows 7.

Наложенное изображение постепенно появляется в правом нижнем углу. А когда свойству `Overlay` присваивается значение `null`, наложение столь же постепенно пропадает.

СОВЕТ

При смене изображений в свойстве `Overlay` эффект плавного затухания не используется. Поэтому быстрое изменение значения этого свойства может создать эффект анимации!

Настройка содержимого эскиза

По умолчанию эскиз, отображаемый при задержке указателя мыши над элементом на панели задачи, – это просто уменьшенное изображение текущего окна приложения. Класс `TaskbarItemInfo` позволяет чуть изменить это поведение. Установив свойство `ThumbnailClipMargin` (типа `Thickness`), можно обрезать эскиз по умолчанию.

На рис. 8.15 демонстрируется одно из возможных применений этой функции. Программа Photo Gallery могла бы установить значение `ThumbnailClipMargin` (и корректировать его при изменении размера окна) при просмотре одной фотографии – чтобы убрать обрамление и привлечь внимание к основному содержимому.



***Рис. 8.15.** Обрезка эскиза по размеру самой фотографии, а не всего окна*

Добавление кнопок управления к эскизу на панели задач

И последнее, что позволяет сделать класс `TaskbarItemInfo`, – поместить под эскизом кнопки, имитирующие интерфейс Windows Media Player: Воспроизведение/Пауза, Предыдущая, Следующая. Для этой цели предназначено свойство `ThumbButtonInfos` – коллекция объектов типа `ThumbButtonInfo`.

Хотя `ThumbButtonInfo` не наследует классу WPF `UIElement`, он обладает основными свойствами, ожидаемыми от кнопки. Единственное ограничение состоит в том, что содержимое может быть только объектом типа `ImageSource`. В классе `ThumbButtonInfo` имеется свойство `ImageSource`, определяющее содержимое, свойство `Description`, задающее всплывающую подсказку, и событие `Click`. (Однако, в отличие от класса `Button`, событие `Click` не маршрутизируется.) Кроме того, в классе `ThumbButtonInfo` имеется свойство `Command` и дополняющие его свойства `CommandTarget` и `CommandParameter`, поэтому кнопки могут использоваться в сочетании с применяемыми в приложении командами.

В классе `ThumbButtonInfo` есть также стандартное свойство `Visibility`, способное принимать любое из трех обычных значений. (Приятная неожиданность, если учесть, что компоновка WPF здесь не применяется.) И ряд булевских свойств: `IsEnabled`, `IsInteractive`, `IsBackgroundVisible` и `DismissWhenClicked`; все они, кроме последнего, по умолчанию равны `true`. Под «фоном» (background), упоминаемым в названии свойства `IsBackgroundVisible`, понимается обрамление кнопки, никакого настраиваемого фона у этих кнопок не существует.

На рис. 8.16 показан результат применения следующей разметки в программе Photo Gallery:

```
<Window ...>
<Window.TaskbarItemInfo>
```

```
  <TaskbarItemInfo>
    <TaskbarItemInfo.ThumbButtonInfos>
      <ThumbButtonInfo Description="Previous" Click="..."
                       ImageSource="Images\previousSmall.gif"/>
      <ThumbButtonInfo Description="Slideshow" Click="..."
                       ImageSource="Images\slideshowSmall.gif"/>
      <ThumbButtonInfo Description="Next" Click="..."
                       ImageSource="Images\nextSmall.gif"/>
      <ThumbButtonInfo Description="Undo" Click="..."
                       ImageSource="Images\counterclockwiseSmall.gif"/>
      <ThumbButtonInfo Description="Redo" Click="..."
                       ImageSource="Images\clockwiseSmall.gif"/>
      <ThumbButtonInfo Description="Delete" Click="..."
                       ImageSource="Images\deleteSmall.gif"/>
    </TaskbarItemInfo.ThumbButtonInfos>
  </TaskbarItemInfo>
</Window.TaskbarItemInfo>
  ...
</Window>
```



Рис. 8.16. *В состав эскиза можно включить кнопки управления*

ПРЕДУПРЕЖДЕНИЕ

Во внимание принимаются лишь первые семь элементов ThumbButtonInfo!

Поскольку в окне эскиза есть место только для семи кнопок управления, все последующие элементы коллекции `ThumbButtonInfos` игнорируются. Но тут есть тонкий момент – дополнительные кнопки игнорируются даже в случае, когда для некоторых из первых семи свойство `Visibility` равно `Collapsed` (то есть теоретически есть место для других кнопок). Поэтому, если вы хотите динамически изменять состав кнопок, общее число которых превышает семь, то должны будете добавлять и удалять элементы коллекции, а не просто манипулировать свойством `Visibility`.

FAQ

Как настроить изменение цвета элемента на панели задач при наведении указателя мыши?

Единственный способ – изменить цвета самого значка. Windows выбирает доминирующий цвет для значка и на его основе определяет цвет подсветки.

Функция Aero Glass

Aero Glass – это размытое, прозрачное обрамление окна, которое можно распространить на клиентскую область. Впервые эта функция появилась в Windows Vista. Чтобы воспользоваться ею в WPF-приложении, проще всего вызывать функцию `DwmExtendFrameIntoClientArea` из Win32 API. (Префикс `Dwm` означает Desktop Window Manager.) Это позволяет сделать все окно `Window` похожим на стеклянный лист (как показано на рис. 8.17) или распространить «стекловидность» на заданную часть клиентской области, примыкающую к любой из четырех сторон окна (как показано на рис. 8.18). В любом случае поверх стекла можно располагать WPF-содержимое, как если бы окно было закрашено сплошным цветом.

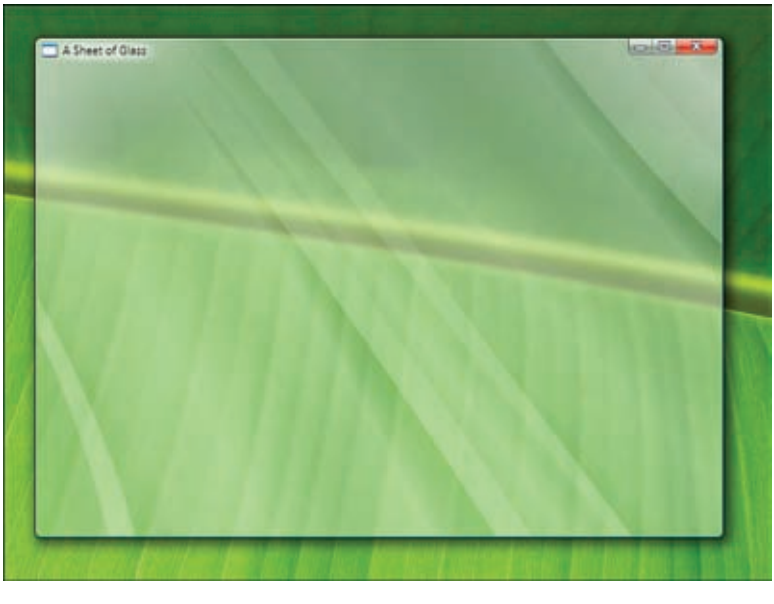


Рис. 8.17. *Стеклянный фон для всего окна Window*

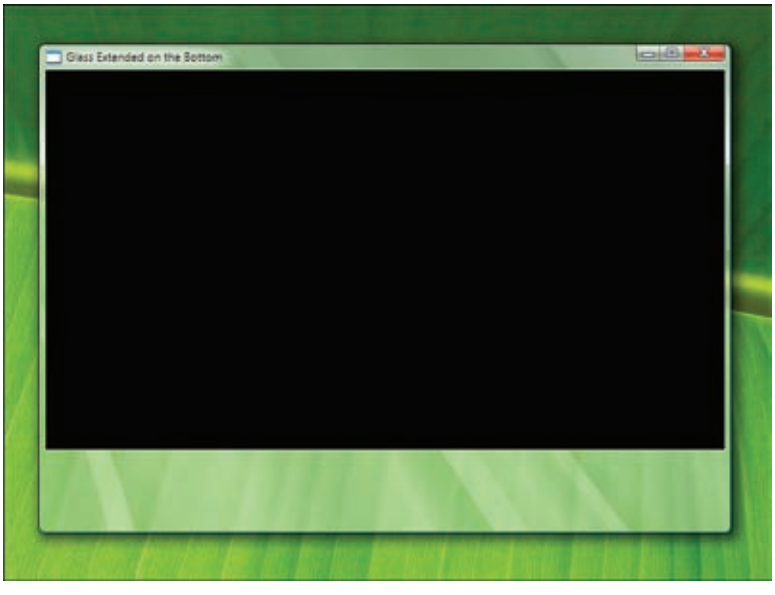


***Рис. 8.18.** Распространение стекла только на нижнюю часть окна*

В программе на Visual C++ вызвать функцию `DwmExtendFrameIntoClientArea` можно непосредственно. Но в языках типа C# или Visual Basic необходимо использовать технологию `PInvoke` (конкретно – задать атрибут `DllImport`). `PInvoke` – ключ к вызову из C# любой функции API из группы Desktop Window Manager. В листинге 8.5 приведены сигнатуры `PInvoke` и простой служебный метод, обертывающий обращения к `PInvoke`.

Листинг 8.5. Использование функции Aero Glass из программы на C#

```
[StructLayout(LayoutKind.Sequential)]
public struct MARGINS
{
  public MARGINS(Thickness t)
  {
    Left = (int)t.Left;
    Right = (int)t.Right;
    Top = (int)t.Top;
    Bottom = (int)t.Bottom;
  }
  public int Left;
  public int Right;
  public int Top;
  public int Bottom;
}

public class GlassHelper
{
```

```
  [DllImport("dwmapi.dll", PreserveSig=false)]
  static extern void DwmExtendFrameIntoClientArea(
    IntPtr hWnd, ref MARGINS pMarInset);

  [DllImport("dwmapi.dll", PreserveSig=false)]
  static extern bool DwmIsCompositionEnabled();

  public static bool ExtendGlassFrame(Window window, Thickness margin)
  {
    if (!DwmIsCompositionEnabled())
      return false;

    IntPtr hwnd = new WindowInteropHelper(window).Handle;
    if (hwnd == IntPtr.Zero)
      throw new InvalidOperationException(
        "The Window must be shown before extending glass.");

    // Устанавливаем прозрачный фон - как с точки зрения WPF, так и в Win32
    window.Background = Brushes.Transparent;
    HwndSource.FromHwnd(hwnd).CompositionTarget.BackgroundColor =
      Colors.Transparent;

    MARGINS margins = new MARGINS(margin);
    DwmExtendFrameIntoClientArea(hwnd, ref margins);
      return true;
  }
}
```

Метод `GlassHelper.ExtendGlassFrame` **принимает объект** `Window` **и уже знакомый нам объект** `Thickness`**; последний определяет части клиентской области, примыкающие к каждой из четырех сторон, на которые следует распространить стекло. (Чтобы получить эффект стеклянного листа, нужно для всех четырех сторон указать значение** `-1`**.) Проверив, что включен режим композиции рабочего стола (необходимое условие для применения Aero Glass), программа создает по объекту** `Thickness` **структуру** `MARGINS`**, получения которой ожидает функция** `DwmExtendFrameIntoClientArea`**, и вызывает эту функцию, передавая ей соответствующий описатель** `HWND`**. Свойству** `Background` **объекта** `Window` **присваивается значение** `Transparent`**, чтобы стекло было прозрачным. Дополнительные сведения об использованной здесь технике см. в главе 19 «Интероперабельность с другими технологиями».**

Любое WPF-окно `Window` **может воспользоваться методом** `GlassHelper.ExtendGlassFrame` **следующим образом:**

```
protected override void OnSourceInitialized(EventArgs e)
{
  base.OnSourceInitialized(e);
  // Это нельзя делать до возникновения события SourceInitialized:
  GlassHelper.ExtendGlassFrame(this, new Thickness(-1));

  // Присоединяем оконную процедуру, чтобы впоследствии
  // можно было понять, включен ли режим композиции рабочего стола
```

```
    IntPtr hwnd = new WindowInteropHelper(this).Handle;
    HwndSource.FromHwnd(hwnd).AddHook(new HwndSourceHook(WndProc));
  }

  private IntPtr WndProc(IntPtr hwnd, int msg, IntPtr wParam,
                         IntPtr lParam, ref bool handled)
  {
    if (msg == WM_DWMCOMPOSITIONCHANGED)
    {
      // Повторно включаем эффект стекла:
      GlassHelper.ExtendGlassFrame(this, new Thickness(-1));
      handled = true;
    }
    return IntPtr.Zero;
  }

  private const int WM_DWMCOMPOSITIONCHANGED = 0x031E;
```

Этот метод следует вызывать не только на этапе инициализации, но и в случае, когда режим композиции стола был сначала выключен, а потом снова включен. Это может произойти как вследствие явного действия пользователя, так и в результате работы программ типа Remote Desktop (Удаленный рабочий стол). Чтобы получать уведомления об изменениях режима композиции рабочего стола, необходимо перехватить сообщение Win32 `WM_DWMCOMPOSITIONCHANGED`. Если хотите лучше разобраться в работе этого кода, обратитесь к главе 19.

На рис. 8.19 показан результат использования этого кода в программе Photo Gallery.



Рис. 8.19. *Программа Photo Gallery с эффектом стекла*

Функция TaskDialog

Часто у разработчика возникает искушение использовать MessageBox там, где уместнее выглядело бы нестандартное диалоговое окно. Но человеку свойственно лениться, поэтому в Windows Vista появился новый, улучшенный вариант класса MessageBox – TaskDialog, – который обладает большей гибкостью. Он соответствует облику современных версий Windows и даже позволяет выполнять глубокую настройку с применением дополнительных элементов управления.

Чтобы воспользоваться этой функциональностью, следует вызвать функцию TaskDialog из Win32 API. Как и при работе с Aero Glass, ключом к вызову этой функции является технология PInvoke. В листинге 8.6 показаны сигнатуры PInvoke для самой функции TaskDialog и связанных с ней типов.

Листинг 8.6. Сигнатуры TaskDialog и относящихся к ней типов на языке C#

```
[DllImport("comctl32.dll", PreserveSig=false, CharSet=CharSet.Unicode)]
static extern TaskDialogResult TaskDialog(IntPtr hwndParent, IntPtr hInstance,
  string title, string mainInstruction,string content,
  TaskDialogButtons buttons, TaskDialogIcon icon);

enum TaskDialogResult
{
  Ok=1,
  Cancel=2,
  Retry=4,
  Yes=6,
  No=7,
  Close=8
}
[Flags]
enum TaskDialogButtons
{
  Ok = 0x0001,
  Yes = 0x0002,
  No = 0x0004,
  Cancel = 0x0008,
  Retry = 0x0010,
  Close = 0x0020
}
enum TaskDialogIcon
{
  Warning = 65535,
  Error = 65534,
  Information = 65533,
  Shield = 65532
}
```

В отличие от MessageBox, функция TaskDialog позволяет задать текст основного сообщения, визуально отделенный от прочего содержимого. Кроме того, можно задавать произвольную комбинацию кнопок. На рис. 8.20 и 8.21 показаны различия между окнами MessageBox и TaskDialog, формируемыми следующим кодом:

```
// Используем MessageBox
result = MessageBox.Show("Are you sure you want to delete '" + filename + "'?",
  "Delete Picture", MessageBoxButton.YesNo, MessageBoxImage.Warning);

// Используем TaskDialog
result = TaskDialog(new System.Windows.Interop.WindowInteropHelper(this).Handle,
  IntPtr.Zero, "Delete Picture",
  "Are you sure you want to delete '" + filename + "'?",
  "This will delete the picture permanently, rather than sending it
   ➥to the Recycle Bin.",
  TaskDialogButtons.Yes | TaskDialogButtons.No,
  TaskDialogIcon.Warning);
```

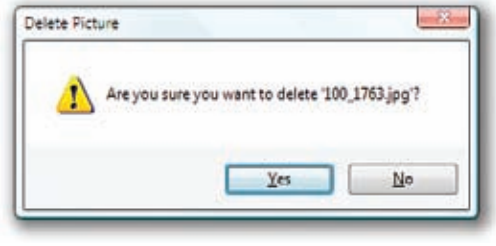


Рис. 8.20. *Окно MessageBox Windows 7 выглядит старомодным и недостаточно проработанным*

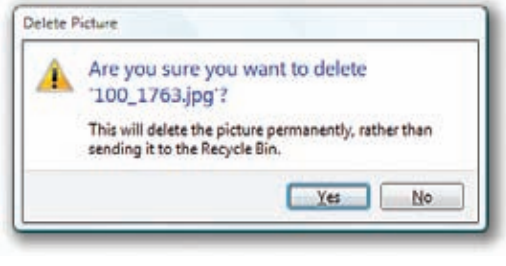


Рис. 8.21. *Аналогичное окно TaskDialog куда дружелюбнее*

ПРЕДУПРЕЖДЕНИЕ

Для работы с TaskDialog необходима версия 6 библиотеки Windows Common Controls DLL (ComCtl32.dll)!

Из соображений совместимости эта версия не подключается по умолчанию. Один из способов подключить версию 6 к своему приложению состоит в том, чтобы поместить в один каталог с исполняемым файлом файл манифеста (с именем *[YourAppName].exe.manifest*), который содержит следующую разметку:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0"
      processorArchitecture="X86" name="YourAppName" type="win32" />
  <description>Your description</description>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
          type="win32" name="Microsoft.Windows.Common-Controls"
          version="6.0.0.0" processorArchitecture="X86"
          publicKeyToken="6595b64144ccf1df" language="*" />
    </dependentAssembly>
  </dependency>
</assembly>
```

Манифест можно также внедрить в сам исполняемый файл в виде ресурса Win32 (с именем `RT_MANIFEST` и `ID`, равным `1`), если вы не хотите включать в дистрибутив еще один файл. Эту работу можно поручить Visual Studio, сославшись на файл манифеста в свойствах проекта.

Если с приложением не связана шестая версия библиотеки стандартных диалоговых окон, то обращение к функции `TaskDialog` приведет к возбуждению исключения с сообщением `"Unable to find an entry point named 'TaskDialog' in DLL 'comctl32.dll'"`. Я рекомендую связывать приложение с этой версией, даже если вы не собираетесь использовать `TaskDialog`. В противном случае *любой* стандартный элемент управления Win32, например `MessageBox`, отображается в старом стиле и в новых версиях ОС выглядит плохо.

СОВЕТ

Для проведения глубокой настройки `TaskDialog` можно также воспользоваться более сложной функцией `TaskDialogIndirect`. В Windows SDK имеются примеры использования этой и других функций Win32 в приложениях .NET. Сигнатуры PInvoke для наиболее популярных функций Win32 API можете посмотреть также на сайте *http://pinvoke.net*.

Резюме

В этой главе рассмотрены новейшие достижения в области пользовательского интерфейса, включенные в ОС Windows 7, и некоторые интересные усовершенствования, появившиеся еще в Windows Vista. К счастью, WPF предоставляет полноценную поддержку для использования этих средств в XAML и процедурном .NET-совместимом языке. Для использования возможностей, добавленных в Windows Vista, необходимо применять технологию PInvoke, которая позволяет обращаться к неуправляемым функциям из Win32 API. Впрочем, доступ к базовой функциональности из управляемого кода все равно несложен.

В этой главе мы рассмотрели все средства Windows 7, к которым есть простой интерфейс из WPF. Однако это лишь малая толика того, что было включено в Win32 API в версии Windows 7 (и Windows Vista). Мы не будем пытаться поведать обо всех хитростях интероперабельности с неуправляемым кодом, необходимых для доступа к некоторым из этих средств, а вместо этого порекомендуем загрузить пакет Windows API Code Pack с сайта *http://code.msdn.microsoft.com/WindowsAPICodePack*. Этот пакет содержит классы и примеры, упрощающие работу со многими новыми функциями Windows 7 и Windows Vista из управляемого кода. Рассматриваются самые разные области – от настройки оболочки и панели задач до датчиков, лингвистических служб и управления питанием.

СОВЕТ

Даже если вы пока не готовы к переносу своего приложения на WPF 4, то все равно можете воспользоваться описанными в этой главе функциями Windows 7, прибегнув к помощи библиотеки WPF Shell Integration Library, которая находится по адресу *http://code.msdn.microsoft.com/WPFShell*.

Эта библиотека содержит совместимую с .NET Framework 3.5 версию классов из пространства имен `System.Windows.Shell` в WPF 4. Между двумя API есть некоторые мелкие несовместимости (например, в библиотеке для версии 3.5 `TaskbarItemInfo` – это присоединенное свойство, а не обычное свойство зависимости), но в целом библиотека удачно прокладывает путь для последующего перехода на новую версию WPF.

СОВЕТ

Планируя воспользоваться средствами, имеющимися в конкретной версии Windows, всегда нужно думать о том, что делать, если программа будет запущена в более ранней версии (если, конечно, вы собираетесь эти более ранние версии поддерживать).

В части списков переходов и средств работы с панелью задач в пространстве имен `System.Windows.Shell` WPF сама заботится о поддержке более старых версий Windows. Если запустить рассмотренные в этой главе примеры в Windows Vista, то код, работающий с классами `JumpList`, `TaskbarItemInfo` и пр., будет выполняться без ошибок, но и без видимого эффекта.

Что же касается прямых обращений к неуправляемым функциям через PInvoke, то вы должны явно проверять версию Windows и соответственно адаптировать поведение программы. В .NET для проверки версии операционной системы можно использовать свойство `System.Environment.OSVersion`. Например:

```
if (System.Environment.OSVersion.Version.Major >= 6)
  // Windows Vista или более поздняя, используем TaskDialog
else
  // Младше Windows Vista, используем MessageBox
```

Основной и дополнительный номер версии Windows 7 равен 6.1, а Windows Vista – 6.0.

III

Элементы управления

Глава 9 «Однодетные элементы управления»

Глава 10 «Многодетные элементы управления»

Глава 11 «Изображения, текст и другие элементы управления»

- Кнопки
- Простые контейнеры
- Контейнеры с заголовками

9

Однодетные элементы управления

Ни один современный презентационный каркас нельзя считать полным без стандартного набора элементов управления, из которых можно быстро собрать традиционный пользовательский интерфейс. И в дистрибутиве Windows Presentation Foundation тоже есть множество таких элементов. С некоторыми из них мы уже встречались в предыдущих главах. А в этой части книги мы проведем обзор большинства встроенных элементов управления, уделив пристальное внимание их уникальным особенностям.

На приведенных в этой книге рисунках показано, как выглядят элементы управления WPF в теме Aero, имеющейся в Windows 7 и Windows Vista. Но у большинства элементов WPF есть и другое обличье, подразумеваемое по умолчанию. Дело в том, что WPF поставляется с DLL-библиотекой тем, которая содержит шаблоны элементов для следующих тем Windows:

- Aero (подразумевается по умолчанию в Windows 7 и Windows Vista)
- Luna (подразумевается по умолчанию в Windows XP)
- Royale (малоизвестная тема из Windows XP Media Center Edition 2005 и Windows XP Tablet PC Edition 2005)
- Classic (имеется в Windows 2000 и более поздних версиях)

На рис. 9.1 показан внешний вид кнопки WPF во всех поддерживаемых темах Windows. Если WPF встречает неподдерживаемую тему, например тему Zune, выпущенную Microsoft в 2006 году, то вместо нее использует тему Classic.



Рис. 9.1. *Внешний вид кнопки WPF в различных темах*

В большинстве случаев различия во внешнем виде малозаметны. Конечно, любому элементу управления можно придать радикально отличающийся облик (основанный на текущей теме или вообще не зависящий от темы) с помощью шаблонов, которые обсуждаются в главе 14 «Стили, шаблоны, обложки и темы».

Встроенные в WPF элементы управления можно грубо разбить на следующие категории, основанные на иерархии наследования:

- Однодетные элементы управления (эта глава)
- Многодетные элементы управления (глава 10 «Многодетные элементы управления»)
- Диапазонные элементы управления (глава 11 «Изображения, текст и другие элементы управления»)
- Все остальное (глава 11)

В этой главе мы рассмотрим *однодетные элементы управления*, то есть элементы, которые могут содержать всего один объект. Все они являются классами, производными от `System.Windows.Controls.ContentControl`, и имеют свойство `Content` типа `Object`, которое может содержать только один объект (одним из таких элементов является кнопка `Button`, продемонстрированная в главе 2 «Все тайны XAML»).

Поскольку содержимое однодетного элемента управления может быть произвольным объектом, то такой элемент, в принципе, может содержать большое дерево объектов. Но *непосредственный* потомок может быть только один. Помимо `Content`, в классе `ContentControl` есть еще один интересный член – булевское свойство `HasContent`. Оно возвращает `false`, если `Content` равно `null`, и `true` в противном случае.

Есть три основных разновидности однодетных элементов управления:

- Кнопки
- Простые контейнеры
- Контейнеры с заголовками

FAQ

? Почему в классе ContentControl определено свойство HasContent? Ведь сравнение Content==null ничуть не сложнее, чем HasContent==false!

Добро пожаловать в мир WPF API, который не всегда выглядит, как привычные .NET API! С точки зрения языка C# свойство `HasContent` избыточно. Но с точки зрения XAML оно очень полезно. Например, с его помощью гораздо проще реализовать триггер свойства, который устанавливает различные значения свойства, когда `HasContent` становится равным `true`.

Класс Window, который мы рассматривали в главе 7 «Структурирование и развертывание приложения», также является однодетным элементом управления. Его свойству Content обычно присваивается ссылка на какую-нибудь панель Panel, например Grid, так что внутри может находиться сколь угодно сложный пользовательский интерфейс.

КОПНЕМ ГЛУБЖЕ

Свойство Content и произвольные объекты

Раз значением свойства Content может быть любой управляемый объект, то, естественно, возникает вопрос, что произойдет, если в роли содержимого выступает невизуальный объект, например Hashtable или TimeZone. Ответ довольно прост: если содержимое является объектом класса, производного от UIElement, то оно визуализируется методом OnRender этого класса. Иначе, если к элементу применим шаблон данных (см. главу 13 «Привязка к данным»), этот шаблон может определять визуализацию от имени данного объекта. Или же вызывается метод ToString объекта-содержимого и возвращенный им текст рисуется внутри элемента TextBlock.

Кнопки

Кнопки, – пожалуй, самый знакомый и неотъемлемый элемент пользовательского интерфейса. Элемент WPF Button, изображенный на рис. 9.1, уже неоднократно встречался на страницах этой книги.

Хотя интуитивно каждый понимает, что такое кнопка, точное определение (по крайней мере, в WPF) может показаться неочевидным. Базовая кнопка – это однодетный элемент управления, по которому можно щелкнуть, но нельзя щелкнуть дважды. Это поведение инкапсулировано в абстрактном классе ButtonBase, которому наследуют несколько разных элементов управления.

В классе ButtonBase имеется событие Click и определено, что понимается под «щелчком». Как и для обычных кнопок Windows, щелчок может возникать в результате нажатия и последующего отпускания левой кнопки мыши или нажатия клавиши Enter либо пробела, если кнопка владеет фокусом.

В классе ButtonBase определено также булевское свойство IsPressed на случай, если вам вдруг понадобится выполнить какие-то действия, когда кнопка нажата (то есть левая кнопка мыши или клавиша пробела нажата, но еще не отпущена).

Однако самым интересным свойством ButtonBase является ClickMode. Оно может принимать значения, определенные в перечислении ClickMode, и управляет тем, при каких условиях генерируется событие Click. Возможные значения: Release (по умолчанию), Press и Hover. Хотя изменение ClickMode для стан-

дартной кнопки может лишь вызвать у пользователя недоумение, эта возможность весьма полезна для кнопок с измененным стилем, которые и на кнопку-то совсем не похожи. В таких случаях принято ожидать, что *нажатие* объекта дает тот же результат, что и *щелчок* по нему.

КОПНЕМ ГЛУБЖЕ

Click и другие события

Чтобы сгенерировать событие `Click`, класс `ButtonBase` анализирует такие более примитивные события, как `MouseLeftButtonDown` и `MouseLeftButtonUp`. Если `ClickMode` равно `Release` или `Press`, то ни одно из этих примитивных событий не всплывает дальше элемента, производного от `ButtonBase`, потому что `ButtonBase` присваивает свойству `MouseButtonEventArgs.Handled` значение `true`. В режиме `Hover` по той же причине не всплывают события `MouseEnter` и `MouseLeave`. Если вы все же хотите обрабатывать примитивные события мыши для элемента, производного от `ButtonBase`, то должны либо обрабатывать `Preview`-версию события (`PreviewMouseLeftButtonDown`, `PreviewMouseLeftButtonUp` и т. д.), либо присоединить свой обработчик в процедурном коде с помощью перегруженного варианта метода `AddHandler`, который игнорирует пометку события как обработанного.

Классу `ButtonBase` прямо или опосредованно наследуют несколько элементов управления, которые мы рассмотрим ниже.

- `Button`
- `RepeatButton`
- `ToggleButton`
- `CheckBox`
- `RadioButton`

Существуют и другие производные от `ButtonBase` элементы, но они предназначены для использования внутри конкретных составных элементов управления, таких как `Calendar` или `DataGrid`.

Класс Button

Класс `Button` в WPF добавляет к тому, что уже дает `ButtonBase`, два простых понятия: *кнопка отмены* и *кнопка по умолчанию*. Оба удобны для применения в диалоговых окнах. Если для некоторой кнопки, находящейся внутри диалогового окна (то есть окна `Window`, показанного методом `ShowDialog`), свойство `Button.IsCancel` равно `true`, то при нажатии этой кнопки окно автоматически закрывается и свойство `DialogResult` принимает значение `false`. Если свойство `Button.IsDefault` равно `true`, то нажатие клавиши Enter приводит к активизации этой кнопки, если только у нее явно не отобран фокус.

FAQ

В чем разница между свойствами IsDefault и IsDefaulted класса Button?

Свойство `IsDefault` доступно для чтения и записи и определяет, будет ли данная кнопка считаться кнопкой по умолчанию. С другой стороны, свойство `IsDefaulted` (с неудачно выбранным именем) предназначено только для чтения. Оно показывает, что кнопка по умолчанию в данный момент находится в таком состоянии, что нажатие клавиши Enter приведет к ее активизации. Другими словами, `IsDefaulted` равно `true` только при выполнении следующих условий: `IsDefault` равно `true` *и* фокусом владеет либо сама кнопка по умолчанию, либо элемент `TextBox` (для которого `AcceptsReturn` равно `false`). Последнее условие позволяет клавише Enter нажать кнопку по умолчанию, не покидая элемент `TextBox`.

FAQ

Как нажать кнопку из программы?

Классу `Button`, как и многим другим элементам управления WPF, соответствует класс в пространстве имен `System.Windows.Automation.Peers`, предназначенный для поддержки автоматизации пользовательского интерфейса: `ButtonAutomationPeer`. Для кнопки с именем `myButton` его можно использовать следующим образом:

```
ButtonAutomationPeer bap = new ButtonAutomationPeer(myButton);
IInvokeProvider iip = bap.GetPattern(PatternInterface.Invoke)
  as IInvokeProvider;
iip.Invoke(); // Это обращение приводит к нажатию кнопки
```

В подобных классах автоматизации есть несколько членов, которые чрезвычайно полезны для автоматизации тестирования и предоставления дополнительных возможностей.

Класс RepeatButton

Класс `RepeatButton` ведет себя так же, как `Button`, но продолжает генерировать событие `Click` до тех пор, пока кнопка нажата. (Кроме того, в нем нет понятия кнопки по умолчанию и кнопки отмены, так как он наследует непосредственно `ButtonBase`.) Частота генерации событий `Click` зависит от значений свойств `Delay` и `Interval`, которые по умолчанию равны `SystemParameters.KeyboardDelay` и `SystemParameters.KeyboardSpeed` соответственно. Внешне элемент `RepeatButton` выглядит точно так же, как `Button` (см. рис. 9.1).

Поведение `RepeatButton` на первый взгляд может показаться странным, но оно полезно (и является стандартным) для кнопок, которые при каждом нажатии увеличивают или уменьшают некоторое значение. Например, кнопки на концах полосы прокрутки демонстрируют именно такое поведение, когда вы

щелкаете по ним и не отпускаете кнопку мыши. Другой пример: если бы вы захотели написать элемент «числовой счетчик» (пока не включенный в WPF), то, наверное, использовали бы пару элементов RepeatButton для управления числовым значением. Класс RepeatButton находится в пространстве имен System.Windows.Controls.Primitives, поскольку, скорее всего, он найдет применение только в составных элементах управления, а не сам по себе.

Класс ToggleButton

ToggleButton – это «залипающая» кнопка, которая сохраняет свое состояние после нажатия (понятия кнопки по умолчанию и кнопки отмены для нее тоже не определены). При первом щелчке свойство IsChecked становится равным true, при следующем – возвращается в false. По умолчанию ToggleButton выглядит точно так же, как Button и RepeatButton.

В классе ToggleButton имеется также свойство IsThreeState; если оно равно true, то IsChecked может принимать три разных значения: true, false или null. На самом деле свойство IsChecked принадлежит типу Nullable<Boolean> (bool? в языке C#). В трехпозиционном режиме первый щелчок устанавливает для IsChecked значение true, второй – null, третий – false и т. д. Чтобы изменить порядок перехода состояний, можно либо перехватывать щелчки, обрабатывая Preview-версии событий мыши, и вручную присваивать свойству IsChecked требуемое значение, либо создать свой подкласс и переопределить в нем метод OnToggle класса ToggleButton, реализовав в нем нужную вам логику.

Помимо свойства IsChecked, в классе ToggleButton определено по одному событию для каждого значения IsChecked: Checked для true, Unchecked для false и Indeterminate для null. Может показаться странным, что не определено единственное событие IsCheckedChanged, но, как выясняется, наличие трех отдельных событий удобно в случае декларативной разметки.

Как и RepeatButton, класс ToggleButton находится в пространстве имен System.Windows.Controls.Primitives, то есть разработчики WPF полагали, что такими кнопками не будут пользоваться без дополнительной настройки. Однако же они вполне естественно выглядят на панели инструментов ToolBar, о чем будет рассказано в главе 10.

Класс CheckBox

Элемент управления CheckBox, изображенный на рис. 9.2, всем хорошо знаком. Но постойте... ведь этот раздел вроде бы посвящен кнопкам, разве не так? Так-то оно так, но давайте приглядимся к характерным особенностям элемента CheckBox в WPF:

- У него имеется единственный вложенный элемент, *задаваемый в разметке* (в отличие от стандартного флажка).
- Для него определено понятие «нажатия» с помощью мыши или клавиатуры.
- После нажатия он сохраняет состояние: отмечен или сброшен.

- Он поддерживает трехпозиционный режим, в котором состояние циклически переключается между вариантами «отмечен», «неизвестно» и «сброшен».

Ничего не напоминает? А должно бы, поскольку `CheckBox` – не что иное, как элемент `ToggleButton` с измененным внешним видом! Класс `CheckBox` наследует `ToggleButton` и переопределяет лишь стиль по умолчанию, используя визуальные элементы, показанные на рис. 9.2.

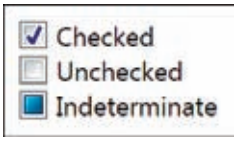


***Рис. 9.2.** Все три состояния элемента управления WPF CheckBox*

КОПНЕМ ГЛУБЖЕ

Поддержка клавиатуры в классе CheckBox

Класс `CheckBox` поддерживает одно дополнительное поведение, отсутствующее в `ToggleButton`, – в целях совместимости с малоизвестной особенностью флажков в Win32. Если флажок `CheckBox` владеет фокусом, то нажатие клавиши плюс (+) отмечает его, а клавиши минус (-) – сбрасывает! Отметим, что это работает, только если свойство `IsThreeState` равно `false`.

Класс RadioButton

`RadioButton` – еще один элемент управления, производный от `ToggleButton`. Его уникальная особенность заключается в поддержке взаимного исключения. Если несколько элементов `RadioButton` помещены в одну группу, то в любой момент времени отмеченным может быть только один из них. Включение какого-то одного переключателя `RadioButton` – даже из программы – автоматически выключает все остальные переключатели в той же группе. На самом деле у пользователя даже нет возможности выключить какой-то элемент `RadioButton`, щелкнув по нему; выключение возможно только из программы. Таким образом, элемент `RadioButton` предназначен для формулирования вопроса, имеющего несколько вариантов ответа. На рис. 9.3 изображен стандартный внешний вид `RadioButton`.

Редко используемое неопределенное состояние элемента `RadioButton` (`IsThreeState=true` и `IsChecked=null`) аналогично выключенному в том смысле, что пользователь не может установить такое состояние щелчком по элементу; это можно сделать только из программы. Щелчок по `RadioButton` переводит его во включенное состояние, но если какой-то переключатель в той же группе находится в неопределенном состоянии, то он в таком состоянии и остается.

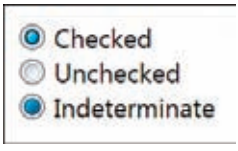


***Рис. 9.3.** Все три состояния элемента управления WPF RadioButton*

Поместить несколько переключателей `RadioButton` в одну группу очень просто. По умолчанию все переключатели, имеющие одного и того же непосредственного логического родителя, автоматически попадают в одну группу. Например, если задана такая разметка:

```
<StackPanel>
  <RadioButton>Option 1</RadioButton>
  <RadioButton>Option 2</RadioButton>
  <RadioButton>Option 3</RadioButton>
</StackPanel>
```

то в любой момент времени включенным может быть только один переключатель.

Но если требуется сгруппировать переключатели каким-то особым образом, то можно воспользоваться свойством `GroupName`, значением которого является строка. Все переключатели с одним и тем же значением `GroupName` помещаются в одну группу (при условии, что у них общий логический корень). Это позволяет группировать переключатели, принадлежащие разным родителям, например:

```
<StackPanel>
  <StackPanel>
    <RadioButton GroupName="A">Option 1</RadioButton>
    <RadioButton GroupName="A">Option 2</RadioButton>
  </StackPanel>
  <StackPanel>
    <RadioButton GroupName="A">Option 3</RadioButton>
  </StackPanel>
</StackPanel>
```

Разные родители

Можно даже создавать подгруппы в пределах одного родителя:

```
<StackPanel>
  <RadioButton GroupName="A">Option 1</RadioButton>
  <RadioButton GroupName="A">Option 2</RadioButton>
  <RadioButton GroupName="B">A Different Option 1</RadioButton>
  <RadioButton GroupName="B">A Different Option 2</RadioButton>
</StackPanel>
```

Разные группы

Разумеется, на практике в последнем примере следует добавить какой-то визуальный элемент, разделяющий подгруппы, иначе пользователь не поймет, что происходит!

Простые контейнеры

В WPF имеется несколько встроенных однодетных элементов управления, для которых, в отличие от кнопки, *не* определено понятие щелчка. У каждого из них есть свои уникальные особенности. Перечислим эти элементы:

- `Label`
- `ToolTip`
- `Frame`

Класс Label

`Label` (метка) – классический элемент управления, который, как и в предшествующих технологиях, может содержать текст. Но, поскольку это однодетный элемент управления в смысле WPF, то в свойстве `Content` может находиться произвольное содержимое – `Button`, `Menu` и т. д., – хотя на практике метка все-таки используется в основном для представления текста.

WPF позволяет размещать текст на экране разными способами, например в элементе `TextBlock`. Уникальная особенность метки – поддержка клавиш доступа. В тексте метки можно выделить одну букву, так что нажатие соответствующей ей комбинации клавиш доступа – эта буква в сочетании с клавишей Alt – будет обрабатываться особым образом. Точнее, можно назначить произвольный элемент, который получит фокус при нажатии этой комбинации. Чтобы выделить букву (которая в зависимости от настроек Windows может быть подчеркнута), достаточно поместить перед ней знак подчеркивания. А чтобы назначить соответствующий целевой элемент, нужно задать свойство метки `Target` (типа `UIElement`).

СОВЕТ

Такие элементы управления, как `Label` и `Button`, поддерживают клавиши доступа путем специальной обработки знака подчеркивания перед буквой, как, например, в названиях `_Open` или `Save_As`. (В Win32 и Windows Forms для этой цели применяется знак амперсанда [&], но подчеркивание значительно удобнее в контексте XML.) Если требуется, чтобы в тексте присутствовал сам знак подчеркивания, то надо записать его два раза подряд, например `__Open` или `Save __As`.

Классическое применение клавиш доступа для метки – сопоставление ей поля ввода `TextBox`. Например, следующая XAML-разметка передает фокус элементу `TextBox` при нажатии сочетания клавиш Alt+U:

```
<Label Target="userNameBox">_User Name:</Label>
<TextBox x:Name="userNameBox"/>
```

Присваивание значения свойству `Target` неявно приводит к вызову конвертера типа `NameReferenceConverter`, который был описан в главе 2. В C# можно просто

записать в свойство ссылку на объект TextBox (в предположении, что метка называется userNameLabel):

```
userNameLabel.Target = userNameBox;
```

Класс ToolTip

Элемент управления ToolTip (всплывающая подсказка) представляет свое содержимое в плавающем прямоугольнике, который появляется, когда пользователь наводит указатель мыши на ассоциированный элемент, и исчезает, когда указатель покидает пределы этого элемента. На рис. 9.4 показано типичное употребление элемента ToolTip, созданного в соответствии со следующей XAML-разметкой:

```
<Button>
  OK
<Button.ToolTip>
  <ToolTip>
    Clicking this will submit your request.
  </ToolTip>
</Button.ToolTip>
</Button>
```

Элемент ToolTip нельзя помещать в дерево элементов UIElement непосредственно. Он должен быть присвоен свойству ToolTip отдельного элемента (это свойство определено в классах FrameworkElement и FrameworkContentElement).



***Рис. 9.4.** Элемент WPF ToolTip*

СОВЕТ

Если задается свойство ToolTip некоторого элемента, то элемент ToolTip можно вообще опустить! Это свойство имеет тип Object, и если присвоить ему ссылку на любой объект, отличный от ToolTip, то система автоматически создаст объект ToolTip и в качестве его содержимого будет использовать значение свойства. Поэтому XAML-код, соответствующий рис. 9.4, можно упростить следующим образом:

```
<Button>
  OK
<Button.ToolTip>
  Clicking this will submit your request.
</Button.ToolTip>
</Button>
```

и даже так:

```
<Button Content="OK" ToolTip="Clicking this will submit your request."/>
```

Благодаря гибкости однодетных элементов управления в WPF элемент ToolTip может содержать все что угодно! В листинге 9.1 показано, как сконструировать всплывающую экранную подсказку (ScreenTip) в стиле Microsoft Office. Результат представлен на рис. 9.5.

Листинг 9.1. Составная всплывающая подсказка в стиле Microsoft Office ScreenTip

```
<CheckBox>
  CheckBox
<CheckBox.ToolTip>
  <StackPanel>
    <Label FontWeight="Bold" Background="Blue" Foreground="White">
      The CheckBox
    </Label>
    <TextBlock Padding="10" TextWrapping="WrapWithOverflow" Width="200">
      CheckBox is a familiar control. But in WPF, it's not much
      more than a ToggleButton styled differently!
    </TextBlock>
    <Line Stroke="Black" StrokeThickness="1" X2="200"/>
    <StackPanel Orientation="Horizontal">
      <Image Margin="2" Source="help.gif"/>
      <Label FontWeight="Bold">Press F1 for more help.</Label>
    </StackPanel>
  </StackPanel>
</CheckBox.ToolTip>
</CheckBox>
```

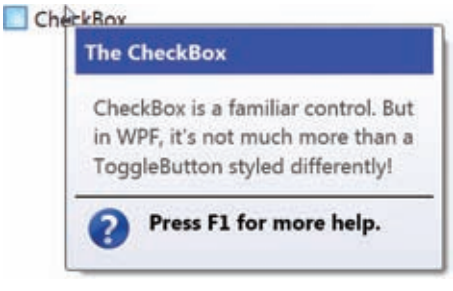


***Рис. 9.5.** В WPF легко создать всплывающую подсказку в стиле экранных подсказок в Microsoft Office*

Хотя ToolTip может содержать интерактивные элементы управления, в частности кнопки, они никогда не получают фокус и щелчок или какие-либо иные действия с ними не дают никакого эффекта.

В классе ToolTip определены события Open и Closed на случай, если вы захотите что-то сделать при появлении и исчезновении подсказки. Есть также несколько свойств для настройки поведения, например: где помещать подсказку, должна ли она оставаться открытой, пока ее явно не закроют, и даже следует ли рисовать тень. Иногда желательно, чтобы один и тот же элемент ToolTip

применялся к нескольким элементам управления, но при этом вел себя поразному в зависимости от того, к чему присоединен. Для таких случаев предусмотрен статический класс ToolTipService.

В классе ToolTipService определено несколько присоединенных свойств, доступных для любого элемента, с которым ассоциирована подсказка ToolTip (но не для самого элемента ToolTip). Некоторые из них совпадают со свойствами ToolTip (причем последним отдается предпочтение в случае, когда значения конфликтуют), но есть и дополнительные. Например, свойство ShowDuration определяет, как долго подсказка должна находиться на экране, когда указатель мыши находится над соответствующим элементом, а свойство InitialShowDelay задает время между приостановкой движения мыши и моментом появления подсказки. В первом примере, относящемся к ToolTip, задать значение ShowDuration можно следующим образом:

```
<Button ToolTipService.ShowDuration="3000">
  ...
</Button>
```

FAQ

Как сделать, чтобы всплывающая подсказка появлялась при задержке указателя мыши над неактивным элементом?

Просто воспользуйтесь присоединенным свойством ShowOnDisabled из класса ToolTipService.

В XAML это будет выглядеть следующим образом:

```
<Button ToolTipService.ShowOnDisabled="True">
...
</Button>
```

А в C# следует вызвать статический метод, соответствующий присоединенному свойству:

```
ToolTipService.SetShowOnDisabled(myButton, true);
```

FAQ

Как принудительно закрыть всплывающую подсказку?

Присвоить false ее свойству IsOpen.

Класс Frame

В элементе управления Frame может находиться произвольное содержимое, как и во всех остальных однодетных элементах управления. Однако он изолирует свое содержимое от остальной части пользовательского интерфейса.

Например, распространение свойств вниз по дереву элементов (обычный механизм наследования свойств в WPF) прекращается по достижении фрейма. Во многих отношениях элемент Frame в WPF ведет себя как HTML-фрейм.

Раз уж речь зашла о HTML, то претензии Frame на всеобщее признание связаны с тем, что он может визуализировать не только WPF-, но и HTML-содержимое. В классе Frame имеется свойство Source типа System.Uri, которому можно присвоить ссылку на любую страницу HTML (или XAML). Например:

```
<Frame Source="http://www.adamnathan.net"/>
```

СОВЕТ

При использовании Frame для переходов между веб-страницами не забывайте обрабатывать событие NavigationFailed, которое происходит при возникновении ошибки, и устанавливать для свойства NavigationFailedEventArgs.Handled значение true. В противном случае необработанное исключение (например, WebException) будет возбуждаться в другом потоке. Объект NavigationFailedEventArgs, который передается обработчику, среди прочего предоставляет доступ к объекту исключения.

В главе 7 было отмечено, что Frame – это навигационный контейнер со встроенным механизмом запоминания истории, пригодный для отображения как HTML-, так и XAML-содержимого. Поэтому можно считать элемент Frame более гибкой версией элемента управления ActiveX Microsoft Web Browser или обертывающего его элемента WPF WebBrowser.

СОВЕТ

По сравнению с фреймами Frame элемент управления WPF WebBrowser (появившийся в WPF 3.5 SP1) предлагает более развитые средства для работы с HTML. Он поддерживает визуализацию HTML-разметки, представленной строкой в памяти или потоком Stream, а также интерактивное взаимодействие с HTML DOM и сценарии. Кроме того, он дает возможность включать Silverlight-содержимое в WPF-приложение: достаточно задать URL, указывающий на XAP-файл Silverlight. Отметим, что WebBrowser *не* является однодетным элементом управления; он не может содержать другие элементы WPF в качестве непосредственных потомков.

К сожалению, когда Frame содержит HTML-разметку, на него распространяется несколько ограничений, не применяемых к другим элементам управления WPF (из-за того, что для визуализации HTML используются механизмы Win32). Например, HTML-содержимое всегда рисуется поверх WPF-содержимого, к нему нельзя применять эффекты, невозможно изменить его свойство Opacity и т. д. Элемент Frame также не поддерживает визуализацию произвольной строки или потока HTML; содержимое должно быть задано в виде пути

или URL-адреса автономного файла. Если необходимо отображать строки, содержащие HTML-код, то лучше воспользоваться элементом управления WPF `WebBrowser`.

КОПНЕМ ГЛУБЖЕ

Свойство Content элемента Frame

Хотя `Frame` – однодетный элемент управления и имеет свойство `Content`, он не рассматривает его как свойство содержимого в смысле XAML. Иными словами, элемент `Frame` в XAML не поддерживает наличие дочернего элемента. Свойство `Content` необходимо задавать явно следующим образом:

```
<Frame>
<Frame.Content>
  ...
</Frame.Content>
</Frame>
```

В классе `Frame` это достигается за счет пометки пустым атрибутом `ContentPropertyAttribute`, который отменяет пометку атрибутом `[ContentProperty("Content")]` базового класса `ContentControl`. Но зачем это сделано?

По словам разработчиков WPF, для того, чтобы предотвратить некорректное использование свойства `Content` класса `Frame`, потому что типичным и ожидаемым способом работы с фреймом является задание свойства `Source`, указывающего на внешний файл. А единственная причина, по которой `Frame` сделан однодетным элементом управления, – необходимость сохранить совместимость с классом `NavigationWindow`, который рассматривался в главе 7. Отметим, что если заданы оба свойства `Source` и `Content`, то предпочтение отдается `Content`.

Контейнеры с заголовками

Все рассмотренные выше однодетные элементы управления добавляют к своему содержимому очень простые визуальные элементы по умолчанию (обрамление кнопки, квадратик флажка и т. д.). Следующие два элемента управления в этом отношении слегка отличаются, потому что добавляют к основному содержимому настраиваемый пользователем *заголовок*. Они наследуют подклассу `ContentControl`, который называется `HeaderedContentControl` и добавляет свойство `Header` типа `Object`.

Класс GroupBox

`GroupBox` – хорошо знакомый элемент для организации групп элементов управления. На рис. 9.6 показан элемент `GroupBox`, охватывающий несколько флажков `CheckBox`. Он создан из следующей XAML-разметки:

```
<GroupBox Header="Grammar">
  <StackPanel>
```

```
    <CheckBox>Check grammar as you type</CheckBox>
    <CheckBox>Hide grammatical errors in this document</CheckBox>
    <CheckBox>Check grammar with spelling</CheckBox>
  </StackPanel>
</GroupBox>
```

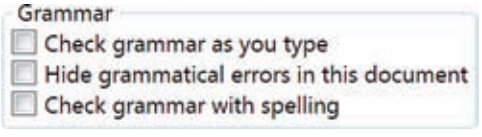


***Рис. 9.6.** Элемент WPF GroupBox*

Как правило, `GroupBox` применяется для охвата нескольких элементов, но поскольку это однодетный элемент управления, то у него может быть всего один непосредственный потомок. Поэтому обычно дочерним элементом `GroupBox` делают какой-то промежуточный элемент, способный содержать несколько потомков. Для этой цели идеально подходит панель, например `StackPanel`.

Как и `Content`, свойство `Header` может ссылаться на произвольный объект, и если это объект класса, производного от `UIElement`, то отображается он естественным образом. Например, если сделать `Header` кнопкой `Button` (см. ниже), то получится результат, показанный на рис. 9.7:

```
<GroupBox>
  <GroupBox.Header>
    <Button>Grammar</Button>
  </GroupBox.Header>
  <StackPanel>
    <CheckBox>Check grammar as you type</CheckBox>
    <CheckBox>Hide grammatical errors in this document</CheckBox>
    <CheckBox>Check grammar with spelling</CheckBox>
  </StackPanel>
</GroupBox>
```

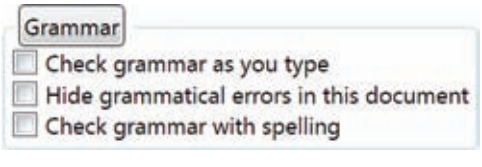


***Рис. 9.7.** Элемент GroupBox с кнопкой Button в заголовке – еще одна демонстрация гибкости модели содержимого в WPF*

На рис. 9.7 находящаяся в заголовке кнопка вполне функциональна. Она может получать фокус, ее можно нажимать и т. д.

Класс Expander

Элемент Expander может заинтриговать, потому что это единственный из рассматриваемых в данной главе элементов управления, не имеющий аналогов в предшествующих технологиях конструирования пользовательских интерфейсов, в том числе в Windows Forms! Expander очень похож на GroupBox, но содержит кнопку, которая позволяет сворачивать и разворачивать внутреннее содержимое (по умолчанию в начальный момент содержимое свернуто).

На рис. 9.8 показаны оба состояния элемента Expander. Этот элемент создан с помощью той же разметки, что и показанный на рис. 9.6, только открывающий и закрывающий теги GroupBox заменены тегами Expander:

```
<Expander Header="Grammar">
  <StackPanel>
    <CheckBox>Check grammar as you type</CheckBox>
    <CheckBox>Hide grammatical errors in this document</CheckBox>
    <CheckBox>Check grammar with spelling</CheckBox>
  </StackPanel>
</Expander>
```

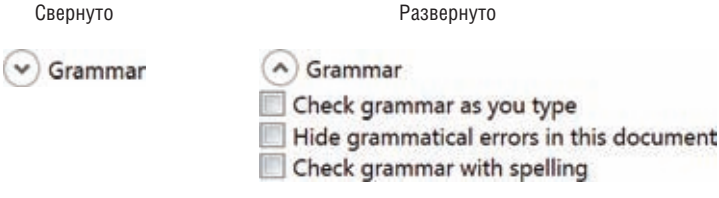


***Рис. 9.8.** Элемент WPF Expander*

В классе Expander определено свойство IsExpanded и события Expanded/Collapsed. Кроме того, он позволяет задать направление развертывания (Up, Down, Left, Right) с помощью свойства ExpandDirection.

Кнопка внутри Expander в действительности представляет собой элемент ToggleButton с измененным стилем. Помимо Expander, еще в нескольких составных элементах используются примитивные элементы управления, такие как ToggleButton и RepeatButton.

Резюме

Никогда еще кнопка не была такой гибкой! В WPF элемент Button, как и все остальные однодетные элементы управления, может содержать все что угодно – при условии, что есть только один непосредственный потомок. Ну а теперь, завершив обзор элементов управления содержимым, перейдем к элементам, которые могут содержать несколько потомков, – *многодетным элементам* (*items controls*).

- Общая функциональность
- Селекторы
- Меню
- Другие многодетные элементы управления

10

Многодетные элементы управления

Помимо элементов управления содержимым, в WPF есть еще одна крупная категория элементов управления – многодетные элементы, которые могут содержать в качестве непосредственных потомков коллекции объектов, а не один-единственный объект. Все многодетные элементы управления наследуют абстрактному классу `ItemsControl`, который, как и `ContentControl`, является прямым подклассом `Control`.

В классе `ItemsControl` содержимое хранится в свойстве `Items` (типа `ItemCollection`). Коллекция может состоять из объектов произвольного типа, по умолчанию они визуализируются так же, как если бы находились внутри однодетного элемента. Иными словами, объекты типа `UIElement` визуализируются естественным образом, а все остальные типы (без учета шаблонов данных) – в виде текстового блока `TextBlock`, содержащего строку, которую возвращает метод `ToString`.

Встречавшийся ранее элемент `ListBox` – пример многодетного элемента управления. В предыдущих главах мы добавляли в список только объекты типы `ListBoxItem`, а вот в следующем примере добавим произвольные объекты:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <Button>Button</Button>
  <Expander Header="Expander"/>
  <sys:DateTime>1/1/2012</sys:DateTime>
  <sys:DateTime>1/2/2012</sys:DateTime>
  <sys:DateTime>1/3/2012</sys:DateTime>
</ListBox>
```

(Поскольку мы использовали конструкцию `sys:DateTime` вместо `x:DateTime`, то эта разметка будет работать в качестве как автономного, так и откомпилированного XAML-кода.)

Дочерние элементы неявно добавляются в коллекцию Items, потому что Items – свойство содержимого. Этот список визуализируется, как показано на рис. 10.1. Оба элемента UIElement (Button и Expander) отображаются нормально и полностью интерактивны. Три объекта DateTime представляются в соответствии с тем, что возвращает их метод ToString.



***Рис. 10.1.** Список ListBox, содержащий произвольные объекты*

В главе 2 «Все тайны XAML» отмечалось, что свойство Items доступно только для чтения. Это означает, что в первоначально пустую коллекцию можно добавлять объекты, из нее можно удалять объекты, но нельзя записать в Items ссылку на совершенно другую коллекцию. В классе ItemsControl имеется еще одно свойство – ItemsSource, – которое поддерживает заполнение элемента объектами из уже существующей произвольной коллекции. Использование свойства ItemsSource более подробно рассматривается в главе 13 «Привязка к данным».

СОВЕТ

Чтобы не усложнять изложение, мы в этой главе заполняем многодетные элементы управления только визуальными элементами. Однако предпочтительным является другой подход: поместить в многодетный элемент невизуальные элементы (например, специализированные бизнес-объекты) и с помощью шаблонов данных определить способ их визуализации. Подробнее шаблоны данных обсуждаются в главе 13.

Общая функциональность

Помимо свойств Items и ItemsSource, в классе ItemsControl есть еще несколько интересных свойств, а именно:

- HasItems – доступное только для чтения булевское свойство, упрощающее анализ наличия элементов в коллекции из декларативного XAML-кода. В программе на C# можно использовать это свойство или просто проверить значение Items.Count.
- IsGrouping – еще одно булевское свойство, доступное только для чтения. Информирует о том, разбиты ли объекты, входящие в элементы управления, на группы верхнего уровня. Группировка производится прямо в классе

ItemsCollection, который включает несколько свойств для управления группами и присвоения им имен. Подробнее о группировке вы узнаете в главе 13.

- AlternationCount и AlternationIndex – эти два свойства позволяют задать чередующиеся стили объектов-потомков в зависимости от индекса в коллекции. Например, если AlternationCount равно 2, то элементам с четным индексом будет назначен один стиль, а элементам с нечетным индексом – другой. Пример использования этих свойств приведен в главе 14 «Стили, шаблоны, обложки и темы».
- DisplayMemberPath – строковое свойство; в него можно занести имя свойства каждого объекта (или более сложное выражение), которое изменяет порядок его визуализации.
- ItemsPanel – свойство, позволяющее изменить способ организации объектов внутри многодетного элемента управления, не заменяя полностью его шаблон.

В следующих двух разделах последние два свойства описываются более подробно.

DisplayMemberPath

На рис. 10.2 показано, что происходит, если в рассмотренном выше списке ListBox задать свойство DisplayMemberPath:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         xmlns:sys="clr-namespace:System;assembly=mscorlib"
         DisplayMemberPath="DayOfWeek">
  <Button>Button</Button>
  <Expander Header="Expander"/>
  <sys:DateTime>1/1/2012</sys:DateTime>
  <sys:DateTime>1/2/2012</sys:DateTime>
  <sys:DateTime>1/3/2012</sys:DateTime>
</ListBox>
```

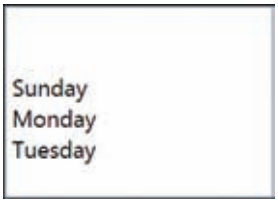


***Рис. 10.2.** Тот же список ListBox, что на рис. 10.1, но добавлено свойство DisplayMemberPath, равное DayOfWeek*

Если DisplayMemberPath равно DayOfWeek, то WPF отображает не сам объект, а значение его свойства DayOfWeek (день недели). Поэтому-то на рис. 10.2 объекты типа DateTime представлены как Sunday, Monday и Tuesday. (Это основанная

на методе ToString визуализация значений перечисления DayOfWeek – типа свойства DayOfWeek.) Поскольку в классах Button и Expander нет свойства DayOfWeek, то они отображаются в виде пустых текстовых блоков TextBlock.

КОПНЕМ ГЛУБЖЕ

Путь к свойству в WPF

Свойство DisplayMemberPath поддерживает синтаксис так называемого пути к свойству, который используется в WPF в нескольких местах, в частности для привязки к данным и анимации. Основная идея состоит в том, чтобы записать последовательность из одного или нескольких свойств, которую можно было бы использовать в процедурном коде для получения требуемого значения. Простейший пример пути – имя одного свойства, но если значением свойства является составной объект, то можно обратиться к его свойствам (и т. д.), разделяя отдельные имена свойств точками, как принято в C#. Этот синтаксис поддерживает даже индексаторы и массивы.

Представьте, к примеру, объект, в котором определено свойство FirstButton типа Button, причем в текущий момент значением свойства Content кнопки является строка "OK". Тогда значение этой строки ("OK") можно представить таким путем к свойству:

```
FirstButton.Content
```

А следующий путь к свойству представляет длину этой строки (2):

```
FirstButton.Content.Length
```

А такой путь – первый символ строки ('O'):

```
FirstButton.Content[0]
```

Эти выражения вполне соответствуют синтаксису C#, разве что приводить типы не требуется.

ItemsPanel

Как и в случае всех остальных элементов управления WPF, смыслом многодетных элементов является не их внешний вид, а способ хранения нескольких объектов, а часто также способ логического выбора объектов. Внешний облик всех элементов управления WPF можно изменить, применив другой шаблон, но для многодетных элементов есть и более короткий способ – заменить лишь часть шаблона, отвечающую за организацию хранящихся в нем объектов. Этот мини-шаблон, который еще называют *внутренней панелью* (items panel), позволяет подменить панель, применяемую для организации объектов, оставив все прочие аспекты элемента управления неизменными.

В качестве внутренней панели разрешается использовать любую из панелей, рассмотренных в главе 5 «Компоновка с помощью панелей» (и вообще любой класс, производный от Panel). Например, список ListBox по умолчанию располагает хранящиеся в нем объекты вертикально, но следующий XAML-код

подставляет вместо этой панели WrapPanel, как в примере программы Photo Gallery из главы 7 «Структурирование и развертывание приложения»:

```
<ListBox>
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <WrapPanel/>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
  ...
</ListBox>
```

Перевод этого XAML-кода на процедурный язык нетривиален, поэтому покажем, как решить ту же задачу в C#:

```
FrameworkElementFactory panelFactory = new
  FrameworkElementFactory(typeof(WrapPanel));
myListBox.ItemsPanel = new ItemsPanelTemplate(panelFactory);
```

А вот пример подстановки нестандартной панели FanCanvas, которую мы реализуем в главе 21 «Компоновка с помощью нестандартных панелей»:

```
<ListBox>
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <custom:FanCanvas/>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
  ...
</ListBox>
```

На рис. 10.3 показан результат применения этой разметки к приложению Photo Gallery (попутно список ListBox обертывается элементом Viewbox) в случае, когда выбран один объект. Несмотря на нестандартную внутреннюю компоновку, элемент ListBox сохраняет все особенности поведения, касающиеся выбора объектов.



Рис. 10.3. *Список ListBox с нестандартной панелью FanCanvas в качестве ItemsPanel*

FAQ

Как заставить ListBox располагать свои объекты по горизонтали, а не по вертикали?

По умолчанию в списке ListBox используется панель VirtualizingStackPanel, которая располагает находящиеся в ней объекты по вертикали. Следующий код подменяет ее другой панелью, VirtualizingStackPanel, в которой свойству Orientation явно присвоено значение Horizontal:

```
<ListBox>
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel Orientation="Horizontal"/>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
  ...
</ListBox>
```

СОВЕТ

В нескольких многодетных элементах управления для повышения производительности в качестве ItemsPanel по умолчанию используется панель VirtualizingStackPanel. В WPF 4 эта панель поддерживает новый режим, еще больше повышающий производительность прокрутки, но устанавливать его нужно явно. Для этого следует присвоить присоединенному свойству VirtualizingStackPanel.VirtualizationMode значение Recycling. В таком случае панель повторно использует («рециклирует») контейнеры, в которых хранятся видимые на экране объекты, а не создает новый контейнер для каждого объекта.

Если взглянуть на подразумеваемый по умолчанию шаблон для такого многодетного элемента управления, как ListBox, то мы увидим элемент ItemsPresenter, задача которого – выбрать подходящую панель ItemsPanel:

```
<ControlTemplate TargetType="{x:Type ListBox}">
  <Border ...>
    <ScrollViewer Padding="{TemplateBinding Padding}" Focusable="false">
      <ItemsPresenter SnapsToDevicePixels="{TemplateBinding SnapsToDevicePixels}"/>
    </ScrollViewer>
  </Border>
  <ControlTemplate.Triggers>
    ...
  </ControlTemplate.Triggers>
</ControlTemplate>
```

Присутствие ScrollViewer в шаблоне элемента по умолчанию объясняет, откуда берется поведение прокрутки. Управлять тем, как осуществляется прокрутка в многодетном элементе управления, позволяют различные присоединенные свойства элемента ScrollViewer.

Управление поведением прокрутки

На примере списка ListBox приведем подразумеваемые по умолчанию значения следующих свойств:

- ScrollViewer.HorizontalScrollBarVisibility — Auto
- ScrollViewer.VerticalScrollBarVisibility — Auto
- ScrollViewer.CanContentScroll — true
- ScrollViewer.IsDeferredScrollingEnabled — false

Если CanContentScroll равно true, то прокрутка производится пообъектно, если же false — то попиксельно. Последний режим обеспечивает более плавную прокрутку, но не гарантирует совмещения первого объекта с краем списка.

Если свойство IsDeferredScrollingEnabled равно false, то прокрутка производится синхронно с перетаскиванием ползунка. Если же оно равно true, то содержимое ScrollViewer обновляется только после прекращения перетаскивания, когда будет отпущена кнопка мыши. Если в многодетном элементе управления используется виртуализирующая панель и он содержит много сложных объектов, то установка для IsDeferredScrollingEnabled значения true может дать существенный прирост производительности за счет отказа от визуализации промежуточных состояний. Например, Microsoft Outlook именно так прокручивает длинные списки.

Ниже приведен пример списка ListBox, в котором установлены все четыре присоединенных свойства, чтобы изменить поведение ScrollViewer в подразумеваемом по умолчанию шаблоне:

```
<ListBox
  ScrollViewer.HorizontalScrollBarVisibility="Disabled"
  ScrollViewer.VerticalScrollBarVisibility="Disabled"
  ScrollViewer.CanContentScroll="False"
  ScrollViewer.IsDeferredScrollingEnabled="True"
>
  ...
</ListBox>
```

ListBox — разумеется, не единственный многодетный элемент управления. Многодетные элементы можно разделить на три основных группы, которые обсуждаются в следующих разделах: селекторы, меню и все остальные.

Селекторы

Селекторами называются многодетные элементы управления, объекты которых можно индексировать и — что более важно — выбирать. Абстрактный класс Selector, производный от ItemsControl, добавляет несколько свойств, необходимых для поддержки выбора. Например, следующие три похожих свойства предназначены для получения и установки текущего выбранного объекта:

- `SelectedIndex` – отсчитываемое от нуля целое число, равное индексу выбранного объекта, или `-1`, если ничего не выбрано. Объекты нумеруются в порядке добавления в коллекцию.
- `SelectedItem` – сам выбранный объект.
- `SelectedValue` – значение выбранного объекта. По умолчанию оно совпадает с самим объектом, то есть `SelectedValue` – то же самое, что и `SelectedItem`. Однако с помощью свойства `SelectedValuePath` можно задать имя произвольного свойства или даже выражение, которое будет представлять значение объекта (`SelectedValuePath` работает аналогично `DisplayMemberPath`).

Все три свойства допускают чтение и запись, поэтому с их помощью можно не только получать текущий выбранный объект, но и устанавливать его.

В классе `Selector` определены также два присоединенных свойства, применяемые к отдельным объектам:

- `IsSelected` – булевское свойство, позволяющее выбрать или отменить выбор объекта (либо узнать, в каком состоянии он сейчас находится).
- `IsSelectionActive` – доступное только для чтения булевское свойство, которое сообщает, владеет ли выбранный объект фокусом.

В классе `Selector` имеется также событие `SelectionChanged`, которое позволяет получать уведомления об изменении выбранного объекта. В главе 6 «События ввода: клавиатура, мышь, стилус и мультисенсорные устройства» мы пользовались им для списка `ListBox`, когда демонстрировали работу с присоединенными событиями.

В состав WPF входит пять элементов управления, производных от `Selector`:

- `ComboBox`
- `ListBox`
- `ListView`
- `TabControl`
- `DataGrid`

Мы опишем их в следующих разделах.

Элемент ComboBox

Изображенный на рис. 10.4 элемент `ComboBox` позволяет выбрать из списка один объект. Он очень популярен, потому что занимает мало места на экране. В *поле выбора* отображается только объект, выбранный в данный момент, а весь остальной список *раскрывается* по требованию. Чтобы раскрыть или закрыть список, можно щелкнуть мышью, а также нажать сочетание клавиш Alt+стрелка вверх, Alt+стрелка вниз либо клавишу F4.

В классе `ComboBox` определены два события – `DropDownOpened` и `DropDownClosed` – и свойство `IsDropDownOpen`. Все вместе они позволяют реагировать на раскрытие или закрытие списка. Например, можно отложить заполнение `ComboBox` до момента раскрытия списка – обработав событие `DropDownOpened`. Отметим, что

свойство IsDropDownOpen допускает чтение и запись, то есть с его помощью можно напрямую управлять состоянием раскрытия.

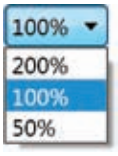


***Рис. 10.4.** Элемент WPF ComboBox в раскрытом виде*

Режимы работы поля выбора

Элемент ComboBox поддерживает режим, в котором пользователь может вводить в поле выбора произвольный текст. Если текст совпадает с каким-то из присутствующих в списке элементов, то этот элемент автоматически становится выбранным. В противном случае ни один элемент не будет выбран, но введенный текст сохраняется в свойстве Text элемента ComboBox, так что программа может получить к нему доступ. Этот режим контролируется двумя неудачно названными свойствами – IsEditable и IsReadOnly, – по умолчанию равными false. Кроме того, имеется свойство StaysOpenOnEdit; если оно равно true, то список остается раскрытым, когда пользователь щелкает по полю выбора (так ведут себя раскрывающиеся списки в Microsoft Office в противоположность стандартным спискам Win32).

Если поле выбора является полем ввода, то выбранный объект можно отображать только в виде простой строки. Это не страшно, если в списке ComboBox и так хранятся строки (или однодетные элементы управления, содержащие строки). Но если в списке находятся более сложные объекты, то необходимо сообщить ComboBox, как получить их строковое представление.

В листинге 10.1 показана XAML-разметка элемента ComboBox, содержащего составные объекты. В каждом объекте отображается кадр презентации PowerPoint, так что все вместе напоминает галерею в духе Microsoft Office, где имеется эскиз и краткое описание объекта. Однако в типичной галерее Office поле выбора может содержать только простой текст, а не выбранный элемент во всей полноте. На рис. 10.5 показан результат визуализации разметки из листинга 10.1, а также то, как он будет выглядеть, если установить для свойства IsEditable значение true.

Листинг 10.1. Список ComboBox с составными элементами, напоминающий галерею Microsoft Office

```
<ComboBox>
  <!-- Объект #1 -->
  <StackPanel Orientation="Horizontal" Margin="5">
    <Image Source="CurtainCall.bmp"/>
    <StackPanel Width="200">
      <TextBlock Margin="5,0" FontSize="14" FontWeight="Bold"
```

```
        VerticalAlignment="center">Curtain Call</TextBlock>
      <TextBlock Margin="5" VerticalAlignment="center"
        TextWrapping="Wrap">
        Whimsical, with a red curtain background that represents
        a stage.
      </TextBlock>
    </StackPanel>
  </StackPanel>
  <!-- Объект #2 -->
  <StackPanel Orientation="Horizontal" Margin="5">
    <Image Source="Fireworks.bmp"/>
    <StackPanel Width="200">
      <TextBlock Margin="5,0" FontSize="14" FontWeight="Bold"
        VerticalAlignment="center">Fireworks</TextBlock>
      <TextBlock Margin="5" VerticalAlignment="center"
        TextWrapping="Wrap">
        Sleek, with a black sky containing fireworks. When you need
        to celebrate PowerPoint-style, this design is for you!
      </TextBlock>
    </StackPanel>
  </StackPanel>
  ...другие объекты...
</ComboBox>
```

IsEditable=False (по умолчанию) IsEditable=True

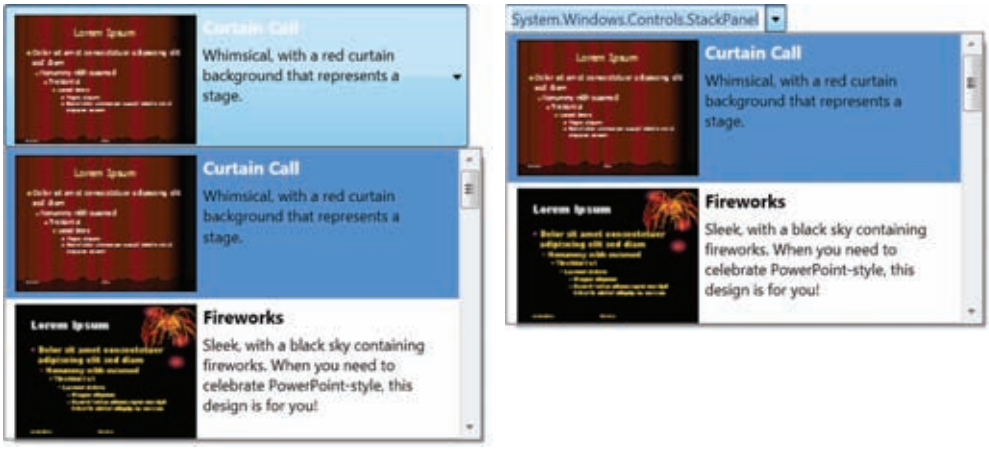


Рис. 10.5. *По умолчанию установка для IsEditable значения true приводит к отображению в поле выбора строки, возвращаемой методом ToString*

Понятно, что выводить в поле выбора имя типа `"System.Windows.Controls.StackPanel"` никуда не годится, и тут приходит на помощь класс `TextSearch`. В нем определены два присоединенных свойства, позволяющих управлять тем, что отображается в редактируемом поле выбора.

FAQ

В чем разница между свойствами IsEditable и IsReadOnly элемента ComboBox?

Если `IsEditable` равно `true`, то поле выбора `ComboBox` превращается в поле ввода. Свойство `IsReadOnly` управляет тем, можно ли изменять текст в этом поле, – точно так же, как свойство `IsReadOnly` элемента `TextBox`. Таким образом, поле `IsReadOnly` не имеет смысла, если `IsEditable` не равно `true`, а тот факт, что `IsEditable` равно `true`, еще не означает, что текст в поле выбора можно редактировать. В табл. 10.1 сведены особенности поведения `ComboBox` при различных комбинациях этих свойств.

Таблица 10.1. Поведение ComboBox при всех возможных комбинациях свойств IsEditable и IsReadOnly

IsEditable	IsReadOnly	Описание
`false`	`false`	В поле выбора отображается копия выбранного объекта, и вводить туда произвольный текст запрещено (это поведение по умолчанию)
`false`	`true`	То же, что и выше
`true`	`false`	В поле выбора отображается текстовое представление выбранного объекта, и разрешено вводить произвольный текст
`true`	`true`	В поле выбора отображается текстовое представление выбранного объекта, но вводить произвольный текст запрещено

К элементу `ComboBox` можно присоединить свойство `TextSearch.TextPath` и тем самым указать, какое свойство (или субсвойство) объекта отображать в поле выбора. Механизм работы такой же, как у свойств `DisplayMemberPath` и `SelectedValuePath`; единственное различие заключается в способе использования конечного значения.

Для объектов в листинге 10.1 напрашивается решение отображать в поле выбора содержимое первого текстового блока `TextBlock`, потому что оно содержит заголовок (`"Curtain Call"` или `"Fireworks"`). Поскольку этот элемент `TextBlock` вложен в две панели `StackPanel`, то в пути к требуемому свойству нужно сначала упомянуть внутренний элемент `StackPanel` (второй потомок каждого объекта), а уже потом сам `TextBlock` (первый потомок внутренней панели `StackPanel`). Таким образом, присоединенное свойство `TextPath` в случае листинга 10.1 будет выглядеть следующим образом:

```
<ComboBox IsEditable="True" TextSearch.TextPath="Children[1].Children[0].Text">
  ...
</ComboBox>
```

Однако такое решение слишком хрупко – путь к свойству перестанет работать, если изменить структуру объекта. Не обрабатывается также случай разнородных объектов; объекты, не соответствующие структуре TextPath, представляются в поле выбора пустыми строками.

В классе TextSearch есть еще одно присоединенное свойство Text; оно более гибкое, но применяться должно к индивидуальным объектам списка ComboBox. Значением свойства Text может быть литеральный текст, отображаемый в поле выбора для данного элемента. В листинге 10.1 им можно воспользоваться следующим образом:

```
<ComboBox IsEditable="True">
  <!-- Объект #1 -->
  <StackPanel TextSearch.Text="Curtain Call" Orientation="Horizontal" Margin="5">
    ...
  </StackPanel>
  <!-- Объект #2 -->
  <StackPanel TextSearch.Text="Fireworks" Orientation="Horizontal" Margin="5">
    ...
  </StackPanel>
  ...другие объекты...
</ComboBox>
```

Можно одновременно задавать и свойство TextSearch.TextPath для ComboBox в целом, и свойство TextSearch.Text для отдельных объектов. В таком случае TextPath дает представление в поле выбора по умолчанию, а Text переопределяет его для тех объектов, где присутствует.

На рис. 10.6 показан результат описанного выше задания свойств TextSearch.TextPath и TextSearch.Text.



***Рис. 10.6.** Благодаря присоединенному свойству TextSearch получается список, похожий на галерею Office*

СОВЕТ

Можно подавить применение свойства TextSearch, задав свойство IsTextSearchEnabled элемента ItemsControl равным false. Свойство IsTextSearchCaseSensitive, также определенное в классе ItemsControl (по умолчанию равно false), указывает, надо ли при сравнении введенного текста с текстами присутствующих в списке объектов принимать во внимание регистр букв.

FAQ



Как получить новый выбранный объект в обработчике события SelectionChanged?

Событие SelectionChanged предназначено для элементов управления, допускающих выбор нескольких объектов, поэтому для селектора типа ComboBox, позволяющего выбрать только один объект, работать с ним не очень удобно. Передаваемый обработчику события объект типа SelectionChangedEventArgs имеет два свойства типа IList: AddedItems и RemovedItems. Свойство AddedItems содержит множество вновь выбранных объектов, а свойство RemovedItems — множество ранее выбранных объектов. Если разрешено выбирать только один объект, то получить его можно следующим образом:

```
void ComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
  if (e.AddedItems.Count > 0)
    object newSelection = e.AddedItems[0];
}
```

Но не следует заранее предполагать, что какие-то элементы вообще выбраны (в приведенном выше коде это проверяется)! Мало того что выбор элемента в списке ComboBox можно отменить программно, так еще это может сделать и пользователь, если IsEditable равно true, а IsReadOnly — false. Если в этом случае пользователь введет в поле выбора значение, не совпадающее ни с одним из присутствующих в списке объектов, то событие SelectionChanged произойдет, но коллекция AddedItems будет пустой.

Класс ComboBoxItem

Класс ComboBox неявно обертывает каждый хранящийся в нем объект объектом ComboBoxItem. (В этом можно убедиться, написав программу, которая пройдет вверх по визуальному дереву, начиная с любого из объектов списка.) Но это можно сделать и явно (кстати, ComboBoxItem — однодетный элемент управления). В листинге 10.1 это будет выглядеть следующим образом:

```
<!-- Объект #1 -->
<ComboBoxItem TextSearch.Text="Curtain Call">
  <StackPanel Orientation="Horizontal" Margin="5">
  ...
  </StackPanel>
```

```
</ComboBoxItem>
<!-- Объект #2 -->
<ComboBoxItem TextSearch.Text="Fireworks">
  <StackPanel Orientation="Horizontal" Margin="5">
  ...
  </StackPanel>
</ComboBoxItem>
...другие объекты...
```

Отметим, что свойство TextSearch.Text теперь нужно присоединять к элементу ComboBoxItem, поскольку StackPanel больше не является самым внешним элементом хранимого объекта. Придется также модифицировать значение свойства TextSearch.TextPath, записав в него путь Content.Children[1].Children[0].Text.

FAQ

А зачем мне самому обертывать объекты в ComboBoxItem?

В классе ComboBoxItem есть полезные свойства – IsSelected и IsHighlighted – и полезные события – Selected и Unselected. Применение ComboBoxItem позволяет также избежать странного поведения при показе однодетных элементов управления в поле выбора (когда IsEditable равно false): если объектом в списке ComboBox является однодетный элемент управления, то в поле выбора показывается не весь элемент, а только его свойство Content. Если же содержательный объект обернут в ComboBoxItem (тоже однодетный элемент), то содержимым последнего будет как раз исходный объект, который и требовалось показать.

Так как ComboBoxItem – однодетный элемент управления, то его удобно использовать для добавления в список ComboBox простых строк (вместо того, чтобы обертывать их в TextBlock или Label). Например:

```
<ComboBox>
  <ComboBoxItem>Item 1</ComboBoxItem>
  <ComboBoxItem>Item 2</ComboBoxItem>
</ComboBox>
```

Элемент ListBox

Уже знакомый нам элемент управления ListBox аналогичен ComboBox, только все объекты отображаются прямо в области, занятой элементом (если они не помещаются, то появится полоса прокрутки). На рис. 10.7 показан элемент ListBox, содержащий те же объекты, что были определены в листинге 10.1.

Пожалуй, самая важная особенность ListBox состоит в том, что он поддерживает выбор нескольких объектов. Этим режимом управляет свойство SelectionMode, которое может принимать три значения (определенных в перечислении SelectionMode):

- Single – одновременно может быть выбран только один объект, как и в ComboBox. Это значение по умолчанию.



Рис. 10.7. *Элемент WPF ListBox*

- `Multiple` – одновременно может быть выбрано несколько объектов. Щелчок по невыбранному объекту добавляет его в коллекцию `SelectedItems`, а щелчок по выбранному объекту удаляет его из этой коллекции.
- `Extended` – одновременно может быть выбрано несколько объектов, но поведение оптимизировано для выбора одного объекта. Чтобы в этом режиме выбрать несколько объектов, следует во время щелчка мышью удерживать нажатой клавишу Shift (чтобы выбирать соседние элементы) или Ctrl (чтобы выбирать произвольные, необязательно соседние элементы). Точно так же ведет себя элемент управления `ListBox` в Win32.

Как у `ComboBox` имеется компаньон `ComboBoxItem`, так и у `ListBox` есть компаньон `ListBoxItem`. С этим классом мы уже встречались в предыдущих главах. На самом деле `ComboBoxItem` наследует классу `ListBoxItem`, в котором и определены свойство `IsSelected` и события `Selected` и `Unselected`.

КОПНЕМ ГЛУБЖЕ

Свойства ListBox и множественный выбор

Хотя в классе `ListBox` имеется свойство `SelectedItems`, которым можно пользоваться вне зависимости от режима `SelectionMode`, он также наследует от класса `Selector` свойства `SelectedIndex`, `SelectedItem` и `SelectedValue`, не укладывающиеся в модель множественного выбора.

Если выбрано несколько объектов, то свойство `SelectedItem` просто указывает на первый элемент в коллекции `SelectedItems` (то есть тот, которым был выбран первым), а свойства `SelectedIndex` и `SelectedValue` возвращают индекс и значение этого объекта. Впрочем, для элементов управления, поддерживающих множественный выбор, этими свойствами лучше не пользоваться. Отметим, что в классе `ListBox` *не* определены свойства `SelectedIndices` и `SelectedValues`.

СОВЕТ

Прием с использованием свойства TextSearch, продемонстрированный ранее для элемента ComboBox, сохраняет актуальность и для ListBox. Например, если объекты на рис. 10.7 аннотировать соответствующими значениями TextSearch.Text, то при нажатии клавиши F в момент, когда ListBox владеет фокусом, выбранным станет объект Fireworks. Если бы TextSearch не было задано, то нажатие клавиши S привело бы к передаче фокуса списку, потому что S – первая буква в строке System.Windows.Controls.StackPanel. (И такое поведение показалось бы пользователю странным!)

FAQ

Как добиться плавной прокрутки ListBox?

По умолчанию ListBox прокручивается пообъектно. Поскольку шаг прокрутки рассчитывается на основе высоты объекта, то в случае больших объектов прокрутка может происходить рывками. Чтобы список прокручивался плавно, с шагом в несколько пикселов, не зависящим от высоты объектов, проще всего присвоить значение false свойству ScrollViewer.CanContentScroll, присоединенному к элементу ListBox, как было показано в предыдущей главе.

Однако имейте в виду, что в таком режиме теряется возможность виртуализации списка. Под *виртуализацией* понимается оптимизация создания дочерних элементов – они создаются только в момент, когда оказываются видны на экране. Виртуализация возможна только в случае, когда для создания объектов, хранящихся в списке, применяется привязка к данным, поэтому установка для свойства CanContentScroll значения false может негативно сказаться на производительности работы списка, привязанного к данным.

FAQ

Как отсортировать объекты в списке ListBox (да и в любом другом элементе типа ItemsControl)?

В основе сортировки лежит механизм, реализованный в классе ItemsCollection, поэтому он равным образом применим ко всем элементам, производным от ItemsControl. В классе ItemsCollection имеется свойство SortDescriptions. Это коллекция, которая может содержать сколько угодно объектов типа System.ComponentModel.SortDescription. Каждый такой объект описывает одно свойство, по которому производится сортировка, а также направление сортировки – по возрастанию или по убыванию. Например, следующий код сортирует последовательность объектов ContentControl по их свойству Content:

```
// Сначала очищаем имеющиеся описания сортировки
myItemsControl.Items.SortDescriptions.Clear();
// Затем сортируем по свойству Content
myItemsControl.Items.SortDescriptions.Add(
  new SortDescription("Content", ListSortDirection.Ascending));
```

FAQ

Как снабдить объекты в элементе ItemsControl идентификаторами автоматизации, видными в инструментальных средствах, например в программе UI Spy?

Самый простой способ снабдить любой элемент, производный от `FrameworkElement`, идентификатором автоматизации – установить его свойство `Name`, поскольку именно оно по умолчанию применяется для целей автоматизации. Но если вы хотите назначить элементу идентификатор, отличный от его имени, то просто запишите желаемое значение в присоединенное свойство `AutomationProperties.AutomationID` (из пространства имен `System.Windows.Automation`).

Элемент ListView

Элемент управления `ListView`, производный от `ListBox`, выглядит и ведет себя, как `ListBox`, с тем отличием, что по умолчанию установлен режим `Extended SelectionMode`. Однако класс `ListView` добавляет также свойство `View`, которое расширяет возможности настройки внешнего вида, не ограничиваясь одним лишь выбором нестандартной панели `ItemsPanel`.

Свойство `View` принадлежит типу `ViewBase`, абстрактному классу. В состав WPF входит один конкретный подкласс этого класса, `GridView`. По умолчанию он очень похож на вид Таблица (Details) в Проводнике Windows. (На самом деле в бета-версиях WPF класс `GridView` даже назывался `DetailsView`.)

На рис. 10.8 представлен простой элемент `ListView`, созданный из следующей XAML-разметки, в которой предполагается, что префикс `sys` соответствует пространству имен `System` из сборки `mscorlib.dll`:

```
<ListView>
  <ListView.View>
    <GridView>
      <GridViewColumn Header="Date"/>
      <GridViewColumn Header="Day of Week"
        DisplayMemberBinding="{Binding DayOfWeek}"/>
      <GridViewColumn Header="Year" DisplayMemberBinding="{Binding Year}"/>
    </GridView>
  </ListView.View>
    <sys:DateTime>1/1/2012</sys:DateTime>
    <sys:DateTime>1/2/2012</sys:DateTime>
    <sys:DateTime>1/3/2012</sys:DateTime>
</ListView>
```

В классе `GridView` имеется свойство содержимого `Columns`, в котором хранится коллекция объектов `GridViewColumn`, а также другие свойства, управляющие поведением заголовков столбцов. В WPF определен элемент `ListViewItem`, производный от `ListBoxItem`. В данном случае объекты `DateTime` неявно обернуты элементами `ListViewItem`, поскольку явно это не указано.



***Рис. 10.8.** Элемент управления WPF ListView с видом GridView*

Объекты, хранящиеся в списке `ListView`, описываются в виде простого списка, как и в случае `ListBox`, поэтому ключом к отображению разных данных в различных столбцах служит свойство `DisplayMemberBinding` класса `GridViewColumn`. Идея в том, что в каждой строке `ListView` может находиться составной объект, а в столбцах отображаются свойства или субсвойства этого объекта. Но, в отличие от свойства `DisplayMemberPath`, определенного в классе `ItemsControl`, для работы со свойством `DisplayMemberBinding` необходима привязка к данным (см. главу 13).

Интересно, что `GridView` автоматически поддерживает кое-какие специальные возможности табличного вида Проводника Windows, а именно:

- Разрешается менять порядок столбцов путем перетаскивания их заголовков.
- Разрешается изменять размеры столбцов путем перетаскивания их разделителей.
- Двойной щелчок по разделителю столбцов приводит к автоматической подгонке их размера под размер содержимого столбца.

Однако `GridView` не поддерживает автоматическую сортировку щелчком по заголовку столбца, что, безусловно, является досадным упущением. Код сортировки объектов в результате щелчка по заголовку столбца совсем не сложен (достаточно воспользоваться вышеупомянутым свойством `SortDescriptions`), но вот рисовать внутри заголовка стрелочку, индицирующую факт и направление сортировки, придется самостоятельно. В общем и целом, `ListView` с видом `GridView` – сильно урезанный вариант элемента `DataGrid`. Но теперь, когда в WPF 4 появился *настоящий* элемент `DataGrid`, нужда в `GridView` сильно поуменьшилась.

Элемент TabControl

Следующий селектор, `TabControl`, полезен для переключения между страницами содержимого. На рис. 10.9 показано, как выглядит элемент `TabControl` в простейшем случае. Обычно вкладки располагаются вдоль верхнего края, но свойство `TabStripPlacment` (типа `Dock`) позволяет разместить их слева (`Left`), справа (`Right`) или снизу (`Bottom`).

Работать с `TabControl` просто. Нужно лишь поместить внутрь него какие-нибудь объекты – и каждый объект автоматически окажется на отдельной вкладке. Например:

```
<TabControl>
  <TextBlock>Content for Tab 1.</TextBlock>
  <TextBlock>Content for Tab 2.</TextBlock>
  <TextBlock>Content for Tab 3.</TextBlock>
</TabControl>
```

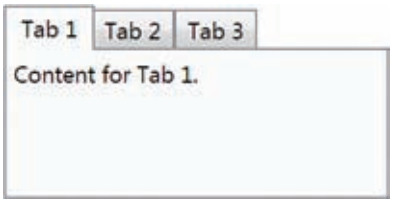


***Рис. 10.9.** Элемент управления WPF TabControl*

Как `ComboBox` со своим `ComboBoxItem`, `ListBox` со своим `ListBoxItem` и т. д., элемент `TabControl` неявно обертывает каждый объект элементом типа `TabItem`. Впрочем, маловероятно, что вы когда-нибудь захотите добавлять объекты типа, отличного от `TabItem`, непосредственно в `TabControl`, потому что без `TabItem` у соответствующей вкладки не будет метки. Например, на рис. 10.9 представлена визуализация следующей XAML-разметки:

```
<TabControl>
  <TabItem Header="Tab 1">Content for Tab 1.</TabItem>
  <TabItem Header="Tab 2">Content for Tab 2.</TabItem>
  <TabItem Header="Tab 3">Content for Tab 3.</TabItem>
</TabControl>
```

`TabItem` – однодетный элемент управления с заголовком, поэтому `Header` может быть произвольным объектом – так же как в случае `GroupBox` или `Expander`.

В отличие от других селекторов, первый элемент `TabItem` по умолчанию оказывается выбранным. Однако в программе можно сделать все вкладки невыбранными, записав значение `null` в свойство `SelectedItem` или значение `-1` в свойство `SelectedIndex`.

Элемент DataGrid

`DataGrid` – весьма гибкий элемент управления для отображения данных в виде таблицы с несколькими столбцами, допускающей сортировку, редактирование и многое другое. Он оптимизирован для связывания с таблицей базы данных в памяти (например, типа `System.Data.DataTable` из ADO.NET). Мастера Visual Studio и такие технологии, как LINQ to SQL, предельно упрощают такое связывание.

В листинге 10.2 показана XAML-разметка элемента `DataGrid`, который содержит коллекцию из двух объектов следующего типа `Record`:

```
public class Record
{
```

```
  public string FirstName   { get; set; }
  public string LastName    { get; set; }
  public Uri Website        { get; set; }
  public bool IsBillionaire { get; set; }
  public Gender Gender      { get; set; }
}
```

где перечисление `Gender` определено следующим образом:

```
public enum Gender
{
  Male,
  Female
}
```

Пять столбцов данных, показанные на рис. 10.10 (по одному для каждого свойства объекта `Record`), определены в коллекции `Columns`.

Листинг 10.2. Элемент DataGrid со статически определенными данными и столбцами разных типов

```
<DataGrid IsReadOnly="True"
  xmlns:local="clr-namespace:Listing10_2"
  xmlns:sys="clr-namespace:System;assembly=mscorlib">

  <!-- Поддержка для показа всех полов в столбце
       DataGridComboBoxColumn: -->
  <DataGrid.Resources>
    <ObjectDataProvider x:Key="genderEnum" MethodName="GetValues"
        ObjectType="{x:Type sys:Enum}">
      <ObjectDataProvider.MethodParameters>
        <x:Type Type="local:Gender"/>
      </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
  </DataGrid.Resources>

  <!-- Столбцы: -->
  <DataGrid.Columns>
    <DataGridTextColumn Header="First Name" Binding="{Binding FirstName}"/>
    <DataGridTextColumn Header="Last Name" Binding="{Binding LastName}"/>
    <DataGridHyperlinkColumn Header="Website" Binding="{Binding Website}"/>
    <DataGridCheckBoxColumn Header="Billionaire?"
      Binding="{Binding IsBillionaire}"/>
    <DataGridComboBoxColumn Header="Gender" SelectedItemBinding="{Binding Gender}"
      ItemsSource="{Binding Source={StaticResource genderEnum}}"/>
    </DataGrid.Columns>

  <!-- Данные: -->
  <local:Record FirstName="Adam" LastName="Nathan"
    Website="http://adamnathan.net" Gender="Male"/>
  <local:Record FirstName="Bill" LastName="Gates"
    Website="http://twitter.com/billgates" IsBillionaire="True" Gender="Male"/>

</DataGrid>
```

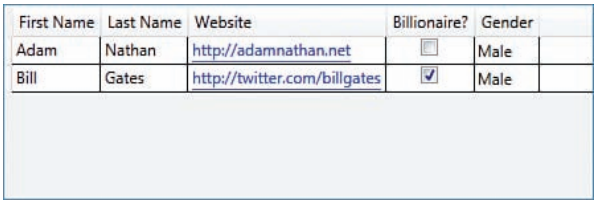


***Рис. 10.10.** Элемент WPF DataGrid, описанный в листинге 10.2*

Элемент `DataGrid` автоматически поддерживает изменение порядка и размера столбцов и сортировку по столбцам, но любую возможность можно отключить, установив значение `false` для некоторых из следующих свойств: `CanUserReorderColumns`, `CanUserResizeColumns`, `CanUserResizeRows` и `CanUserSortColumns`. Свойства `GridLinesVisibility` и `HeadersVisibility` позволяют отключить показ линий сетки и заголовков соответственно.

В листинге 10.2 демонстрируются основные типы столбцов, поддерживаемые элементом `DataGrid`:

- `DataGridTextColumn` – идеален для представления строк, поскольку в обычном режиме используется элемент `TextBlock`, а в режиме редактирования – элемент `TextBox`.
- `DataGridHyperlinkColumn` – представляет обычный текст в виде гиперссылки, по которой можно щелкнуть. Отметим, однако, что со щелчком не ассоциируется никакое поведение по умолчанию (например, открытие браузера). Эти действия вы должны обрабатывать явно.
- `DataGridCheckBoxColumn` – идеален для представления булевских значений, поскольку используется элемент `CheckBox`, который в отмеченном состоянии соответствует значению `true`, а в сброшенном – значению `false`.
- `DataGridComboBoxColumn` – идеален для представления перечислений, поскольку в обычном режиме используется элемент `TextBlock`, а в режиме редактирования – элемент `ComboBox`, содержащий все возможные значения.

В WPF встроен еще один тип столбца:

- `DataGridTemplateColumn` – позволяет задать произвольные шаблоны для представления значения в обычном режиме и в режиме редактирования. Делается это с помощью свойств `CellTemplate` и `CellEditingTemplate`.

Автоматически генерируемые столбцы

Если объекты, отображаемые в элементе `DataGrid`, задаются с помощью свойства `ItemsSource`, то элемент пытается автоматически сгенерировать соответствующие столбцы. В таком случае для представления строк выбирается столбец типа `DataGridTextColumn`, для представления URI – столбец типа `DataGridHyperlinkColumn`, для представления булевских величин – столбец типа `DataGridCheckBoxColumn`, а для представления перечислений – столбец типа `DataGridComboBoxColumn` (при этом источник данных для значений перечисления присоединяется автоматически).

Таким образом, пустой элемент DataGrid:

```
<DataGrid Name="dataGrid" />
```

порождает почти такой же результат, как на рис. 10.10, если установить его свойство ItemsSource, как в следующем застраничном коде:

```
dataGrid.ItemsSource = new Record[]
{
  new Record { FirstName="Adam", LastName="Nathan", Website=
    new Uri("http://adamnathan.net"), Gender=Gender.Male },
  new Record { FirstName="Bill", LastName="Gates", Website=
    new Uri("http://twitter.com/billgates"), Gender=Gender.Male,
    IsBillionaire=true }
};
```

Единственное визуальное отличие – это метки в заголовках, которые теперь совпадают с именами соответствующих свойств. Результат показан на рис. 10.11.

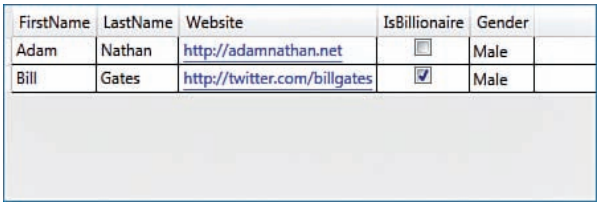


***Рис. 10.11.** Элемент WPF DataGrid с автоматически сгенерированными столбцами, в которых заголовками служат имена свойств класса Record*

Мало того что конструирование элемента оказалось гораздо проще, так еще DataGrid на рис. 10.11 автоматически поддерживает редактирование всех полей каждого элемента. При включении элементов непосредственно в коллекцию Items элемента DataGrid этого не было. Стоит щелкнуть по любой ячейке в первых трех столбцах, как она автоматически превращается в редактируемое поле ввода TextBox, по флажкам CheckBox тоже можно щелкать, а ячейка в столбце Gender (Пол) при щелчке по ней автоматически превращается в список ComboBox с правильным набором значений. Для ячеек, владеющих фокусом, особым образом интерпретируются некоторые клавиши, например пробел или F2. Результат любого завершенного редактирования отражается в коллекции ItemsSource. (К сожалению, отметка флажка IsBillionaire (Миллиардер) против моего имени никак не отразилась на состоянии моего банковского счета. Наверное, в этом примере какая-то ошибка.)

Если в элементе DataGrid уже были явно определены какие-то столбцы, то автоматически сгенерированные добавляются после них. Отдельные автоматически сгенерированные столбцы можно настроить или удалить, обработав событие AutoGeneratingColumn, которое возникает один раз для каждого столбца. После генерации всех столбцов один раз возникает событие AutoGeneratedColumns.

Чтобы вообще отменить автоматическую генерацию столбцов, достаточно присвоить свойству AutoGenerateColumns значение false.

Выбор строк и ячеек

Элемент DataGrid поддерживает несколько моделей выбора с помощью двух свойств – SelectionMode и SelectionUnit. Свойству SelectionMode можно присвоить значение Single – тогда разрешено выбирать только один объект, или значение Extended – в этом случае можно выбирать несколько объектов (это режим по умолчанию). Определение слова «объект» зависит от значения свойства SelectionUnit:

- Cell – разрешено выбирать только отдельные ячейки.
- FullRow – разрешено выбирать только строки целиком.
- CellOrRowHeader – разрешено выбирать и то и другое (для выбора всей строки следует щелкнуть по ее заголовку).

В режиме выбора нескольких объектов щелчок с нажатой клавишей Shift позволяет выбирать соседние объекты, а с нажатой клавишей Ctrl – произвольно расположенные объекты.

При выборе строк генерируется событие Selected, а свойство SelectedItems содержит коллекцию выбранных объектов. Для элемента DataGrid в листинге 10.2 это была бы коллекция объектов типа Record. При выборе отдельных ячеек генерируется событие SelectedCellChanged, а свойство SelectedCells содержит список структур DataGridCellInfo, в которых хранится информация о соответствующих строках и столбцах. Выбираемые объекты DataGridRow и DataGridCell также генерируют свои события Selected, а их свойство IsSelected принимает значение true.

Даже если выбрано несколько ячеек или строк, в каждый момент времени фокус может принадлежать только одной ячейке. Получить или установить эту ячейку позволяет свойство CurrentCell. Кроме того, свойство CurrentColumn позволяет определить, в каком столбце находится ячейка CurrentCell, а свойство CurrentItem возвращает объект данных, соответствующий строке, которая содержит ячейку CurrentCell.

Развитая поддержка множественного выбора и операций над выделенными объектами реализована в базовом классе MultiSelector, который наследует классу Selector и был впервые введен в версии WPF 3.5. Другие элементы управления WPF также поддерживают множественный выбор, но только DataGrid наследует классу MultiSelector.

Дополнительные настройки

Класс DataGrid поддерживает и другие способы настройки, например взаимодействие с буфером обмена, виртуализацию, возможность выводить дополнительную информацию для строк и «замораживать» столбцы.

Взаимодействие с буфером обмена. Настроить, какие именно данные копируются из DataGrid в буфер обмена (например, при нажатии Ctrl+C после выбора

объектов), позволяет свойство ClipboardCopyMode. Оно может принимать следующие значения:

- ExcludeHeader – не включать заголовки столбцов в копируемый текст. Это режим по умолчанию.
- IncludeHeader – включать заголовки столбцов в копируемый текст.
- None – ничего не копировать в буфер обмена.

Виртуализация. По умолчанию строки DataGrid виртуализируются (объекты UIElement не создаются для строк, невидимых на экране, причем в зависимости от источника данных даже выборка данных для этих строк может откладываться), а столбцы – нет. Для изменения этого поведения предназначены свойства EnableRowVirtualization (если оно равно false, то строки не виртуализируются) и EnableColumnVirtualization (если оно равно true, то столбцы виртуализируются). Свойство EnableColumnVirtualization по умолчанию не равно true, потому что в этом режиме может замедляться обновление изображения при горизонтальной прокрутке.

Дополнительная информация для строк. Элемент DataGrid поддерживает показ дополнительной информации в строках за счет установки свойства RowDetailsTemplate. Например:

```
<DataGrid ...>
<DataGrid.RowDetailsTemplate>
  <DataTemplate>
    <TextBlock Margin="10" FontWeight="Bold">Details go here.</TextBlock>
  </DataTemplate>
</DataGrid.RowDetailsTemplate>
  ...
</DataGrid>
```

Обычно элементы внутри шаблона RowDetailsTemplate пользуются механизмом привязки к данным, чтобы изменить содержимое текущей строки, но в данном случае мы задали простой элемент TextBlock. На рис. 10.12 показано, что происходит при выборе строки.

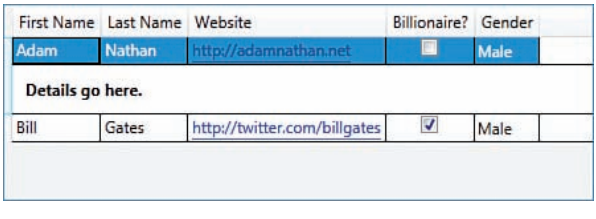


***Рис. 10.12.** Показ дополнительной информации для выбранной строки в элементе DataGrid*

По умолчанию дополнительная информация показывается только для выбранной строки (или строк), но это поведение можно изменить с помощью свойства RowDetailsVisibilityMode, принимающего следующие значения:

- `VisibleWhenSelected` – дополнительная информация показывается только для выбранных строк. Это режим по умолчанию.
- `Visible` – дополнительная информация показывается для всех строк.
- `Collapsed` – дополнительная информация вообще не показывается.

Замораживание столбцов. Элемент `DataGrid` позволяет заморозить любое число столбцов. Это означает, что они не будут выдвинуты за пределы области элемента при горизонтальной прокрутке. Примерно так же заморозка столбцов работает в Microsoft Excel. Но имеется несколько ограничений: замораживать можно только самые левые столбцы и замороженные столбцы нельзя менять местами с незамороженными.

Чтобы заморозить один или несколько столбцов, достаточно присвоить свойству `FrozenColumnCount` любое значение, большее 0. На рис. 10.13 показано, как выглядит элемент `DataGrid` из листинга 10.2, когда `FrozenColumnCount` равно 2. Столбцы, начиная с третьего, можно прокручивать, поэтому-то и не видно заголовка третьего столбца.

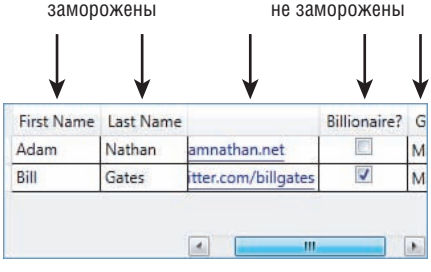


***Рис. 10.13.** Элемент DataGrid из листинга 10.2, в котором FrozenColumnCount="2"*

FAQ

Можно ли в DataGrid замораживать строки?

Нет, такая возможность не предусмотрена. Автоматически заморозить можно только *дополнительную информацию* для строк. Если свойство `AreRowDetailsFrozen` равно `true`, то вся показанная дополнительная информация не смещается при горизонтальной прокрутке.р

Редактирование, добавление и удаление данных

Мы уже видели, что при использовании свойства `ItemsSource` автоматически поддерживается редактирование данных в отдельных объектах. Поскольку коллекция `ItemsSource` допускает также добавление и удаление объектов, то `DataGrid` автоматически поддерживает и эти операции. В предыдущем примере

для получения этой функциональности достаточно обернуть массив списком List<Record> (при этом статический массив служит только для инициализации динамического списка):

```
dataGrid.ItemsSource = new List<Record>(
  new Record[]
  {
    new Record { FirstName="Adam", LastName="Nathan", Website=
      new Uri("http://adamnathan.net"), Gender=Gender.Male },
    new Record { FirstName="Bill", LastName="Gates", Website=
      new Uri("http://twitter.com/billgates"), Gender=Gender.Male,
      IsBillionaire=true }
  }
);
```

Теперь в сетке DataGrid внизу присутствует пустая строка, в которую в любой момент можно добавить данные. В классе DataGrid определены методы и команды для таких типичных действий, как начало редактирования (клавиша F2), отмена редактирования (клавиша Esc), сохранение результатов редактирования (клавиша Enter) и удаление строки (клавиша Delete).

Для предотвращения редактирования следует присвоить свойству IsReadOnly значение true, а чтобы запретить добавление или удаление строк, нужно присвоить значение false свойству CanUserAddRows или CanUserDeleteRows соответственно. В листинге 10.2 для свойства IsReadOnly установлено значение true, чтобы избежать исключений, поскольку описанная в разметке коллекция объектов Record не поддерживает редактирование. Хотя вход в режим редактирования (и переключение ячейки в такой режим) производится автоматически, по ходу дела возникает несколько событий, позволяющих вмешаться в этот процесс: PreparingCellForEdit, BeginningEdit, CellEditEnding/RowEditEnding и InitializeNewItem.

ПРЕДУПРЕЖДЕНИЕ

Свойства CanUserAddRows и CanUserDeleteRows могут быть автоматически сброшены в false!

В зависимости от значений прочих свойств, свойства CanUserAddRows и CanUserDeleteRows могут оказаться равными false, хотя для них было явно установлено значение true! Например, если свойство IsReadOnly или IsEnabled элемента DataGrid равно false, то будут равны false и оба вышеупомянутых свойства. Есть и менее очевидный случай: если источник данных не поддерживает добавление или удаление (на что указывают свойства CanAddNew и CanRemove, определенные в интерфейсе IEditableCollectionView), то и CanUserAddRows, и CanUserDeleteRows будут равны false. Дополнительные сведения о видах коллекций и, в частности, об интерфейсе IEditableCollectionView см. в главе 13.

Меню

В WPF имеются оба элемента, относящихся к меню: Menu и ContextMenu. Но, в отличие от технологий на базе Win32, меню WPF не являются каким-то особым случаем элементов управления со специальными ограничениями. Это просто еще один вид многодетных элементов, предназначенный для иерархического отображения объектов в виде каскадных выпадающих меню.

Элемент Menu

Элемент Menu располагает хранящиеся в нем объекты по горизонтали в строке характерного серого цвета (по умолчанию). К своему базовому классу ItemsControl класс Menu добавляет только свойство IsMainMenu. Если оно равно true (случай по умолчанию), то меню Menu получает фокус при нажатии клавиши Alt или F10, что соответствует ожиданиям пользователей, привыкших к меню в Win32.

Поскольку Menu – обычный многодетный элемент управления, то в качестве объектов в нем может храниться все что угодно, хотя предполагается, что это будут объекты типа MenuItem или Separator. На рис. 10.14 показано типичное меню, созданное из разметки в листинге 10.3.

Листинг 10.3. Типичное меню с дочерними элементами MenuItem и Separator

```
<Menu>
  <MenuItem Header="_File">
    <MenuItem Header="_New..."/>
    <MenuItem Header="_Open..."/>
    <Separator/>
    <MenuItem Header="Sen_d To">
      <MenuItem Header="Mail Recipient"/>
      <MenuItem Header="My Documents"/>
    </MenuItem>
  </MenuItem>
  <MenuItem Header="_Edit">
    ...
  </MenuItem>
  <MenuItem Header="_View">
    ...
  </MenuItem>
</Menu>
```

Класс MenuItem относится к *многодетным элементам управления с заголовком* (наследует классу HeaderedItemsControl) и во многом напоминает однодетные элементы управления с заголовком. В случае MenuItem свойство Header представляет собой основной объект (как правило, текст, см. рис. 10.14). В коллекции Items, если она непустая, хранятся дочерние элементы, отображаемые в виде подменю. Так же как Button и Label, класс MenuItem поддерживает клавиши доступа, обозначаемые предшествующим знаком подчеркивания.

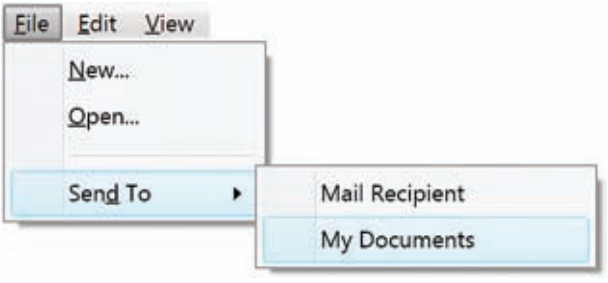


***Рис. 10.14.** Меню WPF*

`Separator` – это простой элемент управления, который, будучи помещен в `MenuItem`, визуализируется в виде горизонтальной черты, как показано на рис. 10.14. Этот класс может использоваться и в двух других многодетных элементах: `ToolBar` и `StatusBar`.

Хотя `Menu` – простой элемент управления, класс `MenuItem` содержит много свойств для настройки своего поведения. Приведем наиболее интересные.

- `Icon` – позволяет добавлять произвольный объект, отображаемый рядом с заголовком `Header`. Объект `Icon` визуализируется так же, как `Header`, хотя обычно представляет собой небольшое изображение.
- `IsCheckable` – наделяет `MenuItem` поведением флажка `CheckBox`.
- `InputGestureText` – позволяет аннотировать элемент меню ассоциированным с ним жестом (чаще всего какой-нибудь комбинацией клавиш, например Ctrl+O).

В классе `MenuItem` определено также пять событий: `Checked`, `Unchecked`, `SubmenuOpened`, `SubmenuClosed` и `Click`. Обычно для наделения пункта меню поведением применяется обработчик события `Click`, но можно также записать команду в свойство `Command` объекта `MenuItem`.

ПРЕДУПРЕЖДЕНИЕ

Задание свойства InputGestureText не ассоциирует с MenuItem соответствующую комбинацию клавиш!

Это досадное отличие WPF от таких систем, как Windows Forms и Visual Basic 6, – запись в свойство `InputGestureText` элемента `MenuItem` строки вида `"Ctrl+O"` еще не означает, что при нажатии комбинации клавиш Ctrl+O будет автоматически вызван данный пункт меню! Эта строка – не более чем документация.

Чтобы связать с `MenuItem` комбинацию клавиш, необходимо ассоциировать ее с командой с помощью свойства `Command`. Если с командой ассоциирован некий жест ввода, то в свойство `InputGestureText` объекта `MenuItem` автоматически записывается соответствующая ему строка, то есть текстовое представление комбинации клавиш отображается без каких-либо дополнительных действий.

СОВЕТ

Когда свойству `Command` объекта `MenuItem` присваивается ссылка на объект типа `RoutedUICommand`, в его свойство `Header` автоматически записывается значение свойства `Text` команды. Это поведение можно переопределить, установив заголовок `Header` явно.

FAQ

Как расположить пункты Menu по вертикали, а не по горизонтали?

Поскольку `Menu` – обычный многодетный элемент управления, можно воспользоваться описанным выше при рассмотрении `ListBox` приемом – подменой панели `ItemsPanel`, только подразумеваемую по умолчанию панель следует заменить на `StackPanel`:

```
<Menu>
  <Menu.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel/>
    </ItemsPanelTemplate>
  </Menu.ItemsPanel>
...
</Menu>
```

По умолчанию `StackPanel` ориентирована вертикально, поэтому в данном случае явно задавать свойство `Orientation` необязательно. Результат показан на рис. 10.15.



Рис. 10.15. *Вертикальное меню*

Если вы хотите, чтобы пункты меню были еще и повернуты на 90° (как в старых программах из пакета Microsoft Office в случае, когда меню перетаскивается и пристыковывается к левой или правой стороне окна), то воспользуйтесь преобразованием `RotateTransform`.

Элемент ContextMenu

Элемент `ContextMenu` работает так же, как `Menu`; это простой контейнер, предназначенный для хранения пунктов меню `MenuItem` и разделителей `Separator`. Однако же включать `ContextMenu` непосредственно в дерево элементов нельзя. Следует связывать его с элементом управления посредством подходящего присоединенного свойства, например свойства `ContextMenu` в классах `FrameworkElement`

и FrameworkContentElement. Контекстное меню элемента отображается, когда пользователь щелкает по элементу правой кнопкой мыши (или нажимает комбинацию клавиш Shift+F10).

На рис. 10.16 изображено контекстное меню, следующим образом ассоциированное со списком ListBox (предполагается, что пункты меню такие же, как в листинге 10.3):

```
<ListBox>
  <ListBox.ContextMenu>
    <ContextMenu>
    ...Три элемента MenuItems из листинга 10.3...
    </ContextMenu>
  </ListBox.ContextMenu>
  ...
</ListBox>
```

Помимо ожидаемого свойства IsOpen и событий Opened/Closed, в классе ContextMenu определено еще много свойств для настройки местоположения меню. По умолчанию левый верхний угол меню совпадает с позицией указателя мыши. Но свойство Placement может принимать и другие значения, кроме MousePoint (например, Absolute). К тому же с помощью свойств HorizontalOffset и VerticalOffset можно задавать смещение от указателя по горизонтали и вертикали.

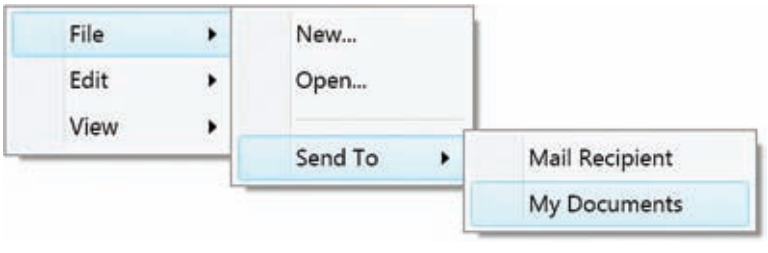


***Рис. 10.16.** Контекстное меню WPF*

FAQ

Как сделать, чтобы контекстное меню появлялось при щелчке правой кнопкой мыши по неактивному элементу?

Как и ToolTipService, класс ContextMenuService содержит присоединенное свойство ShowOnDisabled специально для этой цели. Используется оно следующим образом:

```
<ListBox ContextMenuService.ShowOnDisabled="True">
  <ListBox.ContextMenu>
   ...
  </ListBox.ContextMenu>
  ...
</ListBox>
```

Напомним, что с классом ToolTip связан статический класс ToolTipService, позволяющий управлять свойствами всплывающей подсказки из элемента, с которым она ассоциирована. Точно так же с классом ContextMenu связан статический класс ContextMenuService, предназначенный для той же цели. В нем имеется несколько присоединенных свойств, соответствующих свойствам, определенным в самом классе ContextMenu.

Другие многодетные элементы управления

Оставшиеся многодетные элементы управления – TreeView, ToolBar и StatusBar – не являются ни селекторами, ни меню, но тем не менее могут содержать неограниченное число произвольных объектов.

Элемент TreeView

TreeView – популярный элемент управления, предназначенный для отображения иерархически организованных данных с возможностью раскрывать и сворачивать узлы дерева, как показано на рис. 10.17. В теме Aero состояние узлов обозначается треугольничками, в других темах, например Luna, – привычными знаками плюс и минус.

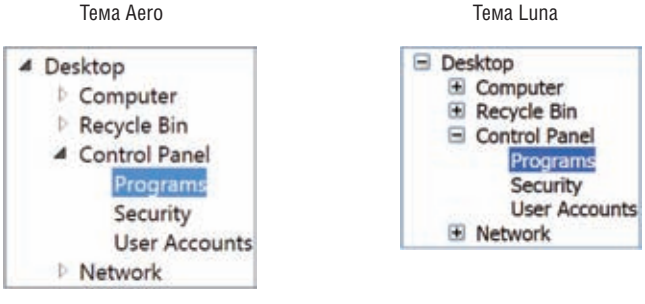


Рис. 10.17. *Элемент WPF TreeView*

TreeView, как и Menu, – очень простой элемент управления. Он может содержать любые объекты и располагает их по вертикали. Однако от TreeView мало пользы, если в нем хранится что-то, кроме объектов типа TreeViewItem.

TreeViewItem, как и MenuItem, – многодетный элемент управления с заголовком. В его свойстве Header хранится сам элемент, а в коллекции Items – его подэлементы (предполагается, что они также являются объектами типа TreeViewItem).

Элемент TreeView, изображенный на рис. 10.17, соответствует следующей XAML-разметке:

```
<TreeView>
  <TreeViewItem Header="Desktop">
    <TreeViewItem Header="Computer">
      ...
```

```
      </TreeViewItem>
      <TreeViewItem Header="Recycle Bin">
       ...
      </TreeViewItem>
      <TreeViewItem Header="Control Panel">
        <TreeViewItem Header="Programs"/>
        <TreeViewItem Header="Security"/>
        <TreeViewItem Header="User Accounts"/>
      </TreeViewItem>
      <TreeViewItem Header="Network">
        ...
      </TreeViewItem>
    </TreeViewItem>
  </TreeView>
```

В классе `TreeViewItem` имеются удобные свойства `IsExpanded` и `IsSelected`, а также четыре события, соответствующие четырем возможным состояниям этих свойств: `Expanded`, `Collapsed`, `Selected` и `Unselected`. Кроме того, класс `TreeViewItem` поддерживает навигацию с помощью клавиатуры: клавиши плюс (+) и минус (–) соответственно раскрывают и сворачивают узел, а клавиши со стрелками, Page Up, Page Down, Home и End дают разные способы передачи фокуса от одного узла другому.

КОПНЕМ ГЛУБЖЕ

Сравнение классов TreeView и Selector

С точки зрения API класс `TreeView` очень похож на класс `Selector`, но не является производным от него, потому что для иерархически организованных объектов не существует естественного понятия целочисленного индекса. Поэтому в `TreeView` определены собственные свойства `SelectedItem` и `SelectedValue` (но не `SelectedIndex`). Также определено событие `SelectedItemChanged`, вместе с которым обработчику передаются не коллекции, а просто объекты `OldValue` и `NewValue`, поскольку `TreeView` поддерживает выбор только одного объекта.

Отсутствие поддержки для выбора нескольких объектов – досадное ограничение, сохранившееся и в версии WPF 4. Если вам это необходимо, то можете воспользоваться каким-нибудь сторонним элементом управления, например `RadTreeView` компании Telerik (*http://telerik.com/products/wpf/treeview.aspx*). Можете также попробовать самостоятельно написать класс `TreeView` с поддержкой множественного выбора, унаследовав классу `ListBox`, но это нелегко.

СОВЕТ

В версии WPF 4 класс `TreeView` начал поддерживать виртуализацию, но включать ее нужно явно – путем установки для присоединенного свойства `VirtualizingStackPanel.IsVirtualizing` объекта `TreeView` значения `true`. Этот режим позволяет заметно сэкономить память и повысить производительность прокрутки, когда количество узлов очень велико.

ПРЕДУПРЕЖДЕНИЕ

Всегда явно обертывайте узлы TreeView элементами TreeViewItem!

Очень заманчиво использовать в качестве листовых узлов простые элементы `TextBlock`, но если вы так поступите, то можете столкнуться с одной тонкостью механизма наследования значений свойств, из-за которой текст в таких элементах, как `TextBlock`, как бы пропадает. По умолчанию выбор родительского элемента меняет его цвет `Foreground` на белый, и если текстовые блоки `TextBlock` являются прямыми логическими потомками этого элемента, то и к ним будет применен белый цвет. (Хотя визуальным родителем каждого текстового блока является неявный элемент `TreeViewItem`, в механизме наследования свойств приоритет отдается логическому родителю.) Понятно, что на подразумеваемом по умолчанию белом фоне такой текст не виден. Если же сделать `TreeViewItem` явным (логическим) родителем каждого блока `TextBlock`, то нежелательный эффект наследования не проявляется.

Элемент ToolBar

Элемент управления `ToolBar` (панель инструментов) обычно применяется для группировки небольших кнопок (или других элементов управления) и служит дополнением к традиционной системе меню. На рис. 10.18 показана панель инструментов, полученная визуализацией следующей XAML-разметки:

```
<ToolBar RenderOptions.BitmapScalingMode="NearestNeighbor">
  <Button><Image Source="copy.gif"/></Button>
  <Separator/>
  <ToggleButton><Image Source="bold.gif"/></ToggleButton>
  <ToggleButton><Image Source="italic.gif"/></ToggleButton>
  <ToggleButton><Image Source="underline.gif"/></ToggleButton>
  <Separator/>
  <ToggleButton><Image Source="left.gif"/></ToggleButton>
  <ToggleButton><Image Source="right.gif"/></ToggleButton>
  <ToggleButton><Image Source="justify.gif"/></ToggleButton>
  <Separator/>
  <Label>Zoom</Label>
  <ComboBox>
    ...
  </ComboBox>
  <Separator/>
  <Button><Image Source="superscript.gif"/></Button>
  <Button><Image Source="subscript.gif"/></Button>
  ...
</ToolBar>
```



Рис. 10.18. Элемент WPF ToolBar

Отметим, что элементы Button и ComboBox на панели инструментов выглядят иначе, чем обычно. Кроме того, разделитель Separator теперь представляется вертикальной линией, а не горизонтальной, как в меню Menu. Элемент ToolBar переопределяет подразумеваемые по умолчанию стили своих дочерних элементов так, чтобы они выглядели привычно для большинства пользователей.

Элементы ToolBar могут находиться в любом месте дерева элементов, но обычно их помещают в специальный контейнер ToolBarTray, производный от FrameworkElement. Объект ToolBarTray содержит коллекцию элементов ToolBar (в своем свойстве содержимого ToolBars) и, если его свойство IsLocked не равно true, позволяет перетаскивать панели инструментов и располагать их в другом месте. (В классе ToolBarTray определено также присоединенное свойство IsLocked, которое можно задавать для отдельных панелей ToolBar.) В классе ToolBarTray имеется свойство Orientation; если присвоить ему значение Vertical, то все содержащиеся в нем панели ToolBar будут ориентированы вертикально.

Если панель инструментов ToolBar содержит больше элементов, чем помещается в занимаемой ею области, то лишние попадают в область переполнения. Это всплывающее окно, для открытия которого нужно щелкнуть по стрелочке в конце панели, как показано на рис. 10.19. По умолчанию первым в область переполнения попадает последний элемент панели, но для отдельных элементов этим поведением можно управлять с помощью присоединенного свойства OverflowMode из класса ToolBar. С его помощью можно определить, что элемент должен перемещаться в область переполнения по мере необходимости (AsNeeded – по умолчанию), всегда (Always) или никогда (Never).

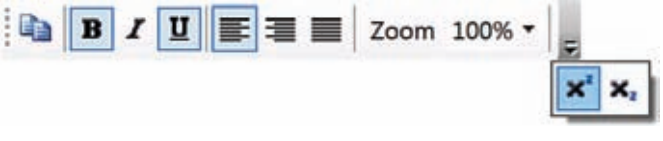


***Рис. 10.19.** У панели инструментов имеется область переполнения, в которую попадают не поместившиеся элементы*

СОВЕТ

Чтобы создать настраиваемую панель инструментов, как в Visual Studio, присвойте свойству ToolBar.OverflowMode значение Never для каждого элемента, затем добавьте элемент Menu с заголовком "_Add or Remove Buttons" (Добавить или удалить кнопки), для которого свойство ToolBar.OverflowMode должно быть равно Always (чтобы он всегда оставался в области переполнения). Далее в это меню можно добавить пункты MenuItem и сделать так, чтобы отметка флажка в таком пункте приводила к добавлению соответствующего пункта на панель инструментов, а сброс флажка – к убиранию элемента с панели.

СОВЕТ

Если элемент содержит небольшой значок, то имеет смысл задать для него присоединенное свойство `RenderOptions.BitmapScalingMode`, равное `NearestNeighbor`. Тогда изображение будет выглядеть гораздо четче, чем в обычном режиме прорисовки. Рассматриваемая в этом разделе панель `ToolBar` визуализируется именно в таком режиме.

Хотя это свойство для удобства присоединено к самому элементу `ToolBar`, более правильно задавать его для каждой кнопки `Button` отдельно. Дело в том, что после перемещения в область переполнения кнопка перестает наследовать это свойство от родителя. (Объемлющий элемент `Popup` не является потомком `ToolBar`.) Из-за этой тонкой особенности последние два значка на рис. 10.19 выглядят менее четко, чем остальные.

КОПНЕМ ГЛУБЖЕ

Настройка навигации с помощью клавиатуры

Следующий элемент `ToolBar` демонстрирует несколько странное поведение в части навигации с помощью клавиш:

```
<ToolBar>
  <Button>A</Button>
  <Menu>
    <MenuItem Header="B"/>
    <MenuItem Header="C"/>
  </Menu>
  <Button>D</Button>
</ToolBar>
```

Если передать фокус панели `ToolBar`, а затем несколько раз нажать клавишу Tab, то фокус «застрянет» – будет передаваться от A к B, затем к C, к D и снова к A и далее по кругу. А если нажимать клавишу со стрелкой влево или вправо, то фокус будет попеременно передаваться B и C.

В классе `KeyboardNavigation` из пространства имен `System.Windows.Input` определено несколько полезных присоединенных свойств для настройки этого и других аспектов поведения клавиатуры. Например, чтобы избежать зацикливания при нажатии клавиши Tab на панели инструментов, можно присвоить свойству `KeyboardNavigation.TabNavigation` для элемента `ToolBar` значение `Continue` (вместо `Cycle`). А чтобы не попасть в цикл при навигации по меню с помощью клавиш со стрелками, задайте для элемента `Menu` свойство `KeyboardNavigation.DirectionalNavigation`, равное `Continue`.

КОПНЕМ ГЛУБЖЕ

Неиспользуемое свойство Header элемента ToolBar

На самом деле `ToolBar` – многодетный элемент управления с заголовком (как `MenuItem` и `TreeViewItem`). Его свойство `Header` никогда не отображается, но может быть полезно для реализации дополнительных возможностей `ToolBarTray`. Например, можно добавить контекстное меню, в котором перечислены все панели инструментов `ToolBar` (представленные своими заголовками `Header`), дав пользователям возможность добавлять или удалять панели. Или реализовать перемещаемые панели инструментов и показывать заголовок, когда панель «плавает».

Элемент StatusBar

Элемент `StatusBar` ведет себя, как `Menu`, но располагает своих потомков по горизонтали, как показано на рис. 10.20. Обычно его помещают вдоль нижнего края окна `Window` и используют для отображения информации о состоянии.



***Рис. 10.20.** Элемент WPF StatusBar*

Строка состояния `StatusBar`, изображенная на рис. 10.20, получена визуализацией следующей XAML-разметки:

```
<StatusBar RenderOptions.BitmapScalingMode="NearestNeighbor">
  <Label>27 Items</Label>
  <Separator/>
  <Label>Zoom</Label>
  <ComboBox>
  ...
  </ComboBox>
  <Separator/>
  <Button><Image Source="justify.gif"/></Button>
</StatusBar>
```

По умолчанию `StatusBar` переопределяет шаблон элемента управления `Separator` так, что он отображается в виде вертикальной линии, как на панели инструментов `ToolBar`. Дочерние элементы `StatusBar` (кроме `Separator`) неявно обертываются объектами `StatusBarItem`, но можно включить их и явно. Тогда можно будет настраивать их позиции с помощью относящихся к компоновке присоединенных свойств, которые мы рассматривали в главе 5.

FAQ

Как сделать, чтобы секции строки состояния пропорционально растягивались?

Очень часто желательно, чтобы отдельные секции строки состояния сохраняли пропорции. Например, левая секция должна занимать 25% ширины `StatusBar`, а правая – 75%. Добиться этого эффекта можно, заменив внутреннюю панель `ItemsPanel` сеткой `Grid` и сконфигурировав ее столбцы следующим образом:

```
<StatusBar>
  <StatusBar.ItemsPanel>
    <ItemsPanelTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="*"/>
          <ColumnDefinition Width="Auto"/>
          <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
      </Grid>
    </ItemsPanelTemplate>
  </StatusBar.ItemsPanel>
  <StatusBarItem Grid.Column="0">...</StatusBarItem>
  <Separator     Grid.Column="1"/>
  <StatusBarItem Grid.Column="2">...</StatusBarItem>
</StatusBar>
```

Отметим, что к элементам внутри `StatusBar` необходимо явно присоединять свойство `Grid.Column` (которое имеет смысл, только если в качестве `ItemsPanel` используется `Grid`), иначе все они окажутся в столбце с индексом 0. Кроме того, имейте в виду, что такие свойства компоновки работают только для потомков типа `StatusBarItem` и `Separator`. Дело в том, что остальные элементы (`Label`, `ComboBox` и `Button` в рассматриваемом примере `StatusBar`) неявно обернуты объектами `StatusBarItem`, к которым нужные свойства не присоединены. Поэтому, чтобы добиться требуемого эффекта, необходимо обернуть их в `StatusBarItem` явно.

Резюме

Понимать, что такое многодетные элементы управления, необходимо практически в любом проекте на основе WPF. Трудно представить себе WPF-приложение, в котором не используются ни однодетные, ни многодетные элементы. Но, в отличие от однодетных, многодетные элементы обладают гораздо более развитой функциональностью! В этой главе неоднократно упоминалось о важности привязки к данным при работе с динамическими списками объектов. Но перед тем как вплотную заняться привязкой к данным, мы должны рассмотреть еще несколько областей WPF. В следующей главе мы поговорим об изображениях, тексте и других элементах управления.

- Элемент управления Image
- Элементы управления Text и Ink
- Документы
- Диапазонные элементы управления
- Календарные элементы управления

11

Изображения, текст и другие элементы управления

В этой главе мы будем рассматривать элементы управления, не являющиеся ни однодетными, ни многодетными. Некоторые из них, например `Image`, кое-какие текстовые элементы, `ProgressBar` и `Slider`, наверное, вам знакомы, но в WPF они обладают более развитой функциональностью, чем вы могли бы предположить. Элементы `Calendar` и `DatePicker` появились только в WPF 4. Мы рассмотрим также ряд элементов, производных от класса `FrameworkContentElement` (а не `Control`), с помощью которых можно создавать потоковые документы. Это мощная, но не слишком часто используемая возможность WPF.

Элемент управления Image

Класс `System.Windows.Controls.Image` позволяет включать в пользовательский интерфейс изображения (в формате BMP, PNG, GIF, JPG и др.). В нем имеется свойство `Source` типа `System.Windows.Media.ImageSource`, но благодаря конвертеру типа `System.Windows.Media.ImageSourceConverter` его можно задавать в XAML в виде простой строки, например:

```
<Image Source="zoom.gif"/>
```

Свойство `ImageSource` может указывать на изображения, представленные URL-адресом, хранящиеся в файловой системе и даже внедренные в сборку. (Извлечение и рисование изображений, внедренных в сборку, рассматривается в следующей главе.) В классе `Image` определены те же самые свойства `Stretch` и `StretchDirection`, с которыми мы встречались в главе 5 «Компоновка с помощью панелей», – они позволяют управлять масштабированием.

В целом работать с классом `Image` несложно, если не считать ряд менее очевидных средств для визуализации изображения, которые он поддерживает. К элементу `Image` можно присоединить свойство `RenderOptions.BitmapScalingMode`, за-

дающее компромисс между скоростью и качеством визуализации. Из всех принимаемых им значений наиболее важным является NearestNeighbor – это режим масштабирования растрового изображения по ближайшей соседней точке, при котором изображение становится более четким. Этот режим мы устанавливали в предыдущей главе при обсуждении элементов ToolBar и StatusBar, а также в приложении Photo Gallery из главы 7 «Структурирование и развертывание приложения». Пример:

```
<Image RenderOptions.BitmapScalingMode="NearestNeighbor" Source="zoom.gif"/>
```

В печатном тексте различие неотчетливо, но на экране компьютера улучшение качества сразу заметно. На рис. 11.1 показаны изображения из программы Photo Gallery в режиме NearestNeighbor и без него.

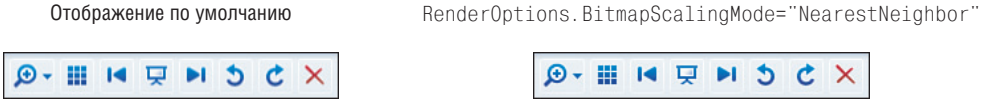


Рис. 11.1. *Если свойство BitmapScalingMode равно NearestNeighbor, то края выглядят четче*

СОВЕТ

Вместо того чтобы пользоваться конвертером типа для преобразования строкового имени файла в объект ImageSource, можно явно присвоить свойству Source объекта Image ссылку на объект одного из подклассов ImageSource, что открывает дополнительные возможности. Например, в подклассе BitmapImage есть ряд свойств, таких как DecodePixelWidth и DecodePixelHeight, с помощью которых можно задать размер изображения, меньший естественного, и тем самым сэкономить память – иногда довольно ощутимо. Подкласс FormatConvertedBitmap позволяет изменять формат пикселов Image, создавая различные эффекты, например переход к полутоновому изображению. В следующей XAML-разметке возможности класса FormatConvertedBitmap применяются для получения результата, показанного на рис. 11.2:

```
<StackPanel Orientation="Horizontal">

  <!-- Нормальное изображение, формат пикселов Pbgra32: -->
  <Image Source="photo.jpg" />

  <!-- Полутоновое изображение: -->
  <Image>
    <Image.Source>
      <FormatConvertedBitmap Source="photo.jpg" DestinationFormat="Gray32Float" />
    </Image.Source>
  </Image>

  <!-- Черно-белое изображение: -->
  <Image>
```

```
      <Image.Source>
        <FormatConvertedBitmap Source="photo.jpg" DestinationFormat="BlackWhite" />
      </Image.Source>
    </Image>

  </StackPanel>
```

Длинный перечень возможных форматов определен в перечислении `System.Windows.Media.PixelFormats`.



Pbgra32 (по умолчанию) Gray32Float BlackWhite

***Рис. 11.2.** Отображение Image с тремя разными форматами пикселов (см. также цветную вклейку)*

Элементы управления Text и Ink

Помимо элементов `TextBlock` и `Label` WPF содержит еще ряд элементов для отображения и редактирования текста – посредством как клавиатурного набора, так и рукописного ввода с помощью стилуса. В этом разделе мы чуть подробнее рассмотрим элемент `TextBlock`, а также поговорим о следующих элементах:

- `TextBox`
- `RichTextBox`
- `PasswordBox`
- `InkCanvas`

Но сначала упомянем о важном усовершенствовании в WPF 4, которое распространяется на все способы визуализации текста. С самого начала пользователи WPF жаловались на размытость текста. (Я сам неоднократно заявлял, что смогу мгновенно распознать созданный на WPF интерфейс по нечеткому тексту!) Механизм визуализации текста в WPF был оптимизирован для крупного кегля и/или экрана со сверхвысоким разрешением с учетом точности масштабирования и отличного качества передачи при печати. Но при работе

с кеглями шрифтов, характерными для большинства приложений, и с разрешениями современных мониторов выявились недостатки такого подхода. Когда хотят вежливо описать эту ситуацию, говорят, что визуализация текста в WPF опередила время.

Рад сообщить вам, что в WPF 4 эти проблемы разрешены. Как и во многих областях, где была повышена производительность, кое-какие усовершенствования в части визуализации текста вы получаете задаром. (Например, WPF теперь автоматически использует растровые изображения, внедренные в некоторые восточноазиатские шрифты, чтобы получать четкий текст при мелких кеглях.) Другие же усовершенствования необходимо активировать явно – в целях сохранения совместимости с существующими приложениями.

Главное, о чем нужно знать, – это присоединенное свойство `TextOptions.TextFormattingMode`. Его можно задавать как для отдельных текстовых элементов, так и – что практикуется чаще – для родительского элемента, например `Window`; в последнем случае оно распространяется на визуализацию текста во всем дереве элементов-потомков. Присвоив свойству `TextFormattingMode` значение `Display`, вы включите новый механизм визуализации текста в WPF 4, в котором применяются метрики текста, совместимые с GDI. С точки зрения четкости текста основная особенность этого механизма состоит в том, что каждый глиф позиционируется на границе пикселов (а его ширина кратна ширине пиксела).

Подразумеваемое по умолчанию значение `TextFormattingMode` – то самое, которое причиняло столько неприятностей разработчикам и пользователям, – по иронии судьбы названо `Ideal`. В этом случае метрики текста обеспечивают максимально точное следование определению шрифта, даже если это означает, что глифы не совмещаются с границами пикселов. В будущем идеальном мире, где плотность размещения пикселов на экране будет куда выше нынешней, это действительно даст оптимальный результат (как и сегодня при отображении текста крупным кеглем).

Присоединенному свойству `TextOptions.TextRenderingMode` можно присвоить значение `ClearType`, `Grayscale`, `Aliased` или `Auto` – для управления режимом сглаживания текста (antialiasing). При заданном значении `Auto` (по умолчанию) будет действовать режим `ClearType`, если эта технология не отключена на данном компьютере, в противном случае – режим `Grayscale`.

На рис. 11.3 показана разница между двумя значениями `TextFormattingMode` и тремя значениями `TextRenderingMode`, отличными от `Auto`, хотя на печатной странице заметить разницу сложно.

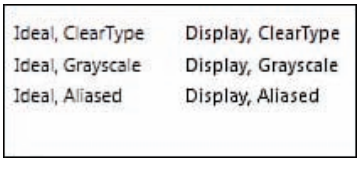


Рис. 11.3. *Настройка визуализации текстовых блоков при кегле FontSize=11*

Далее свойству TextOptions.TextHintingMode можно присвоить значение Fixed, Animated или Auto – для оптимизации отображения в зависимости от того, является текст стационарным или анимированным.

FAQ

Не следует ли всегда задавать значение Display свойства TextFormattingMode, чтобы оптимизировать визуализацию текста?

Нет. Если текст отображается достаточно крупным кеглем (FontSize порядка 15 или больше), то режим Ideal дает такое же четкое изображение, как режим Display, а глифы располагаются лучше. Но еще важнее то, что в случае применения к тексту геометрического преобразования режим Display оказывается хуже, поскольку выравнивание на границы пикселов больше не применяется. Хуже всего выглядит текст в режиме Display после увеличения в результате применения ScaleTransform, потому что WPF просто масштабирует растровое изображение текста, а не перерисовывает его более крупным шрифтом. (Так делается для того, чтобы гарантировать точное масштабирование с заданным коэффициентом, чего невозможно было бы достичь, если бы при большем кегле применялось выравнивание на границы пикселов.) Но для типичных меток, отображаемых мелким шрифтом, у режима Display нет конкурентов.

Элемент TextBlock

У элемента TextBlock есть ряд простых свойств, модифицирующих его внешний вид, например FontFamily, FontSize, FontStyle, FontWeight и FontStretch. Но главный сорприз TextBlock заключается в том, что его свойством содержимого является не Text, а коллекция объектов Inlines. Хотя показанная ниже разметка дает тот же самый результат, что и установка свойства Text, в действительности мы устанавливаем другое свойство:

```
<!-- Здесь устанавливается свойство TextBlock.Inlines: -->
<TextBlock>Text in a TextBlock</TextBlock>
```

Конвертер типа создает иллюзию, будто значением является простая строка, хотя на самом деле это коллекция, состоящая из одного элемента Run. Поэтому следующая XAML-разметка в точности эквивалентна предыдущей:

```
<TextBlock><Run Text="Text in a TextBlock"/></TextBlock>
```

и, в свою очередь, эквивалентна такой (поскольку Text – это свойство содержимого в классе Run):

```
<TextBlock><Run>Text in a TextBlock</Run></TextBlock>
```

Объект Run – это просто фрагмент текста с одним и тем же форматированием. Явное использование одного элемента Run не дает никаких преимуществ, но, когда в одном блоке TextBlock встречается несколько элементов Run, картина становится интереснее. Например, показанный выше TextBlock можно было бы записать и так:

```
<TextBlock>
  <Run>Text</Run>
  <Run> in</Run>
  <Run> a</Run>
  <Run> TextBlock</Run>
</TextBlock>
```

Результат визуализации при этом не изменяется. Однако в классе `Run` имеется несколько свойств форматирования, позволяющих переопределить соответствующие свойства, установленные в родительском элементе `TextBlock`, а именно: `FontFamily`, `FontSize`, `FontStretch`, `FontStyle`, `FontWeight`, `Foreground` и `TextDecorations`. Они используются в следующей XAML-разметке, результат визуализации которой показан на рис. 11.4:

```
<TextBlock>
  <Run FontStyle="Italic" FontFamily="Georgia" Foreground="Red">Rich</Run>
  <Run FontSize="30" FontFamily="Comic Sans MS" Foreground="Blue"> Text </Run>
  <Run FontFamily="Arial Black" Foreground="Orange" FontSize="100">in</Run>
  <Run FontFamily="Courier New" FontWeight="Bold" Foreground="Green"> a </Run>
  <Run FontFamily="Verdana" TextDecorations="Underline">TextBlock</Run>
</TextBlock>
```



***Рис. 11.4.** Несколько фрагментов Run с разным форматированием внутри одного блока TextBlock*

Это, конечно, крайность, но аналогичную технику можно применить, например, для курсивного начертания или подчеркивания одного слова в абзаце. Это гораздо проще, чем пытаться правильно позиционировать несколько элементов `TextBlock`. Кроме того, при использовании одного `TextBlock` вы получаете корректное отсечение и перенос на другую строку, даже если начертание текста неоднородно. Помимо `Run` есть много других объектов типа `Inline`; в разделе «Документы» ниже они рассматриваются более подробно.

КОПНЕМ ГЛУБЖЕ

TextBlock и пустое пространство

Если содержимое `TextBlock` устанавливается с помощью свойства `Text`, то все символы пробела сохраняются. Если же оно устанавливается с помощью свойства `Inlines` в XAML, то пустое пространство не сохраняется. Начальные и конечные пробелы при этом игнорируются, а соседние пробелы заменяются одним (как в HTML).

СОВЕТ

При добавлении содержимого в свойство Inlines элемента TextBlock его неформатированное представление дописывается в конец свойства Text. Поэтому программа по-прежнему может пользоваться свойством Text, даже если явно устанавливается только Inlines. Например, для блока TextBlock на рис. 11.4 значением Text является строка "Rich Text in a TextBlock", как и следовало ожидать.

КОПНЕМ ГЛУБЖЕ

Явно и неявно заданные фрагменты Run

Следующий элемент TextBlock:

```
<TextBlock>Text in a TextBlock</TextBlock>
```

эквивалентен такому:

```
<TextBlock><Run>Text in a TextBlock</Run></TextBlock>
```

но не всегда поведение конвертера типа настолько очевидно. Например, такое использование элемента LineBreak (еще одной разновидности Inline) допустимо:

```
<TextBlock>Text in<LineBreak/>a TextBlock</TextBlock>
```

а такое – нет:



```
<TextBlock><Run>Text in<LineBreak/>a TextBlock</Run></TextBlock>
```

Последний вариант недопустим, потому что свойством содержимого класса Run (Text) является простая строка, а включить элемент LineBreak внутрь строки нельзя. Однако конвертер типа преобразует свойство содержимого класса TextBlock (Inlines) в один или несколько объектов Run, корректно обрабатывая объекты LineBreak. В результате следующая XAML-разметка:

```
<TextBlock>Text in<LineBreak/>a TextBlock</TextBlock>
```

оказывается эквивалентной блоку TextBlock, содержащему два объекта Run, по одному с каждой стороны LineBreak:

```
<TextBlock><Run>Text in</Run><LineBreak/><Run>a TextBlock</Run></TextBlock>
```

Элемент TextBox

Элемент управления TextBox, изображенный на рис. 11.5, позволяет вводить одну или несколько строк текста. В отличие от большинства других элементов управления в WPF, содержимое TextBox хранится не в виде объекта типа System.Object, а в строковом свойстве Text.



Рис. 11.5. Элемент WPF TextBox

Хотя на первый взгляд TextBox выглядит очень просто, в него встроена весьма развитая функциональность, привязки для команд Cut, Copy, Paste, Undo и Redo (см. главу 6 «События ввода: клавиатура, мышь, стилус и мультисенсорные устройства») и даже проверка правописания!

В классе TextBox определено несколько методов и свойств для выбора различных частей текста (выделенного фрагмента, по номеру строку и т. д.), а также методы для поиска физической точки в тексте по номеру строки и символа и наоборот. Определены также события TextChanged и SelectionChanged.

Если на размер элемента TextBox не налагает ограничений окружение (и он не задан явно), то элемент растет по мере добавления в него текста. Если же ширина TextBox ограничена, то можно установить режим переноса строк, присвоив свойству TextWrapping значение Wrap или WrapWithOverflow. В режиме Wrap содержимое ни при каких условиях не может выйти за пределы области, занятой элементом, даже если придется разорвать строку в середине слова. В режиме WrapWithOverflow строка разрывается, только если есть такая возможность, так что длинные слова могут выйти за границы области. (В классе TextBlock также есть свойство TextWrapping.)

FAQ

Как сделать, чтобы элемент TextBox поддерживал ввод нескольких строк текста?

Если присвоить свойству AcceptsReturn значение true, то при нажатии клавиши Enter будет создаваться новая строка. Отметим, что TextBox в любом случае поддерживает создание многострочных текстов из программы. Если записать в свойство Text текст, содержащий символы NewLine, то он отобразится в виде нескольких строк вне зависимости от значения AcceptsReturn. Кроме того, поддержка многострочных текстов никак не связана с переносом строк. Перенос применяется только к отдельным строкам, длина которых превышает ширину TextBox.

КОПНЕМ ГЛУБЖЕ

Проверка правописания

Чтобы включить проверку правописания в TextBox (или RichTextBox), необходимо присвоить присоединенному свойству SpellCheck.IsEnabled значение true. Выглядит это примерно так же, как в Microsoft Word: неправильно написанные слова подчеркиваются красным цветом, а если щелкнуть по такому слову правой кнопкой мыши, будут предложены варианты исправления. WPF пользуется словарем, который сопоставим с применяемым в Microsoft Office и имеется для разных языков (входит в состав соответствующего языкового пакета). Однако пользовательские словари WPF не поддерживает.

Элемент RichTextBox

Элемент RichTextBox предоставляет больше возможностей, чем TextBox, поскольку может содержать форматированный текст (и допускает наличие в тексте произвольных объектов). На рис. 11.6 показан элемент RichTextBox с простым форматированным текстом.



***Рис. 11.6.** Элемент WPF RichTextBox*

У RichTextBox и TextBox общий базовый класс (TextBoxBase), поэтому многие возможности, описанные выше для TextBox, применимы и к RichTextBox. Но некоторые средства TextBox реализованы в RichTextBox более полно. Там, где TextBox предоставляет лишь простые целочисленные свойства CaretIndex, SelectionStart и SelectionEnd, RichTextBox предлагает свойство CaretPosition типа TextPointer и свойство Selection типа TextSelection. Кроме того, содержимое RichTextBox хранится в свойстве Document типа FlowDocument, а не в простом строковом свойстве Text. Содержимое может даже включать объекты типа UIElement, с которыми можно взаимодействовать и которые генерируют события, если свойство IsDocumentEnabled элемента RichTextBox имеет значение true. Класс FlowDocument обсуждается ниже в разделе «Документы».

Элемент PasswordBox

Элемент PasswordBox – это упрощенный вариант TextBox, предназначенный для ввода пароля. Вместо вводимых символов в нем отображаются кружочки, как показано на рис. 11.7.



***Рис. 11.7.** Элемент WPF PasswordBox*

Класс PasswordBox не наследует TextBoxBase, как два предыдущих элемента управления, поэтому не поддерживает ни команды Cut, Copy, Undo и Redo (хотя поддерживает команду Paste), ни проверку правописания. И это вполне разумное поведение элемента, предназначенного для хранения паролей!

Если вам не нравятся кружочки, можете выбрать другой символ с помощью свойства PasswordChar. (По умолчанию предполагается звездочка, которая отображается специальным шрифтом и выглядит, как кружочек.)

Текст элемента PasswordBox хранится в строковом свойстве Password. В действительности для более надежной защиты применяется специальный класс System.Security.SecureString. Содержимое объекта типа SecureString шифруется и намеренно стирается, тогда как объекты System.String не шифруются

и могут оставаться в куче неопределенно долгое время, пока не будут убраны сборщиком мусора.

При изменении пароля генерируется событие `TextboxPasswordChanged`. Его обработчик имеет тип `RoutedEventHandler`, то есть вместе с событием не передается информация о старом и новом паролях. Если нужно узнать текущий пароль, можно просто опросить внутри обработчика свойство `Password`.

Элемент InkCanvas

Основная задача поразительно гибкого элемента `InkCanvas` – предоставить средства для рукописного ввода (с помощью мыши или стилуса, но не мультисенсорного устройства). Его внешний вид показан на рис. 11.8. Строго говоря, `InkCanvas` – не элемент управления, поскольку наследует непосредственно классу `FrameworkElement`, но ведет он себя почти так же, как элементы управления (за исключением того, что его стиль нельзя изменить с помощью шаблона).

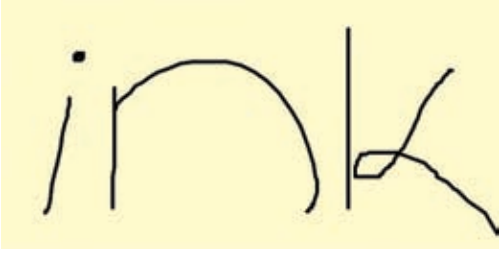


***Рис. 11.8.** Элемент WPF InkCanvas*

В режиме по умолчанию `InkCanvas` позволяет просто писать или рисовать на своей поверхности. При работе со стилусом заостренный конец рисует, а обратный конец стирает. Каждый нанесенный штрих запоминается в виде объекта `System.Windows.Ink.Stroke`, а все такие объекты сохраняются в коллекции `Strokes`. Но `InkCanvas` позволяет также хранить любое число произвольных объектов типа `UIElement` в коллекции `Children` (это его свойство содержимого). В результате очень легко пометить все что угодно рукописной надписью, как показано на рис. 11.9.



***Рис. 11.9.** Нанесение рукописных пометок поверх изображения*

Чтобы получить это изображение, я разрисовал стилусом следующее окно Window:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  SizeToContent="WidthAndHeight">
  <Grid>
    <InkCanvas>
      <Image Source="http://adamnathan.net/blog/images/anathan.png"/>
    </InkCanvas>
  </Grid>
</Window>
```

Здесь я воспользовался очень интересным свойством SizeToContent – если в этом режиме вы начнете рисовать за пределами окна, то объект Window автоматически изменит размер так, чтобы поместились все штрихи!

Свойство DefaultDrawingAttributes позволяет изменить внешний вид будущих штрихов (толщину, цвет и т. д.). В классе Stroke также есть свойство DrawingAttributes, так что внешний вид можно задавать для каждого штриха в отдельности.

Элемент InkCanvas поддерживает несколько режимов, которые можно независимо применять к заостренному концу стилуса (или к мыши) – посредством свойства EditingMode – и к обратному его концу – с помощью свойства EditingModeInverted. Предназначенное только для чтения свойство ActiveEditingMode сообщает, какой режим действует в данный момент. Все три эти свойства имеют тип перечисления InkCanvasEditingMode, в котором определены следующие значения:

- Ink – рисование штрихов мышью или стилусом. Это подразумеваемое по умолчанию значение свойства EditingMode.
- InkAndGesture – аналогично Ink, но распознает также жесты пользователя. Список поддерживаемых жестов (Up, Down, Circle, ScratchOut, Tap и др.) определен в перечислении System.Windows.Ink.ApplicationGesture.
- GestureOnly – только распознает жесты, никакие штрихи не рисуются.
- EraseByStroke – стирает весь штрих, которого коснулся стилус. Это подразумеваемое по умолчанию значение свойства EditingModeInverted.
- EraseByPoint – стирает только часть штриха, находящуюся непосредственно под стилусом (как обычный ластик).
- Select – выделяет штрихи или другие элементы UIElement при касании, так чтобы впоследствии сразу ко всем можно было применить операцию удаления, перемещения или изменения размеров в границах InkCanvas.
- None – никак не реагирует на попытки ввода данных мышью или стилусом.

Применение режима Select к обычным элементам, не имеющим ничего общего с рукописным вводом, – довольно интересная возможность, поскольку она позволяет организовать примитивную область конструирования для размещения элементов. В классе InkCanvas определено также 15 событий, генерируемых при выполнении таких операций, как смена режима редактирования,

корректировка, перемещение и изменение размера выделенных элементов, запоминание и стирание штрихов и распознавание жестов.

Разумеется, поддержку рукописного ввода включают в приложение не только для рисования усов на фотографиях! Зачастую требуется интерпретировать нанесенные штрихи как написанный от руки текст. В WPF встроено распознавание жестов, но механизм распознавания рукописных символов отсутствует.

Документы

Элементы управления `TextBlock` и `Label` предназначены для отображения статического текста, а элементы `TextBox` и `RichTextBox` – для показа редактируемого текста. Но в части работы с текстами WPF может предложить и куда более развитую функциональность!

В WPF имеется обширный набор классов для создания, просмотра, модификации, организации и хранения высококачественных документов. В настоящем разделе мы будем говорить о так называемых *потоковых документах*. Такой документ (представленный объектом типа `FlowDocument`) содержит текст и другие данные, которые требуется расположить так, чтобы оптимально использовалось отведенное под документ место. Например, на мониторах с широким экраном можно автоматически добавлять дополнительные колонки.

Создание потоковых документов

Класс `FlowDocument` наследует классу `FrameworkContentElement`, аналогу `FrameworkElement`, ориентированному на работу с содержимым. Все элементы типа `FrameworkContentElement`, как и элементы типа `FrameworkElement`, поддерживают привязку к данным, анимацию и другие механизмы WPF, но не участвуют в механизме компоновки. При отображении на экране элементы `FrameworkContentElement` располагаются внутри какого-то элемента `FrameworkElement`.

FAQ

Как поддержка потоковых документов в WPF соотносится со спецификацией XML Paper Specification (XPS)?

В отличие от документов с динамической компоновкой, описываемых в этом разделе, XPS-документы имеют фиксированную компоновку и одинаково выглядят как на экране, так и на бумаге. В каркас .NET Framework включены классы для создания и просмотра XPS-документов (они находятся в пространствах имен `System.Windows.Xps` и `System.Windows.Documents`), но для этих целей можно пользоваться также инструментальными средствами, например Microsoft Word. В WPF-приложениях XPS-документы обычно представляются объектами типа `FixedDocument` и просматриваются с помощью элемента управления `DocumentViewer`.

XPS-документы во многом напоминают документы в формате Adobe PDF; для тех и других есть автономные программы для просмотра (на разных платформах), и их можно просматривать в браузере (при наличии подходящего расширения). Одной из уникальных особенностей XPS является тот факт, что это одновременно и «родной» формат спулингового файла Windows (начиная с Windows Vista). Это означает, что XPS-документы можно печатать без потери качества и точности воспроизведения, не требуя никакой дополнительной работы от приложения, отправляющего документ на печать.

Спецификацию XPS и положенную в ее основу спецификацию Open Packaging Conventions (поддерживающие ее классы находятся в пространстве имен `System.IO.Packaging`) можно найти по адресу *http://microsoft.com/xps*.

Еще одним классом, производным от `FrameworkContentElement`, является `TextElement` – абстрактный класс, представляющий содержимое, которое можно поместить в документ `FlowDocument`. В этом разделе мы рассмотрим различные подклассы `TextElement` (все они находятся в пространстве имен `System.Windows.Documents`) и покажем, как с помощью их композиции можно создавать гибкие документы с разнородным наполнением.

Простой потоковый документ

В следующей XAML-разметке показан простой объект `FlowDocument`, представляющий собой коллекцию абзацев `Paragraph` (типа `TextElement`) из черновика первой главы этой книги.

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Paragraph FontSize="22" FontWeight="Bold">Chapter 1</Paragraph>
  <Paragraph FontSize="35" FontWeight="Bold">Why WPF?</Paragraph>
  <Paragraph>
    In movies and on TV, the ...
  </Paragraph>
  <Paragraph>...</Paragraph>
  <Paragraph>...</Paragraph>
...
</FlowDocument>
```

На рис. 11.10 мы видим результат визуализации этого XAML-кода. Если сделать подобный элемент `FlowDocument` корнем XAML-файла, то его можно будет отобразить в подходящей программе просмотра.

Есть два основных типа элементов `TextElement` – `Block` и `Inline` (оба класса являются абстрактными, производными от `TextElement`). `Block` занимает прямоугольную область, которая может разрываться только при переходе на другую страницу, а `Inline` – заполняемую текстом область, которая, в принципе, может оказаться и непрямоугольной (перетекать из конца одной строки в начало следующей). Элемент `FlowDocument` может содержать только блоки `Block` в качестве дочерних элементов. (Его свойство содержимого называется `Blocks` и имеет тип `BlocksCollection`.) Роль элементов `Inline` мы рассмотрим после того, как поближе ознакомимся с блоками.


Chapter 1

Why WPF?

In movies and on TV, the main characters are typically an exaggeration of the people you encounter in real life. They're more attractive, they react quicker, and somehow always know exactly what to do. The same could be said about the software they use.

This especially struck me when watching the movie Disclosure in 1994, starring Michael Douglas, Demi Moore, and an e-mail program that looks nothing like Microsoft Outlook! Throughout the movie, we're treated to various visual features of the program: a spinning 3D "e," messages that unfold in 3D when you open them and crumple in 3D when you delete them, hints of inking support, and slick 2D/3D animations when you print messages. (And believe me, the e-mail program isn't the most unrealistic software in the movie. I'll just say "virtual reality database" and leave it at that.)

Usability issues aside, Hollywood has been telling us for a long time that software in the real world isn't as cool as it should be. But real-world software is starting to catch up! You can already see it in traditional operating systems (Mac OS and more recently Windows Vista), in software for devices such the TiVo or Xbox, and on the

Рис. 11.10. Простой документ типа FlowDocument

Класс Block

В WPF есть пять типов блоков:

- `Paragraph` – содержит коллекцию `Inlines`, которая обычно и составляет содержание документа. В XAML содержимым элемента `Paragraph` часто бывает простой текст, но внутри система обертывает этот текст объектом класса `Run`, производного от `Inline`, который и добавляется в коллекцию `Inlines` – так же, как в случае `TextBlock`.
- `Section` – группирует один или несколько блоков, не вводя никакой дополнительной структуры. Это удобно, когда нужно применить к нескольким блокам одно и то же значение некоторого свойства, например `Background` или `Foreground`.
- `List` – представляет коллекцию объектов типа `ListItem` в виде маркированного, нумерованного или простого списка. Каждый `ListItem` может содержать коллекцию блоков `Block`, так что создание типичного списка `List` подразумевает помещение объекта `Paragraph` внутрь каждого `ListItem`. Свойство `MarkerStyle` (типа `TextMarkerStyle`) позволяет задать различные стили форматирования маркеров – `Box`, `Circle`, `Disc` (подразумевается по умолчанию) и `Square` – и номеров – `Decimal`, `LowerLatin`, `UpperLatin`, `LowerRoman` и `UpperRoman`. Чтобы получить простой список, нужно присвоить свойству `MarkerStyle` значение `None`.
- `Table` – располагает содержимое в виде таблицы из строк и столбцов, наподобие `Grid`, но все же ближе к HTML-таблице. Элементы `Table`, в отличие от `Grid`, могут содержать только блоки `Block` (и элементы, описывающие структуру таблицы).
- `BlockUIContainer` – содержит единственный элемент `UIElement`. Поэтому `BlockUIContainer` – ключ к размещению разнообразного WPF-содержимого внутри `FlowDocument`: изображений `Image`, видео, содержащегося внутри `MediaElement`, кнопок `Button`, трехмерной графики внутри элемента `Viewport3D` и т. д.

В листинге 11.1 демонстрируется использование всех пяти видов блоков в документе `FlowDocument`. Результат визуализации этой разметки показан на рис. 11.11.

Листинг 11.1. Разметка документа FlowDocument, изображенного на рис. 11.11

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Section LineHeight="2" Foreground="White" Background="Black">
    <Paragraph FontSize="18">WPF 4 Unleashed</Paragraph>
    <Paragraph FontSize="30" FontWeight="Bold">Notes from Chapter 1</Paragraph>
  </Section>
  <Paragraph>Here are some highlights of WPF:</Paragraph>
  <List>
    <ListItem>
      <Paragraph>Broad integration</Paragraph>
    </ListItem>
    <ListItem>
      <Paragraph>Resolution independence</Paragraph>
    </ListItem>
    <ListItem>
      <Paragraph>Hardware acceleration</Paragraph>
    </ListItem>
    <ListItem>
      <Paragraph>Declarative programming</Paragraph>
    </ListItem>
    <ListItem>
      <Paragraph>Rich composition and customization</Paragraph>
    </ListItem>
  </List>
  <BlockUIContainer>
    <Viewbox>
      <StackPanel Orientation="Horizontal">
        <Image Source="diagram.jpg" Margin="5"/>
        <TextBlock VerticalAlignment="Center" Width="100" TextWrapping="Wrap">
          The technologies in the .NET Framework.
        </TextBlock>
      </StackPanel>
    </Viewbox>
  </BlockUIContainer>
  <Paragraph>
    Here's another version of the diagram, as a Table:
  </Paragraph>
  <Table CellSpacing="5" Padding="15" FontFamily="Segoe UI">
    <Table.Background>
      <LinearGradientBrush>
        <GradientStop Color="Yellow" Offset="0"/>
        <GradientStop Color="Orange" Offset="1"/>
      </LinearGradientBrush>
    </Table.Background>
```

```
<!-- Определяем четыре столбца: -->
<Table.Columns>
  <TableColumn/>
  <TableColumn/>
  <TableColumn/>
  <TableColumn/>
</Table.Columns>

<!-- Создаем три строки: -->
<TableRowGroup>
  <TableRow>
    <TableCell ColumnSpan="4" TextAlignment="Center">
      <Paragraph FontWeight="Bold">.NET Framework</Paragraph>
    </TableCell>
  </TableRow>

  <TableRow>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center" LineHeight="70">
      <Paragraph FontWeight="Bold">WPF</Paragraph>
    </TableCell>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center">
      <Paragraph FontWeight="Bold">WCF</Paragraph>
    </TableCell>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center">
      <Paragraph FontWeight="Bold">WF</Paragraph>
    </TableCell>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center">
      <Paragraph FontWeight="Bold">WCS</Paragraph>
    </TableCell>
  </TableRow>

  <TableRow>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center">
      <Paragraph FontWeight="Bold">ADO.NET</Paragraph>
    </TableCell>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center">
      <Paragraph FontWeight="Bold">ASP.NET</Paragraph>
    </TableCell>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center">
      <Paragraph FontWeight="Bold">Windows Forms</Paragraph>
    </TableCell>
    <TableCell BorderBrush="Black" BorderThickness="2"
      Background="LightGray" TextAlignment="Center">
      <Paragraph FontWeight="Bold">...</Paragraph>
    </TableCell>
```

```
        </TableRow>
      </TableRowGroup>
    </Table>
  </FlowDocument>
```

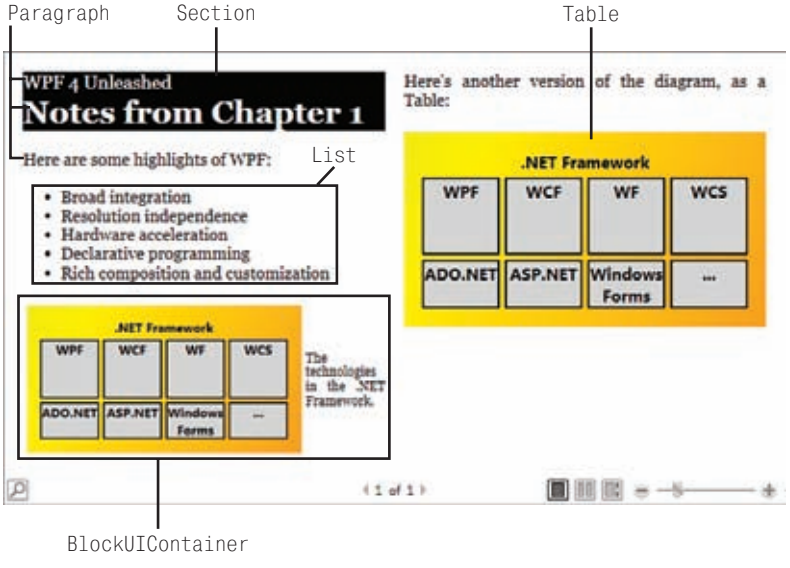


Рис. 11.11. *Документ FlowDocument, в котором встречаются все пять видов блоков*

Элементы `Paragraph` используются в этом документе повсеместно, а элемент `Section` – только в начале, чтобы задать для двух элементов `Paragraph` особые значения свойств `Foreground`, `Background` и `LineHeight`. Далее идет элемент `List` с настройками по умолчанию, соответствующими маркированному списку. Элемент `BlockUIContainer` содержит не только изображение `Image`, но и подпись к нему в виде элемента `TextBlock`. Оба эти элемента скомпонованы на панели `StackPanel` и помещены в элемент `Viewbox`, в результате чего они будут красиво масштабироваться при изменении ширины документа.

Наконец, просто для демонстрации мы имитировали изображение с помощью таблицы `Table`. Отметим, что API класса `Table` (а значит, и структура вложенных в `Table` элементов в XAML-коде) существенно отличается от `Grid`. Для определения столбцов мы помещаем элементы `TableColumn` в коллекцию `Columns` (аналог коллекции `ColumnDefinitions` в элементе `Grid`), но строки определяются непосредственно своим содержимым. Следовательно, `Table` содержит элемент `TableRowGroup`, внутри которого строки таблицы `TableRow` располагаются в том порядке, в котором должны присутствовать в документе, – сверху вниз. Ячейки `TableCell` внутри каждой строки `TableRow` заполняют столбцы последовательно, если только с помощью свойства `ColumnSpan` не задано другое поведение. `TableCell` – единственный элемент, который может содержать блоки `Block`, составляющие содержимое таблицы, – в данном случае это абзацы `Paragraph`.

Элемент Table может содержать несколько групп строки TableRowGroup! Строки, входящие в каждую группу, располагаются сразу под предыдущей группой.

На рис. 11.11 видно, что получившаяся таблица внешне очень похожа на включенное в документ изображение Image. Разумеется, их поведение сильно различается. Текст в таблице Table можно выделять, и он масштабируется при увеличении документа. Но если изображение никогда не разрывается при переходе на другую страницу, то для таблицы это допустимо. Кроме того, если места не хватает, то содержимое отдельных ячеек может переноситься на новую строку. На рис. 11.12 показаны и разрыв, и перенос.

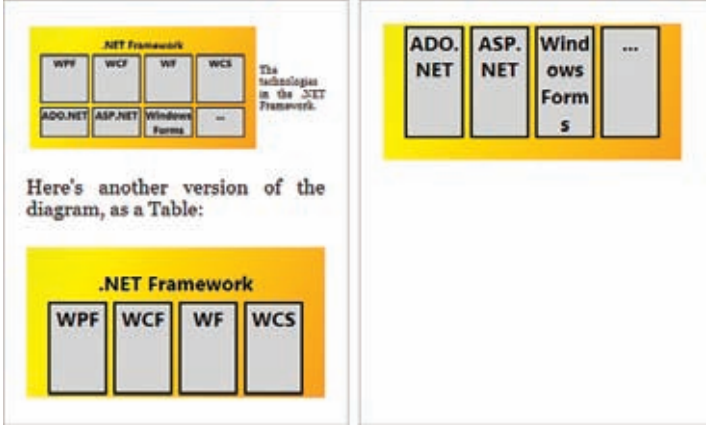


***Рис. 11.12.** Другое представление документа на рис. 11.11 – часть таблицы оказалась на странице 2, а часть – на странице 3*

Класс Inline

Элементы Inline могут находиться внутри Paragraph, позволяя добавлять к тексту эффектное форматирование. В предыдущем разделе было сказано, что на самом деле объект Paragraph содержит не просто строку, а коллекцию объектов Inline. И, хотя абзац Paragraph, представленный в XAML, вроде бы содержит чистый текст, в действительности это одиночный объект класса Run, производного от Inline. В классе Run имеется строковое свойство Text, а его конструктор принимает строку.

Таким образом, элемент Paragraph, определенный в XAML следующим образом:

```
<Paragraph>Here are some highlights of WPF:</Paragraph>
```

эквивалентен такому коду на C#:

```
Paragraph p = new Paragraph(new Run("Here are some highlights of WPF:"));
```

Остальные встраиваемые в абзацы элементы Inline можно разбить на три категории: отрезки (span), заякоренные блоки и все остальное.

Отрезки. Наиболее распространенными отрезками являются элементы `Bold`, `Italic`, `Underline` и уже знакомый `Hyperlink` из главы 7. Все они наследуют классу `Span`, который можно использовать внутри `Paragraph` и напрямую для применения к тексту дополнительных эффектов. Хотя класс `Paragraph` и сам поддерживает возможность изменять начертание своего текста (делать его полужирным, курсивным и т. д.) путем установки таких свойств, как `FontWeight` и `FontStyle`, отрезки позволяют применять нужный эффект к более мелким участкам текста, чем целый абзац.

В следующем элементе `Paragraph`, который показан на рис. 11.13, демонстрируются все виды отрезков:

```
<Paragraph>
  <Bold>bold</Bold>
  <Italic>italic</Italic>
  <Underline>underline</Underline>
  <Hyperlink>hyperlink</Hyperlink>
  <Span BaselineAlignment="Superscript">superscript</Span>
  <Span BaselineAlignment="Subscript">subscript</Span>
  <Span>
    <Span.TextDecorations>
      <TextDecoration Location="Strikethrough"/>
    </Span.TextDecorations>
    strikethrough
  </Span>
</Paragraph>
```

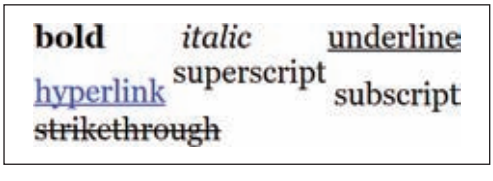


Рис. 11.13. Отрезки с различным форматированием внутри абзаца

СОВЕТ

Так как `TextBlock` хранит свое содержимое в виде коллекции `Inline`, то можно было бы заменить теги `Paragraph` в показанной выше разметке фрагментами с тегами `TextBlock`, и все работало бы по-прежнему. С другой стороны, элемент `Label` такую разметку не поддерживает.

Свойства `BaselineAlignment` и `TextDecorations`, примененные к элементу `Span`, на самом деле являются общими для всех подклассов `Inline`, так что их можно спокойно сочетать с `Bold`, `Italic` и другими эффектами. Кроме того, как и в случае `Paragraph`, содержимое любого отрезка на самом деле представляет собой коллекцию объектов `Inline`, а не простую строку. В показанной выше XAML-

разметке это означает, что каждый потомок Paragraph неявно обернут объектом Run. А заодно и то, что одни отрезки можно вкладывать в другие, как показано на примере следующего элемента Paragraph, визуализированного на рис. 11.14:

```
<Paragraph>
a<Bold>b<Italic>c<Underline>d<Hyperlink>e</Hyperlink>f</Underline>g</Italic>h
</Bold>i
</Paragraph>
```

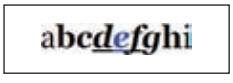


***Рис. 11.14.** Элемент Hyperlink, вложенный в Underline, который вложен в Italic, а тот, в свою очередь, вложен в Bold*

Заякоренные блоки. WPF содержит два подкласса Inline, необычных тем, что они используются как контейнеры для элементов Block. Это классы Figure и Floater – оба наследующие абстрактному классу AnchoredBlock.

Figure – это в каком-то смысле мини-FlowDocument, который можно вкладывать в объемлющий FlowDocument. Внутреннее содержимое изолировано от внешнего, которое обтекает Figure. Например, в документе FlowDocument с текстом главы 1 я мог бы сделать так, чтобы текст абзацев обтекал изображения (рисунки в этой книге именно так и размещены). Этого можно достичь следующим образом:

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Paragraph FontSize="22" FontWeight="Bold">Chapter 1</Paragraph>
  <Paragraph FontSize="35" FontWeight="Bold">Why WPF?</Paragraph>
  <Paragraph>
    <Figure Width="130">
      <BlockUIContainer>
        <Image Source="wpf.png"/>
      </BlockUIContainer>
    </Figure>
    In movies and on TV, the ...
  </Paragraph>
  <Paragraph>...</Paragraph>
  <Paragraph>...</Paragraph>
  ...
</FlowDocument>
```

Поскольку Figure может содержать элементы Block, значит, внутрь него можно помещать Table, Paragraph и другие элементы. Но в данном случае нам достаточно поместить один элемент BlockUIContainer, который содержит изображение. Результат показан на рис. 11.15.

Местоположением Figure можно управлять с помощью свойств HorizontalAnchor и VerticalAnchor (типа FigureHorizontalAnchor и FigureVerticalAnchor соответст-

венно). По умолчанию HorizontalAnchor равно ColumnRight, а VerticalAnchor – ParagraphTop. Оба свойства позволяют задавать различные режимы размещения относительно текущего столбца либо абзаца или даже страницы в целом. На рис. 11.16 показаны некоторые альтернативные способы размещения элемента Figure, изображенного на рис. 11.15, при различных значениях HorizontalAnchor и/или VerticalAnchor.

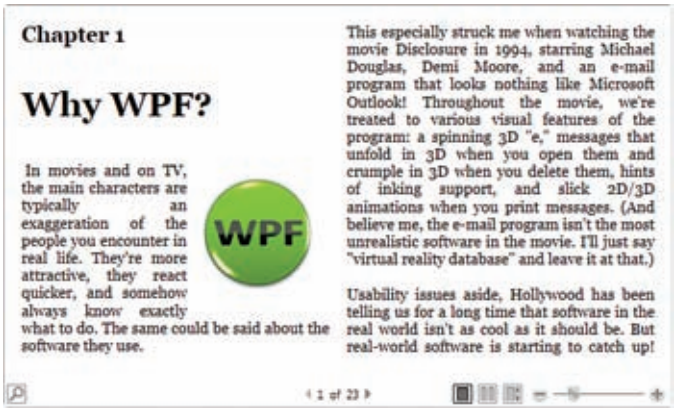


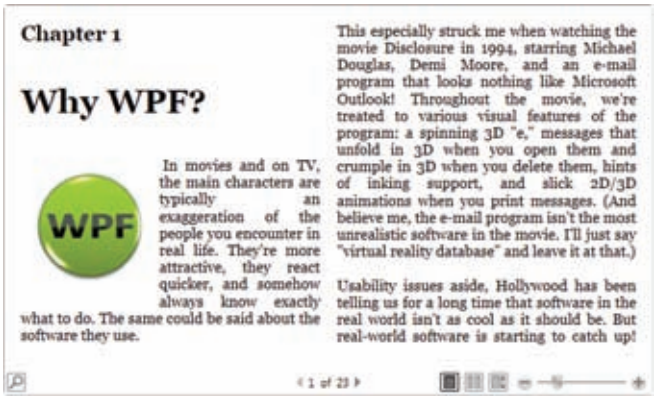
***Рис. 11.15.** В третьем абзаце документа FlowDocument находится элемент Figure, содержащий изображение Image*

Элемент Floater – упрощенный вариант Figure. Он может содержать произвольные блоки Block, но не поддерживает ни позиционирование относительно границ страницы, ни даже распространение на несколько столбцов. Вместо двух свойств HorizontalAnchor и VerticalAnchor в нем есть только одно простое свойство HorizontalAlignment (типа HorizontalAlignment), которое может принимать значения Left, Center, Right и Stretch. Если вам ни к чему полная функциональность Figure, то можете использовать вместо него более легкий элемент Floater.

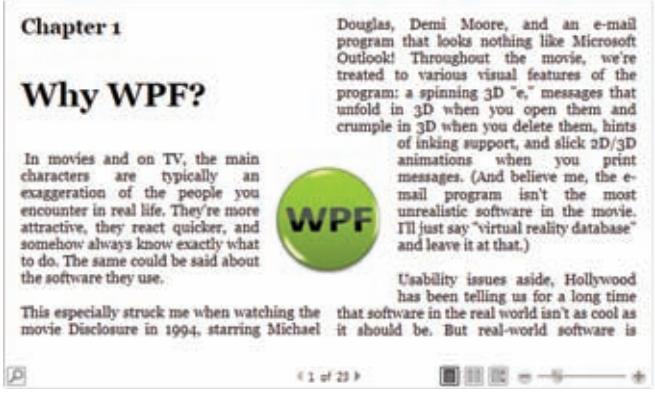
Прочие элементы Inline. Два оставшихся элемента Inline не имеют ничего общего за исключением того факта, что они не наследуют ни одному из классов Span или AnchoredBlock. Один из них – LineBreak, который играет роль символа новой строки. Если поместить пустой элемент LineBreak между двумя символами в абзаце, то второй из них окажется в начале новой строки.

СОВЕТ

Чтобы вставить в FlowDocument разрыв не *строки*, а *страницы*, задайте свойство BreakPageBefore для того элемента Paragraph, перед которым нужно перейти на новую страницу. Свойство BreakPageBefore определено в классе Block, так что применимо также к Section, List, BlockUIContainer и Table.



`HorizontalAnchor="ColumnLeft"`



`HorizontalAnchor="PageCenter"`



`HorizontalAnchor="PageRight"` и `VerticalAnchor="PageTop"`

Рис. 11.16. *Управление местоположением Figure с помощью свойств HorizontalAnchor и VerticalAnchor*

И последний класс, производный от Inline, – это InlineUIContainer, который отличается от BlockUIContainer лишь тем, что может быть вложен внутрь Paragraph и будет размещаться в одном потоке с остальным текстом. Как и BlockUIContainer, он может содержать видео в элементе MediaElement, кнопки Button, трехмерную графику в элементе Viewport3D и т. д., но чаще всего его применяют для того, чтобы включить в поток небольшое изображение. Следующий абзац, показанный на рис. 11.17, демонстрирует, как можно вставить значок RSS-ленты рядом с гиперссылкой Hyperlink на эту ленту:

```
<Paragraph>
  You can read more about this on my blog (
  <Hyperlink NavigateUri="http://blogs.msdn.com/adam_nathan/rss.xml">
    subscribe
  </Hyperlink>
  <InlineUIContainer>
    <Image Width="14" Source="rss.gif"/>
  </InlineUIContainer>
  ), which I try to update once a month.
</Paragraph>
```

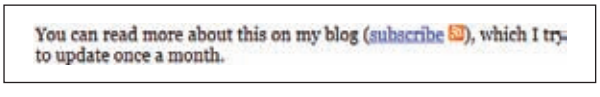


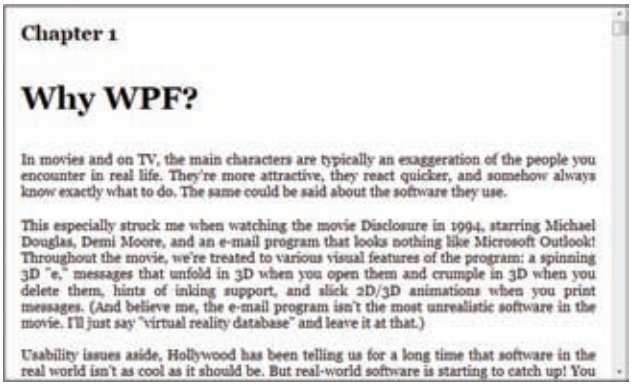
***Рис. 11.17.** Элемент Paragraph с изображением Image в общем потоке – благодаря InlineUIContainer*

Отображение потоковых документов

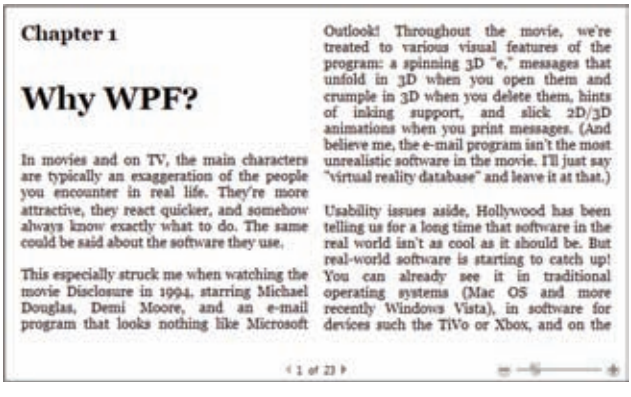
Выше уже упоминалось, что элемент FlowDocument можно отображать (и редактировать) внутри RichTextBox. Хотя редактирование можно запретить, установив для элемента RichTextBox свойство IsReadOnly равным true, RichTextBox не задумывался как основной элемент для чтения документов.

Вместо этого WPF предлагает *три* дополнительных элемента для отображения потоковых документов. Поначалу разобраться в них, возможно, и нелегко, но различия достаточно понятны:

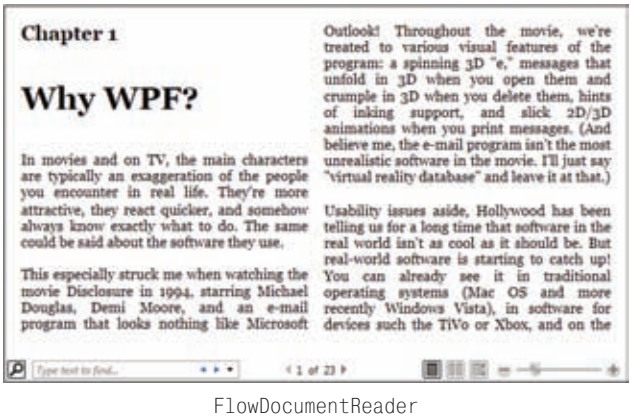
- FlowDocumentScrollViewer – отображает документ как один непрерывный файл с полосой прокрутки, как в режиме веб-документа в Microsoft Word (аналогично доступному только для чтения RichTextBox, помещенному в ScrollViewer).
- FlowDocumentPageViewer – отображает документ в виде набора отдельных страниц, как в режиме полноэкранного чтения в Microsoft Word.
- FlowDocumentReader – объединяет FlowDocumentScrollViewer и FlowDocumentPageViewer в один элемент управления и предлагает дополнительную функциональность, в частности встроенный текстовый поиск. (Такой элемент управления вы получаете по умолчанию, если сделаете FlowDocument корневым элементом XAML-файла.)



FlowDocumentScrollViewer



FlowDocumentPageViewer



FlowDocumentReader

***Рис. 11.18.** Текст главы 1 в каждом из контейнеров документов FlowDocument*

На рис. 11.18 показаны различия между этими элементами управления на примере документа `FlowDocument`, содержащего черновик первой главы. Элемент

FlowDocumentReader наделен весьма развитой функциональностью (наподобие стандартных программ просмотра XPS- и PDF-файлов), но если вам не нужна возможность переключаться между режимами прокрутки и постраничного просмотра, то, пожалуй, лучше ограничиться более простыми средствами просмотра. И FlowDocumentPageViewer, и FlowDocumentReader (в режиме постраничного просмотра) автоматически добавляют или убирают колонки при уменьшении либо увеличении масштаба, чтобы оптимально использовать имеющееся место.

Отметим, что FlowDocumentScrollViewer не содержит инструментов для управления масштабом, как остальные два элемента, но их можно добавить, присвоив свойству IsToolBarVisible значение true.

Добавление комментариев

Все три элемента для просмотра документов типа FlowDocument (а также DocumentViewer, предназначенный для просмотра документов типа FixedDocument) поддерживают добавление комментариев, то есть позволяют пользователю подсвечивать часть содержимого или присоединять заметки в виде печатного либо рукописного текста. Странно то, что для реализации этой возможности вы должны самостоятельно организовать пользовательский интерфейс – никаких элементов управления по умолчанию не предусмотрено.

Конструировать свой интерфейс для ввода комментариев утомительно, но не слишком трудно. На помощь приходит класс AnnotationService в пространстве имен System.Windows.Annotations, в котором имеются команды для всех нужных функций:

- CreateTextStickyNoteCommand присоединяет новый текстовый элемент StickyNoteControl в качестве комментария к выделенному тексту.
- CreateInkStickyNoteCommand присоединяет новый рукописный элемент StickyNoteControl в качестве комментария к выделенному тексту.
- DeleteStickyNotesCommand удаляет выделенные в данный момент элементы StickyNoteControl.
- CreateHighlightCommand подсвечивает выделенный текст цветом, переданным команде в качестве параметра.
- ClearHighlightsCommand удаляет подсветку с выделенного в данный момент текста.

В листинге 11.2 определено окно Window, в котором над элементом FlowDocumentReader размещено несколько простых кнопок. С каждой кнопкой ассоциирована одна из описанных выше команд.

Листинг 11.2. Window1.xaml – пользовательский интерфейс для FlowDocumentReader с возможностью добавления комментариев

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:a="clr-namespace:System.Windows.Annotations;assembly=PresentationFramework"
```

```
    Title="FlowDocumentReader + Annotations"
    x:Class="Window1" Initialized="OnInitialized" Closed="OnClosed">
    <StackPanel>
      <StackPanel Orientation="Horizontal">
        <Label>Control Annotations:</Label>
        <Button Command="a:AnnotationService.CreateTextStickyNoteCommand"
                CommandTarget="{Binding ElementName=reader}">
          Create Text Note
        </Button>
        <Button Command="a:AnnotationService.CreateInkStickyNoteCommand"
                CommandTarget="{Binding ElementName=reader}">
          Create Ink Note
        </Button>
        <Button Command="a:AnnotationService.DeleteStickyNotesCommand"
                CommandTarget="{Binding ElementName=reader}">
          Remove Note
        </Button>
        <Button Command="a:AnnotationService.CreateHighlightCommand"
                CommandParameter="{x:Static Brushes.Yellow}"
                CommandTarget="{Binding ElementName=reader}">
          Create Yellow Highlight
        </Button>
        <Button Command="a:AnnotationService.ClearHighlightsCommand"
                CommandTarget="{Binding ElementName=reader}">
          Remove Highlight
        </Button>
      </StackPanel>

      <FlowDocumentReader x:Name="reader">
        <FlowDocument>
        ...
        </FlowDocument>
      </FlowDocumentReader>
    </StackPanel>
  </Window>
```

Пространству имен .NET `System.Windows.Annotations` сопоставлен префикс пространства имен XML `a`, с помощью которого мы ссылаемся на команды в классе `AnnotationService`. Хотя `AnnotationService` — часть `PresentationFramework`, это пространство имен почему-то не включено в состав стандартного для WPF пространства имен XML. Чтобы команды заработали, в каждой кнопке в качестве цели команды указан элемент `FlowDocumentReader`. Кнопки становятся активными и неактивными автоматически в зависимости от контекста, в котором допустима соответствующая команда.

Осталось только определить методы `OnInitialized` и `OnClosed`, на которые есть ссылки в XAML-файле. В листинге 11.3 приведен застраничный файл для разметки, представленной в листинге 11.2.

Листинг 11.3. Window1.xaml.cs – процедурный код для элемента FlowDocumentReader с возможностью добавления комментариев

```
using System;
using System.IO;
using System.Windows;
using System.Windows.Annotations;
using System.Windows.Annotations.Storage;

public partial class Window1 : Window
{
  FileStream stream;

  public Window1()
  {
    InitializeComponent();
  }

  protected void OnInitialized(object sender, EventArgs e)
  {
    // Включить и загрузить комментарии
    AnnotationService service = AnnotationService.GetService(reader);
    if (service == null)
    {
      stream = new FileStream("storage.xml", FileMode.OpenOrCreate);
      service = new AnnotationService(reader);
      AnnotationStore store = new XmlStreamStore(stream);
      store.AutoFlush = true;
      service.Enable(store);
    }
  }

  protected void OnClosed(object sender, EventArgs e)
  {
    // Выключить и сохранить комментарии
    AnnotationService service = AnnotationService.GetService(reader);
    if (service != null && service.IsEnabled)
    {
      service.Disable();
      stream.Close();
    }
  }
}
```

Основная задача методов `OnInitialized` и `OnClosed` – включать и выключать службу `AnnotationService`, ассоциированную с объектом `FlowDocumentReader`. Однако при включении службы необходимо также указать поток `Stream`, в котором будут сохраняться комментарии. В листинге 11.3 для этого используется отдельный XML-файл в текущем каталоге. При закрытии приложения все комментарии сохраняются и появляются снова при следующем запуске (при условии, что файл *storage.xml* существует и не поврежден).

На рис. 11.19 показано, как выглядит это окно с добавленными комментариями.

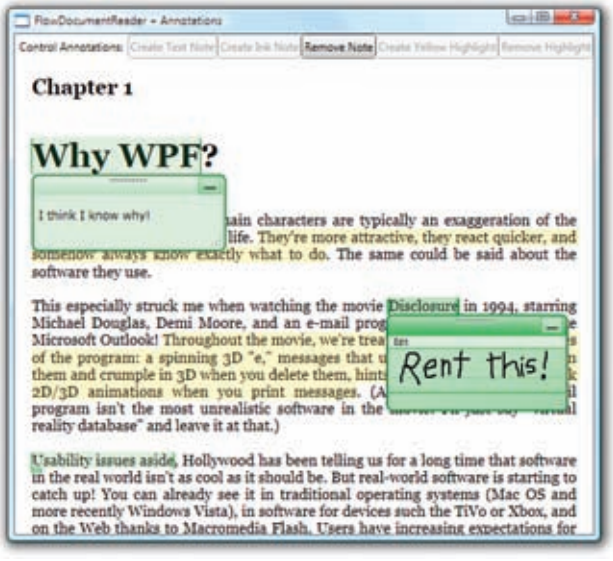


Рис. 11.19. *Кнопки в верхней части окна позволяют работать с комментариями в документе FlowDocument*

СОВЕТ

Элементы `StickyNoteControl`, которыми представлены комментарии, – это полноценные элементы управления WPF (находятся в пространстве имен `System.Windows.Controls`). Поэтому их внешний облик можно полностью изменить, задав другой шаблон.

Диапазонные элементы управления

Диапазонные элементы управления не предназначены для визуализации произвольного содержимого, как однодетные или многодетные элементы. Диапазонный элемент просто хранит и отображает числовое значение, принадлежащее определенному диапазону.

Большую часть своей функциональности диапазонные элементы наследуют от абстрактного класса `RangeBase`. В нем определены свойства типа `double`, в которых хранятся текущее значение и границы диапазона: `Value`, `Minimum`, `Maximum`. Там же определено событие `ValueChanged`.

В этом разделе мы рассмотрим два основных диапазонных элемента: `ProgressBar` и `Slider`. В WPF имеется также примитивный элемент `ScrollBar`, производ-

ный от RangeBase, но маловероятно, что вы захотите воспользоваться им напрямую. Вместо этого лучше прибегнуть к классу ScrollViewer, который был описан в главе 5.

Элемент ProgressBar

Если бы мир был идеален, то вам никогда не пришлось бы использовать в своих программах индикатор выполнения ProgressBar. Однако некоторые операции все-таки выполняются долго, и ProgressBar вселяет в пользователей уверенность в том, что программа что-то делает. Поэтому расположенный в нужном месте индикатор значительно улучшает впечатление пользователей от программы. (Разумеется, впечатление было бы еще лучше, если бы медленная операция выполнялась быстрее!) На рис. 11.20 показано, как выглядит элемент управления WPF ProgressBar по умолчанию.



***Рис. 11.20.** Элемент управления WPF ProgressBar*

По умолчанию свойство Minimum элемента ProgressBar равно 0, а свойство Maximum — 100. Он добавляет к своему базовому классу RangeBase еще два свойства:

- IsIndeterminate — если оно равно true, то в ProgressBar показывается обобщенная анимация, при этом значения свойств Minimum, Maximum и Value не учитываются. Это удобно, когда вы не знаете заранее, сколько времени займет операция, или ленитесь написать код, необходимый для показа реального положения вещей!
- Orientation — по умолчанию равно Horizontal, но может быть сделано равным Vertical, тогда индикатор будет двигаться сверху вниз, а не слева направо. Мне не доводилось встречать приложения, в которых индикаторы выполнения имеют вид «термометра», если не считать старомодных полноэкранных установщиков. Но если очень хочется, это свойство позволяет добиться такого эффекта!

FAQ

Как с помощью ProgressBar показать, что операция приостановлена или остановлена из-за ошибки?

Начиная с Windows Vista, индикатор выполнения в Win32 может визуализировать состояние приостановки (желтым цветом) и остановки/ошибки (красным цветом). К сожалению, в элементе WPF ProgressBar такая возможность не реализована. Если вы хотите получить подобный эффект, то должны будете создать новые шаблоны для этих состояний и применять их к элементу программно с помощью техники, описанной в главе 14 «Стили, шаблоны, обложки и темы».

Элемент Slider

Элемент `Slider` (ползунок) несколько сложнее, чем `ProgressBar`, так как позволяет изменять текущее значение, перемещая *ползунок* на любое число необязательных *делений*. Этот элемент изображен на рис. 11.21.



***Рис. 11.21.** Элемент управления WPF Slider*

По умолчанию значение `Minimum` для него равно `0`, а значение `Maximum` – `10`. Кроме того, в классе `Slider` определено свойство `Orientation` (по умолчанию равное `Horizontal`), а также ряд свойств для задания положения и частоты делений, положения и точности всплывающих подсказок `ToolTip`, которые показывают текущее значение по мере перемещения ползунка, и признак, говорящий о том, должен ли ползунок точно совмещаться с дискретными делениями или может перемещаться плавно. Для работы с клавиатурой в классе `Slider` имеются также свойства `Delay` и `Interval`, которые по своему поведению очень похожи на одноименные свойства элемента `RepeatButton`.

Чтобы появились деления, необходимо присвоить свойству `TickPlacement` значение `TopLeft`, `BottomRight` или `Both`. Странные названия объясняются желанием учесть обе ориентации `Slider`. Когда `TickPlacement` равно `BottomRight`, деления располагаются под ползунком, если он ориентирован горизонтально, и справа от него – если вертикально. Аналогично, когда `TickPlacement` равно `TopLeft`, деления располагаются над ползунком, если он ориентирован горизонтально, и слева от него – если вертикально. Когда `TickPlacement` равно `None` (режим по умолчанию), ползунок выглядит, как показано на рис. 11.22.



***Рис. 11.22.** Элемент Slider без делений*



***Рис. 11.23.** Элемент Slider поддерживает выделение меньшего поддиапазона*

У элемента `Slider` есть одна интересная особенность – он умеет отображать меньший диапазон внутри текущего, как показано на рис. 11.23. Если свойство `IsSelectionRangeEnabled` равно `true`, то свойствам `SelectionStart` и `SelectionEnd` можно присвоить значения границ такого «поддиапазона». В классе нет никаких встроенных средств, позволяющих задавать поддиапазон с помощью клавиатуры или мыши, и не гарантируется, что ползунок всегда остается в пределах поддиапазона. Эта возможность позволяет сделать ползунок похожим на тот, что используется в Windows Media Player, где закрашенная область показывает, какая часть воспроизводимого файла уже загружена.

Календарные элементы управления

В WPF 4 появилось два новых календарных элемента управления, позволяющих очень наглядно выбирать и отображать даты: `Calendar` и `DatePicker`. Их отсутствие в предыдущих версиях WPF ощущалось очень сильно, так что они стали желанным дополнением.

Элемент Calendar

Элемент управления `Calendar`, показанный на рис. 11.24, отображает календарь, очень похожий на стандартный календарь в Windows. С помощью свойства `DisplayMode` он поддерживает три разных режима. Щелкая по тексту в заголовке календаря, пользователь может расширять временной период, переходя от месяца (`Month`) к году (`Year`) или к десятилетию (`Decade`), а щелчок по любой ячейке календаря уменьшает период. В отличие от календаря Windows, элемент WPF `Calendar` не поддерживает режим показа столетия, а готовый стиль, к сожалению, не предусматривает приятной глазу анимации при переключении режима.



`Month` (по умолчанию) `Year` `Decade`

***Рис. 11.24.** Элемент WPF Calendar при различных значениях DisplayMode в том виде, в каком он выглядел бы 20 апреля 2012 года*

Свойство `DisplayDate` элемента `Calendar` (типа `DateTime`) по умолчанию инициализируется текущей датой (на рис. 11.24 это 20 апреля 2012 года). Дата `DisplayDate` при открытии календаря всегда видна, хотя в режиме `Month` она ничем визуально не отличается от других дат. Выделение 20 апреля серым цветом на рис. 11.24 объясняется тем, что `Calendar` подсвечивает сегодняшнюю дату независимо от значения `DisplayDate`. Чтобы отключить подсветку, присвойте свойству `IsTodayHighlighted` значение `false`.

В зависимости от свойства `SelectionMode` в календаре можно выделять одну или несколько дат:

- `SingleDate` – в любой момент времени может быть выделена только одна дата, которая хранится в свойстве `SelectedDate`. Это режим по умолчанию.
- `SingleRange` – можно выделять несколько дат, но они должны образовывать один непрерывный диапазон. Выделенные даты хранятся в свойстве `SelectedDates`.

- `MultipleRange` – выделенные даты не обязаны быть соседними, они хранятся в свойстве `SelectedDates`.
- `None` – выделять даты вообще нельзя.

Чтобы ограничить диапазон дат, отображаемых в элементе `Calendar`, можно задать свойства `DisplayDateStart` и/или `DisplayDateEnd` (типа `DateTime`). На рис. 11.25 показано, как это выглядит в каждом режиме `DisplayMode`. Иногда результат получается нелепым, потому что компоновка «шесть столбцов недель» в режиме `Month` и 4×4 в остальных режимах задана жестко.



`Month` (по умолчанию) `Year` `Decade`

***Рис. 11.25.** Так выглядит календарь, когда DisplayDateStart равно 10 апреля 2012, а DisplayDateEnd – 25 апреля 2012*

Можно вместо этого указать диапазоны, в которых запрещено выделять даты, хотя они и отображаются. Для этого служит свойство `BlackoutDates`, содержащее коллекцию объектов типа `CalendarDateRange`. На рис. 11.26 показано, что получается, когда в `BlackoutDates` записано два диапазона:

```
<Calendar>
  <Calendar.BlackoutDates>
    <CalendarDateRange Start="4/1/2012" End="4/19/2012"/>
    <CalendarDateRange Start="5/1/2012" End="5/5/2012"/>
  </Calendar.BlackoutDates>
</Calendar>
```

Это работает только в режиме `Month`.



***Рис. 11.26.** Так выглядит календарь, когда в коллекции BlackoutDates находятся два диапазона CalendarDateRanges*

СОВЕТ

Типом свойства `BlackoutDates` **является класс** `CalendarBlackoutDatesCollection`**, производный от** `ObservableCollection<CalendarDateRange>`**. В нем есть один особенно полезный метод –** `AddDatesInPast`**. Обратившись к нему, можно запретить все даты раньше текущей. Но поскольку вызвать его можно только в процедурном коде, иногда проще явно указать элемент** `CalendarDateRange`**, задав в нем** `DateTime.MinValue` **(1 января 0001 года) в качестве значения** `Start` **и** `DateTime.Today` **минус один день – в качестве значения** `End`**.**

Свойство `FirstDayOfWeek` **класса** `Calendar` **предназначено для культур, в которых первым днем недели считается воскресенье, но, в принципе, в него можно записать любое значение из перечисления** `System.DayOfWeek`**, и тогда отображение соответственно изменится. В классе** `Calendar` **имеются также события, отражающие все существенные изменения:** `DisplayDateChanged`**,** `DisplayModeChanged`**,** `SelectionModeChanged` **и** `SelectedDatesChanged` **(возникает при выделении как одной, так и нескольких дат).**

Элемент DatePicker

Еще один календарный элемент – `DatePicker` **– по существу, представляет собой поле** `TextBox` **для отображения и ввода даты, с которым ассоциирован всплывающий элемент** `Calendar`**, позволяющий изменять дату визуально. Внешний вид элемента** `DatePicker` **изображен на рис. 11.27.**



***Рис. 11.27.** Элемент WPF DatePicker вместе с ассоциированным всплывающим календарем*

При щелчке по значку календаря появляется уже знакомый нам элемент `Calendar`**, которому** `DatePicker` **и обязан большинством своих интересных возможностей.** `DatePicker` **обладает теми же свойствами и событиями, что и** `Calendar`**, за исключением свойств** `DisplayMode`**,** `SelectionMode` **и соответствующих им событий изменения. Для всплывающего календаря всегда установлены режимы** `DisplayMode=Month` **и** `SelectionMode=SingleDate`**. Поскольку выделить можно только одну дату, то вместо события** `SelectedDatesChanged` **в классе** `DatePicker`

определено событие SelectedDateChanged. По какой-то непонятной причине у DatePicker нет также события DisplayDateChanged, присутствующего в Calendar.

В классе DatePicker имеется также несколько уникальных свойств и событий для управления поведением TextBox и взаимодействия со всплывающим календарем. Булевское свойство IsDropDownOpen позволяет открывать и закрывать всплывающий календарь из программы, а также опрашивать его текущее состояние. События CalendarOpened и CalendarClosed генерируются, когда календарь соответственно открывается или закрывается. Свойство SelectedDateFormat определяет формат строки, помещаемой в TextBox после выбора даты в календаре. По умолчанию оно равно Short, что соответствует формату 4/20/2012. Можно задать значение Long, что даст представление в виде Friday, April 20, 2012. Строку, отображаемую в поле TextBox, можно установить или получить с помощью свойства Text. Если введена строка, которую невозможно интерпретировать как дату, то генерируется событие DateValidationError.

Поле ввода в элементе DatePicker (типа класса DatePickerTextBox, производного от TextBox) нельзя назвать образцом изящества – оно странно выглядит при наведении указателя мыши, а на значке календаря по какой-то необъяснимой прихоти разработчиков всегда отображается число «15». Единственный способ изменить его внешний вид – полностью заменить шаблон.

Резюме

Итак, мы ознакомились с основными встроенными элементами управления, применяемыми для создания традиционных (и не очень традиционных) пользовательских интерфейсов. Их *внешний вид* можно радикально изменить с помощью приемов, описанных в главе 14, но описанное выше *поведение* останется неизменным.

IV

Средства для профессиональных разработчиков

Глава 12 «Ресурсы»

Глава 13 «Привязка к данным»

Глава 14 «Стили, шаблоны, обложки и темы»

- Двоичные ресурсы
- Логические ресурсы

12

Ресурсы

В каркас .NET Framework встроена общая инфраструктура пакетирования и доступа к ресурсам – частям приложения или компонента, отличным от кода. К ним относятся, например, растровые изображения, шрифты, аудио- и видеофайлы и таблицы строк. Как и во многих других случаях, WPF не только пользуется базовой системой ресурсов .NET, но и немного расширяет ее. WPF поддерживает два принципиально разных вида ресурсов: двоичные и логические.

Двоичные ресурсы

Первый тип – двоичные ресурсы – это в точности то, что понимается под ресурсами в остальных частях .NET Framework. В WPF-приложениях в этой роли обычно выступают традиционные ресурсы вроде растровых изображений. Но и откомпилированный XAML-код также хранится в виде двоичного ресурса. Существует три способа пакетирования двоичных ресурсов:

- Внедрить в сборку
- Оставить в виде автономных файлов, известных приложению на этапе компиляции
- Оставить в виде автономных файлов, не известных приложению на этапе компиляции

Двоичные ресурсы приложения часто относят к одной из двух категорий: локализуемые ресурсы, которые должны изменяться в зависимости от текущей культуры, и не зависящие от языка (или нелокализуемые), то есть одинаковые при любой культуре. В этом разделе мы остановимся на различных способах определения, доступа и локализации ресурсов.

Определение двоичного ресурса

Типичная процедура определения двоичного ресурса заключается в том, чтобы добавить файл в проект Visual Studio и выбрать в сетке свойства подходящее действие при построении, как показано на рис. 12.1 на примере файла изображения *logo.jpg*.

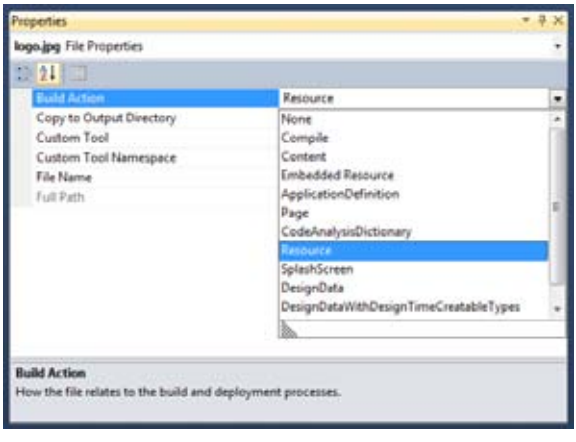


Рис. 12.1. *Пометка файла как двоичного ресурса в Visual Studio*

Visual Studio поддерживает несколько действий при построении для WPF-приложений, два из которых имеют непосредственное отношение к двоичным ресурсам:

- `Resource` – внедрить ресурс в сборку (или в соответствующую конкретной культуре сателлитную сборку).
- `Content` – оставить ресурс в виде автономного файла, но добавить в сборку специальный атрибут (`AssemblyAssociatedContentFile`), в котором говорится о существовании и относительном местоположении файла.

Если вы вручную редактируете проект для программы MSBuild, то такой файл можно добавить следующим образом:

```
<BuildAction Include="logo.jpg"/>
```

где *BuildAction* – название действия при построении. Этот элемент может содержать вложенные элементы, уточняющие поведение, например:

```
<Content Include="logo.jpg">
  <CopyToOutputDirectory>Always</CopyToOutputDirectory>
</Content>
```

Если вы хотите оставить ресурсы в виде автономных файлов, то добавлять их в проект, указывая действие при построении `Content`, необязательно; можно просто поместить их в известное на этапе выполнения место и не добавлять в проект вовсе. Но так поступать не рекомендуется, потому что доступ к ре-

сурсу оказывается менее естественным (см. следующий раздел). Тем не менее иногда невозможно избежать использования ресурсов, неизвестных на этапе компиляции, например файлов, которые динамически генерируются во время работы программы.

ПРЕДУПРЕЖДЕНИЕ

Избегайте действия при построении Embedded Resource!

Действие при построении `Resource` часто путают с похожим действием `EmbeddedResource` (в сетке свойств Visual Studio оно называется `Embedded Resource`). И то и другое приводит к внедрению двоичного ресурса в сборку, но в WPF-проектах второе использовать не следует. Действие `Resource` было добавлено специально для WPF, а `EmbeddedResource` появилось раньше WPF (и применяется для внедрения двоичных ресурсов в проектах Windows Forms).

Классы WPF, ссылающиеся на ресурсы по их унифицированным идентификаторам (см. следующий раздел), предназначены только для работы с ресурсами, для которых было указано действие при построении `Content` или `Resource`. Это также означает, что на ресурсы, внедренные с помощью действия `Content` или `Resource`, можно легко ссылаться из XAML-разметки, тогда как для ресурсов, внедренных с помощью действия `EmbeddedResource`, это невозможно (по крайней мере, без написания дополнительного кода).

Ресурсы следует внедрять в сборку (указывая действие при построении `Resource`), если либо они локализуемые, либо вам кажется, что иметь всего один двоичный файл лучше, чем включать в дистрибутив автономный файл, пусть даже его можно заменять независимо от кода. Если ни одно из этих условий не выполнено или необходим также доступ к содержимому ресурса из внешних программ (быть может, из HTML-страниц, генерируемых приложением), то наиболее подходящим вариантом будет действие при построении `Content`.

Доступ к двоичным ресурсам

Неважно, внедрены ли ресурсы с помощью действия при построении `Resource`, оставлены в виде автономных файлов, связанных с приложением за счет действия при построении `Content`, или оставлены в виде автономных файлов без какой-либо специальной обработки на этапе компиляции, WPF предоставляет механизм для доступа к ним как из кода, так и из XAML, – по унифицированному идентификатору ресурса (URI). Имеется конвертер типа, который позволяет задавать URI в XAML-разметке в виде простой строки, с несколькими встроенными упрощениями для наиболее распространенных случаев.

Это можно видеть на примере исходного кода приложения Photo Gallery из главы 7 «Структурирование и развертывание приложения». В следующем взятом оттуда фрагменте XAML имеются ссылки на несколько изображений, включенных в проект с помощью действия при построении `Resource`:

```
<StackPanel Grid.Column="1" Orientation="Horizontal" HorizontalAlignment="Center">
  <Button x:Name="previousButton" ToolTip="Previous (Left Arrow)" ...>
    <Image Height="21" Source="previous.gif"/>
  </Button>
  <Button x:Name="slideshowButton" ToolTip="Play Slide Show (F11)" ...>
    <Image Height="21" Source="slideshow.gif"/>
  </Button>
  <Button x:Name="nextButton" ToolTip="Next (Right Arrow)" ...>
    <Image Height="21" Source="next.gif"/>
  </Button>
</StackPanel>
```

Отметим, что тот же самый XAML-код будет работать и в случае, когда все GIF-файлы включены в проект с действием при построении `Content`, а не `Resource` (при условии, что автономные файлы скопированы в один каталог с исполняемым файлом). Но если автономные GIF-файлы не включены в проект, то эта разметка работать не будет.

ПРЕДУПРЕЖДЕНИЕ

Откомпилированный XAML–код не может ссылаться на двоичный ресурс в текущем каталоге по имени файла без указания каталога, если этот файл не был включен в проект!

Часто удивляются, что откомпилированный XAML-код, в отличие от автономного, не может следующим образом ссылаться на произвольный файл в текущем каталоге:

```
<Image Height="21" Source="slideshow.gif"/>
```

Если вам необходимо, чтобы ресурс остался автономным, и включать его в проект вы не хотите, то есть несколько альтернативных решений. Одно (плохое) состоит в том, чтобы указать полный путь к файлу:

```
<Image Height="21" Source="C:\Users\Adam\Documents\slideshow.gif"/>
```

Более приемлемая альтернатива – воспользоваться довольно странным синтаксисом, который мы опишем ниже, в разделе «Доступ к ресурсам в первоисточнике»:

```
<Image Height="21" Source="pack://siteOfOrigin:,,,/slideshow.gif"/>
```

Чтобы разобраться в механизме доступа к двоичным ресурсам, неважно, идет ли речь об элементе `Image` или каком-то другом, нужно понимать, как устроен URI, адресующий внедренный или автономный ресурс. В табл. 12.1 перечислены основные форматы URI в XAML-разметке. Отметим, что не все они доступны приложениям с частичным доверием.

Отметим, что первые два варианта в табл. 12.1 годятся как для внедренных, так и для автономных двоичных ресурсов. Это означает, что автономный ресурс можно заменить внедренным (или наоборот), *не внося никаких изменений в XAML-разметку*.

Таблица 12.1. URI для доступа к двоичным ресурсам из XAML-разметки (ресурс называется logo.jpg)

Если URI имеет вид...	То ресурс...
`logo.jpg`	Внедрен в текущую сборку или является автономным и находится в той же папке, что и текущая XAML-страница либо сборка (последнее верно, только если для ресурса в проекте было указано действие при построении `Content`)
`A/B/logo.jpg`	Внедрен в текущую сборку с использованием внутренней структуры подпапок (A\B), определенной на этапе компиляции, или является автономным и находится в подпапке A\B относительно текущей XAML-страницы либо сборки (последнее верно, только если для ресурса в проекте было указано действие при построении `Content`)
`c:\temp\logo.jpg`	Автономный и находится в локальной папке *c:\temp*
`file://c:/temp/logo.jpg`	Автономный и находится в локальной папке *c:\temp*
`\\pc1\images\logo.jpg`	Автономный и находится в общей папке *\\pc1\images*
`http://adamnathan.net/logo.jpg`	Автономный и находится на веб-сайте *adamnathan.net*
`/MyDll;Component/logo.jpg`	Внедрен в другую сборку с именем *MyDll.dll* или *MyDll.exe*
`/MyDll;Component/A/B/logo.jpg`	Внедрен в другую сборку с именем *MyDll.dll* или *MyDll.exe* с использованием внутренней структуры подпапок (A\B), определенной на этапе компиляции
`pack://siteOfOrigin:,,,/logo.jpg`	Автономный в первоисточнике
`pack://siteOfOrigin:,,,/A/B/logo.jpg`	Автономный в первоисточнике в подпапке A\B

FAQ

Что происходит при попытке доступа к ресурсу по медленной или недоступной сети?

В табл. 12.1 сказано, что двоичные ресурсы могут находиться в потенциально ненадежных источниках, таких как веб-сайт или общая папка. При этом доступ производится синхронно, то есть вы будете наблюдать, как приложение «зависает» на время, необходимое для скачивания всего ресурса до последнего бита. Ко всему прочему, ошибка при доступе к ресурсу возбуждает неперехватываемое исключение.

Идея об использовании подпапок в контексте внедренных ресурсов может показаться странной, но на самом деле это удобный способ организовать внедренные ресурсы так же, как вы обычно организуете автономные файлы. Пусть, например, файл *logo.jpg* находился в папке *images* в проекте Visual Studio; в файле проекта это описывается строкой

```
<Resource Include="images\logo.jpg"/>
```

или

```
<Content Include="images\logo.jpg"/>
```

Тогда вне зависимости от того, чем является *logo.jpg* во время выполнения – автономным файлом в подпапке *images* или внедренным в сборку ресурсом, – обращаться к нему можно следующим образом:

```
<Image Source="images\logo.jpg"/>
```

Последние четыре строки в табл. 12.1 нуждаются в дополнительных пояснениях. Первые два варианта позволяют получать доступ к двоичным ресурсам, внедренным в другую сборку, а последние два – к ресурсам, находящимся в так называемом первоисточнике (site of origin).

Доступ к ресурсам, внедренным в другую сборку

Возможность легко получать доступ к двоичным ресурсам, внедренным в другую сборку, очень удобна (и дает большую свободу при обновлении ресурсов, не заставляя заменять основной исполняемый файл), но синтаксис выглядит странновато. Как видно из табл. 12.1, URI записывается в виде

/AssemblyReference;Component/*ResourceName*

где *AssemblyReference* идентифицирует конкретную сборку, а Component – ключевое слово, которое нельзя изменять. *ResourceName* – это имя файла (может включать подпапки).

AssemblyReference может быть простым отображаемым именем сборки или включать необязательные части идентификатора сборки .NET: номер версии и маркер открытого ключа (если это сборка со строгим именем). Таким образом, имеется четыре варианта записи *AssemblyReference*:

- *AssemblyName*
- *AssemblyName*;v*VersionNumber* (префикс v обязателен)
- *AssemblyName*;*PublicKeyToken*
- *AssemblyName*;v*VersionNumber*;*PublicKeyToken*

Доступ к ресурсам в первоисточнике

Приложения с полным доверием могут содержать жестко зашитый унифицированный указатель ресурса (URL) или путь к файлу для автономных двоичных ресурсов, но с точки зрения сопровождения разумнее воспользоваться понятием первоисточника. (К тому же для приложений с частичным доверием альтернативы просто нет.) Во время выполнения первоисточнику могут

быть сопоставлены различные физические места в зависимости от способа развертывания приложения:

- Для приложения с полным доверием, установленного с помощью установщика Windows, первоисточником будет корневая папка приложения.
- Для ClickOnce-приложения с полным доверием первоисточником будет URL или UNC-путь, из которого было развернуто приложение.
- Для XBAP- или ClickOnce-приложения с частичным доверием первоисточником будет URL или UNC-путь к месту, где находится приложение.
- Для автономных XAML-страниц, просматриваемых в браузере, первоисточника нет. При попытке воспользоваться ресурсом возникает исключение.

Формат записи первоисточника еще более чудной, чем при доступе к ресурсу, внедренному в другую сборку! Необходимо указывать префикс `pack://siteOfOrigin:,,,/`, за которым следует имя ресурса (возможно, содержащее подпапки). Отметим, что `siteOfOrigin` – ключевое слово, а не строка, замещаемая другим текстом, так что записывать его нужно без изменений.

FAQ

Откуда взялся этот ужасный синтаксис с тремя запятыми?

Формат URI со схемой `pack` определен в спецификации XML Paper Specification (XPS), которую можно найти по адресу *http://microsoft.com/whdc/xps/xpsspec.mspx*. Вот как он описан:

```
pack://packageURI/partPath
```

Здесь *packageURI* – это фактически один URI внутри другого, поэтому он кодируется путем преобразования символов косой черты в запятые. *packageURI* может указывать на XPS-документ, например `file:///C:/Document.xps`, а в закодированном представлении получится `file:,,,C:,Document.xps`. Что же касается WPF-программ, то это может быть один из двух URI, которые на этой платформе имеют особый смысл:

- `siteOfOrigin:///` (кодируется в виде `siteOfOrigin:,,,`)
- `application:///` (кодируется в виде `application:,,,`)

Следовательно, три запятых – это всего лишь три закодированных косых черты, а не место для подстановки необязательных параметров! (Отметим, что можно было бы задать две косых черты/запятых вместо трех.)

Пакет `application:///` неявно используется во всех ссылках на ресурсы, приведенных в табл. 12.1, за исключением тех, где встречается `siteOfOrigin`. (Объясняется это тем, что участвующие в механизме объекты классов WPF реализуют интерфейс `IUriContext`. В этом интерфейсе определено всего одно свойство `BaseUri`, задающее контекст для относительных URI.) Иными словами, следующий URI, задаваемый в XAML-разметке:

```
logo.jpg
```

на самом деле представляет собой сокращенную запись для

```
pack://application:,,,/logo.jpg
```

а такой URI:

```
/MyDll;Component/logo.jpg
```

будет сокращенной записью для

```
pack://application:,,,/MyDll;Component/logo.jpg
```

При желании можно использовать в XAML и такую более длинную и более явную запись URI, но разумных причин для этого не видно.

Доступ к ресурсам из процедурного кода

При создании URI ресурсов в программе на языке C# не разрешается использовать применяемые в XAML сокращенные формы записи, показанные в табл. 12.1. Вместо этого необходимо конструировать URI из полного URI со схемой pack или из абсолютного пути либо URL.

Например, в следующем фрагменте свойству `Source` объекта `Image` присваивается содержимое файла *logo.jpg*:

```
Image image = new Image();
image.Source = new BitmapImage(new Uri("pack://application:,,,/logo.jpg"));
```

В результате создается объект типа `System.Windows.Media.Imaging.BitmapImage` (этот механизм работает для таких популярных графических форматов, как JPEG, PNG, GIF и BMP), являющегося косвенным потомком абстрактного класса `ImageSource` (это тип свойства `Source`). Сам URI представлен объектом типа `System.Uri`.

Конструкция `pack://application:,,,/` работает только для ресурсов, принадлежащих текущему проекту, для которых указано действие при построении `Resource` или `Content`. Чтобы сослаться на автономные файлы, не имеющие отношения к проекту, по относительному имени, проще всего воспользоваться URI на базе `siteOfOrigin`.

Локализация двоичных ресурсов

Если приложение содержит двоичные ресурсы, относящиеся к определенным культурам, то их можно поместить в сателлитные сборки (по одной на каждую культуру), которые будут автоматически загружаться при необходимости. Если вы именно так и делаете, то, скорее всего, в пользовательском интерфейсе вашего приложения есть еще и строки, нуждающиеся в локализации. Инструментальное средство LocBaml, входящее в состав Windows SDK, позволяет выполнять локализацию строк и других частей приложения, не выдергивая их из XAML и не применяя вручную механизмы косвенного доступа. В этом разделе мы рассмотрим основы работы с LocBaml и сателлитными сборками.

Подготовка проекта для нескольких культур

Чтобы задать подразумеваемую по умолчанию культуру для ресурсов и автоматически построить подходящую сателлитную сборку, необходимо добавить в файл проекта элемент `UICulture`. В Visual Studio нет средств, позволяющих сделать это непосредственно в интегрированной среде, поэтому откройте файл в редакторе и внесите изменения вручную.

СОВЕТ

Можно открыть файл проекта и не покидая Visual Studio. Но для этого нужно сначала выгрузить его из текущего решения (щелкнув по проекту правой кнопкой мыши и выбрав нужный пункт из контекстного меню). Сделав это, снова щелкните по проекту правой кнопкой мыши и выберите из контекстного меню пункт Edit (Правка).

Элемент `UICulture` нужно помещать внутрь некоторых или всех элементов `PropertyGroup`, соответствующих тем конфигурациям построения, которые вас интересуют (Debug, Release и т. д.), или же в группу свойств, вообще не связанную с конфигурацией построения, – тогда элемент автоматически будет относиться ко всем конфигурациям. Выглядеть это должно следующим образом (в данном случае культурой по умолчанию объявлен американский диалект английского языка):

```
<Project ...>
  <PropertyGroup>
    <UICulture>en-US</UICulture>
...
```

Перестроив проект с такой настройкой, вы обнаружите на одном уровне со своей сборкой папку `en-US`, в которой будет находиться сателлитная сборка с именем *AssemblyName*`.resources.dll`.

Следует также пометить свою сборку атрибутом `NeutralResourcesLanguage` уровня сборки, значение которого совпадает с выбранной культурой по умолчанию:

```
[assembly: NeutralResourcesLanguage("en-US",
  UltimateResourceFallbackLocation.Satellite)]
```

Пометка пользовательского интерфейса идентификаторами локализации

Следующий шаг – применение директивы `Uid` из пространства имен языка XAML (`x:Uid`) ко всем объектным элементам, нуждающимся в локализации. Значением каждой такой директивы должен быть уникальный идентификатор. Делать это вручную очень утомительно, но, к счастью, есть и автоматический способ – вызвать MSBuild из командной строки следующим образом:

```
msbuild /t:updateuid ProjectName.csproj
```

В результате *каждый* объектный элемент в *каждом* XAML-файле проекта получит директиву `x:Uid` с уникальным значением. Можете добавить эту задачу в проект перед задачей Build, хотя в этом случае она будет выполняться при каждом построении, что совершенно ни к чему.

Создание новой сателлитной сборки с помощью LocBaml

Откомпилировав проект, в который добавлены `Uid`, можно запустить программу LocBaml из Windows SDK для файла *.resources*, сгенерированного в процессе построения (он находится в каталоге *obj\debug*):

```
LocBaml /parse ProjectName.g.en-US.resources /out:en-US.csv
```

В результате создается простой текстовый CSV-файл, содержащий все значения свойств, подлежащие локализации. Этот файл можно отредактировать так, чтобы он соответствовал новой культуре (никаких хитростей в этой части локализации нет и в помине!). Сохраните файл и запустите LocBaml в обратном направлении, чтобы сгенерировать сателлитную сборку по CSV-файлу! Например, если в CSV-файле находится перевод строк на канадский диалект французского языка, то можно сохранить его с именем *fr-CA.csv*, а затем запустить LocBaml следующим образом:

```
LocBaml /generate ProjectName.resources.dll /trans:fr-CA.csv /cul:fr-CA
```

Новую сателлитную сборку следует скопировать в ту же папку, где находится основная сборка, присвоив ей имя, соответствующее культуре (в данном случае `fr-CA`).

Чтобы протестировать приложение для другой культуры, присвойте свойству `System.Threading.Thread.CurrentThread.CurrentUICulture` (и `System.Threading.Thread.CurrentThread.CurrentCulture`) ссылку на нужный экземпляр класса `CultureInfo`.

Логические ресурсы

Второй тип ресурсов был впервые введен в WPF и поддерживается как WPF, так и Silverlight. В этой главе ресурсы такого типа называются *логическими* за неимением более подходящего термина, хотя в большинстве книг их называют просто *ресурсами*, в отличие от рассмотренных выше *двоичных ресурсов*. (Может возникнуть искушение использовать термин *XAML-ресурсы*, но, как почти все в XAML, их можно создавать и использовать также и в процедурном коде.)

Логические ресурсы представляют собой произвольные объекты .NET, хранящиеся в свойстве элемента `Resources`. Обычно предполагается, что таким ресурсом смогут сообща пользоваться все потомки данного элемента. Свойство `Resources` (типа `System.Windows.ResourceDictionary`) определено в базовых классах `FrameworkElement` и `FrameworkContentElement`, а это означает, что оно есть в большинстве классов WPF. В качестве логических ресурсов часто выступают стили (см. главу 14 «Стили, шаблоны, обложки и темы») или поставщики данных (см. главу 13 «Привязка к данным»). Но в этой главе мы будем хранить в логическом ресурсе простую кисть `Brush`.

В листинге 12.1 показано простое окно Window, в нижней строке которого расположен ряд кнопок – как в пользовательском интерфейсе программы Photo Gallery. В разметке используется прямолинейный способ применения кисти Brush к свойствам Background и BorderBrush каждой кнопки Button (и всего окна Window). Результат изображен на рис. 12.2.



Рис. 12.2. *Результат визуализации разметки из листинга 12.1*

Листинг 12.1. Применение цветной кисти без использования логических ресурсов

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="Simple Window" Background="Yellow">
  <DockPanel>
    <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
      HorizontalAlignment="Center">
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="zoom.gif"/>
      </Button>
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="defaultThumbnailSize.gif"/>
      </Button>
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="previous.gif"/>
      </Button>
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="slideshow.gif"/>
      </Button>
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="next.gif"/>
      </Button>
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="counterclockwise.gif"/>
      </Button>
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="clockwise.gif"/>
      </Button>
      <Button Background="Yellow" BorderBrush="Red" Margin="5">
        <Image Height="21" Source="delete.gif"/>
     </Button>
    </StackPanel>
    <ListBox/>
```

```
    </DockPanel>
  </Window>
```

Вместо этого можно было бы сделать желтую и красную кисти логическими ресурсами элемента `Window` и ссылаться на них из разных элементов. Это правильный способ отделить и собрать в одном месте всю информацию о стилях. Во многом он напоминает применение каскадных таблиц стилей (CSS) для управления всеми цветами и стилями в пределах веб-страницы, не задавая их для каждого элемента в отдельности. Механизм обобществления объектов, реализованный в схеме логических ресурсов, поможет заодно значительно сэкономить память, причем экономия будет тем больше, чем объекты сложнее. В листинге 12.2 предыдущая разметка переработана с использованием логических ресурсов для обеих кистей.

Листинг 12.2. Хранение логических кистей в одном месте с помощью логических ресурсов

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Simple Window">
<Window.Resources>
  <SolidColorBrush x:Key="backgroundBrush">Yellow</SolidColorBrush>
  <SolidColorBrush x:Key="borderBrush">Red</SolidColorBrush>
</Window.Resources>
<Window.Background>
  <StaticResource ResourceKey="backgroundBrush"/>
</Window.Background>
  <DockPanel>
    <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
      HorizontalAlignment="Center">
      <Button Background="{StaticResource backgroundBrush}"
        BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="zoom.gif"/>
      </Button>
      <Button Background="{StaticResource backgroundBrush}"
        BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="defaultThumbnailSize.gif"/>
      </Button>
      <Button Background="{StaticResource backgroundBrush}"
        BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="previous.gif"/>
      </Button>
      <Button Background="{StaticResource backgroundBrush}"
        BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="slideshow.gif"/>
      </Button>
      <Button Background="{StaticResource backgroundBrush}"
        BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="next.gif"/>
      </Button>
      <Button Background="{StaticResource backgroundBrush}"
```

```
        BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="counterclockwise.gif"/>
      </Button>
      <Button Background="{StaticResource backgroundBrush}"
        BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="clockwise.gif"/>
      </Button>
      <Button Background="{StaticResource backgroundBrush}"
         BorderBrush="{StaticResource borderBrush}" Margin="5">
        <Image Height="21" Source="delete.gif"/>
      </Button>
    </StackPanel>
    <ListBox/>
  </DockPanel>
</Window>
```

Способ определения ресурсов и синтаксис `x:Key` уже встречались нам при рассмотрении словаря `ResourceDictionary` в главе 2 «Все тайны XAML». Для применения ресурса к элементам мы используем расширение разметки `StaticResource` (сокращенная запись `System.Windows.StaticResourceExtension`). К свойству `Window.Background` она применяется как элемент свойства, а к свойствам `Button.Background` и `Button.BorderBrush` – как атрибут свойства. Поскольку оба ресурса в этом примере – кисти `Brush`, то и применять их допустимо всюду, где может встречаться кисть.

Поскольку в листинге 12.2 по-прежнему используются простые кисти желтого и красного цветов, то и результат визуализации ничем не отличается от показанного на рис. 12.2. Зато теперь нам достаточно заменить кисть в одном месте, не трогая больше ничего в XAML-файле (при условии, что ключи в словаре ресурсов останутся теми же самыми). Например, если заменить ресурс `backgroundBrush` такой линейно-градиентной кистью:

```
<LinearGradientBrush x:Key="backgroundBrush" StartPoint="0,0" EndPoint="1,1">
  <GradientStop Color="Blue" Offset="0"/>
  <GradientStop Color="White" Offset="0.5"/>
  <GradientStop Color="Red" Offset="1"/>
</LinearGradientBrush>
```

то получится результат, изображенный на рис. 12.3.

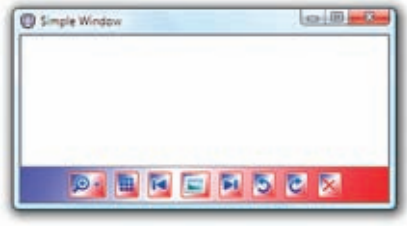


Рис. 12.3. *То же самое окно, что в листинге 12.2, но ресурс backgroundBrush определен по-другому*

Поиск ресурса

Расширение разметки `StaticResource` принимает единственный параметр – ключ объекта в словаре ресурсов. Однако этот объект не обязан находиться в словаре ресурсов текущего элемента. Он может быть в словаре ресурсов любого логического родителя или даже приложения либо системы в целом.

В классе этого расширения разметки реализована возможность обхода логического дерева для поиска нужного объекта. Сначала проверяется коллекция `Resources` текущего элемента (его словарь ресурсов). Если объект не найден, проверяется родительский элемент и т. д., пока не дойдем до корневого элемента. В этот момент проверяется коллекция `Resources` объекта `Application`. Если искомое не найдено и здесь, то проверяется коллекция *ресурсов темы* (см. главу 14). Если объекта нет и там, то на последнем шаге проверяется системная коллекция (в которой находятся системные шрифты, цвета и другие настройки). Если ресурс так и не удалось найти, возбуждается исключение `InvalidOperationException`.

Принимая во внимание описанное поведение, ресурсы обычно хранят в словаре ресурсов корневого элемента или в словаре уровня приложения, чтобы обеспечить максимально широкое обобществление. Отметим, что, хотя в пределах одного словаря ключи ресурсов должны быть уникальны, в разных коллекциях могут встречаться ресурсы с одинаковыми ключами. Приоритет имеет ресурс, оказавшийся в словаре, «ближайшем» к запросившему этот ресурс элементу, – так устроен алгоритм обхода дерева.

ПРЕДУПРЕЖДЕНИЕ

Осторожнее с ресурсами уровня приложения в многопоточных приложениях!

В главе 7 мы говорили, что в WPF-приложении может быть несколько потоков пользовательского интерфейса. В таком случае каждый поток будет напрямую обращаться к ресурсам уровня приложения. Чтобы при этом не возникало ошибок, все такие ресурсы должны принадлежать классу `Freezable` и при этом быть заморожены, либо их следует пометить атрибутом `x:Shared=false`, который мы опишем ниже, в разделе «Необобществляемые ресурсы».

Статические и динамические ресурсы

WPF предлагает два способа доступа к логическому ресурсу:

- Статически с помощью расширения разметки `StaticResource` – это означает, что значение ресурса применяется только один раз (при первом обращении)
- Динамически с помощью расширения разметки `DynamicResource` – это означает, что ресурс заново применяется после каждого изменения

Расширение разметки `DynamicResource` (`System.Windows.DynamicResourceExtension`) реализует такой же обход дерева, как `StaticResource`, поэтому зачастую между

применением DynamicResource и StaticResource нет никакой разницы. Ничто в самих объявлениях ресурсов не делает одно расширение более предпочтительным, чем другое; выбор того или другого определяется тем, хотите вы, чтобы потребитель ресурса видел его изменения, или нет. На самом деле можно даже в каком-то словаре применять к одному и тому же ключу ресурса расширение StaticResource, а в другом – DynamicResource, хотя непонятно, зачем это могло бы понадобиться.

Описание различий

Основное различие между StaticResource и DynamicResource заключается в том, что последующие обновления ресурса применяются только к тем элементам, для которых используется расширение DynamicResource. Причиной обновления может быть как ваш собственный код (например, замена желтой кисти на синюю), так и изменение системных настроек пользователем.

Характеристики производительности классов StaticResource и DynamicResource различаются. С одной стороны, использование DynamicResource влечет за собой большие непроизводительные издержки, так как приходится отслеживать изменения. С другой стороны, использование DynamicResource может уменьшить время загрузки. Ресурсы, на которые есть ссылки типа StaticResource, загружаются вместе с окном Window или страницей Page, тогда как ресурсы со ссылкой типа DynamicResource не загружаются, пока к ним не обратятся.

Кроме того, DynamicResource можно использовать только для установки свойств зависимости, а StaticResource – практически повсеместно. Например, StaticResource может служить в качестве элемента, чтобы абстрагировать целые элементы управления! Следующее окно Window:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  ...
  <Image Height="21" Source="zoom.gif"/>
  ...
</Window>
```

эквивалентно такому:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Window.Resources>
    <Image x:Key="zoom" Height="21" Source="zoom.gif"/>
  </Window.Resources>
  <StackPanel>
    <StaticResource ResourceKey="zoom"/>
  </StackPanel>
</Window>
```

Использование таких элементов, как Image, в виде ресурсов – возможно, и это интересный способ вынести фрагменты XAML-кода в отдельное место, но в качестве средства обобществления объекта он не годится. У элемента Image может быть только один родитель, поскольку он наследует классу Visual (и, значит,

принимает участие в логическом и визуальном деревьях), поэтому любая попытка использовать один и тот же объект в качестве ресурса несколько раз обречена на провал. Например, если вставить другой, но идентичный первому элемент `StaticResource` в предыдущий фрагмент XAML-кода, то мы получим исключение с сообщением `"Specified Visual is already a child of another Visual or the root of a CompositionTarget"` (Указанный элемент Visual уже является дочерним по отношению к другому элементу Visual или корневому элементу CompositionTarget).

КОПНЕМ ГЛУБЖЕ

Вынесение XAML-кода

Механизм ресурсов дает удобный способ вынести фрагменты XAML-кода в одно место на странице, а если сохранить их в виде ресурсов уровня приложения – то даже и в отдельный XAML-файл. Но если вы хотите разнести набор ресурсов по нескольким XAML-файлам вне зависимости от того, в каком месте логического дерева они хранятся (быть может, ради удобства сопровождения или повышения гибкости), то придется воспользоваться свойством `MergedDictionaries` класса `ResourceDictionary`.

Например, окно `Window` могло бы следующим образом организовать свою коллекцию `Resources`, объединив словари ресурсов, хранящиеся в разных файлах:

```
<Window.Resources>
  <ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
  <ResourceDictionary Source="file1.xaml"/>
  <ResourceDictionary Source="file2.xaml"/>
  </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>
```

В отдельных файлах элемент `ResourceDictionary` должен быть корневым. Например, файл *file1.xaml* мог бы выглядеть так:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Image x:Key="logo" Source="logo.jpg"/>
</ResourceDictionary>
```

Если в объединяемых словарях встречаются одинаковые ключи, то предпочтение отдается последнему (в отличие от случая, когда одинаковые ключи встречаются в одном словаре).

Помимо такого решения, другим способом разнести XAML-код по разным файлам является создание нестандартных элементов управления (см. главу 20 «Пользовательские и нестандартные элементы управления»). Универсального механизма включения типа директивы `#include`, обрабатываемой препроцессорами языков C и C++, в XAML нет.

Существует еще одно тонкое различие между статическим и динамическим доступом к ресурсам. Расширение разметки StaticResource в XAML не поддерживает опережающие ссылки. Иными словами, всякому использованию ресурса должно *предшествовать* его объявление в XAML-файле. Это означает, что StaticResource нельзя использовать как атрибут свойства, если ресурс определен в том же самом элементе (потому что в этом случае ссылка на ресурс неизбежно оказывается раньше его определения)! У расширения DynamicResource такого ограничения нет.

Именно из-за этого правила опережающей ссылки в окне Window в листинге 12.2 для задания Background применен синтаксис элемента свойства. Тем самым гарантируется, что ресурс определен раньше, чем используется.

DynamicResource можно было бы использовать точно так же, но можно воспользоваться и синтаксисом атрибута свойства, потому что в этом случае безразлично, находится ссылка на ресурс до его определения или после:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Simple Window" Background="{DynamicResource backgroundBrush}">
<Window.Resources>
  <SolidColorBrush x:Key="backgroundBrush">Yellow</SolidColorBrush>
  <SolidColorBrush x:Key="borderBrush">Red</SolidColorBrush>
</Window.Resources>
  ...
</Window>
```

Необобществляемые ресурсы

По умолчанию, когда ресурс применяется в нескольких местах, всюду используется один и тот же объект. Обычно это желательное поведение. Но можно пометить некоторые объекты в откомпилированном словаре ресурсов ключевым словом x:Shared="False", и тогда при каждом запросе будет генерироваться новый объект, который можно модифицировать независимо от остальных.

Такое поведение может быть полезно в предыдущем примере, где мы использовали в качестве ресурса целиком объект Image (или объект любого другого класса, производного от Visual). Подобный ресурс можно применить в дереве элементов только один раз, потому что у него не может быть больше одного родителя. Однако задание ключевого слова x:Shared="False" изменяет поведение, создавая возможность применять ресурс многократно в виде независимых объектов. Делается это следующим образом:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<Window.Resources>
  <Image x:Shared="False" x:Key="zoom" Height="21" Source="zoom.gif"/>
</Window.Resources>
  <StackPanel>
    <!-- Теперь многократное применение ресурса работает! -->
```

```
      <StaticResource ResourceKey="zoom"/>
      <StaticResource ResourceKey="zoom"/>
      <StaticResource ResourceKey="zoom"/>
    </StackPanel>
  </Window>
```

Отметим, что атрибут x:Shared можно использовать только в откомпилированном XAML-файле. В автономных XAML-файлах эта возможность не поддерживается.

Определение и применение ресурсов в процедурном коде

До сих пор мы говорили о том, как определять и применять логические ресурсы в XAML-коде, но еще не рассмотрели, как то же самое делается в процедурном коде. *Определить* ресурс в коде очень просто. В предположении, что элемент Window называется window, два ресурса типа SolidColorBrush, встречавшиеся в листинге 12.2, можно определить в коде на C# следующим образом:

```
window.Resources.Add("backgroundBrush", new SolidColorBrush(Colors.Yellow));
window.Resources.Add("borderBrush", new SolidColorBrush(Colors.Red));
```

Но вот *применение* ресурсов в процедурном коде – совсем другое дело. Поскольку StaticResource и DynamicResource – расширения разметки, то эквивалентный код поиска и применения ресурса на C# не вполне очевиден.

Чтобы получить поведение, эквивалентное StaticResource, необходимо записать в свойство элемента результат вызова метода FindResource (унаследованного от класса FrameworkElement или FrameworkContentElement).

Таким образом, следующее объявление элемента Button (похожее на то, что встречается в листинге 12.2):

```
<Button Background="{StaticResource backgroundBrush}"
        BorderBrush="{StaticResource borderBrush}"/>
```

эквивалентно такому коду на C# (в предположении, что панели StackPanel, объемлющей этот элемент Button, соответствует переменная stackPanel):

```
Button button = new Button();
// Прежде чем искать ресурсы, необходимо сделать Button потомком Window:
stackPanel.Children.Add(button);
button.Background = (Brush)button.FindResource("backgroundBrush");
button.BorderBrush = (Brush)button.FindResource("borderBrush");
```

Метод FindResource возбуждает исключение, если не удается найти ресурс, но можно вместо него использовать метод TryFindResource, который в этом случае просто вернет null.

В случае DynamicResource обращение к методу элемента SetResourceReference (также унаследованному от FrameworkElement или FrameworkContentElement) организует обновляемую привязку к свойству зависимости.

Таким образом, замена обеих ссылок вида StaticResource на ссылки вида DynamicResource:

```
<Button Background="{DynamicResource backgroundBrush}"
        BorderBrush="{DynamicResource borderBrush}"/>
```

эквивалентна такому коду на C#:

```
Button button = new Button();
button.SetResourceReference(Button.BackgroundProperty, "backgroundBrush");
button.SetResourceReference(Button.BorderBrushProperty, "borderBrush");
```

Этот способ работает при условии, что элемент Button в какой-то момент добавлен в дерево элементов как потомок окна Window (где и определены ресурсы). Но, в отличие от StaticResource, добавление в дерево необязательно должно предшествовать ссылке на ресурс.

Правило опережающих ссылок для StaticResource остается справедливым и в процедурном коде. Обращение к методу FindResource или TryFindResource завершится неудачно, если оно произведено раньше добавления ресурса в соответствующий словарь с требуемым ключом. С другой стороны, метод SetResourceReference можно вызывать и до того, как ресурс добавлен.

КОПНЕМ ГЛУБЖЕ

Прямой доступ к ресурсам

Поскольку словари ресурсов – всего лишь коллекции, доступные как открытые свойства, то ничто не мешает программе обратиться к хранящимся в словаре ресурсов объектам напрямую. Так, свойства Background и BorderBrush кнопки можно было бы установить следующим образом (в предположении, что объект Window называется window):

```
Button button = new Button();
button.Background = (Brush)window.Resources["backgroundBrush"];
button.BorderBrush = (Brush)window.Resources["borderBrush"];
```

Это аналогично использованию расширения разметки StaticResource в XAML (или метода FindResource в процедурном коде) в том смысле, что значение свойства устанавливается только один раз. Однако при этом не производится обход логического дерева, а также поиск поименованного ресурса в словаре уровня приложения или системы. Поэтому такой подход обладает меньшей гибкостью и делает связь между XAML и застраничным кодом более хрупкой, зато дает незначительное повышение производительности за счет отказа от поиска. Никакого способа применить подобную технику в XAML не существует.

Взаимодействие с системными ресурсами

Один из случаев, где напрашивается использование расширения разметки DynamicResource, – доступ к системным настройкам, инкапсулированным в статические свойства трех классов в пространстве имен System.Windows: SystemColors, SystemFonts и SystemParameters. Дело в том, что пользователь может в любой момент изменить эти настройки на Панели управления, даже когда ваше приложение работает.

В классах SystemColors, SystemFonts и SystemParameters свойства определены парами – одно содержит фактическое значение, а другое играет роль ключа ресурса, по которому производится поиск. По принятому соглашению имена свойств-ключей имеют суффикс Key. Например, в классе SystemColors имеются два свойства типа Brush – WindowBrush и WindowTextBrush, а также свойства WindowBrushKey и WindowTextBrushKey типа ResourceKey.

В табл. 12.2 показаны различные гипотетические способы задания для кнопки цвета фона, определенного в текущей системной настройке цвета окна. По моим наблюдениям, чаще всего разработчики применяют второй подход, но полностью корректным является только последний.

Таблица 12.2. Потенциально возможные способы задания цвета фона, совпадающего с определенным в системных настройках

Подход	Результат

XAML:
`<Button Background="SystemColors.WindowBrush"/>`
C#:
`button.Background = (Brush)new BrushConverter().ConvertFrom("SystemColors.WindowBrush");`
	Не работает. Класс BrushConverter этого не поддерживает

XAML:
`<Button Background="{x:Static SystemColors.WindowBrush}"/>`
C#:
`button.Background = SystemColors.WindowBrush;`
	Цвет успешно устанавливается один раз, но не изменяется, когда пользователь меняет системные настройки во время работы программы

XAML:
`<Button Background= "{StaticResource SystemColors.WindowBrushKey}"/>`
C#:
`button.Background = (Brush)FindResource("SystemColors.WindowBrushKey");`
	Не работает, если только ресурс Brush не определен с ключом "SystemColors.WindowBrushKey", не имеющим ни малейшего отношения к статическому свойству, которое вы хотели бы использовать

XAML:
`<Button Background= "{StaticResource {x:Static SystemColors.WindowBrush}}"/>`
C#:
`button.Background = (Brush)FindResource(SystemColors.WindowBrush);`
	SystemColors.WindowBrush не является допустимым ключом, поэтому при таком подходе ресурс не будет найден

Подход	Результат

XAML:
`<Button Background=`
`  "{StaticResource {x:Static SystemColors.WindowBrushKey}}"/>`
C#:
`button.Background =`
`  (Brush)FindResource(SystemColors.WindowBrushKey);`
	Ресурс будет найден. Напоминает второй подход, но, кроме того, позволяет приложению переопределить цвет (на этапе инициализации), например, для простого изменения внешнего вида

XAML:
`<Button Background=`
`  "{DynamicResource {x:Static SystemColors.WindowBrushKey}}"/>`
C#:
`button.SetResourceReference(`
`  Button.BackgroundProperty, SystemColors.WindowBrushKey);`
	Предпочтительный подход. Реагирует на любые изменения настроек, произведенные пользователем, и позволяет приложению в любой момент переопределить их

Резюме

Из всех возможностей WPF, рассматриваемых в этой книге, поддержка ресурсов относится к тем, без которых просто не обойтись. Трудно создать профессионально выглядящее приложение, в котором не было бы хотя бы одной пиктограммы и нескольких изображений!

Но механизм ресурсов позволяет добиться куда большего, чем простое улучшение внешнего вида (или звучания, если вы пользуетесь аудиоресурсами) приложения или элемента управления. Это основа локализации программ для разных языков. Кроме того, этот механизм повышает продуктивность разработки ПО, так как позволяет собрать в одном месте информацию, которую иначе пришлось бы дублировать, и даже дает возможность организовать XAML-код, разбив его на удобные для сопровождения фрагменты. Но самое интересное – и, пожалуй, самое важное – применение логических ресурсов связано с их использованием совместно с такими объектами, как стили и шаблоны (см. главу 14).

- Знакомство с объектом Binding
- Управление визуализацией
- Настройка представления коллекции
- Поставщики данных
- Дополнительные вопросы
- А теперь все вместе: клиент Twitter на чистом XAML

13

Привязка к данным

В WPF термин *данные* обычно употребляется для описания произвольного объекта .NET. Примерами такого соглашения могут служить словосочетания *привязка к данным*, *шаблоны данных* и *триггеры данных*. В роли данных может выступать объект-коллекция, XML-файл, веб-служба, таблица базы данных, объект написанного вами класса и даже элемент WPF, к примеру `Button`.

Поэтому под привязкой к данным понимается установление некоторой связи между произвольными объектами .NET. Классический пример – визуальное представление (например, в виде списка `ListBox` или сетки `DataGrid`) данных, хранящихся в XML-файле, в базе данных или в коллекции в памяти. Вместо того чтобы писать цикл обхода источника данных и вручную добавлять объекты `ListBoxItem` в `ListBox`, не проще ли было бы сказать нечто вроде: «Привет, `ListBox`! Возьми-ка элементы вон оттуда. И будь добр, следи за их актуальностью. Ах да, еще не забудь отформатировать вот так». Механизм привязки к данным позволяет все это и многое другое.

Знакомство с объектом Binding

Ключом к механизму привязки к данным является объект класса `System.Windows.Data.Binding`, который «склеивает» между собой два свойства и открывает коммуникационный канал между ними. Объект `Binding` можно один раз настроить и поручить ему дальнейшую заботу о синхронизации.

Использование объекта Binding в процедурном коде

Допустим, требуется добавить в приложение Photo Gallery, рассмотренное в предыдущих главах, элемент `TextBlock`, в котором будет отображаться имя текущей папки, расположив его над списком `ListBox`:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Background="AliceBlue" FontSize="16" />
```

Можно было бы обновлять этот текстовый блок вручную при каждом изменении свойства SelectedItem элемента TreeView:

```
void treeView_SelectedItemChanged(object sender,
  RoutedPropertyChangedEventArgs<object> e)
{
  currentFolder.Text = (treeView.SelectedItem as TreeViewItem).Header.ToString();
  Refresh();
}
```

Но, воспользовавшись объектом Binding, мы сможем избавиться от этой строки кода, заменив ее однократной инициализацией в конструкторе класса MainWindow:

```
public MainWindow()
{
  InitializeComponent();

  Binding binding = new Binding();
  // Задаем объект-источник
  binding.Source = treeView;
  // Задаем свойство-источник
  binding.Path = new PropertyPath("SelectedItem.Header");
  // Присоединяем к нему свойство-приемник
  currentFolder.SetBinding(TextBlock.TextProperty, binding);
}
```

После такого изменения свойство currentFolder.Text будет автоматически обновляться при каждом изменении свойства treeView.SelectedItem.Header. Если в дереве TreeView будет выбран элемент, не обладающий свойством Header (в приложении Photo Gallery такого быть не может), то привязка завершится неудачно (без каких-либо исключений) и вернет значение свойства, подразумеваемое по умолчанию (в данном случае пустую строку). Впрочем, есть способы получать уведомления о таком развитии события – мы рассмотрим их ниже.

Такая модификация кода вряд ли может считаться улучшением, потому что вместо одной строки нам пришлось написать аж четыре! Но ведь это очень простой случай использования привязки. В последующих примерах мы убедимся, что привязка позволяет значительно уменьшить объем кода, который пришлось бы написать вручную для достижения эквивалентного результата.

Для объекта Binding определены понятия свойства-*источника* и свойства-*приемника*. Свойство-источник (в нашем случае treeView.SelectedItem.Header) задается в два приема: запись ссылки на объект-источник в свойство Source и имени интересующего нас свойства (или цепочки, состоящей из имени свойства и его «субсвойств»), обернутого объектом PropertyPath, в свойство Path. Чтобы ассоциировать Binding со свойством-приемником (в нашем случае currentFolder.Text) Binding, нужно вызвать метод SetBinding (унаследованный от класса

FrameworkElement или FrameworkContentElement), передав ему нужное свойство зависимости и сам объект Binding.

СОВЕТ

На самом деле существует два способа настроить объект Binding в процедурном коде. Первый – вызвать метод экземпляра SetBinding от имени объекта FrameworkElement или FrameworkContentElement, как было показано выше. Второй – вызвать статический метод SetBinding класса BindingOperations. Ему передаются те же аргументы, что и методу экземпляра, плюс дополнительный первый аргумент, представляющий объект-приемник:

```
BindingOperations.SetBinding(currentFolder, TextBlock.TextProperty, binding);
```

Достоинство статического метода в том, что первый параметр имеет тип DependencyObject, то есть открывается возможность осуществлять привязку к объектам, не наследующим ни FrameworkElement, ни FrameworkContentElement (например, класса Freezable).

КОПНЕМ ГЛУБЖЕ

Удаление объекта Binding

Если вы не хотите, чтобы объект Binding существовал на протяжении всего времени работы приложения, то его можно в любой момент «отключить» с помощью статического метода BindingOperations.ClearBinding. (Правда, это делают редко.) Методу передается объект-приемник и его свойство зависимости. Например:

```
BindingOperations.ClearBinding(currentFolder, TextBlock.TextProperty);
```

Если к объекту-приемнику присоединено более одного объекта Binding, то можно очистить их все за один прием, вызвав метод BindingOperations.ClearAllBindings:

```
BindingOperations.ClearAllBindings(currentFolder);
```

Еще один способ убрать привязку – напрямую записать в свойство-приемник новое значение, например:

```
currentFolder.Text = "I am no longer receiving updates.";
```

Но такой способ очищает только односторонние привязки. (Различные типы объектов Binding обсуждаются ниже в разделе «Настройка потока данных».) Подход на основе метода ClearBinding в любом случае более гибкий, поскольку оставляет свойству зависимости возможность принимать данные из других источников с более низким приоритетом (триггеров стилей, механизма наследования значений свойств и т. д.). Напомним, что приоритеты, учитываемые при определении значения свойства, рассматривались в главе 3 «Основные принципы WPF». У объекта Binding, заданного с помощью метода SetBinding, такой же приоритет, как у локального значения, а метод ClearBinding просто исключает этот источник поставки значений точно так же, как метод ClearValue делает это для *любого* локального значения. (На самом деле реализация ClearBinding всего лишь вызывает метод ClearValue для объекта-приемника!)

Использование объекта Binding в XAML

Поскольку вызвать метод SetBinding из XAML-кода невозможно, в WPF включено расширение разметки, позволяющее использовать объект Binding декларативно. На самом деле класс Binding сам является классом расширения разметки (несмотря на то, что в его имени нет стандартного суффикса Extension).

Чтобы использовать объект Binding в XAML, нужно записать в свойство-приемник сам этот объект, а затем с помощью стандартного синтаксиса расширения разметки настроить его свойства. Следовательно, показанный выше процедурный код можно заменить таким дополнением к объявлению элемента currentFolder:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Text="{Binding ElementName=treeView, Path=SelectedItem.Header}"
  Background="AliceBlue" FontSize="16" />
```

Ну что, привязка к данным уже начинает выглядеть более привлекательной, чем кодирование вручную? Соединение между источником и приемником не только записывается лаконично, но еще и вынесено из процедурного кода.

СОВЕТ

Помимо конструктора по умолчанию в классе Binding имеется конструктор, принимающий один аргумент Path. Следовательно, можно оформить расширение разметки и по-другому, передав путь Path конструктору, а не устанавливая его явно в качестве свойства. Иными словами, приведенный выше фрагмент XAML можно было бы переписать и в таком виде:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Text="{Binding SelectedItem.Header, ElementName=treeView}"
  Background="AliceBlue" FontSize="16" />
```

Оба подхода почти идентичны, за исключением некоторых тонких различий в разрешении префиксов пространств имен в путях к свойствам. В целом, явное задание свойства Path надежнее.

Отметим, что в этом XAML-коде для установки объекта-источника применяется свойство ElementName, а не Source, как в предыдущем разделе. И то и другое допустимо в обоих контекстах, но ElementName удобнее в XAML, потому что для него нужно указать только имя элемента-источника. Впрочем, с появлением в WPF 4 расширения разметки x:Reference свойство Source можно задать следующим образом:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
  Text="{Binding Source={x:Reference TreeView}, Path=SelectedItem.Header}"
  Background="AliceBlue" FontSize="16" />
```

СОВЕТ

С помощью свойства TargetNullValue объекта Binding можно указать специальное значение, которое будет возвращаться, если значение реального свойства-источника равно null. Например, в показанном ниже текстовом блоке будет отображаться не пустая строка, а строка «Nothing is selected.» (Ничего не выбрано) в случае, когда значение-источник равно null:

```
<TextBlock Text="{Binding ... TargetNullValue=Nothing is selected.}" .../>
```

Свойство TargetNullValue может выручить и в более сложных ситуациях, когда какое-то свойство объекта-приемника не может принимать значение null.

КОПНЕМ ГЛУБЖЕ

Свойство RelativeSource объекта Binding

Еще один способ задать источник данных – воспользоваться свойством RelativeSource объекта Binding, которое ссылается на элемент, находящийся с элементом-приемником в определенных отношениях. Это свойство имеет тип RelativeSource, который представляет собой класс расширения разметки. Приведем несколько примеров использования RelativeSource.

Чтобы сделать элемент-источник равным элементу-приемнику:

```
{Binding RelativeSource={RelativeSource Self}}
```

Чтобы сделать элемент-источник равным свойству TemplatedParent элемента-приемника (это свойство обсуждается в следующей главе):

```
{Binding RelativeSource={RelativeSource TemplatedParent}}
```

Чтобы сделать элемент-источник равным ближайшему родителю элемента-приемника, имеющему заданный тип:

```
{Binding RelativeSource={RelativeSource FindAncestor,
  AncestorType={x:Type desiredType}}}
```

Чтобы сделать элемент-источник равным *n*-му ближайшему родителю элемента-приемника, имеющему заданный тип:

```
{Binding RelativeSource={RelativeSource FindAncestor,
  AncestorLevel=n, AncestorType={x:Type desiredType}}}
```

Чтобы сделать элемент-источник равным предыдущему объекту в коллекции, привязанной к данным:

```
{Binding RelativeSource={RelativeSource PreviousData}}
```

Особенно полезно свойство RelativeSource в контексте шаблонов элементов управления, которые мы будем обсуждать в следующей главе. Однако использование RelativeSource в режиме Self удобно для привязки одного свойства элемента к другому его же свойству без указания имени элемента. Интересным примером может служить элемент Slider, свойство ToolTip которого привязано к его же текущему значению:

```
<Slider ToolTip="{Binding RelativeSource={RelativeSource Self}, Path=Value}"/>
```

Привязка к обычным свойствам .NET

Пример элементов TreeView и TextBox работает, потому что оба свойства – источник и приемник – являются свойствами зависимости. В главе 3 упоминалось, что в механизм свойств зависимости встроена возможность получать уведомления об изменениях. Именно это и позволяет WPF синхронизировать значения свойства-источника и свойства-приемника.

Однако WPF поддерживает использование любого свойства любого объекта .NET в качестве источника привязки к данным. Допустим, к примеру, что нужно добавить в приложение Photo Gallery элемент Label, в котором будет отображаться количество фотографий в текущей папке. Вместо того чтобы вручную записывать в метку свойство Count коллекции фотографий (типа Photos), можно привязать свойство Content метки к свойству Count коллекции:

```
<Label x:Name="numItemsLabel"
  Content="{Binding Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

(Здесь предполагается, что коллекция определена как ресурс, поэтому на нее можно сослаться из XAML с помощью свойства Source. Свойство ElementName здесь не подойдет, потому что эта коллекция не является объектом класса FrameworkElement или FrameworkContentElement!) На рис. 13.1 показан результат такого добавления. Отметим, что в метке отображается только строка «54», хотя должно быть что-то вроде «54 item(s)». Это можно исправить, поместив рядом метку со статическим текстом «item(s)». Но есть и более удачные решения, которые мы рассмотрим ниже в этой главе.

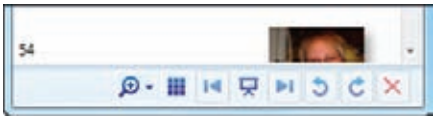


***Рис. 13.1.** Отображение в левом нижнем углу главного окна значения photos.Count с помощью механизма привязки к данным*

Однако на пути использования обычного свойства .NET в качестве источника привязки нас подстерегает неприятность. Поскольку в таких свойствах не предусмотрены средства уведомления об изменениях, то приемник не будет автоматически синхронизироваться с источником – для этого нужна дополнительная работа. Таким образом, значение счетчика, показанное на рис. 13.1, не изменяется при смене папки, что, очевидно, неправильно.

Чтобы синхронизировать источник с приемником, объект-источник должен сделать одно из двух:

- Реализовать интерфейс System.ComponentModel.INotifyPropertyChanged, в котором определено единственное событие PropertyChanged.
- Реализовать событие *XXX*Changed, где *XXX* – имя свойства, значение которого изменяется.

Рекомендуется первое решение, поскольку WPF оптимизирована под него. (События вида *XXX*Changed поддерживаются в WPF только ради обратной совместимости со старыми классами.) Приложение Photo Gallery можно модифицировать, реализовав в коллекции photos интерфейс INotifyPropertyChanged. Для этого следует перехватывать все операции изменения (Add, Remove, Clear, Insert) и генерировать событие PropertyChanged. К счастью, в .NET Framework уже имеется класс, который делает эту работу за вас! Он называется ObservableCollection. Таким образом, для синхронизации привязки к photos.Count достаточно заменить в исходном коде строку

```
public class Photos : Collection<Photo>
```

на такую:

```
public class Photos : ObservableCollection<Photo>
```

КОПНЕМ ГЛУБЖЕ

Как работает привязка к обычным свойствам .NET

Чтобы получить значение свойства-источника, которое является обычным свойством .NET, WPF применяет отражение. Если объект-источник реализует интерфейс ICustomTypeDescriptor, то WPF его и использует (или, более общими словами, любой объект типа TypeDescriptionProvider, зарегистрированный конкретно для данного объекта или для его типа), чтобы узнать, какой дескриптор PropertyDescriptor применять для отражения. Реализация этого интерфейса – дело непростое, но полезное, когда нужно повысить производительность или поддержать нетривиальные ситуации (например, «на лету» изменять состав раскрываемых свойств).

ПРЕДУПРЕЖДЕНИЕ

Источники и приемники данных обрабатываются по-разному!

Свойством-источником действительно может быть любое свойство любого объекта .NET, однако для свойства-приемника это уже не так. Свойство-приемник *обязано* быть свойством зависимости. Отметим также, что источник должен быть настоящим (и притом открытым) свойством, а не просто полем.

Привязка ко всему объекту

Во всех приведенных до сих пор примерах использовались объекты-источники и свойства-источники, но оказывается, что свойство-источник (то есть Path в объекте Binding) необязательно! Можно привязать свойство-приемник ко всему объекту.

Но что это означает? На рис. 13.2 показано, как выглядела бы метка на рис. 13.1, если бы мы опустили задание Path:

```
<Label x:Name="numItemsLabel"
  Content="{Binding Source={StaticResource photos}}"
  DockPanel.Dock="Bottom"/>
```



Рис. 13.2. Отображение в левом нижнем углу главного окна всего объекта photos с помощью механизма привязки к данным

Поскольку класс объекта `photos` не является производным от `UIElement`, то он визуализируется в виде строки, которую возвращает метод `ToString`. В данном случае привязка ко всему объекту не очень полезна, но она крайне важна для элементов, способных распорядиться объектом более удачно, как, например, `ListBox`, который мы рассмотрим ниже.

СОВЕТ

Привязка ко всему объекту удобна, когда нужно в XAML-коде задать некое свойство, требующее объекта, который нельзя получить ни с помощью конвертера типа, ни посредством расширения разметки.

Например, в программе Photo Gallery есть элемент `Popup`, который центрируется над кнопкой `zoomButton`. Для этого мы устанавливаем свойства `Placement` и `PlacementTarget` объекта `Popup`, причем значением последнего должен быть объект типа `UIElement`. На C# это делается легко:

```
Button zoomButton = new Button();
...
Popup zoomPopup = new Popup();
zoomPopup.Placement = PlacementMode.Center;
zoomPopup.PlacementTarget = zoomButton;
```

Но в Photo Gallery для решения этой задачи применяется XAML-код:

```
<Button x:Name="zoomButton" ... >
  ...
</Button>
<Popup PlacementTarget="{Binding ElementName=zoomButton}" Placement="Center" ...>
  ...
</Popup>
```

Мы неоднократно пользовались этой техникой в предыдущих главах. Разумеется, расширение разметки `x:Reference`, появившееся в WPF 4, позволяет выполнить такое присваивание и без `Binding`.

ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны с привязкой к объекту типа UIElement!

Привязывая некоторые свойства-приемники ко всему объекту `UIElement`, вы можете, сами того не желая, попытаться поместить один и тот же элемент в разные места визуального дерева. Например, следующий XAML-код приведет к исключению `InvalidOperationException` с сообщением `"Specified element is already the logical child of another element"` (Указанный элемент уже является логическим дочерним для другого элемента):

```
<Label x:Name="one" Content="{Binding ElementName=two}"/>
<Label x:Name="two" Content="text"/>
```

Однако исключения не будет, если заменить первый элемент `Label` на `TextBlock` (и, следовательно, свойство `Content` – на `Text`):

```
<TextBlock x:Name="one" Text="{Binding ElementName=two}"/>
<Label x:Name="two" Content="text"/>
```

Дело в том, что свойство `Label.Content` имеет тип `Object`, а свойство `TextBlock.Text` – строка. Поэтому, когда метке присваивается строковое значение, выполняется преобразование типа и вызывается метод `ToString`. В данном случае в текстовом блоке отображается строка `"System.Windows.Controls.Label: text"`, по-прежнему не слишком полезная. Чтобы скопировать текст из одного элемента `Label` или `TextBlock` в другой, необходимо осуществить привязку к конкретному свойству (`Label` или `Content`).

Привязка к коллекции

Привязка метки к свойству `photos.Count` – вещь хорошая, но еще лучше было бы привязать список `ListBox` (основной элемент пользовательского интерфейса в окне `Window`) к коллекции `photos`. Эта часть программы Photo Gallery прямотаки напрашивается на использование привязки к данным. В предыдущих версиях этого приложения связь между коллекцией фотографий, представленных в `ListBox`, и физическими фотографиями на диске, поддерживалась вручную. При выборе нового каталога программа очищала `ListBox` и создавала новые объекты `ListBoxItem` для каждой фотографии. Если пользователь удалял или переименовывал фотографию, то генерировалось событие от коллекции-источника (поскольку в ее реализации используется объект `FileSystemWatcher`), и его обработчик «вручную» обновлял содержимое списка `ListBox`.

К счастью, процедура замены этой логики привязкой к данным ничем не отличается от только что показанной.

Непосредственная привязка

Самым правильным было бы создать привязку `Binding`, задав в качестве свойства-приемника `ListBox.Items`, но, увы, `Items` не является свойством зависимости. Однако и `ListBox`, и все прочие многодетные элементы управления имеют свойство зависимости `ItemsSource`, специально предназначенное для привязки

к данным. Это свойство имеет тип IEnumerable, поэтому можно в качестве источника использовать весь объект photos и настроить Binding следующим образом:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
  ...
</ListBox>
```

Чтобы свойство-приемник синхронизировалось с изменениями в коллекции-источнике (то есть отражало добавление и удаление объектов), коллекция-источник должна реализовывать интерфейс INotifyCollectionChanged. Но в классе ObservableCollection реализованы оба интерфейса – INotifyPropertyChanged и InotifyCollectionChanged – так, что произведенного ранее изменения, когда мы унаследовали класс Photos от ObservableCollection<Photo>, достаточно, чтобы и эта привязка работала правильно.

На рис. 13.3 показан результат привязки.

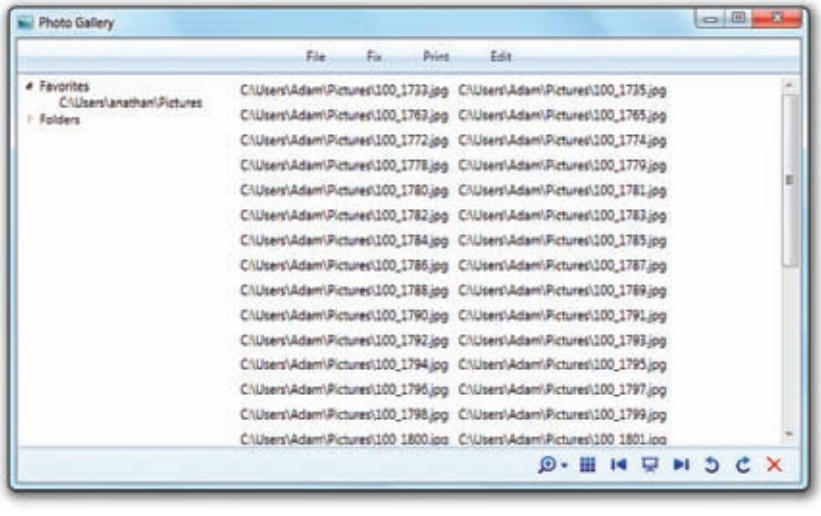


Рис. 13.3. *Если список ListBox привязан ко всему объекту photos, то данные отображаются в необработанном виде*

Улучшение отображения

Очевидно, что подразумеваемое по умолчанию отображение коллекции photos – результаты, возвращаемые методом ToString, – никуда не годится. Один из способов навести порядок – воспользоваться свойством DisplayMemberPath, которое имеется у всех многодетных элементов управления (см. главу 10). Оно работает рука об руку со свойством ItemsSource. Если записать в него путь к некоторому свойству, то для каждого объекта в списке будет отображаться значение именно этого свойства.

Коллекция в программе Photo Gallery состоит из объектов `Photo`, в которых есть такие свойства, как `Name`, `DateTime` и `Size`. Следовательно, показанная ниже XAML-разметка дает результаты, представленные на рис. 13.4, которые уже чуть лучше, чем то, что мы видим на рис. 13.3:

```
<ListBox x:Name="pictureBox" DisplayMemberPath="Name"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
  ...
</ListBox>
```

Но поскольку класс `Photo` определяли мы сами, то точно такого же результата можно было бы достичь, изменив реализацию его метода `ToString` так, чтобы он возвращал `Name` вместо полного пути.

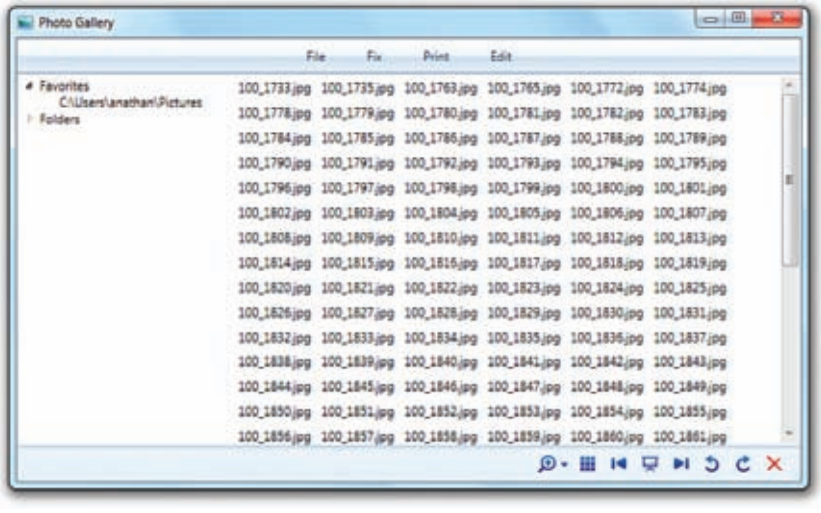


Рис. 13.4. *Свойство DisplayMemberPath дает простой механизм отображения объектов из коллекции, привязанной к данным*

Чтобы показывать в списке сами изображения, можно было бы добавить в класс `Photo` свойство `Image` и указать его в качестве `DisplayMemberPath`. Но есть и более гибкие способы контроля над представлением привязанных данных, которые не требуют вносить изменения в объект-источник. (Это существенно, потому что не всегда класс объекта-источника написан вами. Кроме того, не забывайте, что одна из основных целей WPF – отделение логики от внешнего вида!) Один такой способ (не относящийся собственно к механизму привязки к данным) – воспользоваться шаблоном данных, другой – применить конвертер значений. Ниже в разделе «Управление визуализацией» мы рассмотрим оба способа.

ПРЕДУПРЕЖДЕНИЕ

Свойства Items и ItemsSource объекта ItemsControl нельзя модифицировать одновременно!

Вы должны заранее решить, как будете заполнять многодетный элемент управления: вручную с помощью свойства `Items` или путем привязки к данным с помощью свойства `ItemsSource`, потому что смешивать эти два приема нельзя. Свойству `ItemsSource` можно присвоить значение только в тот момент, когда коллекция `Items` пуста, а свойство `Items` можно модифицировать, лишь если `ItemsSource` равно `null` (иначе вы получите исключение `InvalidOperationException`). Таким образом, если вы собираетесь добавлять объекты в привязанный к данным список `ListBox` или удалять из него объекты, то делать это следует только с помощью коллекции-источника (`ItemsSource`), а не на уровне пользовательского интерфейса (через свойство `Items`). Отметим, что вне зависимости от того, каким способом объекты *помещаются* в многодетный элемент управления, *обращаться* к ним для чтения всегда разрешается с помощью коллекции `Items`.

Управление выбранным объектом

В главе 10 было сказано, что для селекторов `Selector`, к числу которых относится и `ListBox`, определено понятие выбранного объекта или объектов. Когда селектор привязывается к коллекции (любому объекту, реализующему интерфейс `IEnumerable`), WPF отслеживает выбранные элементы, поэтому любой объект-приемник, привязанный к тому же источнику, может воспользоваться этой информацией, не требуя дополнительного кода. Эту возможность можно использовать для создания пользовательских интерфейсов вида главный/подчиненный (мы покажем, как это делается, в конце главы) или для синхронизации нескольких селекторов, чем мы сейчас и займемся.

Чтобы включить рассматриваемую поддержку, присвойте значение `true` свойству `IsSynchronizedWithCurrentItem` (которое наследуется всеми селекторами). В следующем XAML-коде это свойство установлено для трех списков `ListBox`, в каждом из которых отображается по одному свойству объекта из коллекции `photos`:

```
<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="Name"
  ItemsSource="{Binding Source={StaticResource photos}}"></ListBox>
<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="DateTime"
  ItemsSource="{Binding Source={StaticResource photos}}"></ListBox>
<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="Size"
  ItemsSource="{Binding Source={StaticResource photos}}"></ListBox>
```

Поскольку для каждого списка `IsSynchronizedWithCurrentItem="True"` и все они указывают на одну и ту же коллекцию-источник, то изменение выбранного объекта в любом списке приведет к аналогичному изменению в двух других. (Отметим, однако, что прокрутка списков автоматически не синхронизируется!) На рис. 13.5 показано, как это выглядит. Если в каком-то списке не задать свойство `IsSynchronizedWithCurrentItem` вообще или задать его равным `false`, то

изменение в нем выбранного объекта никак не отразится на двух других списках. Верно и обратное – изменение выбранного объекта в любом из двух других списков не влияет на выбранный объект в этом списке.

ПРЕДУПРЕЖДЕНИЕ

Свойство IsSynchronizedWithCurrentItem не поддерживает множественный выбор!

Если в селекторе `Selector` выбрано несколько объектов (как в случае, когда свойство `SelectionMode` элемента `ListBox` равно `Multiple` или `Extended`), то все остальные синхронизированные с ним элементы увидят только первый выбранный объект, даже если сами поддерживают множественный выбор!

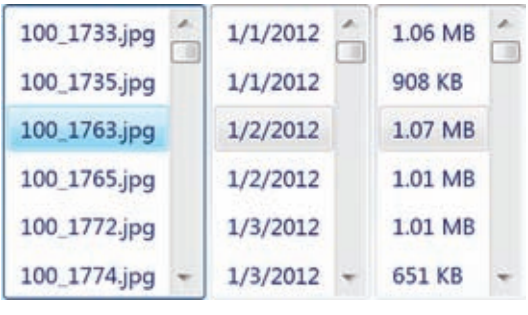


Рис. 13.5. *Благодаря привязке к данным все три списка синхронизированы*

Обобществление источника с помощью DataContext

На данный момент мы применили привязку к данным к нескольким свойствам-приемникам и для всех, кроме одного, указали один и тот же объект-источник (коллекцию `photos`). Вообще практика привязки нескольких элементов в одном пользовательском интерфейсе к общему объекту-источнику (*свойства*-источники могут быть разными, но *объект*-источник один) весьма распространена. Поэтому WPF поддерживает задание неявного источника данных вместо того, чтобы явно указывать в каждом элементе `Binding` свойства `Source`, `RelativeSource` или `ElementName`. Такой неявный источник данных называют также *контекстом данных*.

Чтобы назначить некоторый объект-источник, например коллекцию `photos`, контекстом данных, достаточно найти какой-нибудь общий родительский элемент и записать в его свойство `DataContext` ссылку на объект-источник. (Во всех классах, производных от `FrameworkElement` или `FrameworkContentElement`, имеется свойство `DataContext` типа `Object`.) Обнаружив элемент `Binding`, в котором объект-источник явно не задан, WPF поднимается вверх по логическому дереву, пока не найдет элемент с отличным от `null` свойством `DataContext`.

Поэтому, чтобы привязать элементы Label и ListBox к одному и тому же объекту-источнику, можно было бы установить DataContext следующим образом:

```
<StackPanel DataContext="{StaticResource photos}">
  <Label x:Name="numItemsLabel"
    Content="{Binding Path=Count}" .../>
  ...
  <ListBox x:Name="pictureBox" DisplayMemberPath="Name"
    ItemsSource="{Binding}" ...>
    ...
  </ListBox>
  ...
</StackPanel>
```

Поскольку DataContext – самое обычное свойство, его можно без труда установить и в процедурном коде, избежав тем самым необходимости сохранять объект-источник в качестве ресурса:

```
parent.DataContext = photos;
```

СОВЕТ

Увидев в XAML свойство, значением которого является просто строка {Binding}, легко прийти в недоумение, но это всего лишь означает, что объект-источник задан где-то выше в дереве в виде контекста данных и что привязка производится ко всему этому объекту, а не к отдельному его свойству.

FAQ



Когда следует задавать объект-источник в виде контекста данных, а когда явно в привязке Binding?

Вообще говоря, это дело вкуса. Если объект-источник используется только в одном свойстве-приемнике, то определение контекста данных – это перебор, да и разметка становится менее понятной. Если же объект-источник обобществляется, то контекст данных позволяет описать его в одном месте и, значит, уменьшить вероятность ошибки в случае смены источника.

Одна из ситуаций, когда контекст данных оказывается по-настоящему полезным, – подключение ресурсов, определенных где-то в другом месте. В таких ресурсах привязки Binding можно задавать без явного указания источника или контекста данных. Тем самым предполагается, что разрешение привязки произойдет в *контексте использования*, а не в *контексте объявления*. Контекстом использования будет то место в логическом дереве, куда помещен ресурс; и для каждого такого места может быть определен свой контекст данных. (Впрочем, такой же гибкости можно добиться за счет использования свойства RelativeSource для явного задания источника, определяемого относительным путем.)

Управление визуализацией

Привязка к данным не составляет труда, когда свойство-источник и свойство-приемник имеют совместимые типы и все, что нужно, – это получить визуализацию источника, подразумеваемую по умолчанию. Но очень часто требуется та или иная настройка. В предыдущих разделах необходимость такой настройки была очевидна – ведь нам нужно показывать в списке сами фотографии, а не текстовые строки!

Без привязки к данным такая настройка была бы простым делом, потому что весь код извлечения данных вы пишете сами (как в первоначальной версии Photo Gallery). Однако WPF предоставляет три разных способа получить и отобразить значение источника, поэтому не стоит отказываться от преимуществ привязки к данным, когда с первого взгляда неясно, как добиться желаемого результата в нестандартной ситуации. Вот эти способы: форматирование строк, шаблоны данных и конвертеры значений.

Форматирование строк

Когда результатом привязки к данным должно стать отображение строки, может оказаться полезным свойство `StringFormat` объекта `Binding`. Если оно задано, то WPF вызывает метод `String.Format`, передавая ему значение свойства `StringFormat` в первом аргументе (`format`) и неформатированный объект-приемник – во втором (`args[0]`). Таким образом, `{0}` в форматной строке будет относиться к объекту-приемнику, и вы можете применить все поддерживаемые спецификаторы формата, например `{0:C}` – для форматирования денежных величин, `{0:P}` – для представления в виде процента, `{0:X}` – для представления в шестнадцатеричном виде.

Таким образом, чтобы показать в метке на рис. 13.1 строку «54 item(s)», а не просто «54», нужно заменить ее элементом `TextBlock` и указать в элементе `Binding` простое свойство `StringFormat`:

```
<TextBlock x:Name="numItemsLabel"
  Text="{Binding StringFormat={}{0} item(s),
         Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

Неуместно выглядящие скобки `{}` в начале значения нужны для того, чтобы экранировать знак `{` в начале строки. Напомним (см. главу 2 «Все тайны XAML»), что без этого строка неправильно интерпретировалась бы как расширение разметки. Вставлять `{}` необязательно, если `Binding` используется в виде элемента (а не атрибута) свойства:

```
<TextBlock x:Name="numItemsLabel" DockPanel.Dock="Bottom">
  <TextBlock.Text>
    <Binding Source="{StaticResource photos}" Path="Count">
      <Binding.StringFormat>{0} item(s)</Binding.StringFormat>
    </Binding>
  </TextBlock.Text>
</TextBlock>
```

ПРЕДУПРЕЖДЕНИЕ

Свойство StringFormat работает, только если свойство-приемник — строка!

Основной недостаток подхода на основе свойства `StringFormat` заключается в том, что объект `Binding` попросту игнорирует это свойство, если тип свойства-приемника отличен от `string`. Попытка воспользоваться им для свойства `Content` элемента `Label` не даст никакого эффекта, потому что тип этого свойства – `Object`:



```
<Label x:Name="numItemsLabel"
  Content="{Binding StringFormat={}{0} item(s),
         Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

А вот свойство `Text` элемента `TextBlock` имеет тип `string`, поэтому та же привязка работает для него отлично. Именно поэтому в примерах данного раздела мы заменили `Label` на `TextBlock`. Другое обходное решение – воспользоваться свойством `ContentStringFormat` метки (обсуждается ниже).

Не нужны скобки и тогда, когда строка начинается с любого символа, кроме {:

```
<TextBlock x:Name="numItemsLabel"
  Text="{Binding StringFormat=Number of items: {0},
  Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

Можно также улучшить форматирование, добавив спецификатор `N0`, который расставляет разделители между группами по три цифры. Например, если `Count` равно 54, то будет показана строка «54 item(s)», а если `Count` равно 1001, то строка «1,001 item(s)» (по крайней мере, в культуре en-US):

```
<TextBlock x:Name="numItemsLabel"
  Text="{Binding StringFormat={}{0:N0} item(s),
         Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

ПРЕДУПРЕЖДЕНИЕ

System.Xaml обрабатывает управляющую последовательность {} некорректно!

В библиотеке *System.Xaml*, добавленной в WPF 4, есть ошибка, из-за которой управляющая последовательность {} внутри расширения разметки обрабатывается некорректно. При работе с *System.Xaml* последовательность {} можно использовать для экранирования *всего* строкового значения свойства (предотвращая ее интерпретацию как расширения разметки), но не *внутри* расширения разметки. Например, такой фрагмент XAML-кода разбирается библиотекой *System.Xaml* неправильно:

```
<TextBlock Text="{Binding StringFormat={}{0:C}}" />
```

К счастью, *System.Xaml* еще не применяется в наиболее распространенных ситуациях (например, при компиляции XAML-кода), так что влияние этой ошибки пока ощущается не очень сильно. Обходной путь состоит в том, чтобы в расширениях разметки использовать альтернативную управляющую последовательность, а именно экранировать отдельные символы, например:

```
<TextBlock Text="{Binding StringFormat=\{0:C\}}" />
```

У многих элементов управления имеется также свойство *XXX*StringFormat, где *XXX* представляет форматируемую часть. Например, у однодетных элементов есть свойство ContentStringFormat, которое применяется к свойству Content, а у многодетных – свойство ItemStringFormat, применяемое к каждому объекту в коллекции. В табл. 13.1 перечислены все свойства форматирования, допускающие чтение и запись.

Таблица 13.1. Свойства форматирования строк, имеющиеся в WPF

Свойство	Классы
StringFormat	BindingBase
ContentStringFormat	ContentControl, ContentPresenter, TabControl
ItemStringFormat	ItemsControl, HierarchicalDataTemplate
HeaderStringFormat	HeaderedContentControl, HeaderedItemsControl, DataGridColumn, GridViewColumn, GroupStyle
ColumnHeaderStringFormat	GridView, GridViewHeaderRowPresenter

Вместо замены Label на TextBlock, чтобы иметь возможность воспользоваться свойством StringFormat объекта Binding, можно было бы прибегнуть к собственному свойству элемента Label – ContentStringFormat, поскольку Label – однодетный элемент управления:

```
<Label x:Name="numItemsLabel" ContentStringFormat="{}{0} item(s)"
  Content="{Binding Source={StaticResource photos}, Path=Count}"
  DockPanel.Dock="Bottom"/>
```

Этот механизм работает и безотносительно к привязке к данным. На рис. 13.6 показан результат визуализации следующего списка ListBox для двух языков: американского диалекта английского и корейского:

```
<ListBox ItemStringFormat="{}{0:C}"
  xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <sys:Int32>-9</sys:Int32>
  <sys:Int32>9</sys:Int32>
  <sys:Int32>1234</sys:Int32>
  <sys:Int32>1234567</sys:Int32>
</ListBox>
```

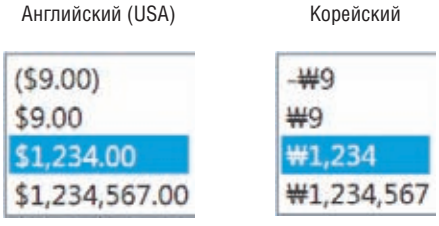


***Рис. 13.6.** Числа в списке ListBox представлены в соответствии с декларативно заданным форматированием строки*

Шаблоны данных

Шаблон данных – это часть пользовательского интерфейса, которую можно применять к произвольному объекту .NET на этапе его визуализации. У многих элементов управления WPF имеются специальные свойства (типа `DataTemplate`) для присоединения шаблона данных. Например, в классе `ContentControl` есть свойство `ContentTemplate` для управления визуализацией объекта содержимого `Content`, а в классе `ItemsControl` – свойство `ItemTemplate`, применяемое к каждому объекту хранимой коллекции. Все подобные свойства перечислены в табл. 13.2. Как видите, свойств вида *XXX*`Template` больше, чем *XXX*`StringFormat`.

Таблица 13.2. Свойства типа DataTemplate

Свойство	Классы
`ContentTemplate`	`ContentControl`, `ContentPresenter`, `TabControl`
`ItemTemplate`	`ItemsControl`, `HierarchicalDataTemplate`
`HeaderTemplate`	`HeaderedContentControl`, `HeaderedItemsControl`, `DataGridRow`, `DataGridColumn`, `GridViewColumn`, `GroupStyle`
`SelectedContentTemplate`	`TabControl`
`DetailsTemplate`	`DataGridRow`
`RowDetailsTemplate`	`DataGrid`
`RowHeaderTemplate`	`DataGrid`
`ColumnHeaderTemplate`	`GridView`, `GridViewHeaderRowPresenter`
`CellTemplate`	`DataGridTemplateColumn`, `GridViewColumn`
`CellEditingTemplate`	`DataGridTemplateColumn`

Поместив в любое из этих свойств ссылку на объект типа `DataTemplate`, можно подключить совершенно новое визуальное дерево. Класс `DataTemplate`, как и класс `ItemsPanelTemplate`, описанный в главе 10, является производным от `FrameworkTemplate`. Поэтому в нем есть свойство содержимого `VisualTree`, в которое можно записать произвольное дерево элементов `FrameworkElement`. Это легко делается в XAML, но довольно неуклюже в процедурном коде.

Давайте применим шаблон DataTemplate к списку ListBox в программе Photo Gallery. На рис. 13.4 в этом списке отображались строки вместо изображений. В следующем фрагменте мы добавляем простой шаблон данных путем задания свойства списка ItemTemplate:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
<ListBox.ItemTemplate>
  <DataTemplate>
    <Image Source="placeholder.jpg" Height="35"/>
  </DataTemplate>
</ListBox.ItemTemplate>
  ...
</ListBox>
```

На рис. 13.7 видно, что начало положено. Правда, пока вместо наших фотографий показывается иллюстрация-заглушка *placeholder.jpg*, но все-таки это уже изображения!



Рис. 13.7. *Простой шаблон данных приводит к показу иллюстрации-заглушки во всех позициях списка*

Итак, мы поместили шаблон Image в нужное место, но как записать в его свойство Source значение свойства FullPath текущего объекта Photo? Разумеется, с помощью привязки к данным! С каждым шаблоном данных неявно ассоциируется контекст данных (то есть объект-источник). Когда шаблон применяется в качестве ItemTemplate, контекстом данных будет текущий объект в источнике ItemsSource. Поэтому для получения результата, показанного на рис. 13.8, достаточно изменить шаблон данных следующим образом:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
```

```
<ListBox.ItemTemplate>
  <DataTemplate>
    <Image Source="{Binding Path=FullPath}" Height="35"/>
  </DataTemplate>
</ListBox.ItemTemplate>
  ...
</ListBox>
```

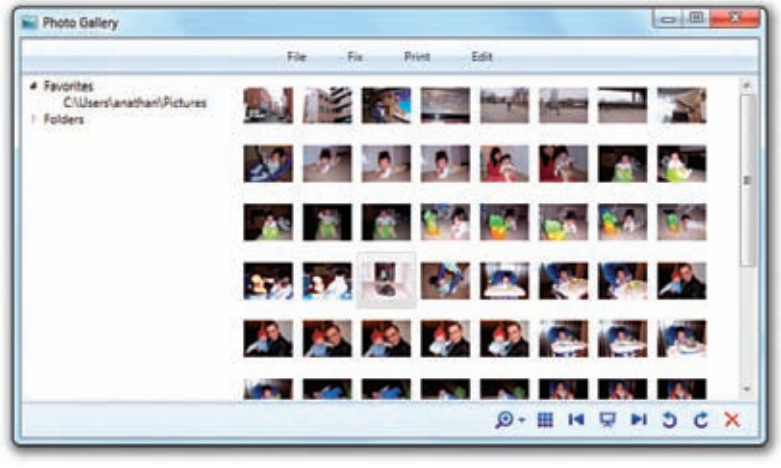


Рис. 13.8. *После модификации шаблон данных дает желаемый результат – в каждой позиции списка отображается нужная фотография*

Разумеется, шаблон данных необязательно определять в том месте, где он используется. Чаще всего шаблоны данных хранятся в виде ресурсов, разделяемых несколькими элементами. Можно даже сделать так, что шаблон данных будет автоматически применяться к любому экземпляру конкретного типа – для этого нужно лишь записать имя этого типа в свойство `DataType` шаблона. Если, к примеру, поместить такой объект `DataTemplate` в коллекцию `Resources` окна `Window`, то он автоматически будет применяться к любому элементу указанного типа, визуализируемому внутри окна, независимо от того, где он находится: в многодетном элементе управления, в однодетном или где-то еще. Если же поместить такой объект `DataTemplate` в коллекцию `Resources` объекта `Application`, то он будет относиться ко всему приложению в целом.

СОВЕТ

Хотя шаблоны данных можно использовать и с объектами, не привязанными к данным (например, со списком `ListBox`, который заполнялся вручную), почти всегда желательно, чтобы *внутри* шаблона привязка к данным была – именно это позволяет изменять визуальное дерево в соответствии с отображаемыми объектами.

Существует специальный подкласс DataTemplate для работы с иерархически организованными данными, например с XML-документами или с файловой системой, – HierarchicalDataTemplate. Он позволяет не только изменять представление таких данных, но и напрямую привязывать иерархию объектов к элементу, который умеет работать с иерархиями, например TreeView или Menu. Ниже в разделе «Класс XmlDataProvider» мы приведем пример использования HierarchicalDataTemplate с XML-данными.

КОПНЕМ ГЛУБЖЕ

Селекторы шаблонов

Иногда желательно выполнить глубокую настройку шаблона данных в зависимости от входных данных. Хотя многое можно сделать в пределах одного лишь шаблона данных, WPF предоставляет также механизм, позволяющий подключить процедурный код, который может выбрать любой шаблон (или создать новый «на лету») во время выполнения программы, когда возникает необходимость визуализировать данные. Для этого нужно создать класс, производный от DataTemplateSelector, и переопределить в нем виртуальный метод SelectTemplate. Затем следует ассоциировать экземпляр этого класса с интересующим вас элементом, установив в этом элементе свойство *XXX*TemplateSelector. Для каждого свойства *XXX*Template, показанного в табл. 13.2, существует соответствующее ему свойство *XXX*TemplateSelector (табл. 13.3).

Таблица 13.3. Свойства, относящиеся к селекторам шаблона данных

Свойство	Классы
ContentTemplateSelector	ContentControl, ContentPresenter, TabControl
ItemTemplateSelector	ItemsControl, HierarchicalDataTemplate
HeaderTemplateSelector	HeaderedContentControl, HeaderedItemsControl, DataGridRow, DataGridColumn, GridViewColumn, GroupStyle
SelectedContentTemplateSelector	TabControl
DetailsTemplateSelector	DataGridRow
RowDetailsTemplateSelector	DataGrid
RowHeaderTemplateSelector	DataGrid
ColumnHeaderTemplateSelector	GridView, GridViewHeaderRowPresenter
CellTemplateSelector	DataGridTemplateColumn, GridViewColumn
CellEditingTemplateSelector	DataGridTemplateColumn

Конвертеры значений

Если шаблоны данных позволяют изменить способ визуализации значений в свойстве-приемнике, то конвертеры значений предназначены для преобра-

зования значения, полученного из источника, в нечто совершенно иное перед доставкой приемнику. С их помощью можно подключить собственный код, не отказываясь от преимуществ привязки к данным.

Конвертеры значений часто применяются для того, чтобы согласовать различные типы данных в источнике и приемнике. Например, можно изменить цвет текста или фона элемента в зависимости от значения источника, тип данных которого отличен от Brush, – примерно так же, как это делается в функции условного форматирования в Microsoft Excel. Или просто расширить отображаемую информацию, не вводя отдельных элементов. В следующих двух разделах мы приведем примеры и того и другого.

Согласование несовместимых типов данных

Допустим, вы задумали менять цвет фона метки (свойство Background) в зависимости от количества фотографий в коллекции photos (значение свойства Count). Следующая привязка не имеет смысла, так как пытается присвоить свойству Background число, а не ожидаемый объект Brush:



```
<Label Background="{Binding Path=Count, Source={StaticResource photos}}" .../>
```

Дело можно поправить, подключив конвертер значений с помощью свойства Converter:

```
<Label Background="{Binding Path=Count, Converter={StaticResource myConverter},
  Source={StaticResource photos}}" .../>
```

Здесь предполагается, что вы написали класс, умеющий преобразовывать целое число в кисть Brush, и включили его в состав ресурсов:

```
<Window.Resources>
  <local:CountToBackgroundConverter x:Key="myConverter"/>
</Window.Resources>
```

Чтобы создать такой класс CountToBackgroundConverter, необходимо реализовать простой интерфейс IValueConverter (определен в пространстве имен System.Windows.Data). В этом интерфейсе есть всего два метода: Convert, принимающий объект-источник, который нужно преобразовать в объект-приемник, и ConvertBack, выполняющий обратную операцию.

Таким образом, на языке C# класс CountToBackgroundConverter можно написать так:

```
public class CountToBackgroundConverter : IValueConverter
{
   public object Convert(object value, Type targetType,
     object parameter, CultureInfo culture)
  {
    if (targetType != typeof(Brush))
      throw new InvalidOperationException("The target must be a Brush!");

    // Если формат входных данных недопустим, то Parse возбудит исключение
    int num = int.Parse(value.ToString());
```

```
        return (num == 0 ? Brushes.Yellow : Brushes.Transparent);
      }

      public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture)
      {
        return DependencyProperty.UnsetValue;
      }
    }
```

Метод `Convert` вызывается при каждом изменении значения источника. Ему передается целочисленное значение, а в ответ он возвращает объект `Brushes.Yellow`, если это значение равно нулю, и `Brushes.Transparent` – в противном случае. (Идея в том, чтобы выделить метку цветом, когда отображаемая папка пуста.) Метод `ConvertBack` нам не нужен, поэтому мы просто возвращаем фиктивное значение. В части VI «Дополнительные вопросы» мы рассмотрим ситуации, когда возникает необходимость в методе `ConvertBack`. На рис. 13.9 показан результат работы конвертера значений.



***Рис. 13.9.** Конвертер значений делает желтым цвет фона метки в левом нижнем углу главного окна, когда в коллекции photos нет ни одной фотографии*

СОВЕТ

Во избежание недоразумений рекомендуется отражать семантику конвертера значений в его названии. Я мог бы назвать описанный выше класс не `CountToBackgroundConverter`, а, скажем, `IntegerToBrushConverter`, потому что технически его можно использовать всюду, где тип данных источника – целое число, а тип данных приемника – `Brush`. Однако это имеет смысл, только если исходное целое число представляет количество объектов, а кисть – цвет фона. (Например, маловероятно, что кто-нибудь захочет присвоить свойству элемента `Foreground` (цвет текста) значение `Transparent` (прозрачный)!) Кроме того, возможно, вам понадобится определить дополнительные конвертеры целого в кисть с иной семантикой.

Методам интерфейса `IValueConverter` передаются аргументы `parameter` и `culture`. По умолчанию `parameter` равен `null`, а `culture` – значению свойства `Language` элемента-приемника. Это свойство (определенное в классах `FrameworkElement` и `FrameworkContentElement`, где часто наследуется от корневого элемента, если вообще задается) по умолчанию равно `"en-US"` (американский диалект английского языка). Однако пользователь привязки может управлять обоими значениями с помощью свойств `Binding.ConverterParameter` и `Binding.Converter-`

Culture. Например, вместо того чтобы жестко зашивать цвет Brushes.Yellow в код метода CountToBackgroundConverter.Convert, можно было бы передавать его в параметре:

```
return (num == 0 ? parameter : Brushes.Transparent);
```

Здесь предполагается, что parameter всегда задается следующим образом:

```
<Label Background="{Binding Path=Count, Converter={StaticResource myConverter},
  ConverterParameter=Yellow, Source={StaticResource photos}}" Content="..." />
```

Присваивание свойству ConverterParameter простой строки "Yellow" работает, а вот почему работает – это тонкий вопрос. Как и все параметры расширения разметки, строка "Yellow" подвергается преобразованию типа, но только в тип свойства ConverterParameter (Object). Следовательно, метод Convert получает параметр в виде строки "Yellow", а не Brush. Поскольку Convert не делает со своим аргументом parameter ничего, кроме его возврата, когда num отлично от нуля, то в конечном итоге он возвращает строку. И вот в этот момент механизм привязки выполняет преобразование типа, чтобы присваивание свойству Background элемента Label завершилось нормально.

В свойство ConverterCulture можно записать любое обозначение языка, стандартизованное Инженерной группой по развитию Интернета (IETF), например "ko-KR", тогда конвертер получит соответствующий объект типа CultureInfo.

СОВЕТ

В состав WPF входит ряд конвертеров значений для нескольких очень распространенных сценариев привязки к данным. Один из них называется BooleanToVisibilityConverter и преобразует трехпозиционное перечисление Visibility (в нем определены значения Visible, Hidden, Collapsed) в тип Boolean, в том числе допускающий значение null (и обратно). В одном направлении true отображается на Visible, а false и null – на Collapsed. В другом направлении Visible отображается на true, а Hidden и Collapsed – на false.

Это бывает полезно для переключения видимости одного элемента в зависимости от состояния другого. Например, в следующей XAML-разметке реализован флажок Show Status Bar (Показывать строку состояния) вообще без процедурного кода:

```
<Window.Resources>
  <BooleanToVisibilityConverter x:Key="booltoVis"/>
</Window.Resources>
...
<CheckBox x:Name="checkBox">Show Status Bar</CheckBox>
...
<StatusBar Visibility="{Binding ElementName=checkBox, Path=IsChecked,
  Converter={StaticResource booltoVis}}">...</StatusBar>
```

В данном случае элемент StatusBar виден тогда и только тогда, когда свойство IsChecked флажка CheckBox равно true.

ПРЕДУПРЕЖДЕНИЕ

Ошибки привязки к данным не возбуждают исключений!

В случае когда во время привязки к данным обнаруживается ошибка, WPF не возбуждает исключение, а выводит пояснительный текст в трассировку отладки, которую можно видеть, только если к программе присоединен отладчик (или другой прослушиватель трассировки). Поэтому если привязка к данным работает не так, как вы ожидали, попробуйте запустить программу под отладчиком и не забудьте посмотреть на трассировку. В Visual Studio трассировка выводится в окно Output (Вывод). В Visual Studio 2010 Ultimate трассировку отладки можно также интегрировать в удобное окно IntelliTrace.

В предыдущем примере бессмысленной привязки (попытки привязать свойство `Background` непосредственно к `photos.Count`) выводится такая отладочная трассировка:

```
System.Windows.Data Error: 5 : Value produced by BindingExpression is not valid
for target property.; Value='39' BindingExpression:Path=Count; DataItem='Photos'
(HashCode=58961324); target element is 'Label' (Name='numItemsLabel'); target
property is 'Background' (type 'Brush')
```

Даже исключения, возбуждаемые объектом-источником (или конвертером значений) по умолчанию, «проглатываются» и отображаются в отладочной трассировке!

Поскольку трассировка реализована с помощью объектов `System.Diagnostics.TraceSource`, то существует несколько стандартных способов получить ту же трассу вне отладчика. Майк Хиллберг (Mike Hillberg), архитектор WPF, поделился подробностями в своем блоге по адресу *http://blogs.msdn.com/mikehillberg/archive/2006/09/14/WpfTraceSources.aspx*. Можно перехватывать трассировки, которые WPF генерирует в разных местах (некоторые из них по умолчанию не видны даже под отладчиком), например информацию о маршрутизации событий, о регистрации свойств зависимости, о поиске ресурсов и многом другом.

Можно также воспользоваться свойством `PresentationTraceSources.TraceLevel` (определено в пространстве имен `System.Diagnostics` в сборке `WindowsBase`), которое присоединяется к любому объекту `Binding` и позволяет увеличить или уменьшить объем информации, помещаемой им в трассировку. Это свойство может принимать значения из перечисления `PresentationTraceLevel`: `None`, `Low`, `Medium`, `High`.

Настройка отображения данных

Конвертеры значений иногда бывают полезны даже в случае, когда типы данных источника и приемника совместимы. Когда мы устанавливали свойство `Content` метки `numItemsLabel` равным свойству `Count` коллекции `photos` (см. рис. 13.1), оно отображалось нормально, но потребовался дополнительный текст, чтобы пользователю было понятно, что означает число. Мы решили эту проблему с помощью свойства `StringFormat`, но можно было бы добиться лучшего результата, чем отображение суффикса item(s). (Не знаю, как вам, а мне строки вида «1 item(s)» всегда кажутся свидетельством лени разработчика.)

Конвертер значений позволяет настроить текст в зависимости от значения, то есть выводить «1 item» (в единственном числе), но «2 items» (во множественном числе). Эту задачу решает класс `RawCountToDescriptionConverter`:

```
public class RawCountToDescriptionConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    // Если формат входных данных недопустим, то Parse возбудит исключение
    int num = int.Parse(value.ToString());
    return num + (num == 1 ? " item" : " items");
  }

  public object ConvertBack(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    return DependencyProperty.UnsetValue;
  }
}
```

Отметим, что здесь жестко зашиты англоязычные строки, тогда как в программе промышленного качества конвертер следовало бы сделать локализуемым ресурсом (или, по крайней мере, воспользоваться переданным параметром `culture`).

СОВЕТ

Конвертеры значений дают возможность подключить к процессу привязки к данным любую логику, выходящую за рамки простого форматирования. Нужно ли вам применить к значению-источнику некоторое преобразование перед отображением или изменить способ обновления приемника при изменении значения-источника – все это можно легко сделать с помощью класса, реализующего интерфейс `IValueConverter`.

СОВЕТ

Можно сделать так, что конвертер значений будет временно отменять привязку к данным, возвращая специальное значение `Binding.DoNothing`. Это не то же самое, что возврат `null`, поскольку `null` может быть вполне допустимым значением свойства-приемника.

`Binding.DoNothing` означает следующее: «Я не хочу сейчас ничего привязывать, сделай вид, что объекта `Binding` не существует». В таком случае значение свойства-приемника не изменяется, если только не найдется другого влияющего на него компонента (анимации, триггера и т. д.). Возврат `Binding.DoNothing` распространяется только на данный вызов `Convert` или `ConvertBack`, так что если объект `Binding` не уничтожен (например, обращением к методу `ClearBinding`), то конвертер будет и дальше вызываться при каждом изменении значения-источника.

FAQ

Как заставить конвертер значений выполнять преобразование для каждого объекта, когда привязка осуществляется к коллекции?

Можно применить шаблон данных к свойству `ItemTemplate` элемента `ItemsControl`, а затем применять конвертеры значений ко всем объектам `Binding`, находящимся внутри шаблона. Если же вместо этого применить конвертер значений к свойству `Binding` элемента `ItemsControl`, то при обновлении коллекции-источника метод `Convert` будет вызван только один раз для всей коллекции (а не для каждого объекта в ней). Можно написать такой конвертер, который будет принимать коллекцию и возвращать тоже коллекцию – видоизмененную, но такой подход вряд ли можно назвать эффективным.

Настройка представления коллекции

Выше, в разделе «Привязка к коллекции», мы видели, что одним поворотом переключателя (установкой для свойства `IsSynchronizedWithCurrentItem` значения `true`) можно сделать так, что несколько селекторов, указывающих на одну и ту же коллекцию-источник, будут видеть один и тот же выбранный элемент. Такое поведение кажется чуть ли не проявлением волшебства, по крайней мере, когда смотришь на него вживую (очень трудно передать синхронизированное движение на статическом снимке экрана!). Коллекция-источник ничего не знает о текущем элементе, так откуда же берется информация и где сохраняется состояние?

А вот откуда. Когда производится привязка к коллекции (любому объекту, реализующему интерфейс `IEnumerable`), между источником и приемником неявно вставляется *представление* по умолчанию. В этом представлении (объекте, реализующем интерфейс `ICollectionView`) хранится вся информация о текущем объекте, но помимо этого оно поддерживает сортировку, группировку, фильтрацию и навигацию. В этом разделе мы рассмотрим все эти вопросы, а заодно покажем, как работать с несколькими представлениями для одного и того же объекта-источника.

СОВЕТ

Представления автоматически ассоциируются с коллекциями-источниками, но не с приемниками. В результате любое изменение представления (например, сортировка или фильтрация) автоматически становится видимым всем приемникам.

Сортировка

В интерфейсе `ICollectionView` определено свойство `SortDescriptions`, которое позволяет управлять сортировкой объектов представления. Основная идея

заключается в том, что выбирается некоторое свойство объекта, хранящегося в коллекции, по которому будет производиться сортировка (скажем, Name, DateTime или Size в случае объекта Photo), и указывается, в каком направлении сортировать: по возрастанию или по убыванию. Эти сведения инкапсулируются в объекте SortDescription, конструктору которого передаются имя свойства и аргумент типа ListSortDirection. Например:

```
SortDescription sort = new SortDescription("Name", ListSortDirection.Ascending);
```

Однако же свойство SortDescriptions — это *коллекция* объектов SortDescription. Так сделано для того, чтобы можно было сортировать сразу по нескольким свойствам. Первый объект SortDescription в коллекции описывает наиболее существенное для сортировки свойство, а последний — наименее существенное. Например, если добавить в коллекцию следующие два объекта SortDescription, то сначала будет произведена сортировка по DateTime в порядке убывания, а если при этом обнаружатся объекты с одинаковыми значениями в данном свойстве, то они будут дополнительно отсортированы по свойству Name (в порядке возрастания):

```
view.SortDescriptions.Add(new SortDescription("DateTime",
  ListSortDirection.Descending));
view.SortDescriptions.Add(new SortDescription("Name",
  ListSortDirection.Ascending));
```

В коллекции SortDescriptions есть метод Clear, возвращающий представление к сортировке по умолчанию. Под этим понимается тот порядок, в котором объекты добавлялись в коллекцию, то есть, возможно, вообще без сортировки!

В листинге 13.1 показано, как в программе Photo Gallery можно реализовать сортировку фотографий по свойствам Name, DateTime или Size, когда пользователь нажимает соответствующую кнопку. Как и в Проводнике Windows, повторное ее нажатие меняет направление сортировки на противоположное.

Листинг 13.1. Сортировка по трем разным свойствам

```
// Обработчики события Click от трех разных кнопок:
void sortByName_Click(object sender, RoutedEventArgs e)
{
  SortHelper("Name");
}
void sortByDateTime_Click(object sender, RoutedEventArgs e)
{
  SortHelper("DateTime");
}
void sortBySize_Click(object sender, RoutedEventArgs e)
{
  SortHelper("Size");
}

void SortHelper(string propertyName)
```

```
{
  // Получаем представление по умолчанию
  ICollectionView view = CollectionViewSource.GetDefaultView(
    this.FindResource("photos"));

  // Проверяем, отстортировано ли уже представление
  // по указанному свойству в порядке возрастания
  if (view.SortDescriptions.Count > 0
      && view.SortDescriptions[0].PropertyName == propertyName
      && view.SortDescriptions[0].Direction == ListSortDirection.Ascending)
  {
    // Уже отсортировано по возрастанию, меняем направление
    // сортировки на вариант «по убыванию»
    view.SortDescriptions.Clear();
    view.SortDescriptions.Add(new SortDescription(
      propertyName, ListSortDirection.Descending));
  }
  else
  {
    // Сортируем по возрастанию
    view.SortDescriptions.Clear();
    view.SortDescriptions.Add(new SortDescription(
      propertyName, ListSortDirection.Ascending));
  }
}
```

Отметим, что в этом коде нигде не упоминается список `ListBox`, в котором отображаются фотографии. Представление, с которым он работает, ассоциировано с коллекцией-источником `photos`, и для его получения вызывается простой статический метод `CollectionViewSource.GetDefaultView`. Если бы к той же коллекции фотографий были привязаны еще какие-то многодетные элементы управления, то они получили бы то же самое представление по умолчанию и сортировались бы синхронно.

КОПНЕМ ГЛУБЖЕ

Нестандартная сортировка

Если вам нужен больший контроль над процессом сортировки, чем может дать `ICollectionView.SortDescriptions` (что маловероятно), то лучше всего воспользоваться поддержкой нестандартной сортировки. Если представляемая коллекция реализует интерфейс `IList` (как большинство коллекций), то объект `ICollectionView`, возвращенный методом `CollectionViewSource.GetDefaultView`, в действительности является экземпляром класса `ListCollectionView`. Если привести `ICollectionView` к типу `ListCollectionView`, то можно будет присоединить объект, реализующий интерфейс `IComparer`, к его свойству `CustomSort`. После этого для определения порядка сортировки будет вызываться ваша реализация метода `IComparer.Compare`. Внутри метода `Compare` можно пользоваться любыми средствами для сортировки объектов.

Группировка

В интерфейсе ICollectionView определено свойство GroupDescriptions, принцип работы которого такой же, как у SortDescriptions. В эту коллекцию можно добавить сколько угодно объектов PropertyGroupDescription с целью разбить все объекты представляемой коллекции на группы и, возможно, подгруппы.

Например, следующий код группирует фотографии в коллекции photos по значению свойства DateTime:

```
// Получаем представление по умолчанию
ICollectionView view = CollectionViewSource.GetDefaultView(
  this.FindResource("photos"));

// Выполняем группировку
view.GroupDescriptions.Clear();
view.GroupDescriptions.Add(new PropertyGroupDescription("DateTime"));
```

Но, в отличие от сортировки, эффект группировки не проявляется, если не модифицировать многодетный элемент управления, в котором отображаются данные. Чтобы добиться правильного поведения, необходимо присвоить свойству GroupStyle многодетного элемента ссылку на объект типа GroupStyle. У такого объекта имеется свойство HeaderTemplate, в которое нужно записать шаблон данных, определяющий внешний вид заголовка группы.

Списку ListBox в приложении Photo Gallery можно назначить такой объект GroupStyle:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
<ListBox.GroupStyle>
  <GroupStyle>
  <GroupStyle.HeaderTemplate>
    <DataTemplate>
      <Border BorderBrush="Black" BorderThickness="1">
        <TextBlock Text="{Binding Path=Name}" FontWeight="Bold"/>
      </Border>
    </DataTemplate>
  </GroupStyle.HeaderTemplate>
  </GroupStyle>
</ListBox.GroupStyle>
  ...
</ListBox>
```

Обратите внимание на использование привязки к данным внутри шаблона данных. В данном случае с шаблоном связывается контекст данных – специальный объект типа CollectionViewGroup, который создается «за кулисами». Детали этого класса нам не очень важны, существенно лишь то, что в нем есть свойство Name, представляющее значение, по которому определяется группа. Следовательно, в этом шаблоне привязка к данным используется для того, чтобы вывести свойство Name в качестве заголовка группы. На рис. 13.10 пред-

ставлен результат визуализации обновленной XAML-разметки вместе с показанным выше кодом.

СОВЕТ

Если вы хотите сгруппировать объекты, хранящиеся в многодетном элементе управления, но равнодушны к оформлению заголовков групп, то можете воспользоваться объектом `GroupStyle`, встроенным в WPF. Его можно получить в виде статического свойства `GroupStyle.Default` и сослаться на него в XAML следующим образом:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos}}" ...>
<ListBox.GroupStyle>
  <x:Static Member="GroupStyle.Default"/>
</ListBox.GroupStyle>
  ...
</ListBox>
```

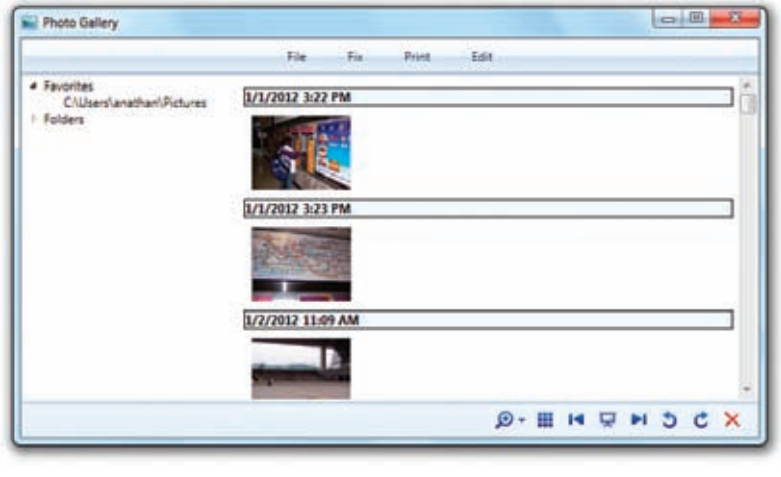


Рис. 13.10. *Первая попытка сгруппировать объекты в списке ListBox*

Но проделав все это, вы, вероятно, придете к выводу, что группировать фотографии по свойству `Photo.DateTime` – не слишком удачная мысль. Поскольку `DateTime` включает не только дату, но и время, то в каждой группе, как правило, оказывается всего одна фотография!

Чтобы исправить ситуацию, можно воспользоваться перегруженным вариантом конструктора класса `PropertyGroupDescription`, который позволяет изменить значение свойства перед тем, как использовать его для группировки.

Для этого конструктор принимает конвертер значений. Стало быть, можно написать класс DateTimeToDateConverter, который преобразует значение DateTime в строку, больше подходящую для группировки:

```
public class DateTimeToDateConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    return ((DateTime)value).ToString("MM/dd/yyyy");
  }

  public object ConvertBack(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    return DependencyProperty.UnsetValue;
  }
}
```

В данном случае мы просто отбрасываем компоненту времени в исходном значении DateTime. Впрочем, имена групп не обязаны быть строками, поэтому метод Convert мог бы отсечь время и вернуть сам объект DateTime:

```
return ((DateTime)value).Date;
```

Этот прием позволяет поддержать и более хитрые группировки, например вычисление диапазона дат и возврат строк вида «За последнюю неделю», «За последний месяц» и т. д. (Разумеется, возвращаемая строка должна формироваться с учетом переданной культуры.)

Определив конвертер значений, его можно использовать для группировки следующим образом:

```
// Получаем представление по умолчанию
ICollectionView view = CollectionViewSource.GetDefaultView(
  this.FindResource("photos"));

// Выполняем группировку
view.GroupDescriptions.Clear();
view.GroupDescriptions.Add(
new PropertyGroupDescription("DateTime", new DateTimeToDateConverter()));
```

Получившийся результат показан на рис. 13.11.

Для сортировки групп можно применить прием, описанный в предыдущем разделе. Сортировка всегда выполняется перед группировкой. В результате первый дескриптор SortDescription применяется к группам, а все остальные – к объектам внутри группы. Только позаботьтесь о том, чтобы свойство (или написанный вами алгоритм), применяемое для сортировки, соответствовало свойству (или алгоритму), используемому для группировки, иначе получившееся упорядочение объектов окажется интуитивно непонятным.

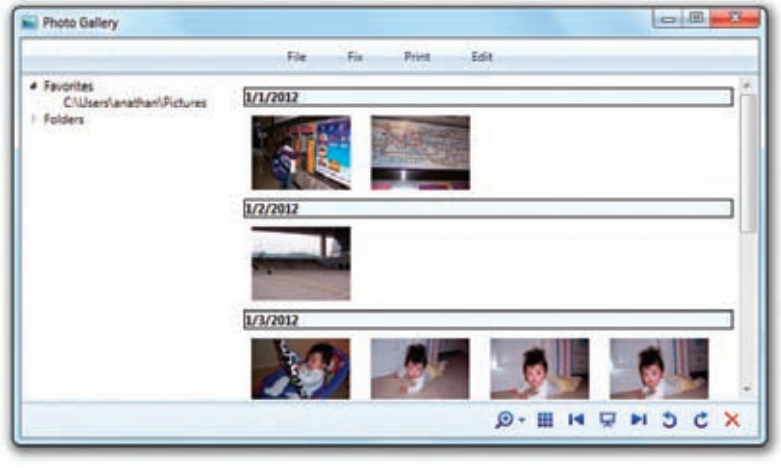


Рис. 13.11. *Улучшенная группировка, учитывающая только дату в свойстве Photo.DateTime*

СОВЕТ

Возможно, вам захочется реализовать группировку по нескольким свойствам. Для этого следует сконструировать объект `PropertyGroupDescription`, в котором имя свойства содержит `null`. В таком случае в качестве параметра `value` ваш конвертер значений получит весь объект-источник (объект типа `Photo` в приложении Photo Gallery), а не значение одного свойства.

Фильтрация

Помимо свойств для поддержки сортировки и группировки в интерфейсе `ICollectionView` имеется свойство, позволяющее реализовать *фильтрацию* – избирательное исключение объектов, удовлетворяющих произвольному условию. Оно называется `Filter` и имеет тип `Predicate<Object>` (иными словами, это делегат, принимающий один параметр типа `Object` и возвращающий булевское значение).

Если `Filter` равно `null` (случай по умолчанию), то в представление включаются все объекты из коллекции-источника. Если же делегат задан, то он вызывается для каждого присутствующего в коллекции объекта. Задача делегата – решить, какие объекты показать (для них он возвращает `true`), а какие – скрыть (возвращает `false`).

Анонимные делегаты в C# позволяют записать фильтр очень компактно. Например, следующий код отфильтровывает все объекты `Photo`, для которых свойство `DateTime` описывает дату, отстоящую от текущей более чем на 7 дней:

```
ICollectionView view = CollectionViewSource.GetDefaultView(this.FindResource
  ("photos"));
view.Filter = delegate(object o) {
  return ((o as Photo).DateTime - DateTime.Now).Days <= 7;
};
```

Можно записать это еще компактнее с помощью лямбда-выражения, хотя с непривычки разобраться в такой нотации сложнее:

```
ICollectionView view = CollectionViewSource.GetDefaultView(this.FindResource
  ("photos"));
view.Filter = (o) => { return ((o as Photo).DateTime -  DateTime.Now).Days <= 7;};
```

Чтобы снять фильтр, достаточно присвоить свойству `view.Filter` значение `null`.

Навигация

В этом контексте под *навигацией* понимается управление текущим объектом, а не та навигация, о которой шла речь в главе 7 «Структурирование и развертывания приложения». В интерфейсе `ICollectionView` определено не только свойство `CurrentItem` (и соответствующее ему свойство `CurrentPosition`, которое возвращает индекс текущего элемента, нумеруемый с нуля), но и ряд методов для программного изменения `CurrentItem`. В версии Photo Gallery с привязкой к данным эти методы используются для реализации обработчиков события `Click` от кнопок Next Photo (Следующая фотография) и Previous Photo (Предыдущая фотография):

```
void previous_Click(object sender, RoutedEventArgs e)
{
  // Получаем представление по умолчанию
  ICollectionView view = CollectionViewSource.GetDefaultView(
this.FindResource("photos"));

  // Идем назад
  view.MoveCurrentToPrevious();

  // Дойдя до начала, переходим в конец
  if (view.IsCurrentBeforeFirst) view.MoveCurrentToLast();
}

void next_Click(object sender, RoutedEventArgs e)
{
  // Получаем представление по умолчанию
  ICollectionView view = CollectionViewSource.GetDefaultView(
this.FindResource("photos"));

  // Идем вперед
  view.MoveCurrentToNext();

  // Дойдя до конца, переходим в начало
  if (view.IsCurrentAfterLast) view.MoveCurrentToFirst();
}
```

Хотя имена этих методов длинноваты, пользоваться ими несложно. Приведенные выше обработчики автоматически обновляют не только выбранный объект в списке ListBox, не ссылаясь на него явно, но и любые другие элементы, желающие знать о текущем объекте, при условии их привязки к тому же источнику. Отметим, что пока в коллекции-источнике не выбран какой-то элемент, свойство CurrentItem равно null, а CurrentPosition имеет значение -1. Однако это справедливо лишь потому, что так работает элемент ListBox. Предоставленная самой себе, коллекция инициализирует CurrentPosition нулем, а CurrentItem – ссылкой на первый объект.

СОВЕТ

Пути к свойствам, задаваемые в привязках Binding, поддерживают специальный синтаксис для ссылки на текущий объект коллекции – косую черту. Например, следующий объект Binding привязывается к текущему объекту в предположении, что источник данных – коллекция:

```
"{Binding Path=/}"
```

А такой объект Binding – к свойству DateTime текущего объекта:

```
"{Binding Path=/DateTime}"
```

А такой объект Binding – к текущему объекту коллекции, которую возвращает свойство Photos другого источника данных, причем сам он коллекцией не является:

```
"{Binding Path=Photos/}"
```

И наконец, следующий объект Binding привязывается к свойству DateTime текущего объекта из предыдущего примера:

```
"{Binding Path=Photos/DateTime}"
```

Эта возможность весьма полезна для реализации интерфейсов вида главный/подчиненный без написания процедурного кода.

ПРЕДУПРЕЖДЕНИЕ

В представлении по умолчанию навигация не предлагается автоматически!

В отличие от сортировки, группировки и фильтрации, средства навигации в представлении по умолчанию доступны, только если свойство IsSynchronizedWithCurrentItem селектора равно true. В противном случае свойство SelectedItem селектора и свойство CurrentItem представления по умолчанию не синхронизированы и могут изменяться независимо друг от друга. Проектировщики WPF хотели, чтобы синхронизация выбранного объекта включалась по явному указанию разработчика, дабы тот не забывал о наличии представления, иначе могло бы возникнуть недоразумение. Но, честно говоря, лично мне путаницей кажется несогласованность с остальными «автоматическими» аспектами поведения представления по умолчанию.

Дополнительные представления

В приведенных выше примерах сортировки, группировки, фильтрации и навигации мы работали только с представлением, которое по умолчанию ассоциировано с коллекцией-источником. Но вполне может статься, что вы захотите иметь *разные* представления одной и той же коллекции-источника для различных элементов управления. Оказывается, класс CollectionViewSource умеет не только возвращать представление по умолчанию, но и конструировать новое представление над любым источником. Для каждого приемника можно задавать свое представление вместо подразумеваемого по умолчанию.

Чтобы создать новое представление над коллекцией photos в приложении Photo Gallery, нужно сделать следующее:

```
CollectionViewSource viewSource = new CollectionViewSource();
viewSource.Source = photos;
// теперь viewSource.View указывает на реализацию интерфейса
// ICollectionView, отличную от подразумеваемой по умолчанию
```

Класс CollectionViewSource устроен так, что можно без труда создавать нестандартные представления декларативно, то есть вместо приведенного выше кода можно было бы воспользоваться такой XAML-разметкой:

```
<Window.Resources>
  <local:Photos x:Key="photos"/>
  <CollectionViewSource x:Key="viewSource" Source="{StaticResource photos}"/>
</Window.Resources>
```

Чтобы применить нестандартное представление к свойству-приемнику, достаточно привязать его к объекту CollectionViewSource, а не к исходному объекту-источнику, над которым построено представление:

```
<ListBox x:Name="pictureBox"
  ItemsSource="{Binding Source={StaticResource photos viewSource}}" ...>
  ...
</ListBox>
```

Отметим, что хотя исходный источник теперь обернут объектом CollectionViewSource, WPF рассматривает класс CollectionViewSource как особый случай, чтобы вам не приходилось изменять пути Path в объекте Binding. Так, привязка к свойству Count по-прежнему обеспечивает ссылку на свойство исходного объекта Photos, а не на свойство объекта CollectionViewSource.

Теперь список ListBox не реагирует на операции сортировки, группировки, фильтрации и навигации, производимые с помощью представления по умолчанию. Если требуется выполнить любое из этих действий над нестандартным представлением, то можно поступать так же, как описано выше, только объект ICollectionView нужно получать не от статического метода CollectionViewSource.GetDefaultView, а от свойства экземпляра CollectionViewSource.

Чтобы можно было конфигурировать сортировку и группировку с помощью нестандартного представления целиком в XAML, класс CollectionViewSource предлагает собственные свойства SortDescriptions и GroupDescriptions, работаю-

щие точно так же, как их аналоги в интерфейсе ICollectionView. Кроме того, в этом классе есть и свой член Filter, но он определен как событие, а не как делегат, чтобы его можно было задавать в XAML-коде. (Разумеется, сам обработчик события все равно должен быть определен процедурно.) Таким образом, сортировку, группировку и фильтрацию можно выразить на XAML следующим образом:

```
<CollectionViewSource x:Key="viewSource" Filter="viewSource_Filter"
  Source="{StaticResource photos}">
<CollectionViewSource.SortDescriptions>
  <componentModel:SortDescription PropertyName="DateTime" Direction="Descending"/>
</CollectionViewSource.SortDescriptions>
<CollectionViewSource.GroupDescriptions>
  <PropertyGroupDescription PropertyName="DateTime"/>
</CollectionViewSource.GroupDescriptions>
</CollectionViewSource>
```

Фильтрация

Сортировка

Группировка

Класс SortDescription находится в пространстве имен .NET, которое не отображено на стандартное пространство имен XML, поэтому необходима следующая директива:

```
xmlns:componentModel="clr-namespace:System.ComponentModel;assembly=WindowsBase"
```

Метод viewSource_Filter, на который ссылается XAML, можно было бы реализовать, как показано ниже; это просто по-другому записанный делегат для исключения фотографий старше семи дней:

```
void viewSource_Filter(object sender, FilterEventArgs e)
{
  e.Accepted = ((e.Item as Photo).DateTime - DateTime.Now).Days <= 7;
}
```

В отличие от делегата, которому объект-источник передавался напрямую, обработчик события получает его в виде e.Item. А вместо того чтобы *возвращать* булевское значение, он должен присвоить значение булевскому свойству e.Accepted и таким образом сообщить, оставить данный объект или отбросить.

СОВЕТ

Даже если в нескольких представлениях одной коллекции-источника нет нужды, все равно можно создать и применить нестандартное представление в виде явного объекта CollectionViewSource просто для того, чтобы сортировать и группировать без написания процедурного кода!

FAQ

Если в моем приложении есть коллекция и никто нигде не привязывается к ней напрямую (а только к CollectionViewSource), будет ли существовать представление по умолчанию?

Нет. Из соображений производительности представление по умолчанию создается только по запросу. Этим оно отличается от падающего дерева в лесу, которое, как мне говорили, трещит, даже когда рядом никого нет и, стало быть, некому услышать.

ПРЕДУПРЕЖДЕНИЕ

В нестандартном представлении навигация работает по-другому!

По умолчанию изменение текущего объекта в нестандартном представлении автоматически отражается на всех привязанных к нему селекторах; чтобы отменить синхронизацию, нужно явно присвоить свойству селектора `IsSynchronizedWithCurrentItem` значение `false`. То есть поведение прямо противоположно тому, что наблюдается в представлении по умолчанию!

Хотя свойство `IsSynchronizedWithCurrentItem` по умолчанию равно `false`, WPF автоматически меняет его значение на `true`, когда в свойство селектора `ItemsSource` записывается ссылка на нестандартное представление, если только ему не было явно присвоено другое значение (или свойство `SelectionMode` селектора равно `Single`). Идея в том, что, работая с нестандартным представлением, вы заведомо знаете о его существовании, поэтому и получаете ожидаемое поведение навигации по умолчанию. (Из-за таких вот шалостей я и хотел бы, чтобы свойство `IsSynchronizedWithCurrentItem` по умолчанию было равно `true` для любого представления.)

Поставщики данных

Поскольку объект-источник может быть произвольным объектом .NET, то, написав соответствующий код, можно привязаться практически к чему угодно – к базе данных, к реестру Windows, к таблице Excel и т. д. Нужно лишь иметь объект .NET, который предоставляет необходимые свойства и события и берет на себя заботу обо всех деталях реализации! (Но если вы будете писать все сами, то трудоемкость разработки подобного кода может свести на нет все преимущества привязки к данным.)

Чтобы уменьшить потребность в написании своего кода, WPF предлагает два класса, в которых необходимые члены реализованы обобщенным – «дружественным к привязке к данным» – образом: `XmlDataProvider` и `ObjectDataProvider`.

СОВЕТ

Начиная с версии WPF 3.5 SP1 механизм привязки к данным отлично работает с технологией Language Integrated Query (LINQ). Можно записать в свойство `Source` объекта `Binding` (или в свойство `DataContext` элемента) LINQ-запрос и использовать возвращенный результат типа `IEnumerable` точно так же, как и любую другую коллекцию. Следовательно, наличие технологий LINQ to SQL, LINQ to XML и других, позволяющих использовать в качестве поставщиков данных классы LINQ, а не WPF, открывает альтернативную возможность для привязки к таблицам базы данных, XML-документам и т. д.

Класс XmlDataProvider

Класс `XmlDataProvider` дает простой способ привязаться к фрагменту XML-документа, будь то объект в памяти или файл. В листинге 13.2 приведен пример использования `XmlDataProvider` для привязки к внедренному острову XML-данных:

Листинг 13.2. Привязка к внедренному острову XML-данных

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="XML Data Binding">
<Window.Resources>
  <XmlDataProvider x:Key="dataProvider" XPath="GameStats">
    <x:XData>
      <GameStats xmlns="">
        <!-- По одной статистике на каждый тип игры -->
        <GameStat Type="Beginner">
          <HighScore>1203</HighScore>
        </GameStat>
        <GameStat Type="Intermediate">
          <HighScore>1089</HighScore>
        </GameStat>
        <GameStat Type="Advanced">
          <HighScore>541</HighScore>
        </GameStat>
      </GameStats>
    </x:XData>
  </XmlDataProvider>
</Window.Resources>
  <Grid>
    <ListBox ItemsSource="{Binding Source={StaticResource dataProvider},
      XPath=GameStat/HighScore}" />
  </Grid>
</Window>
```

Остров XML-данных

Привязка к XML

В качестве значения свойства содержимого объекта `XmlDataProvider` указан остров XML-данных, который содержится внутри элемента `XData`; это условие

необходимо, чтобы его можно было отличить от окружающего XAML-кода. (Если опустить теги XData, то компилятор сообщит об ошибке.) В свойство XPath элемента XmlDataProvider записывается XPath-запрос, который сообщает, где искать нужные данные в дереве XML. Язык XPath (сокращение от XML Path Language) описан в рекомендации W3C по адресу *http://www.w3.org/TR/xpath*.

СОВЕТ

Когда остров XML-данных внедряется в XAML-разметку, его корневой узел следует помечать пустым атрибутом xmlns, как сделано в листинге 13.2. В противном случае элементы оказываются в пространстве имен, подразумеваемом по умолчанию (в данном случае http://schemas.microsoft.com/winfx/2006/xaml/presentation), в результате чего XPath-запросы работают не так, как вы ожидаете.

Получение данных от XmlDataProvider выглядит так же, как получение данных из любого другого источника, за тем исключением, что для доступа к нужным частям источника используется не свойство Path объекта Binding, а свойство XPath. В листинге 13.2 свойство XPath применяется для отображения содержимого узлов HighScore в виде элементов списка ListBox, как показано на рис. 13.12.

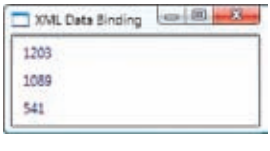


Рис. 13.12. *Результат привязки к острову XML-данных в разметке из листинга 13.2*

Если XML-документ находится в отдельном файле (как оно обычно и бывает), то можно просто записать в свойство Source элемента XmlDataProvider соответствующий унифицированный идентификатор ресурса (URI), а не устанавливать свойство содержимого. Этот URI, как и все прочие, может указывать на локальный файл, на файл в Интернете, на внедренный ресурс и т. д. В листинге 13.2 можно было бы заменить элемент XmlDataProvider таким:

```
<XmlDataProvider x:Key="dataProvider" XPath="GameStats" Source="GameStats.xml"/>
```

XPath – это язык запросов, куда более мощный, чем синтаксис задания свойств в показанных ранее привязках. Например, в листинге 13.2 можно было присвоить свойству XPath выражение "GameStat/@Type", чтобы заполнить список ListBox значениями атрибута Type каждого объекта GameStat (Beginner, Intermediate и Advanced). Можно даже использовать выражение "comment()", чтобы показать содержимое первого XML-комментария!

КОПНЕМ ГЛУБЖЕ

Взаимодействие между свойствами XPath и Path

Свойства XPath и Path можно задавать одновременно в одной и той же привязке Binding. XML-данные поставляются объектом XmlDataProvider в виде объектов, определенных в сборке System.Xml.dll (из пространства имен System.Xml), например XmlNode. Это важно иметь в виду, когда вы работаете с данными из программы, и это же означает, что свойство Path объекта Binding может ссылаться на текущий экземпляр XmlNode или XmlNodeList. Например, при работе с определенным ранее поставщиком данных свойство OuterXml объекта XmlNode в следующем элементе Label используется для отображения <HighScore>1203</HighScore>, а не просто 1203:

```
<Label Content="{Binding Source={StaticResource dataProvider},
                 XPath=GameStat/HighScore, Path=OuterXml}"/>
```

В дополнение к этой поддержке свойство DisplayMemberPath элемента ItemsControl также поддерживает синтаксис как Path, так и XPath.

Если требуется привязать целый фрагмент XML-данных к элементу, умеющему работать с иерархиями (TreeView или Menu) без написания кода, то следует использовать один или несколько шаблонов HierarchicalDataTemplates. В разметке в листинге 13.3, являющемся развитием листинга 13.2, добавлено три шаблона данных (два иерархических HierarchicalDataTemplates и один простой DataTemplate), а элемент ListBox заменен на TreeView, свойство XPath которого включает все XML-содержимое.

Листинг 13.3. Привязка к иерархии с помощью HierarchicalDataTemplate

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="XML Data Binding">
  <Window.Resources>
    <HierarchicalDataTemplate DataType="GameStats"
      ItemsSource="{Binding XPath=*}">
      <TextBlock FontStyle="Italic" Text="All Game Stats"/>
    </HierarchicalDataTemplate>
    <HierarchicalDataTemplate DataType="GameStat" ItemsSource="{Binding XPath=*}">
      <TextBlock FontWeight="Bold" FontSize="20" Text="{Binding XPath=@Type}"/>
    </HierarchicalDataTemplate>
    <DataTemplate DataType="HighScore">
      <TextBlock Foreground="Blue" Text="{Binding XPath=.}"/>
    </DataTemplate>
    <XmlDataProvider x:Key="dataProvider" XPath="GameStats">
      <x:XData>
        <GameStats xmlns="">
          <!-- По одной статистике на каждый тип игры -->
          <GameStat Type="Beginner">
            <HighScore>1203</HighScore>
          </GameStat>
```

```
            <GameStat Type="Intermediate">
              <HighScore>1089</HighScore>
            </GameStat>
            <GameStat Type="Advanced">
              <HighScore>541</HighScore>
            </GameStat>
          </GameStats>
        </x:XData>
      </XmlDataProvider>
    </Window.Resources>
    <Grid>
      <TreeView ItemsSource="{Binding Source={StaticResource dataProvider},
        XPath=.}" />
    </Grid>
  </Window>
```

Идея в том, чтобы применить шаблон `HierarchicalDataTemplate` для каждого типа данных в иерархии, а потом воспользоваться простым шаблоном `DataTemplate` для всех листовых узлов. Любой шаблон данных дает возможность настроить визуализацию типа данных, но `HierarchicalDataTemplate` еще и позволяет определить своих потомков в иерархии с помощью свойства `ItemsSource`. Оба шаблона `HierarchicalDataTemplates` в листинге 13.3 привязывают свое свойство `ItemsSource` к выражению `XPath *`, что означает включение всех потомков из источника XML-данных.

Значение `DataType` в каждом шаблоне данных автоматически применяет его ко всем экземплярам указанного типа в области действия шаблона (в данном случае это элемент `Window`). При использовании совместно с `XmlDataProvider` значение `DataType` соответствует имени элемента XML. Отметим, что ни в одном из трех шаблонов не указаны явные ключи, хотя они и находятся в словаре ресурсов `ResourceDictionary`. Это работает потому, что в качестве ключа шаблона неявно используется значение атрибута `DataType`.

На рис. 13.13 показана визуализация XAML-разметки из листинга 13.3. Также показано, что произойдет, если заменить элемент `TreeView` на `Menu`, а все остальное оставить как есть.



`TreeView` в листинге 13.3



После замены `TreeView` на `Menu`

***Рис. 13.13.** Использование шаблонов HierarchicalDataTemplate позволяет автоматически заполнить элементы TreeView и Menu иерархией объектов, привязанных к данным*

СОВЕТ

Часто в XML-данных пространство имен для элементов уже определено. Например, в RSS-лентах от Twitter определены даже два пространства имен:

```
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom"
                   xmlns:georss="http://www.georss.org/georss">
  ...
</rss>
```

Чтобы сослаться на элементы в этих пространствах имен (например, `atom:link`) из запроса в свойстве `XPath`, можно задать свойство `XmlNamespaceManager` поставщика данных `XmlDataProvider` или конкретной привязки `Binding`. Например:

```
<XmlDataProvider Source="http://twitter.com/statuses/user_timeline/24326956.rss"
  XmlNamespaceManager="{StaticResource namespaceMapping}"
  XPath="rss/channel" x:Key="dataProvider"/>
```

Наиболее распространенный способ получить экземпляр `XmlNamespaceManager` состоит в том, чтобы воспользоваться производным классом `XmlNamespaceMappingCollection`, который назначает префикс каждому пространству имен. Например:

```
<XmlNamespaceMappingCollection x:Key="namespaceMapping">
  <XmlNamespaceMapping Uri="http://www.w3.org/2005/Atom" Prefix="atom"/>
  <XmlNamespaceMapping Uri="http://www.georss.org/georss" Prefix="georss"/>
</XmlNamespaceMappingCollection>
```

Лучше выбирать префиксы так, чтобы они соответствовали используемым в XML, но это необязательно – можете задавать любые префиксы, которые вам нравятся. Выбранные префиксы можно использовать в выражениях XPath, например:

```
"{Binding XPath=atom:link}"
```

Если в значении свойства `XPath` префикс отсутствует, то подразумевается пустое пространство имен. Таким образом, даже если в XML-источнике определено пространство имен по умолчанию, все равно нужно назначить `XmlNamespaceManager`, иначе запросы не будут работать.

Класс ObjectDataProvider

Если класс `XmlDataProvider` предоставляет в качестве источника данных XML-документ, то `ObjectDataProvider` раскрывает в качестве такового объект .NET. «Но это же бессмысленно! – воскликнете вы. – Я и так могу сделать произвольный объект .NET источником данных. Что добавляет `ObjectDataProvider`?» А добавляет он целый ряд возможностей, которых нет в случае прямой привязки к объекту. Например, он позволяет:

- Декларативно создавать объект-источник с помощью конструктора с параметрами
- Осуществлять привязку к методу объекта-источника
- Реализовывать асинхронную привязку к данным

КОПНЕМ ГЛУБЖЕ

Асинхронная привязка к данным

Если привязка к данным оказывается медленной операцией, то ее следует производить асинхронно, чтобы не «подвешивать» пользовательский интерфейс. WPF предлагает два независимых механизма асинхронной привязки: в классе `Binding` есть свойство `IsAsync`, а в классах `XmlDataProvider` и `ObjectDataProvider` — свойство `IsAsynchronous`. (Какая прелесть эта согласованность, не правда ли?)

Если `IsAsynchronous` равно `true`, то поставщик данных создает объект-источник в фоновом потоке. По умолчанию `IsAsynchronous` равно `false` для `ObjectDataProvider`, но `true` для `XmlDataProvider` (потому что последний часто используется для привязки к удаленным XML-файлам, например RSS-лентам, обработка которых происходит медленно). С другой стороны, если свойство `IsAsync` (по умолчанию всегда равное `false`) равно `true`, то в фоновом потоке производится доступ к *свойству*-источнику.

Свойство `Binding.IsAsync` предназначено для того, чтобы приложение могло более-менее эффективно работать с плохо спроектированными объектами. Методы чтения свойств предназначены для быстрых операций: в них не следует выполнять длительные вычисления, обращаться с запросами в сеть и т. д. Если бы все придерживались этих рекомендаций, то разработчикам WPF не пришлось бы создавать свойство `IsAsync`.

Если у вас возникает искушение использовать `IsAsync` со своими собственными объектами, подумайте лучше, как переделать медленно работающие свойства. Например, можно порекомендовать завести метод `Recompute`, который будет выполнять длительные вычисления (быть может, в рабочем потоке) и кэшировать результаты. Когда он закончится, можно сгенерировать соответствующие события `PropertyChanged`. В этом случае аксессору свойства останется только прочитать значение из кэша, так что он всегда будет работать быстро.

Использование в XAML конструктора с параметрами

У большинства источников данных, с которыми вам приходилось работать, скорее всего, имеется конструктор по умолчанию. Есть такой и у коллекции `photos`, рассмотренной в этой главе выше. Следующий XAML-код «обертывает» эту коллекцию объектом `ObjectDataProvider`:

```
<Window.Resources>
  <local:Photos x:Key="photos"/>
  <ObjectDataProvider x:Key="dataProvider"
    ObjectInstance="{StaticResource photos}"/>
</Window.Resources>
```

В данном случае безразлично, к чему привязываться: к `photos` или к `dataProvider`. Результат будет один и тот же. Даже свойство `Path` объекта `Binding` не меняется, потому что `Binding` автоматически «разворачивает» объекты, хранящиеся внутри таких поставщиков данных, как `ObjectDataProvider`.

Поставщику ObjectDataProvider можно также передать тип объекта, который он должен обертывать (вместо самого объекта), и попросить сконструировать объект этого типа от вашего имени:

```
<Window.Resources>
  <!-- ObjectDataProvider сам создает объект коллекции: -->
  <ObjectDataProvider x:Key="dataProvider" ObjectType="{x:Type local:Photos}"/>
</Window.Resources>
```

Если ObjectDataProvider используется таким образом, то создавать объект можно с помощью конструктора с параметрами. Для этого нужно лишь присвоить свойству ConstructorParameters ссылку на коллекцию объектов. Например, если бы конструктор класса Photos требовал задавать объем, то ObjectDataProvider можно было бы использовать следующим образом:

```
<ObjectDataProvider x:Key="dataProvider" ObjectType="{x:Type local:Photos}">
<ObjectDataProvider.ConstructorParameters>
  <sys:Int32>23</sys:Int32>
</ObjectDataProvider.ConstructorParameters>
</ObjectDataProvider>
```

Этот механизм очень похож на использование ключевого слова x:Arguments в XAML2009, но в XAML2006 он тоже работает. Следовательно, он весьма полезен для тех источников данных, определение которых вы не можете контролировать. (Если бы вы контролировали определение источника данных, то, скорее всего, добавили бы подходящий конструктор по умолчанию.) Конечно, если вы согласны обойтись без объявления источника в XAML-коде, то всегда можно сконструировать его программно и установить в качестве контекста данных для элементов, определенных в XAML.

Привязка к методу

Одно из решений, когда ObjectDataProvider позволяет добиться результата там, где декларативная и даже программная реализация затруднительны, – это привязка к методу. Как и поддержка конструкторов с параметрами, эта возможность находит применение в основном для уже написанных классов, которые разработаны без учета привязки к данным и не могут быть изменены. В своих собственных типах вполне можно было бы раскрыть потенциальные источники данных в виде свойств. Но давайте предположим, что в коллекции photos имеется метод GetFolderName, который возвращает строку, представляющую папку с фотографиями. Превратить этот метод в источник данных можно следующим образом:

```
<ObjectDataProvider x:Key="dataProvider" ObjectType="{x:Type local:Photos}"
  MethodName="GetFolderName"/>
```

Если методу необходимо передать параметры, то можно воспользоваться свойством MethodParameters объекта ObjectDataProvider (оно работает точно так же, как свойство ConstructorParameters). Чтобы привязаться к этому методу, нужно просто выполнить привязку ко всему объекту ObjectDataProvider:

```
<TextBlock Text="{Binding Source={StaticResource dataProvider}}"/>
```

Задание свойства Path в этом случае означало бы применение к объекту, возвращенному методом.

КОПНЕМ ГЛУБЖЕ

Подавление автоматического разворачивания поставщиков данных

Если вы хотите осуществлять привязку непосредственно к свойствам объекта ObjectDataProvider, а не к свойствам обернутого им источника данных, то можете присвоить значение true свойству BindsDirectlyToSource объекта Binding и тем самым подавить автоматическое разворачивание. Этот способ работает для любого источника, производного от класса DataSourceProvider (или от класса CollectionViewSource), то есть для ObjectDataProvider, XmlDataProvider и любых производных классов, написанных вами самостоятельно.

Дополнительные вопросы

В последнем разделе этой главы мы рассмотрим некоторые экзотические, но тем не менее чрезвычайно полезные возможности привязки к данным. Сюда мы отнесли настройку потока данных между источником и приемником, подключение нестандартной логики проверки и объединение разных источников в единую сущность, допускающую привязку.

Настройка потока данных

Во всех рассмотренных до сих пор примерах привязки к данным поток обновлений направлен от источника к приемнику. Но в некоторых случаях пользователь может напрямую изменить свойство-приемник, и хорошо бы, чтобы такое изменение распространялось назад к источнику. Класс Binding поддерживает этот и другие режимы с помощью свойства Mode, которое может принимать следующие значения, определенные в перечислении BindingMode:

- OneWay – приемник обновляется при изменении источника.
- TwoWay – изменение в источнике или приемнике отражается на противоположной стороне.
- OneWayToSource – этот режим противоположен OneWay. Источник обновляется, когда изменяется приемник.
- OneTime – работает, как OneWay с тем отличием, что изменения источника не отражаются на приемнике. Приемник сохраняет мгновенный снимок источника в момент инициализации привязки.

Режим TwoWay хорош для редактируемых сеток DataGrid и других привязанных к данным форм, где имеются элементы TextBox, заполненные данными, которые пользователю разрешено изменять. На самом деле большинство свойств зависимости по умолчанию привязываются в режиме OneWay, но такие свойства зависимости, как TextBox.Text, привязываются в режиме TwoWay. (Хотя в начале этого раздела мы сказали, что речь пойдет об экзотических воз-

можностях, привязка в режиме TwoWay весьма распространена. Она используется почти во всех приложениях, которые реагируют на ввод данных пользователем и стремятся отделить пользовательский интерфейс от отображаемых в нем данных.)

ПРЕДУПРЕЖДЕНИЕ

Следите за режимом BindingMode, подразумеваемым по умолчанию!

Тот факт, что для разных свойств зависимости подразумеваемый по умолчанию режим BindingMode различен, может стать причиной недоразумений. Например, в отличие от свойства Label.Content, привязка TextBox.Text к свойству Count коллекции приведет к ошибке, если явно не установить режим BindingMode OneWay (или OneTime), потому что свойство Count допускает только чтение, а режимы TwoWay и OneWayToSource требуют, чтобы свойство-источник можно было изменять.

Именно наличием разных режимов объясняется то, что у конвертеров значений есть два метода: Convert и ConvertBack. В случае двусторонней (TwoWay) привязки вызываются оба, а в случае привязки OneWayToSource – только ConvertBack.

FAQ

Зачем может понадобиться привязка в режиме OneWayToSource? Получается, что на самом деле приемник должен быть источником, а источник – приемником.

Одна из причин заключается в том, что может быть несколько привязок, причем в одних поток данных направлен от источника к приемнику, а в других – от приемника к источнику. Например, к одному и тому же источнику может быть привязано несколько приемников, из которых лишь одному разрешено обновлять источник с помощью механизма привязки к данным.

Режим OneWayToSource может также применяться как хитроумный способ обойти ограничение, согласно которому свойство-приемник обязано быть свойством зависимости. Если требуется привязать свойство-источник, являющееся свойством зависимости, к свойству-приемнику, которое свойством зависимости *не является*, то режим OneWayToSource позволяет это сделать, пометив «реальный источник» как приемник, а «реальный приемник» – как источник!

При использовании привязки в режимах TwoWay или OneWayToSource иногда требуется точнее указать, когда и как обновляется источник. Например, если пользователь вводит данные в поле TextBox, привязанное в режиме TwoWay, то когда следует обновлять источник: при каждом нажатии клавиши или после завершения ввода? Класс Binding позволяет управлять этими аспектами поведения с помощью свойства UpdateSourceTrigger.

Свойство UpdateSourceTrigger может принимать следующие значения, определенные в перечислении UpdateSourceTrigger:

- PropertyChanged – источник обновляется при каждом изменении свойства-приемника.
- LostFocus – источник обновляется только после того, как элемент-приемник теряет фокус (при условии, что его значение изменилось).
- Explicit – источник обновляется в результате явного вызова метода BindingExpression.UpdateSource. Экземпляр объекта BindingExpression можно получить, обратившись к статическому методу BindingOperations.GetBindingExpression либо к методу GetBindingExpression объекта любого класса, производного от FrameworkElement или FrameworkContentElement.

Подразумеваемые по умолчанию значения UpdateSourceTrigger различны для разных свойств точно так же, как значения Mode. Так, для свойства TextBox.Text по умолчанию подразумевается режим LostFocus.

КОПНЕМ ГЛУБЖЕ

Свойства зависимости и настройки по умолчанию

Настройки по умолчанию для свойств зависимости хранятся в специальном наборе метаданных, о чем мы говорили в главе 3. Чтобы проверить настройку любого свойства зависимости из программы, можно вызвать его метод GetMetadata (например, TextBox.TextProperty.GetMetadata()), а затем опросить значение нужного свойства, скажем BindsTwoWayByDefault или DefaultUpdateSourceTrigger. Но, конечно, проще всего получить эту информацию, воспользовавшись такой программой, как .NET Reflector.

СОВЕТ

Хотя данные в источнике и/или приемнике при использовании привязки к данным обновляются автоматически, иногда желательно предпринять дополнительные действия в момент обновления. Быть может, записать что-то в журнал или визуально подтвердить, что обновление произошло.

К счастью, в классах FrameworkElement и FrameworkContentElement определены события SourceUpdated и TargetUpdated, которые можно обработать. Однако из соображений производительности они генерируются, только если булевскому свойству NotifyOnSourceUpdated (или NotifyOnTargetUpdated) объекта Binding присвоено значение true.

Добавление в привязку правил проверки

Когда пользователь вводит данные, желательно своевременно проверять их и сообщать об ошибках. Когда заполнение веб-форм только еще входило в моду, случались ужасные казусы – ошибки обнаруживались лишь после того, как

форма была отправлена на сервер, и пользователю приходилось вводить все заново! К счастью, технология привязки к данным снабжена встроенным механизмом проверки, который позволяет сравнительно легко реализовать удобный интерфейс с высокой степенью интерактивности. Но способов сделать это и соответствующих средств конфигурирования так много, что они способны затруднить работу.

Допустим, вы хотите, чтобы пользователь мог вводить имя существующего JPG-файла в привязанное к данным поле `TextBox`. Могут возникнуть две очевидные ошибки: введено имя несуществующего файла или расширение файла отлично от *.jpg*. Если бы поле `TextBox` не было привязано к данным, то никто не мешал бы написать код, который проверит оба условия перед тем, как обновлять источник данных. Но коль скоро механизм привязки к данным распространяет обновления автоматически, необходим какой-то способ включить проверку в этот процесс. Можно было бы написать конвертер значений, который проверяет данные и возбуждает исключение, если они некорректны. Но даже если не обращать внимания на тот факт, что конвертеры значений предназначены совсем для другого, все равно остается нерешенной проблема показа сообщения об ошибке пользователю.

В этой ситуации можно действовать несколькими способами. Во-первых, вы можете написать свое правило проверки, а во-вторых, – воспользоваться исключениями, которые, возможно, и так возбуждаются при попытке некорректного обновления источника.

СОВЕТ

Описанная в этом разделе техника применима только к распространению изменения от приемника к источнику. Иначе говоря, она работает лишь в том случае, если свойство `BindingMode` равно `OneWayToSource` или `TwoWay`.

Написание собственного правила проверки

В классе `Binding` имеется свойство `ValidationRules`, в котором можно сохранить один или несколько объектов, производных от класса `ValidationRule`. Каждое правило может проверить выполнение каких-то условий и пометить, что данные некорректны. Так, чтобы гарантированно соблюдались описанные выше требования к введенному имени файла, можно написать класс `JpgValidationRule`, производный от `ValidationRule`, переопределив в нем абстрактный метод `Validate`:

```
public class JpgValidationRule : ValidationRule
{
  public override ValidationResult Validate(object value, CultureInfo cultureInfo)
  {
    string filename = value.ToString();
```

```
        // Отвергаем несуществующие файлы:
        if (!File.Exists(filename))
            return new ValidationResult(false, "Value is not a valid file.");

        // Отвергаем файлы с расширением, отличным от .jpg:
        if (!filename.EndsWith(".jpg", StringComparison.InvariantCultureIgnoreCase))
            return new ValidationResult(false, "Value is not a .jpg file.");

        // Данные прошли все проверки!
        return new ValidationResult(true, null);
    }
}
```

О том, что данные некорректны, мы сообщаем, присваивая значение `false` объекту `ValidationResult`; если же все правильно, то этот объект получит значение `true`. (Сравнение расширения имени файла со строкой `".jpg"` – не самый лучший способ убедиться, что файл действительно содержит изображение в формате JPEG, но идея тем не менее понятна.)

Теперь этот класс можно следующим образом указать в элементе `Binding`:

```
<TextBox>
<TextBox.Text>
  <Binding ...>
  <Binding.ValidationRules>
     <local:JpgValidationRule/>
   </Binding.ValidationRules>
  </Binding>
</TextBox.Text>
</TextBox>
```

Проверка будет выполняться при любой попытке обновить информацию (в данном случае это произойдет в момент, когда `TextBox` теряет фокус, потому что мы присвоили свойству `UpdateSourceTrigger` значение `LostFocus`). Это происходит еще до вызова конвертера значений (если он присутствует), и для запрета обновления достаточно одного правила.

А что происходит, когда данные помечены как некорректные? Поверх элемента с недопустимым значением свойства-приемника рисуется индикатор ошибки. По умолчанию он выглядит как тонкая красная рамка. Однако можно указать специальный шаблон для использования в подобных случаях – нужно лишь записать ссылку на него в свойство `Validation.ErrorTemplate`, присоединенное к элементу-приемнику. (Шаблоны элементов управления обсуждаются в следующей главе.) Если вы будете пользоваться механизмом проверки, то, наверное, захотите разработать свой шаблон, потому что подразумеваемый по умолчанию не слишком хорош.

Кроме того, если данные помечены как некорректные, то свойство `Validation.HasError`, присоединенное к элементу-приемнику, становится равным `true` и генерируется присоединенное событие `Validation.Error` (но только в случае, когда свойство `NotifyOnValidationError` объекта `Binding` равно `true`). Таким обра-

зом, можно реализовать логику уведомления с помощью как триггера, так и обработчика события. Получить детальную информацию о найденных в ходе проверки ошибках, например строки сообщений, возвращенные классом JpgValidationRule, позволяет присоединенное к элементу-приемнику свойство Validation.Errors. Оно автоматически очищается, если при последующей привязке ошибок не будет.

Передача уже имеющейся обработки ошибок через систему проверки

Возможно, что источник данных (или конвертер значений) уже обрабатывает ошибки, тогда написание правила проверки было бы дублированием усилий. Если какой-то из вышеупомянутых компонентов возбуждает исключение при тех же условиях, которые вы считаете ошибочными, то можно воспользоваться встроенным объектом ExceptionValidationRule. Например:

```
<TextBox>
<TextBox.Text>
  <Binding ...>
  <Binding.ValidationRules>
    <ExceptionValidationRule/>
  </Binding.ValidationRules>
  </Binding>
</TextBox.Text>
</TextBox>
```

Объект ExceptionValidationRule просто помечает данные как некорректные, если при попытке обновить свойство-источник возникло любое исключение. Следовательно, этот механизм позволяет адекватно отреагировать на исключение, а не «глотать» его, оставляя следы только в отладочной трассировке.

Аналогично, если источник данных предоставляет информацию об ошибке путем реализации простого интерфейса System.ComponentModel.IDataErrorInfo (поддерживается несколькими источниками и находит применение также в Windows Forms), то можно воспользоваться встроенным объектом DataErrorValidationRule и с его помощью пометить данные как некорректные. В следующем элементе TextBox задействованы оба способа:

```
<TextBox>
<TextBox.Text>
  <Binding ...>
  <Binding.ValidationRules>
    <ExceptionValidationRule/>
    <DataErrorValidationRule/>
  </Binding.ValidationRules>
  </Binding>
</TextBox.Text>
</TextBox>
```

Хотя приятно видеть, что встроенная проверка пользуется теми же механизмами, которые вы применили бы в написанном вручную коде, разработчики

WPF пришли к выводу, что этот синтаксис слишком многословный и громоздкий. Поэтому в версии WPF 3.5 SP1 в класс Binding были добавлены два новых булевских свойства – ValidatesOnExceptions и ValidatesOnDataErrors, – которые позволяют добавить в коллекцию ValidationRules те же самые правила проверки, но более лаконично. Таким образом, предыдущий XAML-код можно переписать и так:

```
<TextBox>
<TextBox.Text>
  <Binding ValidatesOnExceptions="True" ValidatesOnDataErrors="True" .../>
</TextBox.Text>
</TextBox>
```

КОПНЕМ ГЛУБЖЕ

Существует несколько способов обработки исключений

Еще один способ обработки исключений, возбуждаемых при обновлении источника, заключается в том, чтобы присоединить делегат к свойству UpdateSourceExceptionFilter объекта Binding. Этот делегат вызывается, когда попытка обновить свойство-источник приводит к исключению, при этом в качестве аргумента делегату передается сам объект Exception. Таким образом можно реализовать собственную схему уведомления об ошибках, вообще не прибегая к правилам ValidationRule. Пользоваться свойством UpdateSourceExceptionFilter из программы, наверное, проще, но для декларативной разметки пригоден только подход на основе ExceptionValidationRule.

Интересно, что все же существует связь между делегатом UpdateSourceExceptionFilter и другими схемами проверки. Если делегат возвращает объект ValidationError, то система будет рассматривать этот делегат как правило проверки и добавит полученный объект ValidationError в коллекцию Validation.Errors объекта-приемника, установит для свойства Validation.HasError значение true и, возможно, сгенерирует событие Validation.Error.

Подведем итоги: если источник данных или конвертер значений уже возбуждает исключения при обнаружении некорректных данных, то можно поступить одним из следующих способов:

- Воспользоваться делегатом UpdateSourceExceptionFilter для реализации собственной логики уведомления об ошибках.
- Установить свойство ValidatesOnExceptions или воспользоваться встроенным объектом ExceptionValidationRule, определить шаблон ErrorTemplate и/или подключить дополнительную логику, опрашивая свойство Validation.HasError либо обрабатывая событие Validation.Error (если свойство NotifyOnValidationError равно true).

Если источник данных реализует интерфейс IDataErrorInfo, то можно вместо этого установить свойство ValidatesOnDataErrors или воспользоваться встроенным объектом DataErrorValidationRule. Но даже если ни источник данных, ни

конвертер значений не обрабатывают ошибки, все равно можно добавить свое правило проверки.

Проверка для группы привязок Binding

Описанный выше механизм проверки применяется к каждой привязке индивидуально, но иногда полезно выполнить «коллективную» проверку, например для всех ячеек в строке DataGrid или какой-то иной форме, в которой есть несколько зависящих друг от друга значений.

Такая коллективная проверка поддерживается с помощью объекта BindingGroup. Как и в случае Binding, этому объекту можно передать набор правил ValidationRule, которые должны применяться к группе привязок Binding. Экземпляр BindingGroup можно связать с любым элементом FrameworkElement (или FrameworkContentElement), записав его в свойство BindingGroup (в классе ItemsControl определено также свойство ItemBindingGroup, которое применяется к каждому хранимому объекту, а не к самому элементу ItemsControl). В результате в группу BindingGroup автоматически попадают все объекты Binding, разделяющие тот же контекст данных DataContext, что и элемент-владелец. Если задать для группы BindingGroup свойство Name, то в нее можно будет добавлять и другие объекты Binding – вне зависимости от источника данных; достаточно записать в свойство BindingGroupName каждого такого объекта имя Name группы.

При вызове каждого правила ValidationRule, ассоциированного с группой BindingGroup, методу Validate в качестве значения передается данный экземпляр BindingGroup. Поскольку в классе BindingGroup имеется целый ряд полезных методов и свойств, например коллекция Items, которая содержит все объекты Binding в данной группе, то в правиле ValidationRule можно реализовать произвольную логику для выяснения того, является ли допустимой группа значений в целом. Класс BindingGroup поддерживает также транзакционное редактирование (и это используется в DataGrid), если источник данных реализует интерфейс IEditableObject.

Работа с несколькими источниками

WPF предлагает ряд интересных способов объединения нескольких источников данных. В основе этого механизма лежат следующие классы:

- CompositeCollection
- MultiBinding
- PriorityBinding

Класс CompositeCollection

Класс CompositeCollection дает простой способ представить не связанные между собой коллекции и/или произвольные объекты в виде одной коллекции. Это бывает полезно, когда нужно осуществить привязку к коллекции объектов, поступающих из нескольких источников. В XAML-разметке ниже определена составная коллекция CompositeCollection, содержащая все, что хранится в коллекции photos, и еще два объекта:

```
<CompositeCollection>
  <CollectionContainer Collection="{Binding Source={StaticResource photos}}"/>
  <local:Photo .../>
  <local:Photo .../>
</CompositeCollection>
```

Коллекция photos обернута объектом CollectionContainer, чтобы частью CompositeCollection считались находящиеся в ней объекты, а не сама коллекция. Если бы мы добавили коллекцию photos непосредственно в CompositeCollection, то последняя содержала бы всего три объекта!

Класс MultiBinding

Класс MultiBinding позволяет агрегировать несколько объектов Binding, так что приемник будет получать единственное значение. Для этого необходим конвертер значений, потому что иначе WPF не будет знать, как объединить несколько входных значений. В следующем фрагменте XAML показано, как с помощью MultiBinding вычислить суммарное значение, отображаемое индикатором ProgressBar, когда есть три независимых источника данных, описанных в виде ресурсов, и конвертер значений:

```
<ProgressBar ...>
<ProgressBar.Value>
  <MultiBinding Converter="{StaticResource converter}">
    <Binding Source="{StaticResource worker1}"/>
    <Binding Source="{StaticResource worker2}"/>
    <Binding Source="{StaticResource worker3}"/>
  </MultiBinding>
</ProgressBar.Value>
</ProgressBar>
```

Однако конвертеры значений для MultiBinding пишутся несколько иначе, чем для Binding. Они должны реализовывать интерфейс IMultiValueConverter, методы которого принимают и возвращают не одно значение, а целый *массив*. Вот как мог бы выглядеть конвертер значений для показанной выше разметки:

```
public class ProgressConverter : IMultiValueConverter
{
  public object Convert(object[] values, Type targetType,
    object parameter, CultureInfo culture)
  {
    int totalProgress = 0;

    // Требуется, чтобы каждое поданное на вход значение
    // было экземпляром класса Worker
    foreach (Worker worker in values)
      totalProgress += worker.Progress;

    return totalProgress;
  }
  public object[] ConvertBack(object value, Type[] targetTypes,
    object parameter, CultureInfo culture)
```

```
    {
      return DependencyProperty.UnsetValue;
    }
  }
```

СОВЕТ

Совместно с `MultiBinding` можно использовать метод `StringFormat`. В этом случае `{0}` будет представлять первый объект `Binding`, `{1}` – второй и т. д.

Класс PriorityBinding

Класс `PriorityBinding` похож на `MultiBinding` тем, что инкапсулирует несколько объектов `Binding`. Но вместо того чтобы агрегировать их, этот класс организует *состязание* нескольких привязок за право установить значение свойства-приемника!

Если привязка осуществляется к медленному источнику данных (и не в ваших силах ускорить его работу), то можно подключить более быстрые источники, которые дадут «приближенную» версию данных, пока программа ожидает поступления точной информации. Такая техника применяется во многих программах. Например, при открытии большого документа в Microsoft Word сначала в течение нескольких секунд в левом нижнем углу окна показывается сообщение типа «Приблизительное число знаков: 77257», потом нечто вроде «Стр. 1/3» (общее число страниц еще неправильное) и наконец долгожданное «Стр. 1/46». В приложении Photo Gallery можно было бы сначала выполнить быструю привязку к коллекции уменьшенных эскизов, а впоследствии – когда завершится более медленная операция – заменить ее коллекцией полноразмерных фотографий.

В следующем XAML-коде показано типичное объявление элемента `PriorityBinding`:

```
<PriorityBinding>
  <Binding Source="HighPri" Path="SlowSpeed" IsAsync="True"/>
  <Binding Source="MediumPri" Path="MediumSpeed" IsAsync="True"/>
  <Binding Source="LowPri" Path="FastSpeed"/>
</PriorityBinding>
```

Привязки `Binding` обрабатываются в порядке следования, то есть первая привязка в списке имеет высший приоритет (и, значит, будет самой медленной), а последняя – низший приоритет (и, стало быть, является самой быстрой). По мере получения значений от разных привязок более приоритетные заменяют менее приоритетные.

СОВЕТ

При использовании элемента `PriorityBinding` **для всех привязок** `Binding`**, кроме последней, необходимо присваивать свойству** `IsAsync` **значение** `true`**, чтобы они обрабатывались в фоновом режиме. В противном случае наиболее приоритетная привязка будет выполняться синхронно (возможно, «подвешивая» пользовательский интерфейс), а после ее завершения результаты низкоприоритетных привязок уже не будут представлять никакого интереса!**

А теперь все вместе: клиент Twitter на чистом XAML

Канонический пример мощи механизма привязки к данным в WPF – полнофункциональная программа чтения RSS-лент, не содержащая ни единой строчки процедурного кода. В листинге 13.4 приведена моя версия такой программы, сконфигурированная для чтения RSS-ленты сайта Twitter. В результате получился вполне пристойный «клиент Twitter», показанный на рис. 13.14. Я скопировал XAML-код в очень удобный инструмент Kaxaml (*http://kaxaml.com*), поэтому в строке заголовка окна отображается его значок.

Листинг 13.4. Полная реализация программы чтения RSS-лент (клиент Twitter)

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="RSS Reader">
<Window.Resources>
  <XmlDataProvider x:Key="Feed"
    Source="http://twitter.com/statuses/user_timeline/24326956.rss"/>
</Window.Resources>
  <DockPanel
    DataContext="{Binding Source={StaticResource Feed},
                XPath=/rss/channel/item}">
    <TextBox DockPanel.Dock="Top" Text="{Binding Source={StaticResource Feed},
      BindsDirectlyToSource=true, Path=Source,
      UpdateSourceTrigger=PropertyChanged}"/>
    <Label DockPanel.Dock="Top" Content="{Binding XPath=/rss/channel/title}"
      FontSize="14" FontWeight="Bold"/>
    <Label DockPanel.Dock="Top"
      Content="{Binding XPath=/rss/channel/description}"/>
    <ListBox DockPanel.Dock="Left" DisplayMemberPath="title"
      ItemsSource="{Binding}" IsSynchronizedWithCurrentItem="True" Width="300"/>
    <Frame Source="{Binding XPath=link}"/>
  </DockPanel>
</Window>
```

Как и следовало ожидать, для получения RSS-ленты используется поставщик данных `XmlDataProvider`**.**

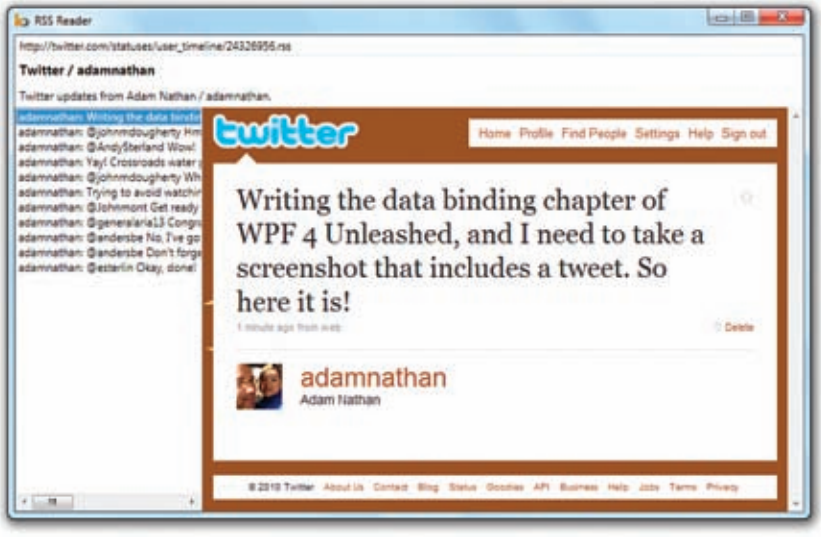


***Рис. 13.14.** Так выглядит клиент, читающий RSS-ленты Twitter, написанный на чистом XAML*

Отметим некоторые интересные особенности решения:

- Первоначально свойство `Text` поля ввода `TextBox` привязывается к свойству `Source` поставщика данных `XmlDataProvider`. При этом используется подразумеваемая по умолчанию двусторонняя привязка, чтобы пользователь в любой момент мог изменить адрес источника.
- Чтобы привязка осуществлялась именно к полю `Source` поставщика данных, в объекте `Binding` для `TextBox` свойству `BindsDirectlyToSource` присвоено значение `true`. В противном случае его свойство `Path` указывало бы на RSS-ленту, а это неправильно.
- В заданном для `TextBox` объекте `Binding` свойство `UpdateSourceTrigger` равно `PropertyChanged`, поэтому обновление данных будет производиться при нажатии каждой клавиши. Наверное, было бы лучше задать режим `UpdateSourceTrigger=Explicit` и добавить кнопку Go (Перейти), чтобы источник можно было обновлять явно. Но тогда потребовалось бы написать одну строчку процедурного кода, а это противоречит идее примера!
- Значением свойства `DisplayMemberPath` элемента `ListBox` является выражение XPath, позволяющее извлекать элемент `title` из каждой статьи ленты, представленной в формате XML.
- Совместно элементы `ListBox` и `Frame` образуют пару главный/подчиненный с общим источником данных.
- Можно было не использовать `Frame`, а отобразить исходное содержимое каждой RSS-статьи в чем-то типа `TextBlock`. Но такую содержащую HTML-теги разметку было бы трудно читать. И не существует никакого иного декла-

ративного способа правильно визуализировать HTML, кроме как воспользоваться элементом `Frame` или `WebBrowser` с указанием URL-адреса файла (его нам любезно предоставляет элемент `link`, входящий в состав статьи).

- При выборе новой статьи в ленте (или вообще другой ленты) имеющиеся во фрейме кнопки навигации автоматически запоминают ваши действия.

Резюме

Привязка к данным – исключительно мощный механизм, хотя его использование необязательно. В конце концов, не так уж трудно написать код, связывающий между собой два объекта. Но это может занять много времени, возможны ошибки, а сопровождать такой код будет сложно, особенно если требуется управлять несколькими источниками данных, которые необходимо синхронизировать при добавлении, удалении и изменении данных. Подобный код зачастую ведет к тесной связи между бизнес-логикой и пользовательским интерфейсом, что делает программу более хрупкой.

Класс `XmlDataProvider` можно считать квинтэссенцией привязки к данным, потому что благодаря ему извлечение, разбор, навигация и отображение удаленных XML-данных осуществляется как никогда просто. Возможность добиться асинхронного поведения любого объекта `Binding` или поставщика данных путем простой установки булевского свойства также делает привязку к данным соблазнительной альтернативой кодированию вручную.

Но привязка к данным не сводится к одному лишь сокращению объема написанного вручную кода. Привлекательность привязки к данным в WPF в немалой степени обусловлена еще и тем, что значительную ее часть можно реализовать декларативно. Отсюда вытекают важные следствия. Такие инструменты конструирования, как Expression Blend, могут воспользоваться возможностями привязки к данным (и делают это), так что даже неопытный программист сумеет наделить пользовательский интерфейс весьма развитой функциональностью. Наличие такой поддержки в Blend также позволяет проектировщикам определить легко заменяемые фиктивные данные для тестирования пользовательских интерфейсов с привязкой к данным. Этот же механизм наделяет автономные XAML-страницы, в которых использование процедурного кода недопустимо, возможностью задействовать такие средства, благодаря которым страницы становятся похожи скорее на миниатюрные приложения, чем на документы.

- Стили
- Шаблоны
- Обложки
- Темы

14

Стили, шаблоны, обложки и темы

Пожалуй, самой знаменитой особенностью WPF является возможность придавать любому элементу пользовательского интерфейса радикально новый внешний облик, не изменяя его поведение. Даже при наличии каскадных таблиц стилей (CSS) языку HTML далеко до такой гибкости, и именно поэтому на многих сайтах для представления кнопок применяют графические изображения, а не «настоящие кнопки». Разумеется в HTML довольно просто имитировать поведение кнопки с помощью изображения, но как быть, если нужно кардинально изменить вид элемента SELECT (HTML-аналога)? Потребуется изрядно поработать, если вы хотите добиться чего-то большего, чем изменение простых свойств (например, цвета текста и фона).

В этой главе мы расскажем о четырех краеугольных камнях, на которых покоится поддержка стилизации в WPF:

- **Стили** – простой механизм отделения значений свойств от элементов пользовательского интерфейса (аналогичный взаимоотношениям CSS и HTML). Стили также лежат в основе других механизмов, описанных в этой главе.
- **Шаблоны** – наделенные широкими возможностями объекты, которые обычно имеют в виду, говоря о «модификации внешнего вида» в WPF.
- **Обложки** – специфические для конкретного приложения коллекции стилей и/или шаблонов, которые обычно можно динамически заменять.
- **Темы** – визуальные характеристики операционной системы, которые потенциально могут настраиваться пользователем.

Как будет показано ниже, механизм применения стилей в WPF во многом опирается на семантику ресурсов.

FAQ

Зачем WPF позволяет полностью изменять внешний вид стандартных элементов управления? Ведь несогласованность между разными приложениями только запутывает пользователей!

Эта «знаменитая» особенность WPF заставляет многих нервничать. Станет ли такая гибкость провозвестником новой эры прекрасных программ или будет использована во вред, лишь раздражая и обескураживая пользователей (как приснопамятный элемент `BLINK` в HTML)?

Конечно, и то и другое. Я и сам скептически отнесся к WPF в далеком 2003 году, когда демонстрации по большей части состояли из подпрыгивающих кнопок и крутящихся списков, раскрашенных во все цвета радуги. Полезно, однако, знать, что такие безумные интерфейсы создать *можно*, хотя вряд ли *нужно*. Философия WPF заключается в том, что разнообразие выразительных средств приложения должно быть ограничено только навыками его проектировщиков, а не лежащей в основе платформой. С такой позицией трудно не согласиться.

Но даже если вы не можете нанять графических дизайнеров, то и тогда предлагаемый по умолчанию внешний вид WPF-приложения будет отвечать ожиданиям пользователей Windows. То же самое относится и к Silverlight, где элементы управления легко адаптируются к конкретному окружению, например к Windows-телефонам. Однако если средства позволяют пригласить дизайнеров, то WPF позволит им без труда изменить облик всего приложения (а не только добавить новые значки или заставку).

Что же касается несогласованности внешнего вида приложений, то ведь то же самое можно сказать и о веб-приложениях, которые стремятся навязать пользователю те или иные предпочтения куда в большей степени, чем традиционные приложения Windows. Несмотря на отсутствие общей линии (и даже *благодаря* такому отсутствию), веб-сайты, удобные для пользователя, отлично себя чувствуют. К тому же разработчики *в любом случае* пытаются писать приложения Windows, которые не походили бы ни на какие другие. А без поддержки со стороны платформы для достижения желаемого эффекта приходится преодолевать многочисленные трудности, и в результате программы получаются с ошибками или причудливыми побочными эффектами.

Стили

Стиль – объект, представленный классом `System.Windows.Style`, – довольно простая штука. Его основная задача – сгруппировать значения свойств, которые в противном случае пришлось бы задавать по отдельности. А конечная цель – предоставить эту группу в распоряжение нескольких элементов.

Возьмем, к примеру, три видоизмененных кнопки `Button` на рис. 14.1. Для придания им такого облика пришлось установить семь свойств. Не будь стилей, мы были бы вынуждены продублировать установку этих свойств для всех кнопок, как показано в листинге 14.1.



***Рис. 14.1.** Три кнопки с измененным внешним видом*

Листинг 14.1. Торжество «копипастинга»!

```
<StackPanel Orientation="Horizontal">
  <Button FontSize="22" Background="Purple" Foreground="White"
    Height="50" Width="50" RenderTransformOrigin=".5,.5">
    <Button.RenderTransform>
      <RotateTransform Angle="10"/>
    </Button.RenderTransform>
    1
  </Button>
  <Button FontSize="22" Background="Purple" Foreground="White"
    Height="50" Width="50" RenderTransformOrigin=".5,.5">
    <Button.RenderTransform>
      <RotateTransform Angle="10"/>
    </Button.RenderTransform>
    2
  </Button>
  <Button FontSize="22" Background="Purple" Foreground="White"
    Height="50" Width="50" RenderTransformOrigin=".5,.5">
    <Button.RenderTransform>
      <RotateTransform Angle="10"/>
    </Button.RenderTransform>
    3
  </Button>
</StackPanel>
```

А стиль позволяет ввести еще один уровень косвенности – задать все свойства в одном месте и из каждой кнопки сослаться на новый элемент, как показано в листинге 14.2. В элементе `Style` мы видим коллекцию элементов `Setter`, устанавливающих интересующие нас свойства. Для создания `Setter` достаточно указать имя свойства зависимости (с помощью имени его класса) и его значение.

Стили удобны по нескольким причинам, в частности потому, что если вы передумаете поворачивать кнопки или захотите поменять цвет фона, то изменения придется вносить только в одном месте. А определение стиля в виде ресурса дает вам всю гибкость механизма ресурсов. Например, можно определить один вариант стиля `buttonStyle` на уровне приложения, а в коллекции `Resources` конкретного окна `Window` переопределить его по-другому (сохранив все тот же ключ `buttonStyle`).

Отметим, что, несмотря на название, ничего специфически визуального в стилях нет. Правда, обычно они все же применяются для задания свойств, затрагивающих визуальные аспекты. Ведь с помощью стилей можно устанавливать лишь свойства зависимости, которые по природе своей визуальны.

Листинг 14.2. Консолидация присваивания значений свойствам в стиле

```
<StackPanel Orientation="Horizontal">
  <StackPanel.Resources>
    <Style x:Key="buttonStyle">
      <Setter Property="Button.FontSize" Value="22"/>
      <Setter Property="Button.Background" Value="Purple"/>
      <Setter Property="Button.Foreground" Value="White"/>
      <Setter Property="Button.Height" Value="50"/>
      <Setter Property="Button.Width" Value="50"/>
      <Setter Property="Button.RenderTransformOrigin" Value=".5,.5"/>
      <Setter Property="Button.RenderTransform">
        <Setter.Value>
          <RotateTransform Angle="10"/>
        </Setter.Value>
      </Setter>
    </Style>
  </StackPanel.Resources>
  <Button Style="{StaticResource buttonStyle}">1</Button>
  <Button Style="{StaticResource buttonStyle}">2</Button>
  <Button Style="{StaticResource buttonStyle}">3</Button>
</StackPanel>
```

Определение стиля

Применение стиля

СОВЕТ

Стили могут также наследовать один другому! Так, следующий элемент `Style` добавляет полужирное начертание к стилю `buttonStyle`, определенному в листинге 14.2. Делается это с помощью свойства `BasedOn`, определяющего стиль, на котором основан данный:

```
<Style x:Key="buttonStyleWithBold" BasedOn="{StaticResource buttonStyle}">
  <!-- Семь свойств, установленных в стиле buttonStyle, наследуются -->
  <Setter Property="Button.FontWeight" Value="Bold"/>
</Style>
```

Обобществление стилей

Хотя любое свойство элемента `Style` можно задать прямо в его определении в XAML-коде (с помощью синтаксиса элемента свойства), идея стилей заключается как раз в том, чтобы ими могли пользоваться разные элементы, как сделано в листинге 14.2. Класс `Style` поддерживает несколько механизмов, позволяющих управлять способом обобществления.

Совместное использование разнородными элементами

Хотя элемент `Style` в листинге 14.2 сообща используется тремя кнопками, после небольшой доработки его можно предоставить в совместное пользование разнородным элементам. В листинге 14.3 мы для этого заменили все вхождения

Button.*XXX* в элементе Style на Control.*XXX*, а затем применили новый стиль к элементам разных типов. Результат показан на рис. 14.2.



***Рис. 14.2.** Разнородные элементы с одним и тем же стилем*

***Листинг 14.3.** Совместное использование одного стиля разнородными элементами*

```
<StackPanel Orientation="Horizontal">
  <StackPanel.Resources>
    <Style x:Key="controlStyle">
      <Setter Property="Control.FontSize" Value="22"/>
      <Setter Property="Control.Background" Value="Purple"/>
      <Setter Property="Control.Foreground" Value="White"/>
      <Setter Property="Control.Height" Value="50"/>
      <Setter Property="Control.Width" Value="50"/>
      <Setter Property="Control.RenderTransformOrigin" Value=".5,.5"/>
      <Setter Property="Control.RenderTransform">
        <Setter.Value>
          <RotateTransform Angle="10"/>
        </Setter.Value>
      </Setter>
    </Style>
  </StackPanel.Resources>
  <Button Style="{StaticResource controlStyle}">1</Button>
  <ComboBox Style="{StaticResource controlStyle}">
    <ComboBox.Items>2</ComboBox.Items>
  </ComboBox>
  <Expander Style="{StaticResource controlStyle}" Content="3"/>
  <TabControl Style="{StaticResource controlStyle}">
    <TabControl.Items>4</TabControl.Items>
  </TabControl>
  <ToolBar Style="{StaticResource controlStyle}">
    <ToolBar.Items>5</ToolBar.Items>
  </ToolBar>
  <InkCanvas Style="{StaticResource controlStyle}"/>
  <TextBox Style="{StaticResource controlStyle}" Text="7"/>
</StackPanel>
```

Можно не беспокоиться о том, что стиль будет применен к элементу, у которого нет каких-то свойств зависимости, определенных в стиле; они будут просто проигнорированы. Например, у элемента InkCanvas нет свойств Foreground и FontSize. И тем не менее в результате применения к нему стиля, определенного в листинге 14.3, все имеющиеся у InkCanvas свойства (Background, Height, Width и т. д.) устанавливаются правильно. Точно так же добавление следующего эле-

мента Setter в стиль Style в листинге 14.3 отражается на элементе TextBox, но все остальные элементы при этом остаются такими же, как показано на рис. 14.2:

```
<Setter Property="TextBox.TextAlignment" Value="Right"/>
```

КОПНЕМ ГЛУБЖЕ

Странное (но правильное) поведение Setter

Внимательный читатель, возможно, задался вопросом, как вообще *какие-то* элементы Setter в листинге 14.3 могут воздействовать на InkCanvas, коль скоро все это свойства класса Control, а InkCanvas даже не наследует Control! А все объясняется еще одним «волшебным» аспектом свойств зависимости (лишний раз подчеркивающим, насколько они отличаются от обычных свойств .NET).

Хотя класс InkCanvas регистрирует несколько собственных свойств зависимости (методом DependencyProperty.Register), у него есть также свойства, например Background, «владение» которыми он разделяет с другими типами (поэтому для их регистрации вызывается метод DependencyProperty.AddOwner). Если несколько типов владеют одним и тем же свойством зависимости, то не имеет значения, имя какого типа указано в Setter.Property, важно лишь, чтобы это было имя одного из владельцев. К сожалению, владение свойствами зависимости – деталь реализации, не слишком хорошо документированная.

А следствием этого решения могут быть еще более озадачивающие эффекты. Например, в листинге 14.3 элементы Setter вообще необязательно изменять, в отличие от листинга 14.2. Если все они будут ссылаться на Button.*XXX*, а не на Control.*XXX*, то результат останется точно таким же. Кроме того, если добавить в листинг 14.3 элемент TextBlock, то обнаружится, что установка свойства Button.Foreground изменяет цвет текста в TextBlock, но установка свойства Button.Background *не* изменяет цвет фона TextBlock! Дело в том, что TextBlock разделяет со всеми элементами Control общую реализацию свойства зависимости Foreground, но не реализацию свойства Background. (Control разделяет реализацию Background с такими классами, как Panel и InkCanvas, тогда как его реализация в классе TextBlock совершенно независима, но зато разделяется с классами TextElement, FlowDocument и др.)

Мой совет – избегайте всех этих недоразумений и создавайте разные стили для различных типов.

СОВЕТ

Любой элемент может переопределить свойства, заданные в его стиле, напрямую присвоив свойству локальное значение. Например, кнопка в листинге 14.3 могла бы позаимствовать угол поворота, размер и прочее у стиля controlStyle, но при этом установить для себя красный, а не фиолетовый фон:

```
<Button Style="{StaticResource controlStyle}" Background="Red">1</Button>
```

Это работает благодаря принципу приоритета в порядке наследования значений свойств, который был описан в главе 3 «Основные принципы WPF». Локальные значения более приоритетны, чем унаследованные от стиля.

СОВЕТ

Чтобы обеспечить совместное использование значений составных свойств даже *внутри* стиля, в классе `Style` имеется собственное свойство `Resources`. Этой коллекцией можно воспользоваться, чтобы сделать стиль полностью самодостаточным, а не создавать потенциально хрупкую зависимость от ресурсов, определенных где-то в другом месте.

Ограничение применимости стилей

Если требуется, чтобы стиль мог применяться только к какому-то одному конкретному типу, то можно установить свойство `TargetType`. Например, следующий стиль будет применяться только к элементам `Button` (и подклассам `Button`):

```
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">
  <Setter Property="Button.FontSize" Value="22"/>
  <Setter Property="Button.Background" Value="Purple"/>
  <Setter Property="Button.Foreground" Value="White"/>
  <Setter Property="Button.Height" Value="50"/>
  <Setter Property="Button.Width" Value="50"/>
  <Setter Property="Button.RenderTransformOrigin" Value=".5,.5"/>
  <Setter Property="Button.RenderTransform">
    <Setter.Value>
      <RotateTransform Angle="10"/>
    </Setter.Value>
  </Setter>
</Style>
```

При любой попытке применить этот стиль к элементу, отличному от `Button`, компилятор выдаст ошибку. Таким образом, если для стиля в листинге 14.3 установить свойство `TargetType="{x:Type Control}"`, то он будет работать для всех элементов, кроме `InkCanvas`.

Кроме того, если в стиле установлено свойство `TargetType`, то уже нет необходимости указывать имя типа в качестве префикса имени свойства внутри элементов `Setter`. Следовательно, показанный выше XAML-код можно переписать короче с сохранением семантики:

```
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">
  <Setter Property="FontSize" Value="22"/>
  <Setter Property="Background" Value="Purple"/>
  <Setter Property="Foreground" Value="White"/>
  <Setter Property="Height" Value="50"/>
  <Setter Property="Width" Value="50"/>
  <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
  <Setter Property="RenderTransform">
    <Setter.Value>
      <RotateTransform Angle="10"/>
    </Setter.Value>
```

```
    </Setter>
  </Style>
```

Создание неявных стилей

Задание свойства TargetType для стиля дает еще одну возможность. Если опустить атрибут Key, то стиль будет неявно применяться ко всем элементам указанного типа в той же области видимости. Обычно такой стиль называют *типизированным*, в отличие от *именованных стилей*, с которыми мы встречались до сих пор.

Область видимости стиля определяется местоположением ресурса Style. Например, если он находится в коллекции Window.Resources, то будет неявно применяться ко всем подходящим элементам в данном окне Window. А ресурс, определенный на уровне приложения, как показано в следующем листинге, применяется ко всему приложению.

```
<Application ...>
<Application.Resources>
  <Style TargetType="{x:Type Button}">          x:Key отсутствует!
    <Setter Property="FontSize" Value="22"/>
    <Setter Property="Background" Value="Purple"/>
    <Setter Property="Foreground" Value="White"/>
    <Setter Property="Height" Value="50"/>
    <Setter Property="Width" Value="50"/>
    <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
    <Setter Property="RenderTransform">
      <Setter.Value>
        <RotateTransform Angle="10"/>
      </Setter.Value>
    </Setter>
  </Style>
</Application.Resources>
</Application>
```

В таком приложении все кнопки по умолчанию получат этот стиль. Но каждая кнопка может переопределить свой облик, явно задав другой стиль или установив для себя какие-то свойства. Чтобы восстановить стиль кнопки, подразумеваемый по умолчанию, следует записать в ее свойство Style значение null.

КОПНЕМ ГЛУБЖЕ

Какая магия стоит за ресурсами без ключей?

Возможно, вам любопытно, как стиль может входить в словарь ResourceDictionary, не имея ключа. На самом деле ключ у него есть – только неявный. И в качестве неявного ключа выступает значение свойства TargetType (то есть объект класса Type, а не строка). Чтобы явно обратиться к стилю без ключа, для которого TargetType равно Button, можно было бы написать:

```
<Button Style="{StaticResource {x:Type Button}}" .../>
```

Для каждого значения TargetType в словаре ResourceDictionary можно определить только один стиль без ключа, иначе возникнет ошибка с сообщением о попытке создать дубликат ключа в одном словаре.

Объекты Style ведут себя таким образом потому, что этот класс помечен следующим атрибутом:

```
[DictionaryKeyProperty("TargetType")]
```

Атрибутом DictionaryKeyPropertyAttribute можно помечать одно из свойств класса, которое будет выступать в роли поставщика ключа по умолчанию, когда экземпляр класса помещается в некоторый словарь.

ПРЕДУПРЕЖДЕНИЕ

Типизированный стиль применяется строго к элементам типа, указанного в свойстве TargetType!

Если для именованного стиля допустимо, чтобы целевой элемент был подклассом TargetType, то типизированный стиль обычно применяется только к элементам того типа, который указан в его свойстве TargetType. Так сделано для того, чтобы избежать разного рода сюрпризов. Допустим, что вы создали стиль для всех элементов ToggleButton в приложении, но не хотите, чтобы он применялся к элементам CheckBox (напомним, что CheckBox – подкласс ToggleButton). Это поведение контролируется каждым элементом (путем выбора ключа стиля по умолчанию, о чем еще будет сказано ниже в разделе «Темы»). Таким образом, можно написать нестандартный элемент, который будет наследовать типизированный стиль от своего базового класса.

СОВЕТ

Стили можно применять в разных местах. Например, у всех элементов управления, производных от FrameworkElement или FrameworkContentElement, имеется свойство FocusVisualStyle помимо свойства Style. Стиль, записанный в свойство FocusVisualStyle, активен в случае, когда элемент владеет фокусом; это очень удобно, когда хочется переопределить стандартную пунктирную прямоугольную рамочку, обозначающую фокус (которая может выглядеть чужеродно, если стиль самого элемента управления был радикально изменен).

В некоторых элементах управления есть еще и дополнительные места для подключения стилей. Так, в классе ItemsControl определено свойство ItemContainerStyle, которое применяется к контейнеру каждого объекта (например, ListBoxItem или ComboBoxItem). В других классах, скажем в ToolBar, имеются свойства вида ResourceKey, представляющие ключи некоторых внутренних стилей, например ButtonStyleKey и TextBoxStyleKey. Хотя сами свойства *XXX*StyleKey допускают только чтение, никто не мешает вам создать собственный стиль с таким ключом и тем самым переопределить стандартный стиль. Например:

```
<Application ...>
<Application.Resources>
  <Style x:Key="{x:Static ToolBar.ButtonStyleKey}" TargetType="{x:Type Button}">
  ...
  </Style>
</Application.Resources>
</Application>
```

Одна из причин, по которым в классе ToolBar используется свойство ResourceKey, а не Style, состоит в том, что свойства зависимости не поддерживают задание ссылки на динамический ресурс в качестве значения по умолчанию. Класс ItemsControl обходит эту проблему, по умолчанию присваивая свойству ItemContainerStyle значение null, поскольку стиль по умолчанию для контейнера элементов в нем всегда один и тот же. Однако в случае ToolBar необходимы разные стили по умолчанию, зависящие от темы.

Триггеры

У триггеров, с которыми мы познакомились в главе 3, тоже есть коллекция элементов Setter – такая же, как у стилей (и/или коллекции действий TriggerAction). Но если стиль применяет заданные в нем значения свойств безусловно, то работа триггера зависит от одного или нескольких условий.

Напомним, что существует три типа триггеров:

- **Триггеры свойств** – вызываются при изменении значения свойства зависимости.
- **Триггеры данных** – вызываются при изменении значения обычного свойства .NET.
- **Триггеры событий** – вызываются при возникновении маршрутизируемого события.

В каждом из классов FrameworkElement, Style, DataTemplate и ControlTemplate (рассматриваются в следующем разделе) имеется коллекция Triggers, но если классы стилей и шаблонов принимают триггеры любого типа, то FrameworkElement – только триггеры событий. К счастью, объект Style оказывается наиболее подходящим местом для размещения триггеров, даже если бы был выбор, потому что стили легко обобществляются и напрямую связаны с визуальными аспектами элементов.

Поэтому рассмотрим несколько примеров триггеров свойств и данных внутри стилей. А обсуждение триггеров событий отложим до главы 17 «Анимация».

Триггеры свойств

Триггер свойства (представленный классом Trigger) выполняет свою коллекцию элементов Setter, когда указанное свойство принимает заданное значение. А когда это значение изменяется, триггер свойства «отменяет» действие своих элементов Setter.

Например, описанное ниже изменение стиля `buttonStyle` происходит только тогда, когда указатель мыши находится над кнопкой; при этом кнопка поворачивается, а ее свойство `Foreground` принимает значение `Black` вместо `White`:

```
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">
<Style.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
    <Setter Property="RenderTransform">
    <Setter.Value>
      <RotateTransform Angle="10"/>
    </Setter.Value>
    </Setter>
    <Setter Property="Foreground" Value="Black"/>
  </Trigger>
</Style.Triggers>
  <Setter Property="FontSize" Value="22"/>
  <Setter Property="Background" Value="Purple"/>
  <Setter Property="Foreground" Value="White"/>
  <Setter Property="Height" Value="50"/>
  <Setter Property="Width" Value="50"/>
  <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
</Style>
```

На рис. 14.3 показан результат применения этого стиля к кнопке. Триггер присваивает свойству `Foreground` значение `Black`, чтобы текст было проще различить на более светлом фоне, который кнопка получает в Windows 7 по умолчанию при наведении на нее указателя мыши. Такой «фон при наведении указателя мыши» не зашит в класс `Button`, а берется из триггера, определенного в стиле темы для `Button` (см. раздел «Темы» в конце главы). Его можно переопределить, явно установив свойство `Background` в только что созданном триггере.



***Рис. 14.3.** Простой триггер свойства может изменять визуальное представление кнопки при наведении на нее указателя мыши*

Более сложный пример триггера свойства работает в сочетании с правилами проверки во время привязки к данным. В предыдущей главе мы создали класс `JpgValidationRule` и присоединили его к привязанному к данным элементу `TextBox`, чтобы не допустить ввода некорректного имени JPG-файла. Чтобы декларативно определить и визуально показать, что проверка не прошла, можно создать триггер свойства, основанный на присоединенном свойстве `Validation.HasError`:

```
<Style x:Key="textBoxStyle" TargetType="{x:Type TextBox}">
<Style.Triggers>
```

```
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="Background" Value="Red"/>
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource={RelativeSource Self},
        Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
  </Style>
```

В этом триггере используется привязка к данным, чтобы показать нужное сообщение во всплывающей подсказке `ToolTip`. Обратите внимание, как атрибут `RelativeSource` позволяет получить присоединенное свойство `Validation.Errors` из любого элемента, к которому применяется данный стиль.

Если применить этот стиль к полю `TextBox`, как показано ниже, то при обнаружении ошибки получится результат, представленный на рис. 14.4:

```
<TextBox Style="{StaticResource textBoxStyle}">
<TextBox.Text>
  <Binding ...>
  <Binding.ValidationRules>
    <local:JpgValidationRule/>
  </Binding.ValidationRules>
  </Binding>
</TextBox.Text>
</TextBox>
```



***Рис. 14.4.** Декларативное описание реакции на ошибку при проверке*

В листинге 14.4 демонстрируется еще одно применение триггеров свойств в стиле – с использованием свойства `AlternationIndex` класса `ItemsControl`, с которым мы познакомились в главе 10 «Многодетные элементы управления». Здесь же показано использование свойства `ItemContainerStyle` класса `ItemsControl` для применения стиля к контейнерам объектов, иногда неявных. (Напомним, что, например, при добавлении в список `ListBox` произвольные объекты обертываются контейнерами `ListBoxItem`.) На рис. 14.5 изображен результат.

Листинг 14.4. Стиль, который поочередно меняет цвета контейнеров объектов, в применении к ListBoxItem и TreeViewItem

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            Orientation="Horizontal">
  <StackPanel.Resources>
    <Style x:Key="AlternatingRowStyle" TargetType="{x:Type Control}">
      <Setter Property="Background" Value="Green"/>
```

```
      <Setter Property="Foreground" Value="White"/>
      <Style.Triggers>
        <Trigger Property="ItemsControl.AlternationIndex" Value="1">
          <Setter Property="Background" Value="White"/>
          <Setter Property="Foreground" Value="Black"/>
        </Trigger>
      </Style.Triggers>
    </Style>
  </StackPanel.Resources>
  <ListBox AlternationCount="2" Margin="10" Width="200"
    ItemContainerStyle="{StaticResource AlternatingRowStyle}">
    <ListBoxItem>Item 1</ListBoxItem>
    <ListBoxItem>Item 2</ListBoxItem>
    <ListBoxItem>Item 3</ListBoxItem>
    <ListBoxItem>Item 4</ListBoxItem>
    <ListBoxItem>Item 5</ListBoxItem>
  </ListBox>
  <TreeView AlternationCount="2" Margin="10" Width="200"
    ItemContainerStyle="{StaticResource AlternatingRowStyle}">
    <TreeViewItem Header="Root 1" AlternationCount="2"
      ItemContainerStyle="{StaticResource AlternatingRowStyle}">
      <TreeViewItem Header="Subitem 1"/>
      <TreeViewItem Header="Subitem 2"/>
      <TreeViewItem Header="Subitem 3"/>
    </TreeViewItem>
    <TreeViewItem Header="Root 2" AlternationCount="2"
      ItemContainerStyle="{StaticResource AlternatingRowStyle}">
      <TreeViewItem Header="Subitem 1"/>
      <TreeViewItem Header="Subitem 2"/>
      <TreeViewItem Header="Subitem 3"/>
    </TreeViewItem>
  </TreeView>
</StackPanel>
```

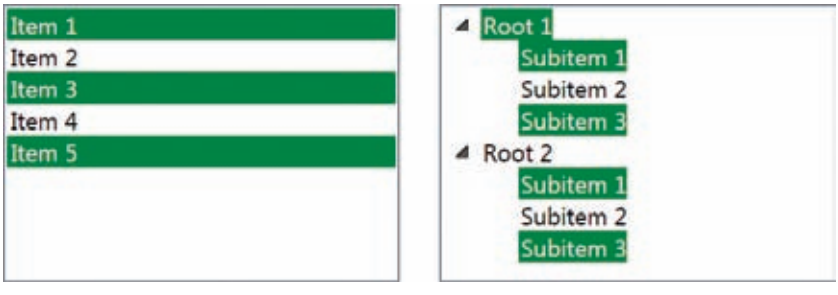


Рис. 14.5. *Элементы ListBox и TreeView, для которых контейнерам объектов назначен один и тот же стиль чередования строк*

В этом стиле по умолчанию определен белый текст на зеленом фоне, но если присоединенное свойство `AlternationIndex` равно 1, то триггеры изменяют цвет

текста на черный, а цвет фона – на зеленый. Поэтому предполагается, что такой стиль будет применяться в качестве стиля контейнера объектов к многодетным элементам управления, для которых AlternationCount равно 2 (что дает последовательность 0, 1, 0, 1, ...).

Отметим, что для того чтобы стиль можно было применять и к ListBoxItem, и к TreeViewItem, в нем указан более общий класс Control (базовый класс, самый близкий к обоим), а в качестве присоединенного свойства используется ItemsControl.AlternationIndex, а не какое-то более специфическое (например, ListBox.AlternationIndex). Чтобы результат оказался таким, как на рис. 14.5, у каждого элемента TreeViewItem, имеющего потомков, свойство AlternationCount должно быть равно 2, а в свойство ItemContainerStyle должен быть записан подходящий стиль. Объясняется это тем, что TreeViewItem (да и сам многодетный элемент) не наследует эти настройки от родителя.

Триггеры данных

Триггеры данных похожи на триггеры свойств, но могут срабатывать при изменении *любого* свойства .NET, а не только свойства зависимости. (Впрочем, элементы Setter внутри триггера данных по-прежнему могут устанавливать только свойства зависимости.)

Чтобы воспользоваться триггером данных, следует добавить в коллекцию Triggers объект DataTrigger и указать пару свойство/значение. Для поддержки обычных свойств .NET интересующее свойство задается с помощью объекта Binding, а не просто именем.

Ниже для поля ввода TextBox задан стиль, который изменяет значение свойства IsEnabled в зависимости от свойства Text, которое не является свойством зависимости. Если свойство Text содержит строку "disabled", то для IsEnabled устанавливается значение false (согласен, не слишком естественное применение триггера данных):

```
<StackPanel Width="200">
<StackPanel.Resources>
  <Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <DataTrigger
      Binding="{Binding RelativeSource={RelativeSource Self}, Path=Text}"
      Value="disabled">
      <Setter Property="IsEnabled" Value="False"/>
    </DataTrigger>
  </Style.Triggers>
  <Setter Property="Background"
    Value="{Binding RelativeSource={RelativeSource Self}, Path=Text}"/>
  </Style>
</StackPanel.Resources>
  <TextBox Margin="3"/>
</StackPanel>
```

Такая же привязка Binding к свойству Text присутствует и вне триггера; она устанавливает цвет фона Background поля TextBox в соответствии со значением

в этом поле (благодаря конвертеру строки в тип Brush). Если в поле задано недопустимое название цвета, то восстанавливается цвет фона по умолчанию, поскольку именно так обрабатываются ошибки привязки к данным. (Помещение подобной привязки в обычный элемент Setter может создать впечатление, будто он является частью триггера, хотя на самом деле это не так.) На рис. 14.6 изображены поля ввода с различными значениями свойства Text.



***Рис. 14.6.** Триггер данных в стиле TextBox делает поле неактивным, когда в нем находится строка "disabled"*

Выражение более сложной логики с помощью триггеров

В рассмотренных выше триггерах логика работы выглядела так: «если *свойство=значение*, установить следующие свойства». Но существуют и другие возможности:

- К одному и тому же элементу можно применять несколько триггеров (для получения логического ИЛИ).
- В одном триггере можно вычислять несколько свойств (для получения логического И).

Логическое ИЛИ. Поскольку коллекция Style.Triggers может содержать несколько триггеров, то открывается возможность создавать разные триггеры с одним и тем же набором элементов Setter – для выражения логической связки ИЛИ:

```
<Style.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
    <Setter Property="RenderTransform">
    <Setter.Value>
      <RotateTransform Angle="10"/>
    </Setter.Value>
    </Setter>
    <Setter Property="Foreground" Value="Black"/>
  </Trigger>
  <Trigger Property="IsFocused" Value="True">
    <Setter Property="RenderTransform">
    <Setter.Value>
      <RotateTransform Angle="10"/>
    </Setter.Value>
    </Setter>
    <Setter Property="Foreground" Value="Black"/>
```

```
    </Trigger>
  </Style.Triggers>
```

Это означает: «если IsMouseOver равно true *или* IsFocused равно true, то повернуть и установить черный цвет фона».

КОПНЕМ ГЛУБЖЕ

Конфликтующие триггеры

Если одновременно активны несколько триггеров с конфликтующими наборами элементов Setter, то побеждает последний. То же самое относится к конфликтующим наборам Setter внутри одного триггера.

Логическое И. Чтобы выразить отношение «логическое И», можно воспользоваться разновидностью элемента Trigger под названием MultiTrigger или разновидностью элемента DataTrigger под названием MultiDataTrigger. И MultiTrigger, и MultiDataTrigger содержат коллекцию условий Conditions, где хранится информация, которая обычно помещается внутрь Trigger или DataTrigger. Таким образом, элемент MultiTrigger можно использовать следующим образом:

```
<Style.Triggers>
  <MultiTrigger>
  <MultiTrigger.Conditions>
    <Condition Property="IsMouseOver" Value="True"/>
    <Condition Property="IsFocused" Value="True"/>
  </MultiTrigger.Conditions>
    <Setter Property="RenderTransform">
    <Setter.Value>
      <RotateTransform Angle="10"/>
    </Setter.Value>
    </Setter>
    <Setter Property="Foreground" Value="Black"/>
  </MultiTrigger>
</Style.Triggers>
```

Это означает: «если IsMouseOver равно true *и* IsFocused равно true, то повернуть и установить черный цвет фона». Элемент MultiDataTrigger работает так же, как MultiTrigger, но поддерживает обычные свойства .NET.

СОВЕТ

Если требуется наделить стиль еще более сложным поведением в части реакции на события, то можно воспользоваться классом EventSetter (у него тот же базовый класс, что и у Setter), позволяющим присоединить обработчик события к любому элементу, для которого задается данный стиль. Элементы EventSetter включаются в стиль так же, как элементы Setter:

```
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">
  <Setter Property="FontSize" Value="22"/>
  <EventSetter Event="MouseEnter" Handler="Button_MouseEnter"/>
</Style>
```

Хотя для обработки события приходится писать процедурный код, это все же неплохой способ задать общий обработчик для нескольких элементов, не прибегая к копированию и вставке.

Шаблоны

В классах, производных от `Control`, имеется много свойств, позволяющих изменить внешний вид элементов управления. У кнопки `Button` конфигурируются кисти `Background` и `Foreground` (можно даже использовать экстравагантные градиентные кисти), вкладки `TabControl` можно по-разному располагать с помощью свойства `TabStripPlacement` и т. д. Но все же возможности настройки за счет одних только свойств ограничены.

С другой стороны, шаблон позволяет полностью подменить визуальное дерево элемента всем, что взбредет вам в голову, сохранив при этом всю функциональность. Более того, шаблоны (как и многое в WPF) – это не просто механизм для реализации сторонних надстроек; стандартный внешний облик всех элементов управления WPF определен в шаблонах (и настроен для каждой темы Windows). Исходный код элементов управления полностью отделен от представления визуального дерева (или «визуального исходного кода»).

Наличие шаблонов и желание отделить внешний вид от логики – это причины того, что элементы управления WPF содержат не слишком много простых свойства для настройки их вида. Например, неплохо было бы иметь возможность поменять цвет стрелки в элементе `Expander` на рис. 14.2, потому что серый плохо смотрится на фиолетовом фоне. Но такое сравнительно простое изменение можно реализовать только путем определения нового шаблона для `Expander`. В классе `Expander` нет свойства `ArrowBrush` или `ArrowColor`, потому что в нестандартном шаблоне `Expander` стрелки может вообще не быть!

Существует несколько видов шаблонов. Те, о которых шла речь выше и которым посвящен настоящий раздел, называются *шаблонами элементов управления*. Они представлены классом `ControlTemplate`, наследующим абстрактному классу `FrameworkTemplate`. Другие классы, производные от `FrameworkTemplate`, рассматривались ранее: `DataTemplate` (в предыдущей главе) и `ItemsPanelTemplate` (в главе 10). Шаблоны данных позволяют изменять внешний вид произвольных объектов .NET, что особенно важно в случае неграфических элементов, для которых шаблон по умолчанию – это просто `TextBlock`, содержащий строку, возвращенную методом `ToString`. Шаблон `ItemsPanelTemplate` можно назначить свойству `ItemsPanel` элемента, производного от `ItemsControl`; это простой способ изменить его компоновку.

В особо изощренных визуальных шаблонах наверняка будет применяться двумерная (и трехмерная!) графика, анимация и другие мультимедийные средства, которые мы рассмотрим в следующей главе. Ну а пока ограничимся простыми двумерными рисунками.

Введение в шаблоны элементов управления

Существенная часть класса `ControlTemplate` – его свойство содержимого `VisualTree`, которое содержит дерево элементов, определяющее внешний вид элемента. Описав шаблон `ControlTemplate` (без сомнения, на XAML), вы сможете присоединить его к любому элементу `Control` или `Page`, установив свойство `Template`. В листинге 14.5 определен простой, но приятный шаблон элемента управления в виде ресурса, который затем применен к одной кнопке `Button`. На рис. 14.7 показан результат.

Листинг 14.5. Применение простого шаблона ControlTemplate к кнопке

```
<Grid>
<Grid.Resources>
  <ControlTemplate x:Key="buttonTemplate">
    <Grid>
      <Ellipse Width="100" Height="100">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0" Color="Blue"/>
          <GradientStop Offset="1" Color="Red"/>
        </LinearGradientBrush>
      </Ellipse.Fill>
      </Ellipse>
      <Ellipse Width="80" Height="80">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0" Color="White"/>
          <GradientStop Offset="1" Color="Transparent"/>
        </LinearGradientBrush>
      </Ellipse.Fill>
      </Ellipse>
    </Grid>
  </ControlTemplate>
</Grid.Resources>
  <Button Template="{StaticResource buttonTemplate}">OK</Button>
</Grid>
```

Чтобы добиться такого вида, в визуальное дерево включены два круга (созданные с помощью элементов `Ellipse`), которые помещены в сетку `Grid` из одной ячейки. Приобретя нестандартный облик, кнопка по-прежнему реагирует на событие `Click`, имеет свойство `IsDefault` и обладает всей остальной ожидаемой функциональностью. Ведь это же по-прежнему экземпляр класса `Button`!



***Рис. 14.7.** Забавная круглая кнопка, созданная с помощью ControlTemplate*

СОВЕТ

В листинге 14.5 `Button` **считается** *шаблоном-родителем* **всех элементов в визуальном дереве шаблона элемента управления. В классах** `FrameworkElement` **и** `FrameworkContentElement` **имеется свойство** `TemplatedParent` **для представления этого отношения.**

Обеспечение интерактивности с помощью триггеров

Как и стили, шаблоны могут содержать любые виды триггеров в коллекции `Triggers`**. В листинге 14.6 в показанный выше шаблон** `ControlTemplate` **включены триггеры для реакции на события наведения указателя и щелчка мышью. Триггер свойства** `Button.IsMouseOver` **окрашивает кнопку в оранжевый цвет, а триггер свойства** `Button.IsPressed` **уменьшает кнопку, применяя преобразование** `ScaleTransform`**, чтобы она выглядела нажатой. Результат показан на рис. 14.8.**

Листинг 14.6. Шаблон ControlTemplate, дополненный триггерами

```
<Grid>
  <Grid.Resources>
    <ControlTemplate x:Key="buttonTemplate">
      <Grid>
        <Ellipse x:Name="outerCircle" Width="100" Height="100">
        <Ellipse.Fill>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="Blue"/>
            <GradientStop Offset="1" Color="Red"/>
          </LinearGradientBrush>
        </Ellipse.Fill>
        </Ellipse>
        <Ellipse Width="80" Height="80">
        <Ellipse.Fill>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="White"/>
            <GradientStop Offset="1" Color="Transparent"/>
          </LinearGradientBrush>
        </Ellipse.Fill>
        </Ellipse>
```

```
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property="Button.IsMouseOver" Value="True">
            <Setter TargetName="outerCircle" Property="Fill" Value="Orange"/>
          </Trigger>
          <Trigger Property="Button.IsPressed" Value="True">
            <Setter Property="RenderTransform">
              <Setter.Value>
                <ScaleTransform ScaleX=".9" ScaleY=".9"/>
              </Setter.Value>
            </Setter>
            <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Grid.Resources>
    <Button Template="{StaticResource buttonTemplate}">OK</Button>
  </Grid>
```



IsMouseOver=True

IsPressed=True
(и IsMouseOver=True)

***Рис. 14.8.** Эффекты наведения указателя мыши и нажатия, реализованные в шаблоне ControlTemplate в листинге 14.6*

Обратите внимание, что больший круг в визуальном дереве шаблона имеет имя – `outerCircle`. Это сделано для того, чтобы на него можно было сослаться из триггера. В первом триггере используется свойство `TargetName` элемента `Setter` (которое имеет смысл только внутри шаблона), чтобы присваивание свойству `Fill` значения `Orange` производилось только для элемента `outerCircle`. Если в этом случае опустить `TargetName`, то произойдет ошибка, потому что система попытается применить триггер ко всей кнопке, а в классе `Button` свойства `Fill` нет. Возможность указывать в триггерах подэлементы шаблона очень важна для разработки сложных шаблонов.

Второму триггеру указывать подэлемент не нужно. Преобразование `ScaleTransform` применяется (в режиме `RenderTransform`) ко всей кнопке, как и установка свойства `RenderTransformOrigin`, задающего центр масштабирования. Передать это на рис. 14.8 трудно, но поверьте, что небольшое уменьшение диаметра кнопки (в данном случае на 10%) дает очень удачный визуальный эффект нажатия.

СОВЕТ

По аналогии со свойством TargetName элемента Setter в классе Trigger (а также в классах EventTrigger и Condition) имеется свойство SourceName, которое дает возможность реагировать на изменение конкретного подэлемента шаблона, а не шаблона в целом. Например, можно создать триггеры свойства IsMouseOver для отдельных подэлементов – это дало бы более интересный эффект при наведении указателя мыши.

КОПНЕМ ГЛУБЖЕ

Именованные элементы в шаблонах

Вне шаблона употребление ключевого слова x:Name для задания имени элемента создает поле, к которому можно обратиться из программы. Но при использовании x:Name внутри шаблона дело обстоит иначе. Дело в том, что шаблон может применяться к нескольким элементам в одной и той же области видимости. Основная цель именования элементов в шаблоне состоит в том, чтобы на них можно было ссылаться из триггеров (как правило, определенных в XAML). Но если вам все-таки нужен программный доступ к именованному элементу внутри шаблона, то можно воспользоваться методом FindName класса Template после того, как шаблон будет применен к целевому элементу.

Ограничение типа целевого элемента

Как и в классе Style, в классе ControlTemplate имеется свойство TargetType, позволяющее ограничить область применения шаблона. Кроме того, оно дает возможность опускать уточняющее имя класса во всех ссылках на свойства внутри шаблона (например, в значениях свойств Trigger.Property и Setter.Property). Таким образом, шаблон в листинге 14.6 можно было бы переписать следующим образом:

```
<ControlTemplate x:Key="buttonTemplate" TargetType="{x:Type Button}">
  <Grid>
  ...
  </Grid>
  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter TargetName="outerCircle" Property="Fill" Value="Orange"/>
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter Property="RenderTransform">
      <Setter.Value>
        <ScaleTransform ScaleX=".9" ScaleY=".9"/>
      </Setter.Value>
      </Setter>
      <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
    </Trigger>
```

```
    </ControlTemplate.Triggers>
  </ControlTemplate>
```

Обратите внимание, что и в предыдущих листингах значения свойства Property в элементах Setter не были квалифицированы именем класса. Объясняется это тем, что данные свойства либо квалифицированы с помощью TargetName, либо являются общими для всех подклассов Control. (Без явного указания TargetType неявно подразумевается целевой тип Control.)

В отличие от Style, использование TargetType не дает возможности отказаться от атрибута x:Key в шаблоне (если он включается в словарь). Не существует такого понятия, как подразумеваемый по умолчанию шаблон элемента управления; чтобы получить такое поведение, необходимо задать шаблон внутри типизированного стиля.

Учет свойств шаблона-родителя

С шаблонами, которые мы видели до сих пор, связана небольшая проблема. Все кнопки, к которым они применяются, выглядят абсолютно одинаково, какие бы значения свойств мы для них ни задавали. Например, в последних двух листингах содержимым кнопки была строка "OK", однако же она не отображается. Если вы собираетесь создать шаблон, предназначенный для широкого распространения, то необходимо проделать некоторую работу, чтобы учитывались различные свойства целевого элемента управления.

Учет свойства Content класса ContentControl

Ключом к вставке значений свойств целевого элемента управления в шаблон является механизм привязки к данным. К счастью, класс TemplateBindingExtension существенно упрощает эту задачу.

Этот класс представляет собой расширение разметки, аналогичное расширению Binding, но более простое, облегченное и рассчитанное специально на шаблоны. Часто его называют просто TemplateBinding, поскольку суффикс Extension в XAML принято опускать.

Источником данных для TemplateBinding всегда является целевой элемент, а путем может быть любое его свойство зависимости, задаваемое в свойстве Property объекта TemplateBinding. Стало быть, в шаблон из листинга 14.6 можно следующим образом добавить элемент TextBlock, который будет содержать значение свойства Content кнопки Button:

```
<TextBlock Text="{TemplateBinding Property=Button.Content}"/>
```

Или, поскольку в классе TemplateBinding имеется конструктор, принимающий свойство зависимости, можно написать короче:

```
<TextBlock Text="{TemplateBinding Button.Content}"/>
```

Если TargetType ограничивает применение шаблона только кнопками (или другими элементами, производными от ContentControl), то код можно еще больше упростить:

```
<TextBlock Text="{TemplateBinding Content}"/>
```

Разумеется, содержимым кнопки может быть не только текст, поэтому применение элемента TextBlock для его отображения вводит искусственное ограничение. Чтобы гарантировать правильное отображение в шаблоне свойства Content любого типа, можно вместо TextBlock использовать более общий класс ContentControl. В листинге 14.7 так и сделано. Для элемента ContentControl задано поле Margin, и он обернут элементом Viewbox, чтобы размер устанавливался с учетом остальных частей кнопки.

Листинг 14.7. Модифицированный шаблон ControlTemplate, отображающий содержимое целевой кнопки

```
<ControlTemplate x:Key="buttonTemplate" TargetType="{x:Type Button}">
  <Grid>
    <Ellipse x:Name="outerCircle" Width="100" Height="100">
    <Ellipse.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0" Color="Blue"/>
        <GradientStop Offset="1" Color="Red"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
    </Ellipse>
    <Ellipse Width="80" Height="80">
    <Ellipse.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0" Color="White"/>
        <GradientStop Offset="1" Color="Transparent"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
    </Ellipse>
    <Viewbox>
      <ContentControl Margin="20" Content="{TemplateBinding Content}"/>
    </Viewbox>
  </Grid>
  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter TargetName="outerCircle" Property="Fill" Value="Orange"/>
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter Property="RenderTransform">
      <Setter.Value>
        <ScaleTransform ScaleX=".9" ScaleY=".9"/>
      </Setter.Value>
      </Setter>
      <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

На рис. 14.9 показано, как выглядят две кнопки в результате применения нового шаблона. В одной отображается простая строка "OK", а в другой – изобра-

жение Image. В обоих случаях содержимое правильно показывается в новом визуальном дереве.



***Рис. 14.9.** Две разных кнопки, отображаемые по шаблону в листинге 14.7*

СОВЕТ

Вместо того чтобы использовать внутри шаблона элемент ContentControl, следовало бы применить более легкий элемент ContentPresenter. Он отображает содержимое так же, как ContentControl, но разработан специально для использования в шаблонах элементов управления. ContentPresenter – это примитивный строительный блок, тогда как ContentControl – полноценный элемент управления с собственным шаблоном (который, кстати, содержит ContentPresenter)!

В листинге 14.7 можно заменить строку

```
<ContentControl Margin="20" Content="{TemplateBinding Content}"/>
```

такой:

```
<ContentPresenter Margin="20" Content="{TemplateBinding Content}"/>
```

В элемент ContentPresenter даже встроена возможность сокращенной записи: если опустить присваивание Content значения {TemplateBinding Content}, то оно будет неявно подразумеваться. Так что предыдущую строку можно сократить:

```
<ContentPresenter Margin="20"/>
```

Но это работает, только если в шаблоне явно прописан тип TargetType целевого элемента, производный от ContentControl (например, Button).

Далее в этой главе мы будем во всех шаблонах использовать ContentPresenter вместо ContentControl, поскольку именно так делается в реальной практике.

ПРЕДУПРЕЖДЕНИЕ

Расширение TemplateBinding разрешено только внутри визуального дерева шаблона и не работает со свойствами, тип которых является подклассом Freezable!

Расширение привязки TemplateBinding не работает вне шаблона или вне его свойства VisualTree, так что его нельзя использовать даже в триггере шаблона. Более того, TemplateBinding нельзя применять к свойствам, производным от класса Freezable. Например, попытка применить это расширение для привязки свойства Color любой явно заданной кисти Brush потерпит неудачу.

Однако TemplateBinding – всего лишь менее мощная, но удобная замена обычному расширению разметки Binding. Точно такого же эффекта можно добиться, применив обычный объект Binding, в котором свойство RelativeSource равно {RelativeSource TemplatedParent}, а Path содержит путь к свойству зависимости, значение которого мы хотим получить. Такой объект Binding работает и в тех случаях, где TemplateBinding отказывается.

Учет остальных свойств

Для какого бы элемента управления ни создавался шаблон, наверняка у целевого элемента есть и другие свойства, которые следует учитывать, если ставится задача сделать шаблон повторно используемым: Height и Width, возможно, Background, Padding и т. д. Некоторые свойства (например, Foreground, FontSize, FontWeight и прочие) могут автоматически наследовать нужные значения благодаря механизму наследования значений свойств в визуальном дереве, другие же требуют явного внимания.

В листинге 14.8 приведен модифицированный код шаблона из листинга 14.7, в котором учтены свойства Background, Padding и Content целевой кнопки Button. Здесь же неявно учитывается размер целевого элемента – за счет того, что мы *убрали* явную установку свойств Height и Width, положившись на систему компоновки. В листинге 14.8 мы применили ContentPresenter вместо ContentControl, хотя результат в обоих случаях получается одинаковым.

Листинг 14.8. Модифицированный шаблон ControlTemplate, более пригодный для повторного использования

```
<ControlTemplate x:Key="buttonTemplate" TargetType="{x:Type Button}">
  <Grid>
    <Ellipse x:Name="outerCircle">
    <Ellipse.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0"
          Color="{Binding RelativeSource={RelativeSource TemplatedParent},
            Path=Background.Color}"/>
        <GradientStop Offset="1" Color="Red"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
    </Ellipse>
    <Ellipse RenderTransformOrigin=".5,.5">
    <Ellipse.RenderTransform>
      <ScaleTransform ScaleX=".8" ScaleY=".8"/>
    </Ellipse.RenderTransform>
    <Ellipse.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0" Color="White"/>
        <GradientStop Offset="1" Color="Transparent"/>
      </LinearGradientBrush>
```

```
      </Ellipse.Fill>
      </Ellipse>
      <Viewbox>
        <ContentPresenter Margin="{TemplateBinding Padding}"/>
      </Viewbox>
    </Grid>
    <ControlTemplate.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
        <Setter TargetName="outerCircle" Property="Fill" Value="Orange"/>
      </Trigger>
      <Trigger Property="IsPressed" Value="True">
        <Setter Property="RenderTransform">
        <Setter.Value>
          <ScaleTransform ScaleX=".9" ScaleY=".9"/>
        </Setter.Value>
        </Setter>
        <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
      </Trigger>
    </ControlTemplate.Triggers>
  </ControlTemplate>
```

Свойство `Padding` целевой кнопки теперь выступает в роли свойства `Margin` элемента `ContentPresenter`. Использование отступа `Padding` целевого элемента в качестве поля `Margin` внутреннего элемента шаблона – распространенная практика. В конце концов, это, по существу, и есть определение отступа!

Кроме того, в визуальное дерево шаблона внесен ряд интуитивно неочевидных изменений, чтобы адаптировать его к заданным извне размерам и цвету фона `Background`. Мы могли бы просто присвоить значение `{TemplateBinding Background}` свойству `Fill` внешнего круга `outerCircle`, позволив тем самым задавать кнопке сплошной цвет, градиент и т. д. Но, допустим, «красный ободок» в нижней части мы считаем характерной особенностью, которую следует сохранить при любом изменении шаблона. Другими словами, мы хотим заменить только синюю часть градиента заданным извне цветом `Background`. Однако свойству `GradientStop.Color` нельзя напрямую присвоить значение `{TemplateBinding Background}`, потому что оно имеет тип `Color`, тогда как `Background` имеет тип `Brush` (да к тому же `GradientStop` наследует `Freezable`)! Поэтому в листинге используется обычный объект `Binding`, который поддерживает ссылку на субсвойство `Color`. (Отметим, что этот объект `Binding` работает только в случае, когда в качестве `Background` задана кисть типа `SolidColorBrush`, потому что у всех остальных кистей свойства `Color` нет.)

Для обоих эллипсов (или для родительской сетки `Grid`) можно было бы явно задать высоту `Height` и ширину `Width`, ориентируясь на соответствующие свойства целевой кнопки. Для этого достаточно было бы привязаться к ее свойствам `ActualHeight` и `ActualWidth`. Но вместо этого мы их вообще опустили, потому что корневому элементу в любом случае назначается размер шаблона-родителя! Это означает, что конкретная целевая кнопка может превратить себя в эллипс, задав разные значения `Width` и `Height`. Если бы мы захотели сохранить

форму идеального круга, то могли бы погрузить все визуальное дерево в элемент Viewbox.

И последний трюк, который мы использовали в листинге 14.8, – это применение ScaleTransform к внутреннему кругу, чтобы его размер составлял 80% от размера внешнего. В предыдущих листингах это преобразование было не нужно, потому что размеры внешнего и внутреннего круга были жестко зашиты в код. Но если размер изменяется динамически, то ScaleTransform дает эффективный способ адаптироваться к этому. (Если бы мы хотели, чтобы между границами кругов всегда сохранялось фиксированное расстояние, то достаточно было бы воспользоваться свойством Margin.)

На рис. 14.10 показан результат визуализации следующих кнопок с новым шаблоном:

```
<StackPanel Orientation="Horizontal">
  <Button Template="{StaticResource buttonTemplate}"
    Height="100" Width="100" FontSize="80" Background="Black"
    Padding="20" Margin="5">1</Button>
  <Button Template="{StaticResource buttonTemplate}"
    Height="150" Width="250" FontSize="80" Background="Yellow"
    Padding="20" Margin="5">2</Button>
  <Button Template="{StaticResource buttonTemplate}"
    Height="200" Width="200" FontSize="80" Background="White"
    Padding="20" Margin="5">3</Button>
</StackPanel>
```

Для каждой кнопки на рис. 14.10 заданы значения свойств Background, Padding и Content, которые явно используются в шаблоне. Значения свойств Height и Width учитываются шаблоном неявно, а свойство FontSize неявно же подхватывается элементом ContentPresenter, входящим в состав шаблона. Размер шрифта не отражен явно в результате визуализации, поскольку шаблон помещает ContentPresenter внутрь Viewbox, чтобы текст не выходил за границы внешнего круга. Поле Margin, заданное в каждой кнопке, в шаблоне не используется, но влияет на компоновку StackPanel, так что между кнопками остается небольшой промежуток.



***Рис. 14.10.** Кнопки влияют на свой шаблон, приведенный в листинге 14.8*

КОПНЕМ ГЛУБЖЕ

TemplateBinding и конвертеры значений

Как и Binding, расширение разметки TemplateBinding поддерживает конвертеры значений. В классе TemplateBinding определены свойства Converter и ConverterParameter, но, как ни странно, нет свойства ConverterCulture. Если последнее вам необходимо, пользуйтесь расширением Binding.

Похищение существующих свойств для других целей

Иногда желательно параметризовать некоторые аспекты шаблона элемента управления, несмотря на то, что в целевом элементе нужных свойств нет. Например, в шаблоне в листинге 14.8 зашита оранжевая кисть, представляющая состояние с наведенным указателем мыши. И как разрешить кнопкам настраивать эту кисть? Ведь соответствующего свойства в классе Button нет!

Один из вариантов – определить нестандартный элемент управления, как описано в главе 20 «Пользовательские и нестандартные элементы управления». Не так уж трудно написать класс, производный от Button, добавив в него всего одно свойство HoverBrush. Но все-таки чересчур много работы для такой простой задачи. Другой вариант – определить несколько шаблонов с разными кистями, применяемыми при наведении указателя мыши. Но это разумно, лишь если набор предпочтительных кистей невелик и заранее известен. Еще один вариант – определить где-то подходящее присоединенное свойство, например в каком-нибудь уже существующем служебном классе.

Однако большинство разработчиков предпочитают маленькую уловку – так называемое *похищение* свойства зависимости. Смысл в том, чтобы найти в целевом элементе управления такое свойство зависимости нужного типа, которое можно было бы использовать непредусмотренным способом. Например, во всех классах, производных от Control, есть три свойства типа Brush: Background, Foreground и BorderBrush. Поскольку Background и Foreground уже играют важные роли в листинге 14.8, ни одно из них не будет хорошо выглядеть в качестве кисти, применяемой при наведении указателя мыши. (Кроме того, не осталось бы способа задать такую кисть независимо от двух остальных.) А вот BorderBrush – совсем другое дело. Оно не встречается в нашем шаблоне, так почему бы им не воспользоваться?

Единственное возражение – это то, что применение шаблона может стать более запутанным, а в его коде будет сложнее разобраться. Тем не менее покажем, как можно модифицировать триггер свойства IsMouseOver в листинге 14.8, чтобы похитить свойство BorderBrush:

```
<Trigger Property="IsMouseOver" Value="True">
  <Setter TargetName="outerCircle" Property="Fill"
    Value="{Binding RelativeSource={RelativeSource TemplatedParent},
      Path=BorderBrush}"/>
</Trigger>
```

В этом случае необходимо использовать `Binding`, а не `TemplateBinding`, потому что элемент `Trigger` находится вне визуального дерева.

Если у целевого элемента управления нет подходящего свойства, то можно даже похитить присоединенное свойство, определенное в каком-то совершенно постороннем элементе! Но, выбирая такое свойство, обращайте внимание на его метаданные, например на то, какое у него значение по умолчанию и что происходит при изменении значения (скажем, может потребоваться перекомпоновка).

Если этот трюк кажется вам сомнительным, то никто не мешает прибегнуть к альтернативным решениям. Разработчики WPF его уж точно не рекомендуют! Тем не менее знать о нем полезно на случай, если понадобится что-то быстро подправить.

Учет визуальных состояний с помощью триггеров

При создании шаблонов для кнопок было бы неплохо реагировать на события наведения указателя и щелчка мышью с помощью соответствующих триггеров, хотя это и необязательно. Но давайте задумаемся о том, как будет работать шаблон, приведенный в листинге 14.8, для элемента `CheckBox` или `ToggleButton` (для этого достаточно изменить свойство `TargetType`). Поскольку в шаблоне не предусмотрены визуальные различия между состояниями `Checked`, `Unchecked` и `Indeterminate`, то для этих элементов управления он попросту не годится!

На самом деле шаблон в листинге 14.8 недоделан даже для кнопок `Button`! Проблема в том, что он визуально не показывает, когда свойство `IsEnabled` равно `false`, а когда `IsDefaulted` равно `true`, поэтому пользоваться им будет крайне неудобно.

Следовательно, при проектировании шаблона для какого-то элемента управления необходимо учесть все визуальные состояния, в которых он потенциально может находиться. Это можно сделать с помощью триггеров для соответствующих свойств или событий либо же просто реализовав привязку для этих свойств и событий.

Например, шаблон элемента управления `ProgressBar` должен показывать текущее значение, иначе он будет бесполезен. В листинге 14.9 приведен шаблон индикатора `ProgressBar` (определенный в виде ресурса уровня приложения), который выглядит как секторная диаграмма. Наиболее существенная часть шаблона – заполнение диаграммы в соответствии с текущим значением свойства `Value` – реализована путем привязки к шаблону-родителю, а необходимые тригонометрические вычисления производятся в конвертерах значений. Кроме того, триггеры свойств `IsEnabled` и `IsIndeterminate` изменяют внешний вид элемента в соответствующих состояниях. На рис. 14.11 и 14.12 показаны результаты визуализации таких различных индикаторов, например:

```
<ProgressBar Foreground="{StaticResource foregroundBrush}" Width="100"
  Height="100" Value="10" Template="{StaticResource progressPie}"/>
```

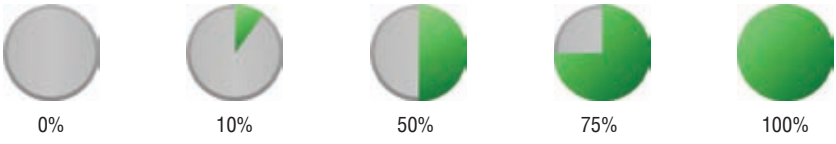


***Рис. 14.11.** Индикатор ProgressBar с нестандартным шаблоном на различных стадиях выполнения*



***Рис. 14.12.** Индикатор ProgressBar с нестандартным шаблоном в неактивном и неопределенном состояниях*

Ресурс `foregroundBrush` определен как простой градиент зеленого цвета:

```
<LinearGradientBrush x:Key="foregroundBrush" StartPoint="0,0" EndPoint="1,1">
  <GradientStop Offset="0" Color="LightGreen"/>
  <GradientStop Offset="1" Color="DarkGreen"/>
</LinearGradientBrush>
```

Листинг 14.9. Шаблон ProgressBar в виде секторной диаграммы

```
<Application x:Class="WindowsApplication1.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WindowsApplication1"
             StartupUri="Window1.xaml">
  <Application.Resources>

    <ControlTemplate x:Key="progressPie" TargetType="{x:Type ProgressBar}">

    <!-- Ресурсы -->
    <ControlTemplate.Resources>
      <local:ValueMinMaxToPointConverter x:Key="converter1"/>
      <local:ValueMinMaxToIsLargeArcConverter x:Key="converter2"/>
    </ControlTemplate.Resources>

    <!-- Визуальное дерево -->
    <Viewbox>
      <Grid Width="20" Height="20">
        <Ellipse x:Name="background" Stroke="{TemplateBinding BorderBrush}"
          StrokeThickness="{TemplateBinding BorderThickness}"
          Width="20" Height="20" Fill="{TemplateBinding Background}"/>
        <Path x:Name="pie" Fill="{TemplateBinding Foreground}">
        <Path.Data>
          <PathGeometry>
            <PathFigure StartPoint="10,10" IsClosed="True">
```

```
            <LineSegment Point="10,0"/>
            <ArcSegment Size="10,10" SweepDirection="Clockwise">
            <ArcSegment.Point>
              <MultiBinding Converter="{StaticResource converter1}">
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Value"/>
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Minimum"/>
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Maximum"/>
              </MultiBinding>
            </ArcSegment.Point>
            <ArcSegment.IsLargeArc>
              <MultiBinding Converter="{StaticResource converter2}">
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Value"/>
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Minimum"/>
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Maximum"/>
              </MultiBinding>
            </ArcSegment.IsLargeArc>
            </ArcSegment>
          </PathFigure>
        </PathGeometry>
        </Path.Data>
      </Path>
    </Grid>
  </Viewbox>

  <!-- Триггеры -->
  <ControlTemplate.Triggers>
    <Trigger Property="IsIndeterminate" Value="True">
      <Setter TargetName="pie" Property="Visibility" Value="Hidden"/>
      <Setter TargetName="background" Property="Fill">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
          <GradientStop Offset="0" Color="Yellow"/>
          <GradientStop Offset="1" Color="Brown"/>
        </LinearGradientBrush>
      </Setter.Value>
      </Setter>
    </Trigger>
    <Trigger Property="IsEnabled" Value="False">
      <Setter TargetName="pie" Property="Fill">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
          <GradientStop Offset="0" Color="Gray"/>
          <GradientStop Offset="1" Color="White"/>
        </LinearGradientBrush>
      </Setter.Value>
      </Setter>
```

```
        </Trigger>
      </ControlTemplate.Triggers>
      </ControlTemplate>

    </Application.Resources>
  </Application>
```

Корнем визуального дерева является элемент `Viewbox`, поэтому сетка `Grid` с единственной ячейкой размером 20×20 будет масштабироваться правильно. Свойства `Background`, `BorderBrush` и `BorderThickness` внутреннего круга, определяющего фон (до масштабирования его радиус равен 10), берутся из шаблона-родителя. Сектор представлен элементом `Path` (мы рассмотрим его в следующей главе), который заимствует у шаблона-родителя свойство `Foreground` и возлагает ответственность за вычисление правильной формы на два элемента `MultiBinding` с конвертерами значений (их определения приведены ниже, в листинге 14.10). Мы использовали объекты `MultiBinding`, а не `TemplateBinding` или `Binding`, чтобы сектор менял форму при изменении любого из трех существенных свойств `ProgressBar`: `Value`, `Minimum` и `Maximum`. Триггеры дают результат, показанный на рис. 14.12, закрашивая элемент жестко зашитой в код кистью `Brush` (а в состоянии `IsIndeterminate` сектор скрывается). Возможно, более правильно было бы изобразить состояние `IsIndeterminate` анимацией в виде перемещения сектора по кругу, но и то решение, на котором мы остановились, демонстрирует *некое* визуальное различие. Отметим, что в этом шаблоне учтены не все свойства `ProgressBar`. Например, свойство `Orientation` игнорируется, но, принимая во внимание выбранное представление, непонятно, как его можно было бы визуализировать.

СОВЕТ

Обратите внимание: в листинге 14.9 предполагается, что конвертеры значений находятся в коллекции `Resources` объекта `ControlTemplate`. Как и класс `Style`, все классы, производные от `FrameworkTemplate`, располагают собственной коллекцией `Resources`. Она позволяет создавать автономные, ни от чего не зависящие шаблоны.

Листинг 14.10. Конвертеры значений, на которые ссылается код в листинге 14.9

```
public class ValueMinMaxToIsLargeArcConverter : IMultiValueConverter
{
  public object Convert(object[] values, Type targetType,
    object parameter, CultureInfo culture)
  {
    double value = (double)values[0];
    double minimum = (double)values[1];
    double maximum = (double)values[2];

    // Возвращаем true, только если value составляет не менее 50% диапазона
    return ((value * 2) >= (maximum - minimum));
```

```
    }

    public object[] ConvertBack(object value, Type[] targetTypes,
      object parameter, CultureInfo culture)
    {
      throw new NotSupportedException();
    }
  }

  public class ValueMinMaxToPointConverter : IMultiValueConverter
  {
    public object Convert(object[] values, Type targetType,
      object parameter, CultureInfo culture)
    {
      double value = (double)values[0];
      double minimum = (double)values[1];
      double maximum = (double)values[2];

      // Приводит value к диапазону от 0 до 360
      double current = (value / (maximum - minimum)) * 360;

      // Корректируем вид в состоянии «полностью выполнено»,
      // чтобы дуга ArcSegment рисовалась в виде полной окружности
      if (current == 360)
          current = 359.999;

      // Поворачиваем на 90 градусов, чтобы 0 оказался в верхней точке окружности
      current = current - 90;

      // Переводим градусы в радианы
      current = current * 0.017453292519943295;

      // Вычисляем координаты точки на окружности
      double x = 10 + 10 * Math.Cos(current);
      double y = 10 + 10 * Math.Sin(current);
      return new Point(x, y);
    }

    public object[] ConvertBack(object value, Type[] targetTypes,
      object parameter, CultureInfo culture)
    {
      throw new NotSupportedException();
    }
  }
```

Первый конвертер значений не представляет никаких сложностей. Свойство `IsLargeArc` **объекта** `ArcSegment` **(см. листинг 14.9) должно быть равно** `true`**, если сектор занимает больше половины круга, и** `false` **в противном случае. Таким образом, класс** `ValueMinMaxToIsLargeArcConverter` **всего лишь выполняет простое вычисление исходя из значений трех свойств целевого индикатора** `ProgressBar` **и возвращает соответствующее булевское значение.**

Второй конвертер значительно сложнее. Он должен вернуть точку Point на окружности, зная текущие значения свойств индикатора. Для этого он интерпретирует свойство Value как угол (в градусах), вносит небольшие корректировки и преобразует значение в радианы. Далее с полученным углом производятся тригонометрические вычисления для нахождения координат точки – в предположении, что радиус равен 10, а центр круга находится в точке (10,10).

Учет визуальных состояний с помощью менеджера визуальных состояний

Проектировщику шаблонов трудно разобраться в том, какие визуальные состояния требуется учитывать. У каждого элемента управления есть множество свойств, и не всегда понятно, какие из них существенны для визуального представления и как управлять ими с помощью триггеров. К счастью, в версии WPF 4 решение этой задачи упростилось с появлением менеджера визуальных состояний (Visual State Manager – VSM), ранее включенного в состав Silverlight.

VSM поддерживает набор типов и членов классов, позволяющих автору элемента управления формально определить его *части* и *состояния*, исключая возможную путаницу. Важно, что VSM поддерживает эффективность инструментальных средств при создании сложных шаблонов. Например, в Expression Blend части и состояния используются с большой пользой.

Части элемента управления

Понятие «части» существовало в WPF с самой первой версии. Идея в том, что элемент управления может поискать в визуальном дереве примененного к нему шаблона элементы со специальными именами и впоследствии применить к ним определенную логику. Рассмотрим несколько примеров.

- Если в шаблоне индикатора ProgressBar есть элементы с именами PART_Indicator и PART_Track, то сам элемент гарантирует, что ширина Width (или высота Height в зависимости от ориентации) элемента PART_Indicator будет составлять правильную процентную долю от ширины (или высоты) элемента PART_Track, вычисляемую с учетом значений свойств Value, Minimum и Maximum индикатора. Для шаблона индикатора в виде секторной диаграммы (см. листинг 14.9) такое поведение, очевидно, нежелательно. Но для шаблона, более похожего на стандартный элемент ProgressBar, наличие такой поддержки существенно упрощает работу (и делает излишним выполнение математических вычислений в процедурном коде).
- Если в шаблоне комбинированного списка ComboBox имеется элемент Popup с именем PART_Popup, то при закрытии этого элемента автоматически генерируется событие DropDownClosed элемента ComboBox. Ели же имеется элемент TextBox с именем PART_EditableTextBox, то он автоматически интегрируется со встроенной в ComboBox возможностью обновлять выбранный элемент, когда пользователь вводит текст с клавиатуры.

- **Большая часть функциональности таких элементов управления, как** TextBox **и** PasswordBox**, основана на наличии в шаблоне элемента с именем** PART_ContentHost**. Если элемента с таким именем в шаблоне не существует, то всю область редактирования вам придется реализовывать с нуля!**

В некоторых случаях именованная часть может быть произвольным элементом типа FrameworkElement**, в других на ее тип накладываются ограничения, при несоблюдении которых часть игнорируется. В табл. 14.1 перечислены все именованные части, используемые встроенными в WPF элементами управления. Производные классы, которые автоматически наследуют логику работы с именованными частями, не упоминаются (например,** TextBox **и** PasswordBox **получают логику работы с частью** PART_ContentHost **от базового класса** TextBoxBase**).**

Таблица 14.1. Именованные части, используемые в элементах управления WPF

Элемент управления	Имя части	Тип части
Calendar	PART_CalendarItem	CalendarItem
	PART_Root	Panel
CalendarItem	DayTitleTemplate	DataTemplate
	PART_DisabledVisual	FrameworkElement
	PART_HeaderButton	Button
	PART_MonthView	Grid
	PART_NextButton	Button
	PART_PreviousButton	Button
	PART_Root	FrameworkElement
	PART_YearView	Grid
ComboBox	PART_EditableTextBox	TextBox
	PART_Popup	Popup
DataGridColumnFloatingHeader	PART_VisualBrushCanvas	Canvas
DataGridColumnHeader	PART_LeftHeaderGripper	Thumb
	PART_RightHeaderGripper	Thumb
DataGridColumnHeadersPresenter	PART_FillerColumnHeader	DataGridColumnHeader
DataGridRowHeader	PART_BottomHeaderGripper	Thumb
	PART_TopHeaderGripper	Thumb
DatePicker	PART_Button	Button
	PART_Popup	Popup
	PART_Root	Grid
	PART_TextBox	DatePickerTextBox
DatePickerTextBox	PART_Watermark	ContentControl
DocumentViewer	PART_ContentHost	ScrollViewer
	PART_FindToolBarHost	ContentControl

Элемент управления	Имя части	Тип части
DocumentViewerBase	PART_FindToolBarHost	Decorator
FlowDocumentReader	PART_ContentHost	Decorator
	PART_FindToolBarHost	Decorator
FlowDocumentScrollViewer	PART_ContentHost	ScrollViewer
	PART_FindToolBarHost	Decorator
	PART_ToolBarHost	Decorator
Frame	PART_FrameCP	ContentPresenter
GridViewColumnHeader	PART_FloatingHeaderCanvas	Canvas
	PART_HeaderGripper	Thumb
MenuItem	PART_Popup	Popup
NavigationWindow	PART_NavWinCP	ContentPresenter
ProgressBar	PART_GlowRect	FrameworkElement
	PART_Indicator	FrameworkElement
	PART_Track	FrameworkElement
ScrollBar	PART_Track	Track
ScrollViewer	PART_HorizontalScrollBar	ScrollBar
	PART_ScrollContentPresenter	ScrollContentPresenter
	PART_VerticalScrollBar	ScrollBar
Slider	PART_SelectionRange	FrameworkElement
	PART_Track	Track
StickyNoteControl	PART_ClipboardSeparator	Separator
	PART_CloseButton	Button
	PART_ContentControl	ContentControl
	PART_CopyMenuItem	MenuItem
	PART_EraseMenuItem	MenuItem
	PART_IconButton	Button
	PART_InkMenuItem	MenuItem
	PART_PasteMenuItem	MenuItem
	PART_ResizeBottomRightThumb	Thumb
	PART_SelectMenuItem	MenuItem
	PART_TitleThumb	Thumb
TabControl	PART_SelectedContentHost	ContentPresenter
TextBoxBase	PART_ContentHost	FrameworkElement
ToolBar	PART_ToolBarOverflowPanel	ToolBarOverflowPanel
	PART_ToolBarPanel	ToolBarPanel
TreeViewItem	PART_Header	FrameworkElement

Таким образом, заявления о том, что элементы управления WPF якобы «безвидные» (то есть сами по себе не имеют внешнего облика), а их реализация полностью отделена от визуального представления (я и сам высказывался в этом духе в начале главы), *не совсем* правдивы! Однако эти «тайные заигрывания» с частями, имеющими предопределенные имена, необязательны. Это важно, поскольку означает, что вы по-прежнему вправе кардинально изменить внешний вид элемента, как мы поступили с шаблоном индикатора выполнения `ProgressBar`.

Чтобы инструменты конструирования могли узнать обо всех доступных именованных частях, элемент управления документирует их в атрибутах `TemplatePartAttribute`, применяемых к классу, – по одному для каждой части. В атрибуте указываются имя и ожидаемый тип части. В WPF также принято соглашение о том, что имена частей имеют вид `PART_XXX` (правда, как показано в таблице выше, оно нарушено в частях элемента `CalendarItem`); в Silverlight такого соглашения об именовании нет.

С одной стороны, именованные части – это деталь реализации, о которой вам знать необязательно. Но, с другой стороны, иногда, воспользовавшись встроенной логикой, можно создать шаблон элемента управления с гораздо меньшими усилиями!

Состояния элементов управления

Состояния – это новшество в модели частей и состояний, появившееся только в WPF 4. Как и в случае частей, элементы управления могут быть наделены внутренней логикой перехода из одного именованного состояния в другое (для этого вызывается статический метод `VisualStateManager.GoToState`). А шаблон может воспользоваться для отображения визуальных характеристик каждого состояния не триггерами, а некоторыми новыми элементами. Писать шаблоны с учетом состояний необязательно, но в WPF 4 такой подход рекомендуется. Подобные шаблоны не только лучше поддерживаются инструментами типа Expression Blend, но и с большей вероятностью будут работать также с элементами управления Silverlight.

Состояния, определенные в каждом элементе управления, объединяются в непересекающиеся *группы состояний*. Например, у кнопки `Button` есть четыре состояния в группе `CommonStates`: `Normal`, `MouseOver`, `Pressed` и `Disabled`, а также два состояния в группе `FocusStates`: `Unfocused` и `Focused`. В любой момент времени `Button` находится ровно в одном состоянии из каждой группы, по умолчанию `Normal` и `Unfocused`. Этот механизм группировки придуман для того, чтобы избежать появления длинных списков, содержащих все возможные сочетания состояний (например, `NormalUnfocused`, `NormalFocused`, `MouseOverUnfocused`, `MouseOverFocused` и т. д).

В табл. 14.2 перечислены все группы и состояния, поддерживаемые встроенными элементами управления WPF. Обратите внимание на «комбинаторный взрыв» в состояниях `DataGridRow` и `DataGridRowHeader`; на самом деле их следовало бы разбить на три группы. (Очевидно, кто-то проглядел служебную записку.) Состояния, унаследованные от базовых классов, не включены; состояния

Button можно найти в строке для ButtonBase. Аналогично для элемента DataGridColumnHeader приведена только группа SortStates, хотя еще две группы он наследует от ButtonBase. Некоторые элементы управления не учитывают состояния, определенные в их базовых классах. Например, ProgressBar поддерживает два состояния из группы CommonStates: Determinate и Indeterminate, но переопределяет функциональность, реализованную в базовом классе RangeBase, таким образом, что его три состояния, отнесенные к группе CommonStates, а также два состояния из группы FocusStates никогда не вызываются.

Таблица 14.2. Состояния и группы состояний, используемые в элементах управления WPF

Элемент управления	Группа состояний	Состояния
ButtonBase	CommonStates	Normal, MouseOver, Pressed, Disabled
	FocusStates	Unfocused, Focused
CalendarButton	SelectionStates	Unselected, Selected
	CalendarButtonFocusStates	CalendarButtonUnfocused, CalendarButtonFocused
	ActiveStates	Inactive, Active
CalendarDayButton	SelectionStates	Unselected, Selected
	CalendarButtonFocusStates	CalendarButtonUnfocused, CalendarButtonFocused
	ActiveStates	Inactive, Active
	DayStates	RegularDay, Today
	BlackoutDayStates	NormalDay, BlackoutDay
CalendarItem	CommonStates	Normal, Disabled
ComboBox	CommonStates	Normal, MouseOver, Disabled
	FocusStates	Unfocused, Focused, FocusedDropDown
	EditStates	Editable, Uneditable
ComboBoxItem	CommonStates	Normal, MouseOver
	SelectionStates	Unselected, Selected, SelectedUnfocused
	FocusStates	Unfocused, Focused
Control	ValidationStates	Valid, InvalidFocused, InvalidUnfocused
DataGrid	CommonStates	Normal, Disabled
DataGridCell	CommonStates	Normal, MouseOver
	SelectionStates	Unselected, Selected
	FocusStates	Unfocused, Focused
	CurrentStates	Regular, Current
	InteractionStates	Display, Editing

Таблица 14.2 (продолжение)

Элемент управления	Имя части	Тип части
DataGridColumnHeader	SortStates	Unsorted, SortAscending, SortDescending
DataGridRow	CommonStates	Normal, Normal_AlternatingRow, Normal_Editing, Normal_Selected, Unfocused_Editing, Unfocused_Selected, MouseOver, MouseOver_Editing, MouseOver_Selected, MouseOver_Unfocused_Editing, MouseOver_Unfocused_Selected
DataGridRowHeader	CommonStates	Normal, Normal_Selected, Normal_EditingRow, Normal_CurrentRow, Normal_CurrentRow_Selected, Unfocused_Selected, Unfocused_EditingRow, Unfocused_CurrentRow_Selected, MouseOver, MouseOver_Selected, MouseOver_EditingRow, MouseOver_CurrentRow, MouseOver_CurrentRow_Selected, MouseOver_Unfocused_Selected, MouseOver_Unfocused_EditingRow, MouseOver_Unfocused_CurrentRow_Selected
DatePicker	CommonStates	Normal, Disabled
DatePickerTextBox	WatermarkStates	Unwatermarked, Watermarked
Expander	CommonStates	Normal, MouseOver, Disabled
	FocusStates	Unfocused, Focused
	ExpansionStates	Expanded, Collapsed
	ExpandDirectionStates	ExpandDown, ExpandUp, ExpandLeft, ExpandRight
GridSpliter	CommonStates	Normal, MouseOver, Disabled
ListBoxItem	CommonStates	Normal, MouseOver, Disabled
	SelectionStates	Unselected, Selected, SelectedUnfocused
	FocusStates	Unfocused, Focused
ProgressBar	CommonStates	Determinate, Indeterminate
RangeBase	CommonStates	Normal, MouseOver, Disabled
	FocusStates	Unfocused, Focused
ScrollBar	CommonStates	Normal, MouseOver, Disabled
	FocusStates	Unfocused, Focused
TabControl	CommonStates	Normal, Disabled
TabItem	CommonStates	Normal, MouseOver, Disabled
	SelectionStates	Unselected, Selected
	FocusStates	Unfocused, Focused

Элемент управления	Имя части	Тип части
TextBox	CommonStates	ReadOnly **(и состояния из** TextBoxBase**)**
TextBoxBase	CommonStates	Normal, MouseOver, Disabled
	FocusStates	Unfocused, Focused
Thumb	CommonStates	Normal, MouseOver, Pressed, Disabled
	FocusStates	Unfocused, Focused
ToggleButton	CheckStates	Checked, Unchecked, Indeterminate
ToolTip	OpenStates	Open, Closed
	FocusStates	Unfocused, Focused
TreeViewItem	CommonStates	Normal, MouseOver, Disabled
	SelectionStates	Unselected, Selected, SelectedInactive
	FocusStates	Unfocused, Focused
	ExpansionStates	Collapsed, Expanded
	HasItemsStates	HasItems, NoItems

Чтобы воспользоваться теми плюсами, которые дают состояния, следует присоединить свойство VisualStateManager.VisualStateGroups **к корневому элементу визуального дерева шаблона и записать в него коллекцию объектов** VisualStateGroup**, в каждом из которых содержится коллекция дочерних элементов** VisualState**.**

В листинге 14.11 приведена модифицированная версия шаблона из листинга 14.9, представляющего индикатор ProgressBar **в виде секторной диаграммы; в данном случае мы воспользовались визуальными состояниями. Поскольку** ProgressBar **поддерживает только состояния** Determinate **и** Indeterminate**, но не различает состояния** Normal **и** Disabled**, то в этом шаблоне все-таки придется оставить один триггер для случая, когда свойство** IsEnabled **становится равным** false**. Но предыдущий триггер, который срабатывал, когда свойство** IsIndeterminate **принимало значение** true**, теперь заменен манипуляциями с визуальным состоянием** Indeterminate**.**

Листинг 14.11. Модифицированный шаблон для представления индикатора выполнения в виде секторной диаграммы – на основе VSM

```
<Application x:Class="WindowsApplication1.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WindowsApplication1"
             StartupUri="Window1.xaml">
  <Application.Resources>

    <ControlTemplate x:Key="progressPie" TargetType="{x:Type ProgressBar}">
```

```
  <!-- Ресурсы -->
  <ControlTemplate.Resources>
    <local:ValueMinMaxToPointConverter x:Key="converter1"/>
    <local:ValueMinMaxToIsLargeArcConverter x:Key="converter2"/>
  </ControlTemplate.Resources>

  <!-- Визуальное дерево -->
  <Viewbox>
<!-- Группы визуальных состояний -->
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="CommonStates">
    <VisualState Name="Determinate"/> <!-- в этом состоянии делать нечего -->
    <VisualState Name="Indeterminate">
      <Storyboard>
        <DoubleAnimation Storyboard.TargetName="pie"
          Storyboard.TargetProperty="Opacity" To="0" Duration="0"/>
        <DoubleAnimation Storyboard.TargetName="backgroundNormal"
          Storyboard.TargetProperty="Opacity" To="0" Duration="0"/>
        <DoubleAnimation Storyboard.TargetName="backgroundIndeterminate"
          Storyboard.TargetProperty="Opacity" To="1" Duration="0"/>
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
    <Grid Width="20" Height="20">
      <Ellipse x:Name="backgroundIndeterminate" Opacity="0"
        Stroke="{TemplateBinding BorderBrush}"
        StrokeThickness="{TemplateBinding BorderThickness}"
        Width="20" Height="20">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
          <GradientStop Offset="0" Color="Yellow"/>
          <GradientStop Offset="1" Color="Brown"/>
        </LinearGradientBrush>
      </Ellipse.Fill>
      </Ellipse>
      <Ellipse x:Name="backgroundNormal" Stroke="{TemplateBinding BorderBrush}"
        StrokeThickness="{TemplateBinding BorderThickness}"
        Width="20" Height="20" Fill="{TemplateBinding Background}"/>
      <Path x:Name="pie" Fill="{TemplateBinding Foreground}">
        <Path.Data>
          <PathGeometry>
            <PathFigure StartPoint="10,10" IsClosed="True">
              <LineSegment Point="10,0"/>
              <ArcSegment Size="10,10" SweepDirection="Clockwise">
              <ArcSegment.Point>
                <MultiBinding Converter="{StaticResource converter1}">
                  <Binding RelativeSource="{RelativeSource TemplatedParent}"
                    Path="Value"/>
                  <Binding RelativeSource="{RelativeSource TemplatedParent}"
                    Path="Minimum"/>
```

```
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Maximum"/>
              </MultiBinding>
            </ArcSegment.Point>
            <ArcSegment.IsLargeArc>
              <MultiBinding Converter="{StaticResource converter2}">
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Value"/>
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Minimum"/>
                <Binding RelativeSource="{RelativeSource TemplatedParent}"
                  Path="Maximum"/>
              </MultiBinding>
            </ArcSegment.IsLargeArc>
            </ArcSegment>
          </PathFigure>
        </PathGeometry>
      </Path.Data>
    </Path>
  </Grid>
</Viewbox>

<!-- Только один триггер -->
<ControlTemplate.Triggers>
  <Trigger Property="IsEnabled" Value="False">
    <Setter TargetName="pie" Property="Fill">
    <Setter.Value>
      <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
        <GradientStop Offset="0" Color="Gray"/>
        <GradientStop Offset="1" Color="White"/>
      </LinearGradientBrush>
    </Setter.Value>
    </Setter>
  </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>

</Application.Resources>
</Application>
```

Внутри каждого элемента `VisualState` находится элемент `Storyboard`, который мы будем подробно рассматривать в главе 17. Он позволяет изменять значения некоторых свойств либо мгновенно (как в листинге 14.11), либо путем плавного перехода. `Storyboard` не дает возможности перейти от произвольной заливки `Fill` эллипса, образующего фон, к заливке кистью `LinearGradientBrush`, поэтому в визуальном дереве теперь использовано *два* эллипса — `backgroundNormal`, показываемый по умолчанию, и `backgroundIndeterminate`, который изначально не виден (так как его свойство `Opacity` равно `0`). Теперь переход в визуальное состояние `Indeterminate` осуществляется путем мгновенной «анимации», которая сводится к присвоению свойству `Opacity` эллипса `backgroundNormal` значения `0`,

а тому же свойству эллипса backgroundIndeterminate — значения 1. Чтобы переход был постепенным, можно увеличить значение свойства Duration обоих элементов DoubleAnimation. В главе 17 мы расскажем о гибкости этих объектов анимации во всех деталях. Там же мы вернемся к шаблону элемента управления Button (см. листинг 14.8) и покажем, как переписать его с использованием VSM.

Как и в случае частей, элементы управления должны документировать свои группы состояний и состояния с помощью атрибута TemplateVisualStateAttribute. Однако в настоящее время встроенные элементы управления WPF этого не делают.

СОВЕТ

В классе VisualStateGroup имеется свойство Transitions — коллекция, в которую можно поместить один или несколько объектов VisualTransition, обеспечивающих анимированные переходы между любыми комбинациями состояний. Подробности см. в главе 17.

Комбинирование шаблонов со стилями

Хотя все рассмотренные до сих пор шаблоны для простоты применялись непосредственно к элементам управления, чаще свойство Template элемента Control задают внутри элемента Style, а затем применяют для стилизации нужных элементов:

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="{x:Type Button}">
      ...
    </ControlTemplate>
  </Setter.Value>
  </Setter>
  ...
</Style>
```

Помимо удобства комбинирования шаблона с произвольными значениями свойств, у такого подхода есть и более весомые достоинства:

- Мы получаем эффект шаблонов по умолчанию. Например, когда некий типизированный стиль применяется к элементам по умолчанию и при этом содержит нестандартный шаблон, то фактически этот шаблон применяется без каких-либо явных пометок в самих элементах!
- Открывается возможность организовать подразумеваемые по умолчанию, но вместе с тем допускающие переопределение значения свойств, контроли-

рующих внешний вид шаблона. Иными словами, можно учитывать свойства, установленные в шаблоне-родителе, но при этом предусматривать собственные значения по умолчанию.

Последнее замечание прямо относится к рассмотренным выше шаблонам. Я хотел, чтобы сектор в шаблоне ProgressBar по умолчанию заливался зеленой градиентной кистью. Если бы такая кисть была зашита в код шаблона, то клиенты не смогли бы изменить способ заливки. С другой стороны, привязываясь к свойству Foreground шаблона-родителя (что сделано в листинге 14.9), я возлагаю на каждый элемент ProgressBar обязанность самостоятельно установить свое свойство Foreground. По умолчанию Foreground в элементе ProgressBar – сплошной зеленый цвет, а не желаемый градиент!

Однако, поместив зеленую градиентную кисть в элемент Setter внутри Style, мы получаем желаемый вид по умолчанию, не запрещая отдельным элементам ProgressBar переопределять заливку путем локальной установки своего свойства Foreground. И выражение {TemplateBinding Foreground} внутри шаблона менять не приходится. Элемент Style мог бы выглядеть следующим образом:

```
<Style x:Key="pieStyle" TargetType="{x:Type ProgressBar}">
  <Setter Property="Foreground">
  <Setter.Value>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Offset="0" Color="LightGreen"/>
      <GradientStop Offset="1" Color="DarkGreen"/>
    </LinearGradientBrush>
  </Setter.Value>
  </Setter>
  <Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="{x:Type ProgressBar}">
    ...
      <Path x:Name="pie" Fill="{TemplateBinding Foreground}">
    ...
    </ControlTemplate>
  </Setter.Value>
  </Setter>
</Style>
```

При применении подобного стиля можно поступить так:

```
<!-- Использовать градиентную заливку по умолчанию -->
<ProgressBar Style="{StaticResource pieStyle}"
  Width="100" Height="100" Value="10"/>
<!-- Использовать вместо этого заливку красным цветом  -->
<ProgressBar Style="{StaticResource pieStyle}" Foreground="Red"
  Width="100" Height="100" Value="10"/>
```

Разумеется, аналогичный подход применим и к другим свойствам, например Width и Height.

КОПНЕМ ГЛУБЖЕ

Взаимодействие между стилями и их шаблонами

Если элемент `Style` содержит шаблон, то одно и то же свойство может устанавливаться в разных местах: в триггерах внутри `Style`, в триггерах, находящихся в шаблоне внутри `Style`, и в элементе `Setter` внутри `Style`! При этом приоритетность соблюдается именно в том порядке, как перечислено в предыдущем предложении. То есть триггеры стилей важнее триггеров шаблонов и любые триггеры важнее элементов `Setter` в стилях.

FAQ

Можно ли внести мелкие поправки в существующий шаблон элемента управления, а не создавать новый с нуля?

Механизма модификации существующих шаблонов (наподобие свойства `BasedOn` в классе `Style`) не существует. Но можно без труда извлечь XAML-разметку любого имеющегося стиля или шаблона, модифицировать ее и применить в качестве нового стиля или шаблона. На самом деле, даже если вы захотите создать совершенно другой внешний облик, лучший способ узнать о том, как проектируются надежные шаблоны элементов управления, – ознакомиться со встроенными в WPF шаблонами, которые используются в стилях разных тем.

Чтобы получить «визуальный исходный код» XAML-разметки любого шаблона элемента управления, достаточно написать такую строку (после того, как завершится компоновка элемента управления, то есть после применения шаблона):

```
string xaml = XamlWriter.Save(someControl.Template);
```

Или можно получить весь стиль любого элемента, программно извлекая нужный ресурс. Следующий код извлекает стиль темы элемента, воспользовавшись свойством зависимости с именем `DefaultStyleKey` (описано в разделе «Темы» ниже), чтобы идентифицировать ресурс `Style`:

```
// Получить ключ стиля по умолчанию
object defaultStyleKey = someElement.GetValue(
  FrameworkElement.DefaultStyleKeyProperty);
// Извлечь ресурс с этим ключом
Style style = (Style)Application.Current.FindResource(defaultStyleKey);
// Сериализовать его XAML-представление в виде строки
string xaml = System.Windows.Markup.XamlWriter.Save(style);
```

Для других типов стилей можно вызвать метод `FindResource` с соответствующим ключом, например `typeof(Button)` для типизированного стиля `Button` (если таковой существует).

Есть также ряд других способов, не требующих написания кода:

- Обратиться к Windows SDK, где имеются XAML-файлы со всеми стилями тем, используемыми в элементах управления WPF.

- Воспользоваться программой .NET Reflector с надстройкой BAML Viewer для просмотра стилей, внедренных в различные сборки, например `PresentationFramework.Aero.dll`.
- Создать интересующий элемент управления в Expression Blend, а затем выбрать команды меню Edit Template→Edit a Copy… (Править шаблон→Править копию…), чтобы получить копию его стиля и вставить ее в свою XAML-разметку. (XAML-файлы для всех тем, поддерживаемых элементами управления WPF, можно найти также в развернутом дистрибутиве Blend в папке *Program Files*.)

Лично мне больше всего нравится последний вариант. В дистрибутив Blend включены также «простые стили» для наиболее распространенных элементов управления, которые гораздо проще понять и модифицировать. Это может послужить неплохой отправной точкой для создания собственных шаблонов.

Обложки

Под *сменой обложки* (скина) понимается изменение внешнего вида приложения «на лету», обычно в программах сторонних фирм. В WPF нет четко определенного понятия обложки, да оно ей и не нужно. Нетрудно написать приложение или компонент, который будет поддерживать динамическую смену обложки с помощью механизма динамических ресурсов WPF (см. главу 12 «Ресурсы») в сочетании со стилями или шаблонами.

Для поддержки смены обложки в приложении нужно прежде всего определиться с форматом данных. Для приложений Win32 или Windows Forms, может быть, и имеет смысл изобретать собственный формат, но в WPF-приложениях уже есть готовый формат – XAML, *если*, конечно, вы не возражаете против загрузки в свой процесс произвольного кода. (Загрузка написанного кем-то XAML-кода сродни загрузке чужой надстройки; он может вызывать не относящийся к программе код и, следовательно, совершить что-то злонамеренное. Дополнительные сведения см. во врезке FAQ в конце этого раздела.)

Но как мог бы выглядеть такой XAML-код?

Часто первой на ум приходит мысль динамически загрузить весь элемент `Window` или `Page` из автономного XAML-файла, а потом подключить нужную логику (применяя технику, описанную в конце главы 2 «Все тайны XAML»). Загрузка пользовательского интерфейса целиком «на лету» дает абсолютную гибкость, но в большинстве случаев она оказывается *чрезмерной*. Авторы таких XAML-файлов должны строго соблюдать соглашения, включив все необходимые элементы с правильными именами, обработчиками событий и т. д. (Или же код, подключающий пользовательский интерфейс, должен быть исключительно снисходительным к ошибкам.) В Visual Studio 2010 такой подход применяется к начальной странице, которая основана на XAML. Загрузив произвольный элемент `Page`, авторы могут полностью изменить внешний вид этой страницы. Если нужно всего лишь сменить обложку, то достаточно скопировать существующий элемент `Page` и модифицировать его.

Но если вы не хотите поощрять полную замену пользовательского интерфейса, то лучше всего сделать корнем представления обложки элемент `ResourceDictionary`. В общем и целом, словарь ресурсов представляет собой отличную точку расширяемости из-за той легкости, с которой его можно загружать и выгружать или объединять с другими словарями. Определяя обложку, имеет смысл включить в `ResourceDictionary` стили и/или шаблоны.

Продемонстрируем технологию смены обложки на следующем примере, где окно `Window` – гипотетическое диалоговое окно, иллюстрирующее ход выполнения процесса (рис. 14.13):

```
<Window x:Class="WindowsApplication1.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Please Wait" Height="200" Width="300" ResizeMode="NoResize">
  <Grid>
    <StackPanel Style="{DynamicResource DialogStyle}">
      <Label Style="{DynamicResource HeadingStyle}">Loading...</Label>
      <ProgressBar Value="35" MinHeight="20" Margin="20"/>
      <Button Style="{DynamicResource CancelButtonStyle}" Width="70"
        Click="Cancel_Click">Cancel</Button>
    </StackPanel>
   </Grid>
</Window>
```

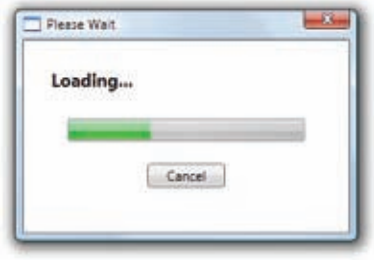


Рис. 14.13. *Диалоговое окно с обложкой, подразумеваемой по умолчанию*

Обратите внимание, что для большинства элементов окна заданы явные стили `Style`. Это необязательно для организации обложки, но часто оказывается полезным приемом, оставляющим авторам обложек больше контроля над ее внешним видом. Предположим, например, что по вашей задумке кнопка `Cancel` (Отмена) должна внешне отличаться от всех остальных кнопок. Для этого будет достаточно явно пометить все кнопки `Cancel` стилем `CancelButtonStyle`. А явно заданные стили можно изменять в любой момент, если элементы ссылаются на них как на *динамические* ресурсы.

Чтобы приведенный выше элемент `Window` визуализировался, как показано на рис. 14.13, его надо объединить с файлом *App.xaml*, в котором находятся определения всех ресурсов `Style`, подразумеваемые по умолчанию:

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="Window1.xaml">
  <Application.Resources>
    <Style x:Key="DialogStyle" TargetType="{x:Type StackPanel}">
      <Setter Property="Margin" Value="20"/>
    </Style>
    <Style x:Key="HeadingStyle" TargetType="{x:Type Label}">
      <Setter Property="FontSize" Value="16"/>
      <Setter Property="FontWeight" Value="Bold"/>
    </Style>
    <Style x:Key="CancelButtonStyle" TargetType="{x:Type Button}"/>
  </Application.Resources>
</Application>
```

Отметим, что стиль `CancelButtonStyle` пуст, поэтому применение его к кнопке не дает никакого эффекта. И это правильно, поскольку идея в том, что обложка подменит этот стиль чем-то более осмысленным.

СОВЕТ

Назначая элементу стиль, который предполагается менять динамически в процессе смены обложки, не забывайте ссылаться на него как на динамический ресурс!

Теперь нетрудно написать и файл обложки:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Style x:Key="DialogStyle" TargetType="{x:Type StackPanel}">
  ...
  </Style>
  <Style x:Key="HeadingStyle" TargetType="{x:Type Label}">
  ...
  </Style>
  <Style x:Key="CancelButtonStyle" TargetType="{x:Type Button}">
  ...
  </Style>
  Дополнительные стили...
</ResourceDictionary>
```

После этого приложению остается только динамически загрузить XAML-файл обложки и назначить его в качестве нового словаря `Application.Resources`. Следующий код делает это для XAML-файла, находящегося в текущем каталоге:

```
ResourceDictionary resources = null;
using (FileStream fs = new FileStream("CustomSkin.xaml",
   FileMode.Open, FileAccess.Read))
{
```

```
    // Получить корневой элемент, который должен быть словарем ResourceDictionary
    resources = (ResourceDictionary)XamlReader.Load(fs);
}
Application.Current.Resources = resources;
```

А можно вместо этого загрузить обложку из Интернета, если известен URL-адрес:

```
ResourceDictionary resources = null;
System.Net.WebClient client = new System.Net.WebClient();
using (Stream s = client.OpenRead("http://adamnathan.net/wpf/CustomSkin.xaml"))
{
  // Получить корневой элемент, который должен быть словарем ResourceDictionary
  resources = (ResourceDictionary)XamlReader.Load(s);
}
Application.Current.Resources = resources;
```

Поскольку присваивание свойству `Application.Current.Resources` уничтожает текущий словарь, необходимо где-то сохранить словарь `ResourceDictionary`, подразумеваемый по умолчанию, на случай если впоследствии захочется его восстановить!

FAQ

Что произойдет, если в обложке не определен именованный стиль, нужный приложению?

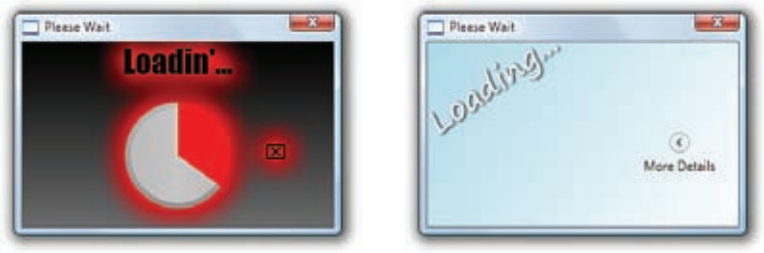
Если вы решите полностью подменять текущий словарь `Application.Resources` новым, а в последнем какие-то стили отсутствуют, то те элементы управления, к которым эти стили применяются, просто вернутся к своему виду по умолчанию. Это справедливо и в отношении ресурсов, которые динамически удаляются во время работы приложения. Однако при этом генерируется отладочная трассировка – так же, как в случае ошибок привязки к данным. Например, попытка применить обложку, в которой нет стиля `CancelButtonStyle`, приведет к появлению такого сообщения в отладчике:

```
System.Windows.ResourceDictionary Warning: 9 : Resource not found;
ResourceKey='CancelButtonStyle'
```

Чтобы избежать этого, можно поступить по-другому: обойти новый словарь ресурсов и отдельно для каждой пары ключ/значение включить ресурс в словарь ресурсов приложения.

В примере диалогового окна с информацией о ходе загрузке (его полный исходный код можно найти по адресу *http://informit.com/title/9780672331190* вместе с остальными примерами из этой книги) обложка переключается при нажатии кнопки `Cancel`. В реальном приложении, наверное, был бы предусмотрен специальный интерфейс для выбора обложки.

В исходном коде к книге есть две альтернативные обложки для диалогового окна на рис. 14.13; обе показаны на рис. 14.14.



Обложка «Электричество» Обложка «Мягкий и пушистый»

Рис. 14.14. *Две альтернативные обложки диалогового окна*

Отметим, что в обложке «Электричество» переопределен стиль элемента `ProgressBar` (с использованием шаблона в виде секторной диаграммы, обсуждавшегося в предыдущем разделе), хотя в самом приложении ему не назначен стиль явно. Для этого стиль сделан типизированным – применяемым ко всем индикаторам `ProgressBar`. К счастью, любые добавления, удаления или изменения типизированных стилей в словаре `ResourceDictionary` автоматически отражаются в интерфейсе точно так же, как явно заданные динамические ресурсы. В стиле `CancelButtonStyle`, присутствующем в обложке, используется преобразование `TranslateTransform`, которое перемещает кнопку так, чтобы она оказалась рядом с индикатором, а не под ним. Кроме того, делается нечто совершенно необычное для стиля метки `Label`: с помощью шаблона содержимое метки пропускается через веб-службу перевода на диалект джайв[1] (разумеется, это работает, только если метка содержит текст).

КОПНЕМ ГЛУБЖЕ

Обложки, нуждающиеся в процедурном коде

Шаблону элемента `ProgressBar` в обложке «Электричество» необходим процедурный код (как показано в предыдущем разделе), поэтому реализовать его в виде автономного XAML-файла невозможно. В таких случаях можно включить в сборку откомпилированную версию словаря `ResourceDictionary` и сделать эту сборку обложкой. Нужно только использовать метод `Application.LoadComponent` для извлечения откомпилированного ресурса, который может находиться в одной сборке с кодом или в отдельной (см. главу 12).

В примере диалогового окна с информацией о ходе загрузки обе обложки находятся в одной и той же сборке, поэтому для их загрузки применяется такой код:

```
ResourceDictionary resources = (ResourceDictionary)Application.LoadComponent(
  new Uri("CustomSkin.xaml", UriKind.RelativeOrAbsolute));
Application.Current.Resources = resources;
```

[1] Комический диалект английского языка, пародия на афро-американский английский. – *Прим. перев.*

Обложка «Мягкий и пушистый» тоже содержит ряд радикальных изменений. В листинге 14.12 приведен ее полный исходный код.

Листинг 14.12. Обложка «Мягкий и пушистый»

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <!-- Фон заливается простой градиентной кистью -->
  <Style x:Key="DialogStyle" TargetType="{x:Type StackPanel}">
    <Setter Property="Margin" Value="0"/>
    <Setter Property="Background">
    <Setter.Value>
      <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
        <GradientStop Offset="0" Color="LightBlue"/>
        <GradientStop Offset="1" Color="White"/>
      </LinearGradientBrush>
    </Setter.Value>
    </Setter>
  </Style>

  <!-- Поворачиваем и сдвигаем основной текст -->
  <Style x:Key="HeadingStyle" TargetType="{x:Type Label}">
    <Setter Property="Foreground" Value="White"/>
    <Setter Property="FontSize" Value="30"/>
    <Setter Property="FontFamily" Value="Segoe Print"/>
    <Setter Property="RenderTransform">
    <Setter.Value>
      <TransformGroup>
        <RotateTransform Angle="-35"/>
        <TranslateTransform X="-19" Y="55"/>
      </TransformGroup>
    </Setter.Value>
    </Setter>
    <Setter Property="Effect">
    <Setter.Value>
      <DropShadowEffect ShadowDepth="2"/>
    </Setter.Value>
    </Setter>
  </Style>

  <!-- Удаляем кнопку Cancel -->
  <Style x:Key="CancelButtonStyle" TargetType="{x:Type Button}">
    <Setter Property="Visibility" Value="Collapsed"/>
  </Style>

  <!-- Помещаем ProgressBar внутрь Expander -->
  <Style TargetType="{x:Type ProgressBar}">
    <Setter Property="Height" Value="100"/>
   <Setter Property="Template">
   <Setter.Value>
```

```
        <ControlTemplate TargetType="{x:Type ProgressBar}">
          <Expander Header="More Details" ExpandDirection="Left">
            <ProgressBar Style="{x:Null}"
              Height="30" Value="{TemplateBinding Value}"
              Minimum="{TemplateBinding Minimum}"
              Maximum="{TemplateBinding Maximum}"
              IsEnabled="{TemplateBinding IsEnabled}"
              IsIndeterminate="{TemplateBinding IsIndeterminate}"/>
          </Expander>
        </ControlTemplate>
      </Setter.Value>
      </Setter>
    </Style>
  </ResourceDictionary>
```

Модификация стилей `DialogStyle` и `HeadingStyle` достаточно проста (хотя в последнем случае мы добавили стильный эффект тени, который будем рассматривать в следующей главе). Но в этой обложке, идея которой – создать минималистский интерфейс, стиль `CancelButtonStyle` полностью *скрывает* кнопку Cancel! В данном случае это вполне оправдано (при условии, конечно, что операция закрытия окна ведет себя обычным образом). Но в других случаях пользователь может и не оценить все остроумие обложки, скрывающей части интерфейса!

В типизированном стиле элементов `ProgressBar` также применен интересный прием с целью упростить пользовательский интерфейс. Мы определили шаблон, помещающий `ProgressBar` внутрь элемента `Expander` (по умолчанию свернутого)! В обернутом таким образом индикаторе есть несколько привязок `TemplateBinding`, отвечающих за синхронизацию с шаблоном-родителем. Отметим, что вложенному индикатору `ProgressBar` назначен стиль `null`. Это необходимо, чтобы избежать бесконечной рекурсии. Не будь такого явного элемента `Style`, к внутреннему `ProgressBar` был бы по умолчанию применен типизированный стиль, то есть мы получили бы `Expander` внутри `Expander` внутри `Expander` и т. д.

FAQ

Как предотвратить вредоносные действия сторонней обложки?

Никакого встроенного механизма не существует. Может возникнуть искушение попробовать написать свой код, который будет исследовать полученный от пользователя словарь `ResourceDictionary` и удалять из него все, что покажется вам подозрительным, но это, в общем-то, бесполезное занятие. Например, желая запретить обложке скрывать какие-нибудь элементы управления, вы можете без особого труда удалить элементы `Setter`, воздействующие на свойство `Visibility`. Но что если обложка нарисует текст таким же цветом, как и фон? Или назначит элементу управления шаблон, из-за которого тот будет выглядеть пустым? Есть много способов попасть впросак!

Но возможность получить неработоспособный пользовательский интерфейс – это еще меньшее из зол. Представьте себе, что в обложке реализован какой-то способ отправлять на веб-сервер конфиденциальную информацию, отображаемую в приложении. Выполнение произвольного кода (в том числе XAML-кода!) в контексте приложения с полным доверием всегда чревато риском. Один из обходных путей – загрузка XAML в отдельный процесс, но в большинстве случаев овчинка не стоит выделки.

Если же вы всерьез озабочены этой проблемой, то, наверное, стоит определить свой формат данных обложки, обладающий гораздо более ограниченными возможностями. Однако если вы предоставите конечному пользователю простой способ удалить «вредоносную обложку», то, скорее всего, и беспокоиться на этот счет не стоит.

Темы

Если *обложки* применяются к одному приложению, то *темы* обычно влияют на те визуальные характеристики операционной системы, которые отражаются в пользовательском интерфейсе всех программ. Например, если установлена тема Windows Classic, то все кнопки и полосы прокрутки выглядят плоскими и прямоугольными. В Windows XP переключение между разными цветовыми схемами применяемой по умолчанию темы (голубая, оливковая, серебристая) влияет на цвета стандартных элементов управления. Для сохранения согласованности с выбранной темой Windows у встроенных элементов управления WPF имеются отдельные шаблоны управления для каждой темы (для кнопки `Button` мы их видели в главе 9 «Однодетные элементы управления»).

Согласованность с темой операционной системы важна в шаблонах, применяемых по умолчанию. Но, создавая собственные шаблоны элементов управления, автор обычно хочет, чтобы они как можно *меньше* походили на операционную систему! И все же имеет смысл учитывать хотя бы некоторые особенности темы операционной системы, чтобы ваши элементы не слишком кололи глаза. При создании собственных элементов управления, которые по умолчанию сочетались бы с темой операционной системы, также важно понимать, как работает механизм тем.

В этом разделе мы покажем, как можно легко создавать стили и шаблоны (а значит, и обложки), адаптирующиеся к текущей теме. Сделать это можно двумя способами. Первый простой, но не слишком гибкий, второй требует чуть больше работы, зато предоставляет любые возможности.

Системные цвета, шрифты и параметры

Свойства, определенные в классах `SystemColors`, `SystemFonts` и `SystemParameters`, автоматически обновляются при смене темы Windows. Поэтому, чтобы обеспечить согласованность с выбранной пользователем темой, достаточно включить их в свои стили и шаблоны.

В следующем модифицированном стиле индикатора ProgressBar в виде секторной диаграммы класс SystemColors используется для задания подразумеваемых по умолчанию цветов заливки (описание этой техники см. в главе 12):

```
<Style TargetType="{x:Type ProgressBar}">
  <Style.Resources>
    <LinearGradientBrush x:Key="foregroundBrush" StartPoint="0,0" EndPoint="1,1">
      <GradientStop Offset="0"
        Color="{DynamicResource {x:Static SystemColors.InactiveCaptionColorKey}}"/>
      <GradientStop Offset="0.5"
        Color="{DynamicResource {x:Static SystemColors.InactiveCaptionColorKey}}"/>
      <GradientStop Offset="1"
        Color="{DynamicResource {x:Static SystemColors.ActiveCaptionColorKey}}"/>
    </LinearGradientBrush>
  </Style.Resources>
  <Setter Property="Foreground" Value="{StaticResource foregroundBrush}"/>
  <Setter Property="Background"
    Value="{DynamicResource {x:Static SystemColors.ControlBrushKey}}"/>
  ...
</Style>
```

На рис. 14.15 показаны тонкие различия в этом стиле при смене темы Windows.



Windows 7 (Aero) Windows Classic

***Рис. 14.15.** Один и тот же элемент управления с одинаковым стилем, но в разных темах*

Стили и шаблоны тем

Многие встроенные элементы WPF в разных темах отличаются не только цветами, шрифтами и простыми линейными размерами. Например в теме Windows 7 Aero они кажутся более блестящими, а в теме Windows Classic – несколько тусклыми. Достигается это за счет того, что для каждой темы определен свой шаблон.

Возможность определять отдельные стили и шаблоны, нетривиальным образом зависящие от текущей темы, может оказаться весьма полезной. Например, кому-то показанный на рис. 14.15 вариант ProgressBar для темы Windows Classic может показаться *слишком* уж помпезным! Человек, выбравший эту тему, наверное, не любит затейливые градиенты и прочие эффекты.

Если вы решили создать собственные стили и шаблоны, адаптированные к темам, то загружать их можно программно в момент смены темы (применяя технику, обсуждавшуюся в разделе «Обложки»). Однако WPF не генерирует

никаких событий при смене тем, поэтому придется перехватывать сообщение Win32 `WM_THEMECHANGE` (так же как мы перехватывали сообщение `WM_DWMCOMPOSITIONCHANGED` в главе 8 «Особенности Windows 7»). Но, к счастью, WPF предлагает основанный на низкоуровневом Win32 API механизм адаптации к темам, который позволяет включать ресурсы для разных тем почти без процедурного кода.

Первым делом следует разнести зависящие от тем ресурсы по разным XAML-файлам, включаемым в откомпилированную сборку. Каждый такой файл должен содержать словарь ресурсов для одной темы. Затем можно сделать каждый такой словарь ресурсов *словарем темы*, поместив его в подкаталог `themes` (который должен находиться в корневом каталоге проекта!) и назвав *`ThemeName.ThemeColor`*`.xaml` (регистр букв не имеет значения). WPF будет автоматически загружать и применять словарь темы при запуске приложения и при каждой смене темы. Стили внутри словаря темы называются *стилями темы*.

Ниже перечислены все созданные Microsoft темы и URI соответствующих им словарей:

- Тема Aero (Windows Vista и Windows 7): `themes\Aero.NormalColor.xaml`
- Тема Windows XP по умолчанию: `themes\Luna.NormalColor.xaml`
- Оливковая тема Windows XP: `themes\Luna.Homestead.xaml`
- Серебристая тема Windows XP: `themes\Luna.Metallic.xaml`
- Тема Windows XP Media Center Edition 2005 и Windows XP Tablet PC Edition 2005: `themes\Royale.NormalColor.xaml`
- Тема Windows Classic: `themes\Classic.xaml`
- Тема Zune Windows XP: `themes\Zune.NormalColor.xaml`

Обратите внимание на особенность темы Windows Classic: в URI соответствующего словаря нет части *`ThemeColor`*.

Можно также задать запасной словарь ресурсов, который будет использоваться, если не существует словаря, соответствующего текущей теме и цвету. Такой словарь часто называют *типовым*; он должен именоваться `themes\Generic.xaml`.

СОВЕТ

Не забывайте о типовом словаре, когда будете создавать словари тем. Это поможет обеспечить предсказуемое поведение, если встретится неожиданная тема.

Подготовив словари тем и типовой словарь, вы должны будете явно включать механизм автоматической адаптации к теме, снабдив сборку атрибутом `ThemeInfoAttribute`. Конструктор этого атрибута принимает два параметра типа `ResourceDictionaryLocation`. Первый сообщает, где WPF должна искать словари тем, второй – где находится типовой словарь. Каждый параметр может принимать следующие значения независимо от другого:

- `None` – не искать словарь ресурсов. Это значение по умолчанию.
- `SourceAssembly` – искать в текущей сборке.
- `ExternalAssembly` – искать в другой сборке, которая должна называться *AssemblyName.ThemeName*.`dll` (где *AssemblyName* совпадает с именем текущей сборки). WPF применяет эту схему ко встроенным словарям тем, которые находятся в сборках `PresentationFramework.Aero.dll`, `PresentationFramework.Luna.dll` и т. д. Этот способ позволяет избежать постоянного присутствия в памяти ресурсов для всех тем.

Типичное использование атрибута `ThemeInfoAttribute` выглядит так:

```
// Искать словари тем и типовой словарь в этой сборке
[assembly:ThemeInfo(ResourceDictionaryLocation.SourceAssembly,
                    ResourceDictionaryLocation.SourceAssembly)]
```

В поддержке тем есть еще один нюанс: она позволяет предоставлять стили элементов *по умолчанию*. Как следует из конструкции атрибута `ThemeInfoAttribute`, стили тем должны находиться в одной сборке с элементом, к которому применяются, или в указанной сателлитной сборке. В отличие от словарей ресурсов уровня приложения (или более низкого), невозможно определить типизированный стиль для элементов, определенных в другом месте. Например, вы не сможете определить в своем словаре темы и типовом словаре стили элементов `Button` или `ProgressBar` так, чтобы они переопределяли заданные для них стили по умолчанию, – если только не воспользуетесь дополнительным механизмом с участием `ThemeDictionaryExtension`.

`ThemeDictionaryExtension` – это расширение разметки, позволяющее переопределять стили темы любых элементов. В нем можно сослаться на сборку, содержащую набор словарей тем, и даже на сборку текущего приложения. Расширение `ThemeDictionaryExtension` можно указать в качестве значения свойства `Source` элемента `ResourceDictionary`, и тогда оно будет распространяться на все элементы, находящиеся в области видимости этого словаря. Например:

```
<Application ...>
<Application.Resources>
  <ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary .../>
    <ResourceDictionary Source="{ThemeDictionary MyApplication}"/>
  </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
</Application>
```

Допустим, вы хотите, чтобы стиль секторной диаграммы для `ProgressBar` изменялся в зависимости от темы Windows. Если сборка `MyApplication` содержит стили тем, в которых тип целевого элемента `TargetType` равен {`x:Type ProgressBar`}, то все индикаторы `ProgressBar` в этом приложении по умолчанию получат модифицированный вами стиль темы – благодаря использованию `ThemeDictionaryExtension`.

Другой способ присоединить стили темы к существующим элементам состоит в том, чтобы определить свой подкласс. Разработка нестандартных элементов управления рассматривается в главе 20, но создать подобный элемент исключительно ради назначения ему стиля темы совсем несложно. Так, в примере секторной диаграммы можно было бы создать следующий нестандартный элемент ProgressPie:

```
public class ProgressPie : ProgressBar
{
  static ProgressPie()
  {
    DefaultStyleKeyProperty.OverrideMetadata(
      typeof(ProgressPie),
      new FrameworkPropertyMetadata(typeof(ProgressPie)));
  }
}
```

Поскольку класс ProgressPie наследует ProgressBar, то он автоматически обладает всей необходимой функциональностью. Но коль скоро тип у него все-таки другой, то мы получаем возможность применить новый стиль темы, отличный от стиля темы ProgressBar. Нужно лишь «произнести одно заклинание» – написать статический конструктор ProgressPie, который устанавливает свойство зависимости DefaultStyleKey. Это защищенное свойство зависимости имеется в классах FrameworkElement и FrameworkContentElement и определяет ключ стиля по умолчанию в словаре ресурсов. (Термины *«стиль по умолчанию»* и *«стиль темы»* часто употребляются как синонимы.)

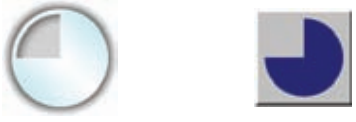
Встроенные элементы WPF указывают в качестве значения этого свойства собственный тип, поэтому в их словарях тем применяются типизированные стили. Если не установить свойство DefaultStyleKey, то класс ProgressPie унаследует его значение от класса ProgressBar, где оно равно typeof(ProgressBar). Поэтому-то ProgressPie задает typeof(ProgressPie) в качестве своего DefaultStyleKey.

В примерах к этой книге есть проект Visual Studio, содержащий показанное выше определение ProgressPie, пример использования атрибута ThemeInfoAttribute и ряд словарей тем, прикомпилированных к приложению. Каждый словарь темы представляет собой автономный XAML-файл со следующей структурой:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:ThemedProgressPie">
  <Style TargetType="{x:Type local:ProgressPie}">
  ...
  </Style>
</ResourceDictionary>
```

На рис. 14.16 показан элемент ProgressPie в стилях двух разных тем. Вы можете покопаться в деталях устройства каждого стиля, но смысл в том, что стили тем обладают гибкостью, достаточной для полного изменения внешнего вида

элемента при смене темы. Я считаю, что на рис. 14.16, в отличие от рис. 14.15, индикатор `ProgressPie` для Windows 7 выглядит очень живенько, а для Windows Classic – совсем уныло. Но шутки в сторону – если стили одного элемента в разных темах *слишком* различны, то это скорее будет мешать, а не помогать пользователю.



Windows 7 (Aero) Windows Classic

***Рис. 14.16.** Один и тот же элемент управления с одинаковым стилем, но в разных темах*

КОПНЕМ ГЛУБЖЕ

Темы и цветовые схемы Windows

В Windows 7 и Windows Vista имеется длинный список цветовых схем в диалоговом окне Дополнительные параметры оформления. Если выбрана схема Windows Aero или Windows Basic, то WPF будет пользоваться словарем темы `Aero.NormalColor`. (Это справедливо и в том случае, когда пользователь изменяет «обычный» цвет окна с помощью настроек на Панели управления.) Если же выбрана Windows Standard, Windows Classic или какая-либо высококонтрастная схема, то WPF будет пользоваться словарем темы `Classic`. Если вы хотите различать цветовые схемы, отображаемые на одну и ту же тему, то оптимальное решение – включить привязку к классу `SystemColors` в свои стили и шаблоны.

Резюме

Сочетание стилей, шаблонов, обложек и тем – чрезвычайно мощное средство, которое часто вызывает затруднения у начинающих изучать WPF. Этому способствует еще и то, что стили могут содержать (и часто содержат) шаблоны, у всех элементов в шаблонах есть стили (явно указанные или неявно наследуемые), а стили тем управляются иначе, чем обычные стили (так, в элементе `Button` свойство `Style` по умолчанию равно `null`, хотя, очевидно, стиль темы к нему применяется).

Все эти механизмы настолько гибкие, что часто удается изменить стиль существующего элемента, вместо того чтобы писать собственный нестандартный элемент управления. И это замечательно, потому что применить стиль к имеющемуся элементу гораздо проще, чем написать новый, причем это вполне по силам графическому дизайнеру без привлечения программиста. Если вы считаете, что писать нестандартный элемент управления все-таки необходимо

(см. главу 20), не забудьте о том, что здесь говорилось по поводу создания надежных шаблонов и адаптации их к темам.

СОВЕТ

Можете поэкспериментировать с альтернативными обложками для многих элементов управления WPF, скачав файл *WPFThemes.zip* с сайта *http://wpf.codeplex.com*. Это «темы», которые в данной главе называются обложками, то есть просто словари ресурсов с определениями новых типизированных стилей для большинства встроенных элементов управления WPF. Чтобы ими воспользоваться, нужно просто сослаться на словарь ресурсов в коллекции `Resources` для элемента `Application`, `Window` или еще какого-то:

```
<Application ...>
<Application.Resources>
  <ResourceDictionary Source="BureauBlack.xaml"/>
</Application.Resources>
</Application>
```

К сожалению, когда писалась эта книга, упомянутые обложки еще не включали стилей для новых элементов управления, появившихся в WPF 4, например `DataGrid`, `Calendar` и `DatePicker`. На рис. 14.17 показано, как выглядят несколько элементов управления в семи различных обложках.



Рис. 14.17. *Обложки из набора «WPF Themes»*

V

Мультимедиа

Глава 15 «Двумерная графика»
Глава 16 «Трехмерная графика»
Глава 17 «Анимация»
Глава 18 «Аудио, видео и речь»

- Класс Drawing
- Класс Visual
- Класс Shape
- Кисти
- Эффекты
- Повышение производительности визуализации

15

Двумерная графика

У приложений и компонентов есть много причин для рисования прямоугольников, эллипсов, линий и других фигур и путей. В большинстве нестандартных шаблонов элементов управления обычно требуется что-то рисовать, чтобы придать элементу необычный внешний вид; в предыдущей главе мы так поступали в шаблонах элементов `Button` и `ProgressBar`. Но иногда приложению нужно что-то нарисовать и для других целей, необязательно в контексте элемента управления. Это может быть логотип продукта или линии, разделяющие области окна. В Сети для этого, как правило, используют готовые изображения, но средства рисования, встроенные в WPF, позволяют получать векторные рисунки, отлично масштабирующиеся под любой размер.

Возможность создавать и использовать векторную двумерную графику не является уникальной особенностью WPF; даже технология GDI позволяла рисовать пути и фигуры. Основное отличие WPF от GDI и других предшествующих технологий Windows в части рисования – это то, что в WPF применяется графическая система, работающая полностью в режиме *запоминания*, а не *непосредственной визуализации*.

В системах с непосредственной визуализацией (GDI, GDI+, DirectX и т. д.) можно рисовать «прямо» на экране, но вы должны сами сохранять состояние всех визуальных элементов. Иначе говоря, обязанность нарисовать правильные пикселы после того, как область экрана объявлена недействительной, возлагается на вас. Недействительность может быть следствием действий пользователя, например изменения размера окна, или приложения, которому потребовалось обновить визуальные элементы.

В системе, работающей в режиме запоминания, вы можете формулировать высокоуровневые указания, например: «Помести синий квадрат размером 10×10 в точку (0,0)», – и система сама запомнит и будет поддерживать это состояние. Иначе говоря, на самом деле ваши слова означают: «Помести синий квадрат размером 10×10 в точку (0,0) *и следи за тем, чтобы он там оставался*». Вам

нет нужды возиться с недействительными областями и перерисовкой, а это экономит немало времени. Эта идея лежит также в основе органичной поддержки в WPF таких особенностей, как перекрывающиеся объекты, прозрачность, видео, независимость от разрешающей способности устройства и многое другое.

Разнообразие вообще характерно для WPF, и способов создания и использования двумерной графики тоже несколько. В этой главе мы сосредоточим внимание на трех важных типах данных: `Drawing`, `Visual` и `Shape`. Между ними существуют довольно сложные взаимосвязи. По большей части рисунки `Drawing` – это просто описания путей и фигур с ассоциированными кистями `Brush` для заливки и контура. Визуальное представление `Visual` – это один из способов нарисовать объект `Drawing` на экране, но класс `Visual` открывает также возможность низкоуровневого и менее ресурсоемкого подхода к рисованию, позволяющего обходиться вообще без объектов `Drawing`. Наконец, фигуры `Shape` – это готовые объекты `Visual`, предлагающие самый простой (но и самый ресурсоемкий) подход к рисованию на экране. Кстати, из этих трех типов в Silverlight поддерживается лишь `Shape`. При изучении классов `Drawing`, `Visual` и `Shape` мы рассмотрим простое изображение и обсудим, как можно его создать и использовать во всех трех контекстах.

В конце главы мы поговорим о кистях `Brush`, специальных эффектах и средствах повышения производительности приложений с интенсивным использованием графики. Кисти – неотъемлемая часть всех затрагиваемых в этой главе тем; они уже использовались в предыдущих главах для решения таких прозаических задач, как установка свойств `Foreground` и `Background` элементов управления. В WPF имеется множество кистей с самой разной функциональностью, потому мы и посвятили им отдельный раздел. Эффекты, например отбрасываемая тень или размывание, используются сравнительно редко, но способны добавить пользовательскому интерфейсу элегантный штрих, который без них реализовать было бы сложно.

Класс Drawing

Абстрактный класс `Drawing` представляет двумерный рисунок. Этот класс, а особенно его подкласс `GeometryDrawing`, играет в WPF роль клипарта (clip art). Его достаточно для описания любой двумерной иллюстрации, и, как и все классы, производные от `Animatable`, он поддерживает еще и анимацию, привязку к данным, ссылки на ресурсы и многое другое!

В WPF включено пять конкретных подклассов `Drawing`:

- `GeometryDrawing` – объединяет объект `Geometry` с кистью `Brush`, служащей для заливки его внутренней области, и пером `Pen`, которым рисуется контур. Этот подкласс больше других относится к теме настоящей главы.
- `ImageDrawing` – объединяет объект `ImageSource` и прямоугольник `Rect`, определяющий границы изображения.
- `VideoDrawing` – объединяет объект `MediaPlayer` (см. главу 18 «Аудио, видео и речь») и ограничивающий прямоугольник `Rect`.

- `GlyphRunDrawing` – объединяет объект `GlyphRun`, низкоуровневый класс для представления текста, с кистью `Brush`, которой рисуется текст.
- `DrawingGroup` – содержит коллекцию объектов `Drawing` и обладает рядом свойств для их группового изменения (`Opacity`, `Transform` и другие). Класс `DrawingGroup` сам является частным случаем `Drawing`, поэтому его можно использовать всюду, где допустим `Drawing`. (Это полный аналог отношения, существующего между классами `TransformGroup` и `Transform`.)

Ниже приведен пример элемента `GeometryDrawing`, в который вложены элемент `Geometry`, описывающий эллипс (`EllipseGeometry`), оранжевая кисть `Brush` и черное перо `Pen`:

```
<GeometryDrawing Brush="Orange">                                    Brush
  <GeometryDrawing.Pen>
    <Pen Brush="Black" Thickness="10"/>                             Pen
  </GeometryDrawing.Pen>
  <GeometryDrawing.Geometry>
    <EllipseGeometry RadiusX="100" RadiusY="50"/>                   Geometry
  </GeometryDrawing.Geometry>
</GeometryDrawing>
```

Элементы `Drawing` не наследуют классу `UIElement`; у них нет собственного поведения визуализации. Поэтому если вы попытаетесь поместить `GeometryDrawing` прямо в `Window` или другой элемент типа `ContentControl`, то получите простой текстовый блок `TextBlock`, содержащий строку `"System.Windows.Media.GeometryDrawing"` (выводится строка, возвращаемая методом `ToString`).

Чтобы элемент `Drawing` визуализировался правильно, его необходимо размещать в одном из следующих объектов-владельцев:

- `DrawingImage` – подкласс `ImageSource`, поэтому его можно использовать внутри `Image` вместо типичного `BitmapImage`.
- `DrawingBrush` – подкласс `Brush`, поэтому его можно применять в самых разных местах, например в качестве кисти `Foreground`, `Background` или `BorderBrush` для любого элемента, производного от `Control`.
- `DrawingVisual` – подкласс `Visual`, рассматривается ниже в разделе «Класс Visual».

Следовательно, чтобы показанный выше элемент `DrawingImage` появился на экране, его можно использовать следующим образом:

```
<Image>
  <Image.Source>
    <DrawingImage>
    <DrawingImage.Drawing>
      <GeometryDrawing Brush="Orange">
      <GeometryDrawing.Pen>
        <Pen Brush="Black" Thickness="10"/>
      </GeometryDrawing.Pen>
      <GeometryDrawing.Geometry>
        <EllipseGeometry RadiusX="100" RadiusY="50"/>
```

```
        </GeometryDrawing.Geometry>
        </GeometryDrawing>
      </DrawingImage.Drawing>
      </DrawingImage>
    </Image.Source>
  </Image>
```

На рис. 15.1 показан результат визуализации этого элемента `Image` – как видите, внутри находится `GeometryDrawing`.



Рис. 15.1. *Простой элемент EllipseGeometry, вложенный в GeometryDrawing, который сам вложен в DrawingImage, находящийся внутри Image*

КОПНЕМ ГЛУБЖЕ

DrawingImage и ImageDrawing

Поначалу трудно разобраться, в чем разница между классом `DrawingImage` и упоминавшимся ранее классом `ImageDrawing`. Оба они интересны тем, что позволяют смешивать векторную и растровую графику.

`DrawingImage` – подкласс `ImageSource` и в качестве содержимого допускает типичный векторный рисунок `Drawing`, а не растровое изображение. Напротив, `ImageDrawing` – подкласс `Drawing`, и его содержимым может быть растровый объект `ImageSource`, а не векторный.

Уловить разницу позволяет следующий нехитрый прием: почти для всех графических классов в WPF (двумерных и трехмерных) составное имя вида *FooBar* означает, что этот класс *является* подклассом *Bar*, который может *содержать* или *работать как Foo*. Следовательно, `DrawingImage` – это `ImageSource`, содержащий `Drawing`, а `ImageDrawing` – это `Drawing`, содержащий `ImageSource`.

Тот факт, что `DrawingImage` – частный случай `ImageSource`, открывает возможность генерировать изображения для векторного содержимого и использовать их в совершенно неожиданных местах. `Window.Icon` – это `ImageSource`, равно как `TaskbarItemInfo.Overlay` и `ThumbButtonInfo.ImageSource` (см. главу 8 «Особенности Windows 7»). На рис. 15.2 показано, что произойдет, если применить элемент `DrawingImage` с одним и тем же рисунком `GeometryDrawing` ко всем трем вышеперечисленным свойствам:

```
<Window ...>
<Window.Icon>
  <DrawingImage>
  <DrawingImage.Drawing>
    <GeometryDrawing Brush="Orange">
```

```
      <GeometryDrawing.Pen>
        <Pen Brush="Black" Thickness="10"/>
      </GeometryDrawing.Pen>
      <GeometryDrawing.Geometry>
        <EllipseGeometry RadiusX="100" RadiusY="50"/>
      </GeometryDrawing.Geometry>
      </GeometryDrawing>
      </DrawingImage.Drawing>
    </DrawingImage>
  </Window.Icon>
    ...
  </Window>
```



Рис. 15.2. *Применение одного и того же эллипса EllipseGeometry в качестве значка Window, наложения на панели задач и всех кнопок-манипуляторов*

В предыдущих главах мы столько раз пользовались кистями, что вы, наверное, уже освоились с этим понятием. Однако в классе `Brush` скрыто куда больше возможностей, чем упоминалось выше, но прямого отношения к `Drawing` они не имеют. Поэтому мы рассмотрим их в разделе «Кисти» в конце главы. А пока познакомимся с двумя другими компонентами класса `GeometryDrawing`: `Geometry` и `Pen`.

Класс Geometry

`Geometry` – это простейшее абстрактное представление фигуры и пути. В этом классе имеются методы, позволяющие задавать разнообразные вопросы геометрического характера, например: «Какова твоя площадь?» или «Эта точка находится внутри тебя?». У класса `Geometry` есть ряд подклассов, которые можно отнести к двум категориям – простые и составные геометрические объекты.

Простые геометрические объекты

К простым геометрическим объектам относятся следующие:

- `RectangleGeometry` – имеет свойство `Rect`, описывающее размеры прямоугольника, и свойства `RadiusX` и `RadiusY` – радиусы закругления углов.
- `EllipseGeometry` – имеет свойства `RadiusX` и `RadiusY`, а также свойство `Center`.

КОПНЕМ ГЛУБЖЕ

Применения объектов Geometry

Хотя объекты класса Geometry чаще всего используются внутри рисунков Drawing, они встречаются и в других местах WPF. Например, в классе System.Windows.Ink.Stroke рукописные штрихи представляются в виде объектов Geometry (возвращаемых методом GetGeometry), а в классе DrawingGroup и подклассах Visual имеется свойство Clip, которое позволяет обрезать визуальные элементы в соответствии с произвольным объектом Geometry.

- LineGeometry – имеет свойства StartPoint и EndPoint, описывающие отрезок прямой.
- PathGeometry – хранит коллекцию объектов типа PathFigure в свойстве содержимого Figures; наиболее общий геометрический объект.

Первые три класса в действительности являются частными случаями PathGeometry, предлагаемыми просто для удобства. Любой прямоугольник, эллипс и отрезок прямой можно выразить с помощью PathGeometry. Поэтому поговорим более подробно о компонентах этого мощного класса PathGeometry.

Классы PathFigure и PathSegment. Каждый компонент PathFigure в составе PathGeometry хранит один или несколько соединенных объектов PathSegment в своем свойстве содержимого Segments. Абстрактный класс PathSegment представляет один прямолинейный или криволинейный отрезок и имеет семь конкретных подклассов:

- LineSegment – представляет отрезок прямой.
- PolyLineSegment – представляет последовательность соединенных между собой отрезков LineSegment (ломаная линия).
- ArcSegment – представляет отрезок дуги воображаемого эллипса.
- BezierSegment – представляет кубическую кривую Безье.
- PolyBezierSegment – представляет последовательность соединенных между собой отрезков BezierSegment.
- QuadraticBezierSegment – представляет квадратичную кривую Безье.
- PolyQuadraticBezierSegment – представляет последовательность соединенных между собой отрезков QuadraticBezierSegment.

КОПНЕМ ГЛУБЖЕ

Кривые Безье

Кривые Безье (названные в честь инженера Пьера Безье) повсеместно применяются в компьютерной графике для представления гладких кривых. Они даже используются в шрифтах для математического описания формы глифов.

Основная идея заключается в том, что помимо двух концевых точек у кривой Безье есть одна или несколько *управляющих точек*, которые и придают прямолинейному отрезку форму кривой. Управляющие точки не видны (и могут не лежать на самой кривой), а лишь входят в формулу, описывающую кривую. Интуитивно можно представлять себе, что каждая управляющая точка является центром притяжения, то есть отрезок как бы «притягивается» к этим точкам.

Несмотря на пугающее название, класс `QuadraticBezierSegment` на самом деле проще, чем `BezierSegment`, и дешевле с точки зрения объема вычислений. Но квадратичная кривая Безье может принимать только *U*-образную форму (или быть отрезком прямой), тогда как кубическая – *S*-образную форму.

Ниже показан элемент `GeometryDrawing`, который содержит путь `PathGeometry`, состоящий из двух простых отрезков `LineSegment`, образующих букву *L* (рис. 15.3):

```
<GeometryDrawing>
<GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <PathGeometry>
    <PathFigure>
      <LineSegment Point="0,100"/>
      <LineSegment Point="100,100"/>
    </PathFigure>
  </PathGeometry>
</GeometryDrawing.Geometry>
</GeometryDrawing>
```

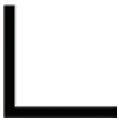


***Рис. 15.3.** Элемент GeometryDrawing, состоящий из двух отрезков LineSegments*

Разумеется, чтобы на экране появилось изображение, показанное на рис. 15.3, элемент `GeometryDrawing` необходимо поместить внутрь чего-то вроде `DrawingImage`, как мы и поступали ранее.

Отметим, что в определении каждого отрезка `LineSegment` присутствует только одна точка `Point`. Объясняется это тем, что упомянутая точка неявно соединена отрезком с предыдущей. Первый отрезок соединяет подразумеваемую по умолчанию точку (0,0) с точкой (0,100), второй – точку (0,100) с точкой (100,100). (Остальные шесть классов `PathSegment` работают точно так же.) Если требуется изменить положение начальной точки, достаточно записать в свойство `StartPoint` объекта `PathFigure` значение типа `Point`, отличное от (0,0).

Возможно, вы думаете, что применять кисть к такому рисунку бессмысленно. Но рис. 15.4 демонстрирует, как кисть фактически закрашивает многоугольник в предположении, что существует еще один отрезок, соединяющий конечную точку с начальной. Рисунок 15.4 был создан путем добавления в показанный выше XAML-код следующей кисти `Brush`:

```
<GeometryDrawing Brush="Orange">
  ...
</GeometryDrawing>
```



Рис. 15.4. *Элемент GeometryDrawing с рис. 15.3, закрашенный оранжевой кистью*



Рис. 15.5. *Элемент GeometryDrawing с рис. 15.4, в котором IsClosed="True"*

Чтобы превратить воображаемый отрезок прямой в реальный, нужно явно добавить в элемент `PathFigure` третий элемент `LineSegment` или же просто присвоить свойству `IsClosed` элемента `PathFigure` значение `true`. Результат показан на рис. 15.5.

Поскольку все отрезки `PathSegment` в пути `PathFigure` должны быть соединены между собой, то для получения несвязных фигур или путей внутри одного объекта `Geometry` следует поместить внутрь `PathGeometry` несколько элементов `PathFigure`. Можно также накладывать один объект `PathFigure` и получать результат, которого было бы сложно добиться с помощью одного `PathFigure`. Например, в следующей XAML-разметке треугольник, показанный на рис. 15.5, наложен на треугольник с другой начальной точкой `StartPoint`, а в остальном идентичный первому:

```
<GeometryDrawing Brush="Orange">
<GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <PathGeometry>
    <!-- Треугольник 1 -->
    <PathFigure IsClosed="True">
      <LineSegment Point="0,100"/>
      <LineSegment Point="100,100"/>
    </PathFigure>
    <!-- Треугольник 2 -->
    <PathFigure StartPoint="70,0" IsClosed="True">
      <LineSegment Point="0,100"/>
      <LineSegment Point="100,100"/>
    </PathFigure>
  </PathGeometry>
</GeometryDrawing.Geometry>
</GeometryDrawing>
```

Этот элемент GeometryDrawing с двумя вложенными PathFigure показан на рис. 15.6. Если вам не нравятся заостренные углы, то можете для каждого отрезка LineSegment установить значение true для свойства IsSmoothJoin (наследуемого всеми подклассами PathSegment). На рис. 15.6 справа показано, что при этом получается.



Соответствует исходной XAML-разметке



После добавления IsSmoothJoin="True" для всех LineSegments

Рис. 15.6. *Перекрывающиеся треугольники, полученные в результате использования двух элементов PathFigures*

Расположение оранжевой заливки, возможно, вызвало у вас удивление. Класс PathGeometry позволяет управлять поведением заливки с помощью свойства FillRule.

Свойство FillRule. Когда граница геометрического объекта имеет точки самопересечения – неважно, образовались они в результате наложения нескольких PathFigure или наличия пересекающихся отрезков PathSegment в одном PathFigure, – возможны различные интерпретации понятий *внутренней области* фигуры (которая и является объектом заливки) и ее *внешней области*.

Свойство FillRule класса PathGeometry (принимающее значения из перечисления FillRule), позволяет выбрать один из двух режимов заливки:

- EvenOdd – заливать область только в том случае, если для перехода из этой области в область, внешнюю по отношению к фигуре, потребуется пересечь нечетное число отрезков. Это режим по умолчанию.
- NonZero – более сложный алгоритм, принимающий во внимание направление отрезков, которые приходится пересекать, чтобы выйти за пределы фигуры. Для многих фигур он приводит к заливке всех находящихся внутри фигуры областей.

Различие между режимами EvenOdd и NonZero показано на рис. 15.7 на примере тех же перекрывающихся треугольников, что и на рис. 15.6.



EvenOdd



NonZero

Рис. 15.7. *Изображение перекрывающихся треугольников при различных значениях PathGeometry.FillRule*

Составные геометрические объекты

Названия двух имеющихся в WPF классов для представления составных геометрических объектов – `GeometryGroup` и `CombinedGeometry` – звучат похоже, но ведут они себя совершенно различно. Правда, по аналогии с отношениями, связывающими классы `TransformGroup` и `Transform`, а также `DrawingGroup` и `Drawing`, оба класса составных геометрических объектов наследуют `Geometry`, поэтому их можно использовать всюду, где допустим класс `Geometry`.

КОПНЕМ ГЛУБЖЕ

StreamGeometry

Для сложных геометрических объектов, которые не нужно будет модифицировать после создания, имеет смысл прибегнуть к классу `StreamGeometry`, а не `PathGeometry` – ради повышения производительности. Класс `StreamGeometry` работает так же, как `PathGeometry`, но заполнить его можно только из процедурного кода. Его странное название связано с деталью реализации: чтобы уменьшить потребление памяти (и ресурсов процессора), вложенные объекты `PathFigure` и `PathSegment` хранятся в виде компактного потока байтов, а не графа объектов .NET.

Приведенный ниже код конструирует объект `StreamGeometry` из перекрывающихся прямоугольников, изображенных на рис. 15.6:

```
StreamGeometry g = new StreamGeometry();
using (StreamGeometryContext context = g.Open())
{
  // Треугольник 1
  context.BeginFigure(new Point(0, 0), true /*isFilled*/, true /*isClosed*/);
  context.LineTo(new Point(0, 100), true /*isStroked*/, true /*isSmoothJoin*/);
  context.LineTo(new Point(100, 100), true /*isStroked*/, true /*isSmoothJoin*/);

  // Треугольник 2
  context.BeginFigure(new Point(70, 0), true /*isFilled*/, true /*isClosed*/);
  context.LineTo(new Point(0, 100), true /*isStroked*/, true /*isSmoothJoin*/);
  context.LineTo(new Point(100, 100), true /*isStroked*/, true /*isSmoothJoin*/);
}
// Сделать этот объект Geometry содержимым существующего
// объекта GeometryDrawing:
geometryDrawing.Geometry = g;
```

Вместо того чтобы создавать объекты `LineSegment`, `ArcSegment`, `BezierSegment` и прочие, можно вызывать такие методы, как `LineTo`, `ArcTo` и `BezierTo`. Для повышения производительности во многих местах внутри WPF используется именно класс `StreamGeometry`.

Класс `GeometryGroup`. Объект класса `GeometryGroup` состоит из одного или нескольких объектов `Geometry`. Например, приведенный выше XAML-код для создания перекрывающихся треугольников на рис. 15.6 можно было бы переписать с использованием двух геометрических объектов (каждый в отдельном элементе `PathFigure`) вместо одного:

```
<GeometryDrawing Brush="Orange">
<GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <GeometryGroup>
    <!-- Треугольник 1 -->
    <PathGeometry>
      <PathFigure IsClosed="True">
        <LineSegment Point="0,100"/>
        <LineSegment Point="100,100"/>
      </PathFigure>
    </PathGeometry>
    <!-- Треугольник 2 -->
    <PathGeometry>
      <PathFigure StartPoint="70,0" IsClosed="True">
        <LineSegment Point="0,100"/>
        <LineSegment Point="100,100"/>
      </PathFigure>
    </PathGeometry>
  </GeometryGroup>
</GeometryDrawing.Geometry>
</GeometryDrawing>
```

В классе `GeometryGroup`, как и в `PathGeometry`, имеется свойство `FillRule`, по умолчанию равное `EvenOdd`. Оно переопределяет задание свойства `FillRule` в дочерних элементах.

Естественно, напрашивается вопрос: «Зачем создавать `GeometryGroup`, когда ничуть не сложнее создать один элемент `PathGeometry`, содержащий несколько `PathFigure`?» Небольшое преимущество состоит в том, что `GeometryGroup` позволяет агрегировать и другие геометрические объекты, например `RectangleGeometry` и `EllipseGeometry`, что немного упрощает применение. Но гораздо важнее тот факт, что `GeometryGroup` дает возможность задавать различные свойства, определенные в классе `Geometry`, независимо для каждого дочернего элемента.

Например, следующий элемент `GeometryGroup` состоит из двух одинаковых треугольников, для одного из которых задано преобразование поворота на 25°:

```
<GeometryDrawing Brush="Orange">
<GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <GeometryGroup>
    <!-- Треугольник 1 -->
    <PathGeometry>
      <PathFigure IsClosed="True">
        <LineSegment Point="0,100" IsSmoothJoin="True"/>
        <LineSegment Point="100,100" IsSmoothJoin="True"/>
      </PathFigure>
    </PathGeometry>
```

```
      <!-- Треугольник 2 -->
      <PathGeometry>
        <PathGeometry.Transform>
          <RotateTransform Angle="25"/>
        </PathGeometry.Transform>
        <PathFigure IsClosed="True">
          <LineSegment Point="0,100" IsSmoothJoin="True"/>
          <LineSegment Point="100,100" IsSmoothJoin="True"/>
        </PathFigure>
      </PathGeometry>
    </GeometryGroup>
  </GeometryDrawing.Geometry>
  </GeometryDrawing>
```

Результат представлен на рис. 15.8. Создать такой рисунок с помощью одного объекта `PathGeometry` с одним `PathFigure` было бы затруднительно. Проще воспользоваться одним `PathGeometry` с двумя `PathFigure`, но и тогда для выполнения поворота пришлось бы проделать математические вычисления вручную. А с применением `GeometryGroup` задача становится тривиальной.



***Рис. 15.8.** Группа GeometryGroup из двух одинаковых треугольников, один из которых повернут*

СОВЕТ

Поскольку кисть `Brush` и перо `Pen` задаются на уровне `Drawing`, а не `Geometry`, группа `GeometryGroup` не позволяет объединять фигуры с разными заливками или контурами. Чтобы добиться этого, нужно воспользоваться классом `DrawingGroup`, который разрешает объединять несколько рисунков (каждый из которых может состоять из одного или нескольких геометрических объектов).

СОВЕТ

В отличие от объектов `UIElement`, которые могут иметь единственного родителя, объекты классов `Geometry`, `PathFigure` и связанных с ними допустимо использовать совместно. Их обобществление может дать заметный выигрыш в производительности, особенно для сложных геометрических объектов. Если не предполагается ничего изменять, то имеет смысл их заморозить и тем самым еще повысить производительность.

Для группы GeometryGroup, использованной при построении рис. 15.8, дублировать одинаковые экземпляры PathFigure вовсе не обязательно. Можно было определить PathFigure как ресурс с ключом figure и переписать GeometryGroup следующим образом:

```
<GeometryGroup>
  <!-- Треугольник 1 -->
  <PathGeometry>
    <StaticResource ResourceKey="figure"/>
  </PathGeometry>
  <!-- Треугольник 2 -->
  <PathGeometry>
  <PathGeometry.Transform>
    <RotateTransform Angle="25"/>
  </PathGeometry.Transform>
  <StaticResource ResourceKey="figure"/>
</PathGeometry>
```

Класс CombinedGeometry. Класс CombinedGeometry, в отличие от GeometryGroup, не является агрегатором общего вида. Он комбинирует два (и только два) геометрических объекта с помощью одной из операций, описываемых перечислением GeometryCombineMode:

- Union – комбинированный объект является объединением обоих составляющих. Это режим по умолчанию.
- Intersect – комбинированный объект является пересечением составляющих.
- Xor – комбинированный объект содержит те точки, которые принадлежат хотя бы одному, но не принадлежат обоим составляющим объектам.
- Exclude – комбинированный объект содержит те точки, которые принадлежат первому, но не принадлежат второму объекту.

В классе CombinedGeometry определены свойства Geometry1 и Geometry2, которые возвращают составляющие объекты, и свойство GeometryCombineMode, возвращающее режим комбинирования. На рис. 15.9 показан результат применения каждой операции к перекрывающимся треугольникам на рис. 15.8.

```
<GeometryDrawing Brush="Orange">
 <GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <CombinedGeometry GeometryCombineMode="XXX">
    <CombinedGeometry.Geometry1>
      <!-- Треугольник 1 -->
      <PathGeometry>
        ...
      </PathGeometry>
    </CombinedGeometry.Geometry1>
```

```
      <CombinedGeometry.Geometry2>
        <!-- Треугольник 2 -->
        <PathGeometry>
          ...
        </PathGeometry>
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </GeometryDrawing.Geometry>
  </GeometryDrawing>
```

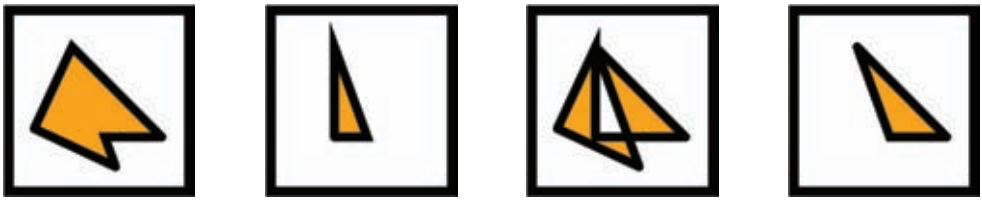


Рис. 15.9. *Результат работы класса CombinedGeometry в каждом из режимов GeometryCombineMode; внешний квадрат служит системой отсчета*

Представление геометрических объектов в виде строк

Можно представлять каждый отрезок внутри `Geometry` отдельным элементом для простых фигур и путей, но если рисунок становится сложнее, то представление получается слишком длинным. Хотя разработчики обычно не пишут XAML-разметку геометрических объектов вручную, а пользуются каким-то инструментом, все равно желательно, чтобы размер получающегося файла был как можно меньше.

Поэтому в WPF имеется конвертер типа `GeometryConverter`, который поддерживает гибкий синтаксис, позволяющий представить практически любой объект `PathGeometry` в виде строки. Для использования из программы в классе `Geometry` даже имеется статический метод `Parse`, который понимает тот же синтаксис и возвращает экземпляр `Geometry`. (Хотя это и деталь реализации, отметим, что объект, возвращаемый конвертером типа и методом `Geometry.Parse`, на самом деле является экземпляром класса `StreamGeometry`, обеспечивающего эффективное хранение.)

Например, объект `PathGeometry`, соответствующий простому треугольнику на рис. 15.6:

```
<GeometryDrawing>
<GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
<GeometryDrawing.Geometry>
  <PathGeometry>
    <PathFigure IsClosed="True">
```

```
        <LineSegment Point="0,100"/>
        <LineSegment Point="100,100"/>
      </PathFigure>
    </PathGeometry>
  </GeometryDrawing.Geometry>
  </GeometryDrawing>
```

можно компактно представить следующим образом:

```
<GeometryDrawing Geometry="M 0,0 L 0,100 L 100,100 Z">
<GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
</GeometryDrawing>
```

Для представления перекрывающихся треугольников на рис. 15.6 нужна чуть более длинная запись:

```
<GeometryDrawing Geometry="M 0,0 L 0,100 L 100,100 Z M 70,0 L 0,100 L 100,100 Z">
<GeometryDrawing.Pen>
  <Pen Brush="Black" Thickness="10"/>
</GeometryDrawing.Pen>
</GeometryDrawing>
```

Эти строки содержат команды, которые управляют свойствами объекта PathGeometry и составляющих его объектов PathFigure, а также команды заливки. Синтаксис простой, но весьма выразительный. В табл. 15.1 описаны все имеющиеся команды.

Таблица 15.1. Команды для представления геометрических объектов в строке

Команда	Назначение
Свойства PathGeometry и PathFigure	
F *n*	Устанавливает FillRule, причем 0 означает EvenOdd, а 1 — NonZero. Если команда встречается, то должна быть первой в строке.
M *x,y*	Начать новый PathFigure и установить для StartPoint значение (x,y). Должна предшествовать всем остальным командам (кроме F). M означает *move* (переместить.)
Z	Закончить PathFigure и установить для IsClosed значение true. За этой командой может следовать команда M, которая начинает следующий PathFigure с другой точки, или какая-нибудь другая команда, начинающая новый PathFigure в текущей точке. Если не требуется замыкать PathFigure, то команду Z можно вообще опустить.
PathSegment	
L *x,y*	Создать отрезок LineSegment, заканчивающийся в точке (*x,y*).
A *rx,ry d f1 f2 x,y*	Создать отрезок ArcSegment, заканчивающийся в точке (*x,y*) и являющийся дугой эллипса с радиусами *rx* и *ry*, повернутого на угол *d* градусов. Флаги *f1* и *f2* могут принимать значения 0 (false) или 1 (true) и соответствуют двум свойствам ArcSegment: IsLargeArc (большая дуга) и Clockwise (по часовой стрелке).

Таблица 15.1 (продолжение)

Команда	Назначение
`C x1,y1 x2,y2 x,y`	Создать отрезок `BezierSegment`, заканчивающийся в точке (*x*,*y*), используя в качестве управляющих точки с координатами (*x1*,*y1*) и (*x2*,*y2*). Буква `C` означает *cubic* (кубическая кривая Безье).
`Q x1,y1 x,y`	Создать отрезок `QuadraticBezierSegment`, который заканчивается в точке (*x*,*y*), используя в качестве управляющей точку с координатами (*x1*,*y1*).
Дополнительные стенографические команды	
`H x`	Создать отрезок `LineSegment`, заканчивающийся в точке (*x*,*y*), где *y* совпадает с ординатой текущей точки. Буква `H` означает *horizontal* (горизонтальный отрезок).
`V y`	Создать отрезок `LineSegment`, заканчивающийся в точке (*x*,*y*), где *x* совпадает с абсциссой текущей точки. Буква `V` означает *vertical* (вертикальный отрезок).
`S x2,y2 x,y`	Создать отрезок `BezierSegment`, который заканчивается в точке (*x*,*y*), используя в качестве управляющих точки с координатами (*x1*,*y1*) и (*x2*,*y2*). *x1* и *y1* автоматически вычисляются так, чтобы обеспечить гладкое сопряжение. (Эта точка является либо второй управляющей точкой предыдущего отрезка, либо текущей точкой, если предыдущий отрезок не является объектом `BezierSegment`.) Буква `S` означает *smooth* (гладкая кубическая кривая Безье).
Команды, записываемые строчными буквами	Если команда записана строчной буквой, то ее параметры интерпретируются как координаты относительно текущей точки, а не абсолютные. Смысл команд `F`, `M` и `Z` при этом не изменяется, тем не менее их тоже можно записывать строчными буквами.

КОПНЕМ ГЛУБЖЕ

Пробелы и запятые в строках описания геометрических объектов

Пробел между командой и первым параметром необязателен, запятые также необязательны, но соседние параметры должны разделяться хотя бы одним пробелом или запятой. Таким образом, строка `M 0,0 L 0,100 L 100,100 Z` эквивалентна строке `M0 0L0 100L100 100Z`, хотя последняя выглядит куда менее понятно.

Класс Pen

Из трех компонентов класса `GeometryDrawing` геометрические объекты и кисти заслуживают отдельного обширного обсуждения, а вот перья (класс `Pen`) сравнительно просты. Перо – это, по сути дела, кисть, для которой задана толщина `Thickness`. Действительно, у элементов `Pen`, встречавшихся в предыдущих примерах, было два свойства: `Brush` (типа `Brush`) и `Thickness` (типа `double`). Но

в классе Pen определены и другие свойства для более точного управления внешним видом:

- StartLineCap и EndLineCap – описывает открытый конец отрезка и может принимать значения, определенные в перечислении PenLineCap: Flat (плоский, по умолчанию), Square, Round и Triangle. Вид линии в точке соединения двух отрезков управляется свойством LineJoin.
- LineJoin – описывает способ соединения отрезков в углах ломаной. Может принимать значения из перечисления PenLineJoin: Miter (по умолчанию), Round и Bevel. Отдельное свойство MiterLimit определяет, насколько выступает уголок соединения типа Miter; если эту длину не ограничить, то для острых углов выступ может оказаться очень большим. По умолчанию принимается значение 10.
- DashStyle – описывает стили линии, рисуемой пером (она необязательно сплошная). Значением этого свойства должен быть объект типа DashStyle. Начертание конечных точек штриха можно регулировать свойством DashCap класса Pen; оно аналогично свойствам StartLineCap и EndLineCap, только по умолчанию равно Square, а не Flat.

FAQ

В чем разница между значениями Flat и Square в перечислении PenLineCap?

В режиме Flat линия заканчивается точно в конечной точке, а в режиме Square – продолжается за конечной точкой. Представьте себе квадрат, длина стороны которого равна толщине пера, с центром в конечной точке. Стало быть, линия простирается за конечную точку на *половину* толщины пера.

На рис. 15.10 показаны все варианты PenLineCap, заданные одинаково в начальной и конечной точках отрезка LineSegment. На рис. 15.11 иллюстрируются различные значения LineJoin в углах треугольника. Значение Round свойства LineJoin дает тот же эффект, что и присвоение true свойству IsSmoothJoin во всех отрезках PathSegment. Но второй способ позволяет настраивать вид каждого уголка по отдельности, тогда как установка свойства LineJoin пера применяется ко всему нарисованному им геометрическому объекту.

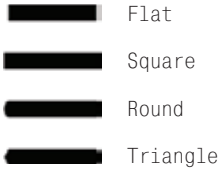


***Рис. 15.10.** Различные значения PenLineCap, заданные одинаково на обоих концах отрезка*

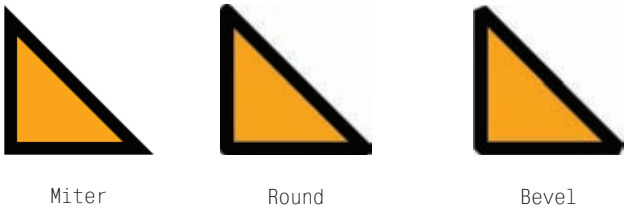


***Рис. 15.11.** Все варианты LineJoin для треугольника*

В классе `DashStyle` имеется свойство `Dashes`. Это простая коллекция `DoubleCollection`, в которой хранится последовательность чисел, представляющих длины штрихов и промежутков между ними. В нечетных элементах находятся длины штрихов (относительно толщины пера), а в четных – длины промежутков. Заданная последовательность повторяется до бесконечности. В классе `DashStyle` имеется также свойство `Offset` типа `double`, определяющее, где начинает рисоваться последовательность.

Странность поведения `DashStyle` заключается в том, что поскольку свойство `DashCap` по умолчанию равно `Square`, то штрих получается длиннее промежутка с точно таким же числовым значением. Более того, для длины штриха вполне можно задать значение `0`, и тогда она окажется равной толщине пера. Имеется еще класс `DashStyles`, и в его статическом свойстве `DashStyle` определены некоторые типичные варианты начертания штрихпунктирных линий. Например, можно выбрать вариант `DashDotDot`:

```
<Pen Brush="Black" Thickness="10" DashStyle="{x:Static DashStyles.DashDotDot}"/>
```

На рис. 15.12 показаны все определенные в классе `DashStyles` варианты и соответствующие им числовые значения в свойстве `Dashes`.

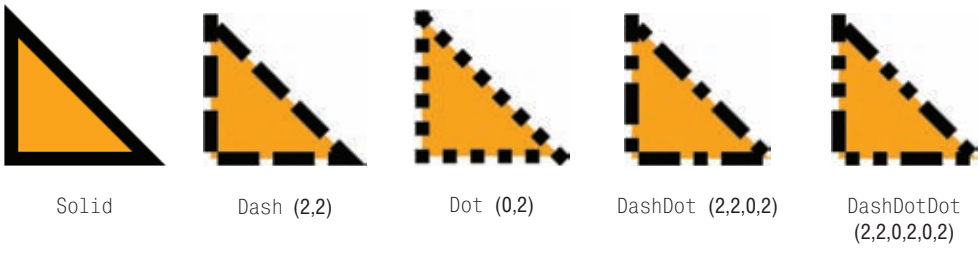


***Рис. 15.12.** Все встроенные варианты начертания штрихпунктирных линий, примененные к перу с подразумеваемыми по умолчанию значениями свойств DashCap и Miter: Square и LineJoin соответственно*

Пример изображения

Теперь, когда мы знаем все о классе `GeometryDrawing`, давайте создадим простенькое изображение. В листинге 15.1 приведен элемент `DrawingGroup`, содер-

жащийся внутри `Image`, в котором есть три элемента `GeometryDrawing`, изображающих привидение (рис. 15.13).

Листинг 15.1. Изображение привидения с помощью элемента Drawing, вложенного в Image

```
<Image>
<Image.Source>
  <DrawingImage>
    <DrawingImage.Drawing>
      <DrawingGroup>

        <!-- Тело -->
        <GeometryDrawing Brush="Blue" Geometry="M 240,250
          C 200,375 200,250 175,200
          C 100,400 100,250 100,200
          C 0,350 0,250 30,130
          C 75,0 100,0 150,0
          C 200,0 250,0 250,150 Z"/>

        <!-- Глаза -->
        <GeometryDrawing Brush="Black">
          <GeometryDrawing.Pen>
            <Pen Brush="White" Thickness="10"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <GeometryGroup>
              <!-- Левый глаз -->
              <EllipseGeometry RadiusX="15" RadiusY="15" Center="95,95"/>
              <!-- Правый глаз -->
              <EllipseGeometry RadiusX="15" RadiusY="15" Center="170,105"/>
            </GeometryGroup>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>

        <!-- Рот -->
        <GeometryDrawing>
          <GeometryDrawing.Pen>
            <Pen Brush="Black" StartLineCap="Round"
              EndLineCap="Round" Thickness="10"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <LineGeometry StartPoint="75,160" EndPoint="175,150"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>

      </DrawingGroup>
    </DrawingImage.Drawing>
  </DrawingImage>
</Image.Source>
</Image>
```



***Рис. 15.13.** Привидение, созданное в виде элемента DrawingGroup, содержащего три элемента GeometryDrawing*

Класс Visual

`Visual`, абстрактный базовый класс `UIElement` (который сам является базовым классом для `FrameworkElement`), содержит низкоуровневые инфраструктурные средства для рисования на экране. В предыдущем разделе для визуализации объектов `Drawing` на экране мы пользовались элементами `Image`. Класс `Image` в конечном итоге наследует `Visual`, но два его промежуточных базовых класса, `FrameworkElement` и `UIElement`, содержат целый ряд средств, которые для рисования обычно не нужны – стили, привязку к данным, ресурсы, участие в компоновке, поддержку ввода с помощью клавиатуры, мыши, стилуса и мультисенсорных устройств, поддержку понятия фокуса, маршрутизируемых событий и т. д.

А теперь представьте приложение или компонент, которому нужно много рисовать, например игру-скроллер в стиле «Супербратья Марио» либо картографическую программу типа Bing Maps. Будучи реализована на базе векторной графики WPF, такая программа могла бы в каждый момент времени хранить сотни или тысячи объектов `Drawing`. Если бы все они были вложены в один объект `Image`, то мы не смогли бы обеспечить интерактивное взаимодействие с каждым отдельным рисунком. Если же помещать каждый рисунок в отдельный объект `Image`, то непроизводительные издержки, связанные с ненужными средствами, возросли бы неимоверно.

К счастью, у класса `Visual` есть другой подкласс, реализующий менее ресурсоемкий механизм визуализации рисунков `Drawing` на экране: `DrawingVisual`. В этом классе есть полезные свойства для управления различными аспектами визуализации, например `Opacity` и `Clip` (они же, кстати, есть и в классе `DrawingGroup`). Но помимо этого реализована еще и минимальная поддержка взаимодействия с устройствами ввода – в виде проверки нахождения точки с заданными координатами в области рисунка (*проверки попадания в `Visual`*).

Поскольку класс `DrawingVisual` функционирует на гораздо более низком уровне, чем типичные средства WPF, порядок работы с ним интуитивно не очевиден. В этом разделе мы объясним, как наполнить `DrawingVisual` содержимым, как визуализировать это содержимое на экране и как выполнить проверку попадания в `Visual`.

Наполнение DrawingVisual содержимым

В классе DrawingVisual нет простого свойства Drawing, в которое можно было бы записать ссылку на объект Drawing (точнее, свойство-то такое *есть*, но оно предназначено только для чтения). Вместо этого необходимо вызвать метод RenderOpen, который возвращает объект класса DrawingContext. На этом объекте можно порисовать, а затем закрыть его методом Close.

Например, приведенный ниже код помещает весь рисунок Drawing из листинга 15.1 внутрь DrawingVisual в предположении, что этот рисунок определен как ресурс с ключом ghostDrawing:

```
DrawingGroup ghostDrawing = FindResource("ghostDrawing") as DrawingGroup;
DrawingVisual ghostVisual = new DrawingVisual();
using (DrawingContext dc = ghostVisual.RenderOpen())
{
  dc.DrawDrawing(ghostDrawing);
}
```

В этом коде учитывается тот факт, что класс DrawingContext реализует интерфейс IDisposable и вызывает метод Close из метода Dispose (а последний неявно вызывается в блоке finally при выходе из области действия using).

В листинге 15.1 все три объекта GeometryDrawing, описывающих привидение, помещены в группу DrawingGroup – просто для того, чтобы ее можно было назначить тем единственным объектом Drawing, который помещается внутрь DrawingImage. Но при использовании DrawingVisual такая группировка GeometryDrawing необязательна. Ниже мы добавляем каждый из трех объектов GeometryDrawing в контекст DrawingContext по отдельности, предполагая, что они определены как ресурсы со своими индивидуальными ключами:

```
GeometryDrawing bodyDrawing = FindResource("bodyDrawing") as GeometryDrawing;
GeometryDrawing eyesDrawing = FindResource("eyesDrawing") as GeometryDrawing;
GeometryDrawing mouthDrawing = FindResource("mouthDrawing") as GeometryDrawing;
DrawingVisual ghostVisual = new DrawingVisual();
using (DrawingContext dc = ghostVisual.RenderOpen())
{
  dc.DrawDrawing(bodyDrawing);
  dc.DrawDrawing(eyesDrawing);
  dc.DrawDrawing(mouthDrawing);
}
```

Каждое следующее изображение рисуется поверх помещенных ранее, так что здесь сохранен правильный *Z*-порядок.

Но можно избавиться не только от лишней группы DrawingGroup, но и от самих объектов Drawing! Ведь объект Drawing – не более чем обертка для команд рисования, которые можно выполнять средствами самого объекта DrawingContext. В классе DrawingContext есть несколько методов для рисования геометрических объектов, изображений и даже видео и текста. (Иначе говоря, эти методы покрывают всю функциональность, предоставляемую различными упоминавшимися выше подклассами Drawing: GeometryDrawing, ImageDrawing, VideoDrawing

и GlyphRunDrawing.) Он также поддерживает вставку и удаление различных эффектов. Все методы класса DrawingContext перечислены в табл. 15.2.

Таблица 15.2. Методы класса DrawingContext

Задача	Методы
Нарисовать простой объект GeometryDrawing без экземпляра Geometry или Drawing	DrawRectangle, DrawRoundedRectangle, DrawEllipse, DrawLine
Нарисовать произвольные объекты Drawing без экземпляра Drawing	DrawGeometry, DrawImage, DrawVideo, DrawGlyphRun, DrawText
Нарисовать произвольные объекты Drawing с экземпляром Drawing	DrawDrawing
Применить эффекты к командам рисования	PushClip, PushEffect, PushGuidelineSet, PushOpacity, PushOpacityMask, PushTransform, Pop
Закончить последовательность команд рисования	Close

Методы Push*XXX* и Pop позволяют не только применять один эффект, скажем полупрозрачность или поворот, к последовательности команд, но и организовать вложенные эффекты. В версии WPF 4 метод PushEffect объявлен устаревшим и ничего не делает, но все остальные работают, как положено.

В листинге 15.2 приведена другая реализация изображения привидения из листинга 15.1 – целиком в процедурном коде. В конструкторе Window создается объект DrawingVisual, который затем заполняется экземплярами Drawing. Отметим, что окно Window, рисуемое этой программой, пока еще абсолютно пустое, потому что мы не предприняли никаких шагов по визуализации DrawingVisual! Эту задачу мы отложим до следующего раздела.

Листинг 15.2. WindowHostingVisual.cs – изображение привидения из листинга 15.1 средствами класса DrawingContext

```
using System;
using System.Windows;
using System.Windows.Media;

public class WindowHostingVisual : Window
{
  public WindowHostingVisual()
  {
    Title = "Hosting DrawingVisuals";
    Width = 300;
    Height = 350;

    DrawingVisual ghostVisual = new DrawingVisual();
    using (DrawingContext dc = ghostVisual.RenderOpen())
    {
```

```
        // Тело
        dc.DrawGeometry(Brushes.Blue, null, Geometry.Parse(
          @"M 240,250
            C 200,375 200,250 175,200
            C 100,400 100,250 100,200
            C 0,350 0,250 30,130
            C 75,0 100,0 150,0
            C 200,0 250,0 250,150 Z"));
        // Левый глаз
        dc.DrawEllipse(Brushes.Black, new Pen(Brushes.White, 10),
          new Point(95, 95), 15, 15);
        // Правый глаз
        dc.DrawEllipse(Brushes.Black, new Pen(Brushes.White, 10),
          new Point(170, 105), 15, 15);
        // Рот
        Pen p = new Pen(Brushes.Black, 10);
        p.StartLineCap = PenLineCap.Round;
        p.EndLineCap = PenLineCap.Round;
        dc.DrawLine(p, new Point(75, 160), new Point(175, 150));
      }
    }
  }
```

В этом коде мы для рисования тела привидения вызываем метод `DrawGeometry` – простейший из всех методов рисования сложных фигур. Обратите внимание на использование метода `Geometry.Parse` – ему можно передать ту же самую строку с описанием пути, что и в листинге 15.1, а не возиться с явным заданием объекта `PathFigure`, содержащего набор отрезков `BezierSegment`. Для рисования глаз и рта нам даже не нужны объекты `Geometry`; достаточно просто вызвать методы `DrawEllipse` и `DrawLine`. Для инициализации пера, которым рисуется рот, необходимо несколько дополнительных строк, потому что конструктор класса `Pen` не позволяет задавать такие характеристики, как форма окончания линий.

В отличие от XAML-разметки листинга 15.1, код в листинге 15.2 – не самый лучший способ создания совместно используемых изображений. Но такая техника очень полезна в приложениях с большим объемом рисования. Возвращаясь к примеру картографической программы, мы могли бы применить метод `DrawGeometry` из класса `DrawingContext` для рисования путей, представляющих дороги, озера, границы административных образований, а метод `DrawText` – для нанесения текстовых надписей. Или, если в качестве карт выступают спутниковые снимки, можно было бы воспользоваться методом `DrawImage` для позиционирования изображений без непроизводительных издержек, связанных с использованием отдельного элемента `Image` для каждого снимка. (Класс `DrawImage` принимает объект `ImageSource`, а не `Image`.)

Таким образом, класс `DrawingContext` в WPF – ближайший аналог контекста устройства в Win32 или объекта `Graphics` в Windows Forms. Заметьте, использование `DrawingContext` не отменяет тот факт, что вы работаете с системой в режиме запоминания. Заданная операция рисования не выполняется немедленно; WPF сохраняет команды до того момента, когда они понадобятся.

СОВЕТ

Использование DrawingContext для рисования оказывается столь нетребовательным к ресурсам, поскольку не выделяет из управляемой кучи память для каждого объекта Drawing, представляющего линии, фигуры и т. д. Так что для визуализации десятков тысяч объектов это самый подходящий способ.

Отображение объекта Visual на экране

Отображение на экране объекта Visual, который одновременно является UIElement, не вызывает сложностей; если добавить его в качестве значения свойства Content в любой однодетный элемент управления, например Window, или в качестве дочернего элемента панели либо одного из объектов, хранящихся в многодетном элементе управления, то он будет визуализироваться в соответствии с тем, как реализован его метод OnRender. Но если имеется объект Visual, не являющийся UIElement, например наш DrawingVisual с изображением привидения, то любое из вышеупомянутых действий приведет лишь к выводу строки, возвращенной методом ToString.

Чтобы такой объект Visual был нарисован правильно, необходимо вручную добавить его в визуальное дерево какого-то элемента UIElement. Тут я слышу удивленные возгласы: «Постойте, постойте! А я-то думал, что весь смысл использования DrawingVisual в том и заключается, чтобы избежать непроизводительных издержек, связанных с UIElement!» Так-то оно так, но хотя бы *один* UIElement все-таки нужен, пусть даже окно верхнего уровня. В картографической программе можно было бы нарисовать тысячи объектов Visual на одном холсте Canvas или в одном окне Window, а не размещать в этом контейнере тысячи объектов UIElement.

Хитрость добавления Visual в элемент состоит в том, что вы должны написать свой подкласс какого-нибудь существующего UIElement и переопределить в нем два защищенных виртуальных члена: VisualChildrenCount и GetVisualChild. В листинге 15.3 это сделано для подкласса Window, определенного в листинге 15.2. Это все, что необходимо для того, чтобы получить изображение объекта DrawingVisual, показанное на рис. 15.14. Отметим, что цвет фона оказался черным – в отличие от случая, когда DrawingImage вкладывался в элемент Image.

Листинг 15.3. WindowHostingVisual.cs – модификация, необходимая для визуализации привидения

```
using System;
using System.Windows;
using System.Windows.Media;

public class WindowHostingVisual : Window
{
  DrawingVisual ghostVisual = null;
```

```
  public WindowHostingVisual()
  {
    Title = "Hosting DrawingVisuals";
    Width = 300;
    Height = 350;

    ghostVisual = new DrawingVisual();
    using (DrawingContext dc = ghostVisual.RenderOpen())
    {
      Те же команды рисования, что и в листинге 15.2...
    }

    // Служебная часть:
    AddVisualChild(ghostVisual);
    AddLogicalChild(ghostVisual);
  }

  // Два обязательных переопределения, реализованных в расчете
  // на отображение одного объекта Visual:
  protected override int VisualChildrenCount
  {
    get { return 1; }
  }

  protected override Visual GetVisualChild(int index)
  {
    if (index != 0)
      throw new ArgumentOutOfRangeException("index");
    return ghostVisual;
  }
}
```



Рис. 15.14. *Объект DrawingVisual, представляющий привидение, появляется в окне после того, как переопределены члены VisualChildrenCount и GetVisualChild*

Свойство VisualChildrenCount должно возвращать количество объектов Visual, находящихся в окне Window. В данном примере есть всего один объект типа DrawingVisual, поэтому мы всегда возвращаем 1. Метод GetVisualChild должен вернуть объект Visual с указанным индексом (нумерация начинается с 0). Поэтому мы возвращаем DrawingVisual, если на вход передан 0, и возбуждаем исключение в противном случае. Если требуется поддержать несколько объектов Visual, то можно поместить их в некую коллекцию и изменить эти два члена так, чтобы они работали с коллекцией. Если нужно взаимодействовать с системой компоновки, то следует переопределить еще два члена – MeasureOverride и ArrangeOverride; они рассматриваются в главе 21 «Компоновка с помощью нестандартных панелей».

Имейте в виду, что та реализация VisualChildrenCount/GetVisualChild, которая приведена в листинге 15.3, не показывает содержимое свойства Content окна Window, даже если оно задано. Если это не годится, то проще всего перенести код для поддержки владения объектом Visual в какой-нибудь другой класс, производный от UIElement, а затем поместить этот элемент в окно, как обычно. Для картографической программы можно было бы, например, сделать владельцем всех объектов Visual класс, производный от Canvas, а потом поместить его в сетку Grid (или другую панель) внутри Window, чтобы можно было расположить еще разные кнопки и другие элементы управления картой.

Помимо переопределения двух членов класса в листинге 15.3 есть еще одна особенность: объект DrawingVisual передается двум защищенным методам, определенным в базовых классах Window: AddVisualChild (определен в классе Visual) и AddLogicalChild (определен в классе FrameworkElement). Вызывать их для визуализации DrawingVisual, строго говоря, необязательно, но это необходимо сделать, чтобы «зарегистрировать» существование визуального объекта в логическом и визуальном деревьях. Тогда такие механизмы, как маршрутизация событий, проверка попадания в область и наследование свойств, будут работать, как ожидается. Если вы поддерживаете в своем классе коллекцию объектов Visual и из этой коллекции иногда удаляются объекты, то необходимо также вызывать методы RemoveVisualChild и RemoveLogicalChild.

КОПНЕМ ГЛУБЖЕ

Другие применения Visual

Объекты Visual, не являющиеся подклассами UIElement, необходимо вкладывать в UIElement, чтобы они появились на экране, это правда. Но с такими облегченными объектами Visual можно делать и другие вещи, не требующие наличия элемента-владельца UIElement. Например, любой объект Visual можно отправить на принтер методом PrintVisual класса PrintDialog или использовать в приложении Win32 (как описано в главе 19 «Интероперабельность с другими технологиями»). Класс DrawingVisual удобен также при работе с классом RenderTargetBitmap, о чем будет рассказано ниже в этой главе.

ПРЕДУПРЕЖДЕНИЕ

Для добавления визуального потомка недостаточно вызвать метод Visual.AddVisualChild!

Имя метода `AddVisualChild` наводит на мысль, будто его вполне достаточно для добавления в элемент дочернего объекта типа `Visual`. Но это не так. Необходимо еще правильно реализовать члены `VisualChildrenCount` и `GetVisualChild`.

КОПНЕМ ГЛУБЖЕ

Еще один способ визуализации

Если в приложении или компоненте используется только один объект `DrawingVisual`, то есть не требуется, чтобы с рисунками можно было интерактивно взаимодействовать независимо от самого приложения, то можно поместить этот объект в элемент `UIElement`. Если имеется всего один экземпляр `UIElement`, то дополнительные издержки малы по сравнению с `DrawingVisual`. А работать с `UIElement` проще, чем с `DrawingVisual`: нужно лишь переопределить метод `OnRender`, произвести рисование в контексте, переданном этому методу в параметре `DrawingContext`, и поместить элемент внутрь какого-нибудь владельца, например записать в свойство `Content` элемента типа `ContentControl`. Все это, конечно, так, но размещение `Drawing` внутри `Image` еще проще, не требует написания процедурного кода и при наличии всего одного объекта непроизводительные издержки по сравнению с `UIElement` невысоки.

Проверка попадания в Visual

Под *проверкой попадания* (*hit testing*) понимают ответ на вопрос, находится ли некая точка (или множество точек) внутри данного объекта. Обычно этот вопрос возникает при обработке событий мыши, стилуса или касания, когда нужно узнать, где находится указатель мыши, кончик стилуса или палец.

В WPF есть два вида проверки попадания: *проверка попадания в `Visual`*, которая поддерживается всеми классами, производными от `Visual`, и *проверка попадания при вводе*, поддерживаемая только подклассами `UIElement`. В этом разделе мы опишем лишь проверку попадания в `Visual`, а проверку попадания при вводе отложим до раздела «Класс Shape».

Без механизма проверки попадания объекты `Visual` не смогли бы реагировать на такие действия пользователя, как щелчок мышью, касание или наведение указателя мыши, потому что в этом классе нет событий ввода, которыми располагает класс `UIElement` (`MouseLeftButtonDown`, `MouseEnter`, `MouseLeave`, `MouseMove` и прочие). Обработав такое событие в элементе-владельце `UIElement` и воспользовавшись затем проверкой попадания в `Visual`, вы сможете узнать, «задеты» ли какие-то интересующие вас дочерние объекты `Visual`, и соответственно отреагировать на событие.

Простая проверка попадания

Проверку попадания в Visual можно произвести с помощью статического метода VisualTreeHelper.HitTest. Простейший вариант этого метода принимает объект Visual, находящийся в корне исследуемого дерева, а также проверяемые координаты (которые должны быть выражены в системе координат переданного корневого объекта). В ответ возвращается объект HitTestResult, содержащий Visual самого верхнего уровня, в который попала точка.

Таким образом, мы можем добавить в листинг 15.3 следующий метод для обработки щелчков мышью по привидению (элементу DrawingVisual), который будет поворачивать рисунок на 1° (просто ради демонстрации):

```
protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
{
  base.OnMouseLeftButtonDown(e);

  // Получить положение указателя мыши относительно окна
  Point location = e.GetPosition(this);

  // Выполнить проверку попадания в окно
  HitTestResult result = VisualTreeHelper.HitTest(this, location);

  // Если мы попали в ghostVisual, повернем его
  if (result.VisualHit == ghostVisual)
  {
    if (ghostVisual.Transform == null)
      ghostVisual.Transform = new RotateTransform();

    (ghostVisual.Transform as RotateTransform).Angle++;
  }
}
```

Поскольку класс Image опосредованно наследует Visual, то можно было бы реализовать ту же схему для случая, когда привидение представлено объектом DrawingImage, а тот помещен в Image (листинг 15.1). (Или можно было бы просто присоединить обработчик события MouseLeftButtonDown к элементу Image.) Однако существует важное различие между проделыванием этого с Image и проверкой попадания в объект DrawingVisual. В показанном выше коде считается, что попадание произошло в DrawingVisual, если указатель мыши действительно оказался где-то в области тела привидения. А в случае Image попадание считается состоявшимся, если указатель мыши находится где-то внутри прямоугольника, ограничивающего изображение.

Проверка попадания при наличии нескольких объектов Visual

Умение проверять попадание точки в непрямоугольную область – это замечательно, но, быть может, вам хочется узнать, в какую конкретно часть привидения попал указатель мыши: на глаза, рот или тело. Для этого придется разбить один объект DrawingVisual на три. В листинге 15.4 мы так и поступили

и поворачиваем на 1° только тот объект, по которому щелкнули. На рис. 15.15 показано, к чему привела возможность манипулировать частями рисунка независимо – привидение стало напоминать картину Пикассо.

Листинг 15.4. WindowHostingVisual.cs – разбиение привидения на три объекта DrawingVisual с целью независимой проверки попадания

```
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Collections.Generic;

public class WindowHostingVisual : Window
{
  List<Visual> visuals = new List<Visual>();

  public WindowHostingVisual()
  {
    Title = "Hosting DrawingVisuals";
    Width = 300;
    Height = 350;

    DrawingVisual bodyVisual = new DrawingVisual();
    DrawingVisual eyesVisual = new DrawingVisual();
    DrawingVisual mouthVisual = new DrawingVisual();

    using (DrawingContext dc = bodyVisual.RenderOpen())
    {
      // Тело
      dc.DrawGeometry(Brushes.Blue, null, Geometry.Parse(
        @"M 240,250
          C 200,375 200,250 175,200
          C 100,400 100,250 100,200
          C 0,350 0,250 30,130
          C 75,0 100,0 150,0
          C 200,0 250,0 250,150 Z"));
    }

    using (DrawingContext dc = eyesVisual.RenderOpen())
    {
      // Левый глаз
      dc.DrawEllipse(Brushes.Black, new Pen(Brushes.White, 10),
        new Point(95, 95), 15, 15);

      // Правый глаз
      dc.DrawEllipse(Brushes.Black, new Pen(Brushes.White, 10),
        new Point(170, 105), 15, 15);
    }
```

```
    using (DrawingContext dc = mouthVisual.RenderOpen())
    {
      // Рот
      Pen p = new Pen(Brushes.Black, 10);
      p.StartLineCap = PenLineCap.Round;
      p.EndLineCap = PenLineCap.Round;
      dc.DrawLine(p, new Point(75, 160), new Point(175, 150));
    }

    visuals.Add(bodyVisual);
    visuals.Add(eyesVisual);
    visuals.Add(mouthVisual);

    // Служебная часть:
    foreach (Visual v in visuals)
    {
      AddVisualChild(v);
      AddLogicalChild(v);
    }
  }

  // Два обязательных переопределения, реализованных в расчете
  // на отображение нескольких объектов Visual:
  protected override int VisualChildrenCount
  {
    get { return visuals.Count; }
  }

  protected override Visual GetVisualChild(int index)
  {
    if (index < 0 || index >= visuals.Count)
      throw new ArgumentOutOfRangeException("index");
    return visuals[index];
  }

  protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
  {
    base.OnMouseLeftButtonDown(e);

    // Получить положение указателя мыши относительно окна
    Point location = e.GetPosition(this);

    // Выполнить проверку попадания в Visual
    HitTestResult result = VisualTreeHelper.HitTest(this, location);

    // Если попали в какой-то объект DrawingVisual, поворачиваем его
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
      DrawingVisual dv = result.VisualHit as DrawingVisual;
      if (dv.Transform == null)
        dv.Transform = new RotateTransform();
```

```
            (dv.Transform as RotateTransform).Angle++;
        }
    }
}
```



Рис. 15.15. *Привидение, представленное тремя независимыми объектами DrawingVisual, после нескольких щелчков по телу и глазам*

Поскольку в этом окне теперь три дочерних элемента `Visual`, а не один, то мы храним их в коллекции `List<Visual>` – это необходимо для реализации `VisualChildrenCount` и `GetVisualChild`. Чтобы рисовать в трех объектах `DrawingVisual` вместо одного, понадобилось внести простое изменение: команды работы с `DrawingContext` разбиты на три блока `using`, по одному на каждый `DrawingVisual`. При обработке объекта `HitTestResult` программа применяет преобразование поворота к тому объекту `DrawingVisual`, в который попал указатель мыши.

КОПНЕМ ГЛУБЖЕ

Объекты DrawingVisual как дочерние элементы DrawingVisual

Класс `DrawingVisual` является производным от класса `ContainerVisual`, в котором есть свойство-коллекция `Children` для хранения произвольного числа объектов `Visual`. (Класс `ContainerVisual` правильнее было бы назвать `VisualGroup` по аналогии с такими классами WPF, как `TransformGroup`, `DrawingGroup` и `GeometryGroup`.) Поэтому еще один способ реализовать программу в листинге 15.4 – добавить объекты `eyesVisual` и `mouthVisual` в качестве потомков `bodyVisual`, а *не* помещать их в коллекцию `visuals`. Это к тому же означает, что мы могли бы вернуться к варианту, когда есть единственный объект-потомок `Visual`, а не целая коллекция! Визуализация и проверка попадания автоматически работают для всех потомков `DrawingVisual`, потому что в классе `ContainerVisual` члены `VisualChildrenCount` и `GetVisualChild` уже переопределены, как в листинге 15.4. Нужно лишь присоединить `Window` к корневому `DrawingVisual` и поручить `DrawingVisual` сделать все остальное!

Проверка попадания в случае перекрывающихся объектов Visual

Механизм проверки попадания в Visual может сообщить обо *всех* объектах Visual, внутри которых оказалась точка, а не только об объекте самого верхнего уровня. В примере привидения, состоящего из трех Visual, можно настроить проверку попадания так, чтобы в случае щелчка по глазу сообщалось о том, что указатель мыши находится *одновременно* в областях глаза и тела. Даже если объект Visual полностью закрыт другим объектом, в него все равно можно попасть.

Чтобы воспользоваться этой возможностью, необходимо вызвать другой вариант метода HitTest, который принимает делегат типа HitTestResultCallback. Перед тем как HitTest вернет управление, он вызовет делегат по одному разу для каждого обнаруженного объекта Visual, начиная с верхнего и кончая нижним.

Ниже приведен модифицированный код метода OnMouseLeftButtonDown из листинга 15.4 с поддержкой проверки попадания в перекрывающиеся объекты Visual:

```
protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
{
  base.OnMouseLeftButtonDown(e);

  // Получить положение указателя мыши относительно окна
  Point location = e.GetPosition(this);

  // Выполнить проверку попадания в  Visual
  VisualTreeHelper.HitTest(this, null,
    new HitTestResultCallback(HitTestCallback),
    new PointHitTestParameters(location));
}

public HitTestResultBehavior HitTestCallback(HitTestResult result)
{
  // Если попали в какой-то объект DrawingVisual, поворачиваем его
  if (result.VisualHit.GetType() == typeof(DrawingVisual))
  {
    DrawingVisual dv = result.VisualHit as DrawingVisual;
    if (dv.Transform == null)
      dv.Transform = new RotateTransform();

    (dv.Transform as RotateTransform).Angle++;
  }
  // Продолжаем анализировать попадания
  return HitTestResultBehavior.Continue;
}
```

Число отличий от предыдущего варианта невелико. Самое заметное заключается в том, что логика обработки HitTestResult перенесена в метод обратного вызова, поскольку данный перегруженный вариант метода HitTest ничего не

возвращает. Метод обратного вызова должен вернуть одно из двух значений типа HitTestResultBehavior: Continue или Stop. Таким образом, в любой момент можно прекратить анализ попадания в последующие объекты Visual. Если метод обратного вызова *всегда* возвращает Stop, то обрабатывается только самый верхний объект Visual, то есть делается точно то же самое, что при более простом подходе, описанном выше. Второй параметр этого перегруженного варианта метода HitTest, в котором мы передали null, может содержать делегат HitTestFilterCallback, позволяющий пропустить обработку некоторых частей визуального дерева, не прекращая ее вовсе. С его помощью можно реализовать весьма изощренные схемы проверки попадания.

Отметим, что этому варианту HitTest проверяемая точка Point передается не напрямую, а в объекте PointHitTestParameters. Объясняется это тем, что метод принимает экземпляр абстрактного класса HitTestParameters, для которого в WPF определены два подкласса: PointHitTestParameters и GeometryHitTestParameters. Второй из них можно использовать для проверки пересечения с произвольной областью. Это бывает полезно для поддержки более сложных действий ввода, например буксировки выделяющего прямоугольника или набрасывания «лассо» для выделения нескольких объектов.

FAQ

Почему в более сложной форме проверки попадания применяется неудобный механизм обратного вызова вместо того, чтобы просто вернуть массив объектов HitTestResult?

Схема с обратным вызовом была выбрана из соображений производительности. В этом случае WPF не нужно выделять дополнительную память, что существенно, когда количество объектов Visual велико или проверка попадания производится часто. Кроме того, такая схема допускает оптимизацию под конкретный сценарий, предоставляя методу обратного вызова возможность остановить обработку, вернув HitTestResultBehavior.Stop.

СОВЕТ

Если требуется, чтобы механизм проверки сообщал о попадании в Visual, когда точка находится в пределах ограничивающего прямоугольника, а не точно в пределах области, занимаемой геометрическим объектом, то можно переопределить метод HitTestCore класса Visual, который вызывается при попадании в ограничивающий прямоугольник. (Этот метод позволяет настраивать алгоритм проверки попадания и другими способами.)

Можно добиться того же результата и проще – нарисовать внутри Visual прозрачный прямоугольник того же размера, что и ограничивающий прямоугольник. При проверке попадания в Visual прозрачность объектов не принимается во внимание: все они трактуются одинаково, как если бы были оконными стеклами.

ПРЕДУПРЕЖДЕНИЕ

Не модифицируйте визуальное дерево в методах обратного вызова!

Методы обратного вызова используются механизмом проверки попадания в процессе обхода визуального дерева, поэтому любое изменение его структуры может привести к ошибкам. Если действительно необходимо модифицировать дерево в зависимости от результатов проверки попадания, то следует сохранить необходимую для этого информацию в методе обратного вызова, а саму модификацию произвести по завершении `HitTest`. Сделать это довольно просто, потому что `HitTest` возвращает управление только после того, как все методы обратного вызова получили возможность отработать.

Класс Shape

Класс `Shape`, как и `GeometryDrawing`, – это двумерный рисунок, который сочетает в себе объект `Geometry` с пером `Pen` и кистью `Brush`. Но, в отличие от `GeometryDrawing`, класс `Shape` наследует классу `FrameworkElement` и потому его можно помещать в пользовательский интерфейс непосредственно – без написания специального кода или создания сложной иерархии объектов. Например, в главе 2 «Все тайны XAML» было показано, как легко поместить внутрь кнопки `Button` квадратик, представленный объектом класса `Rectangle` (производного от `Shape`):

```
<Button MinWidth="75">
  <Rectangle Height="20" Width="20" Fill="Black"/>
</Button>
```

В состав WPF входит шесть классов, производных от абстрактного класса `System.Windows.Shapes.Shape`:

- `Rectangle`
- `Ellipse`
- `Line`
- `Polyline`
- `Polygon`
- `Path`

Большая часть из них вам уже знакома, потому что они повторяют рассмотренные выше подклассы `Geometry`. В последующих разделах мы рассмотрим каждый подкласс отдельно, потому что они работают не совсем так, как их аналоги из иерархии `Geometry`. (А кроме того, `Polyline` и `Polygon` – это специфические для `Shape` абстракции, не имеющие прямых аналогов в `PathGeometry`.) В самом классе `Shape` определено много свойств для управления внешним видом его конкретных подклассов. Наиболее важными являются свойства `Fill` и `Stroke`, оба типа `Brush`.

FAQ

Почему Shape.Stroke имеет тип Brush, а не Pen?

Свойства `Fill` и `Stroke` класса `Shape` играют ту же роль, что и свойства `Brush` и `Pen` класса `GeometryDrawing`: `Fill` используется для закрашивания внутренней области, а `Stroke` — для рисования контура. На уровне реализации для создания контура фигуры `Shape` действительно используется перо `Pen`. Но вместо того чтобы раскрывать это перо напрямую, класс `Shape` определяет свойство `Stroke` как имеющее тип `Brush` и предоставляет восемь дополнительных свойств для настройки внутреннего пера, стоящего за свойством `Stroke`: `StrokeStartLineCap`, `StrokeEndLineCap`, `StrokeThickness` и т. д.

Причиной этой кажущейся непоследовательности является тот факт, что задавать относящиеся к перу свойства прямо для объекта `Shape` проще, чем для отдельного объекта `Pen`, особенно в типичном случае, когда требуется только установить кисть `Brush` и толщину пера `Thickness`.

ПРЕДУПРЕЖДЕНИЕ

Чрезмерное увлечение объектами Shape может привести к снижению производительности!

Очень заманчиво использовать объекты `Shape` как кирпичики, из которых складываются все прочие двумерные рисунки. С ними гораздо проще работать, чем с объектами `Drawing`, и они прекрасно укладываются в модель содержимого, которую разработчики на WPF считают предпочтительной. Кроме того, инструменты конструирования и программы экспорта в формате XAML по умолчанию часто используют `Shape` для представления иллюстраций, так что подобные объекты могут проникнуть в ваше приложение без вашего ведома. Например, если выбрать в меню программы Microsoft Expression Design команду `XAML Export`, то в результирующем XAML-файле будут присутствовать элементы `Shape` на панели `Canvas`, если только для переключателя `Document Format` не было явно задано значение `Resource Dictionary`. В последнем случае можно выбирать между `DrawingImage` и `DrawingBrush`. Обычно `DrawingImage` — более разумный выбор, потому что в этом режиме, как правило, удается избежать рисования на промежуточной поверхности перед рисованием в буфере.

Когда изображение состоит из объектов `Shape`, *каждый отдельный* объект поддерживает стили, привязку к данным, ресурсы, компоновку, ввод и передачу фокуса, маршрутизируемые события и т. д. Замечательно, конечно, что все это вы получаете без малейших усилий со своей стороны, но, как отмечалось в разделе «Класс Visual» выше, обычно эти дополнительные возможности не востребованы. Не забывайте об этом, когда подумываете об использовании в программе значительного количества объектов `Shape`.

Класс Rectangle

В рассмотренном выше классе RectangleGeometry имеется свойство Rect, определяющее размеры занимаемой прямоугольной области. Класс Rectangle делегирует управление своим размером и положением системе компоновки WPF. Для этого могут использоваться его свойства Width и Height, унаследованные (в числе прочих) от класса FrameworkElement, или, например, присоединенные свойства Canvas.Left и Canvas.Top.

Однако, как и в случае RectangleGeometry, в классе Rectangle определены собственные свойства RadiusX и RadiusY типа double, что позволяет задавать скругленные углы. На рис. 15.16 показано несколько прямоугольников на панели StackPanel с разными значениями RadiusX и RadiusY:

```
<StackPanel>
  <Rectangle Width="200" Height="100"
    Fill="Orange" Stroke="Black" StrokeThickness="10" Margin="4"/>
  <Rectangle Width="200" Height="100" RadiusX="10" RadiusY="30"
    Fill="Orange" Stroke="Black" StrokeThickness="10" Margin="4"/>
  <Rectangle Width="200" Height="100" RadiusX="30" RadiusY="10"
    Fill="Orange" Stroke="Black" StrokeThickness="10" Margin="4"/>
  <Rectangle Width="200" Height="100" RadiusX="100" RadiusY="50"
    Fill="Orange" Stroke="Black" StrokeThickness="10" Margin="4"/>
</StackPanel>
```

RadiusX не должен превышать половины ширины Width прямоугольника, а RadiusY – половины высоты Height. Задавать большее значение бессмысленно – эффекта не будет.



***Рис. 15.16.** Четыре прямоугольника с разными значениями RadiusX и RadiusY*

ПРЕДУПРЕЖДЕНИЕ

Чтобы объект Shape был виден, необходимо явно задать свойство Stroke или Fill!

Для тех, кто привык работать с объектами GeometryDrawing, это замечание может показаться очевидным, но разработчики, воспринимающие фигуры Shape так же, как кнопки Button и списки ListBox, часто попадают в эту ловушку. Хотя внутри каждого объекта Shape и хранится соответствующий объект Geometry, его свойства Stroke и Fill по умолчанию равны null.

Класс Ellipse

Узнав о гибкости класса Rectangle и обнаружив, что его можно превратить в эллипс (или круг), вы, наверное, придете к выводу, что отдельный класс Ellipse — излишество. И будете правы! Единственное назначение класса Ellipse — упростить получение фигуры эллиптической формы. В нем нет ни одного допускающего установку свойства помимо тех, что уже предоставлены классом Shape и его базовыми классами. В отличие от класса EllipseGeometry, который имеет также свойства RadiusX, RadiusY и Center, класс Ellipse просто вписывает в свою прямоугольную область эллипс максимально возможного размера.

Следующий элемент Ellipse можно было бы подставить вместо последнего элемента Rectangle в предыдущем фрагменте XAML, и результат на рис. 15.16 ничуть не изменился бы:

```
<Ellipse Width="200" Height="100"
  Fill="Orange" Stroke="Black" StrokeThickness="10" Margin="4"/>
```

Мы лишь поменяли имя элемента и убрали свойства RadiusX и RadiusY.

КОПНЕМ ГЛУБЖЕ

Как работает класс Shape

Класс Shape переопределяет метод OnRender класса UIElement и для рисования требуемого геометрического объекта пользуется методами класса DrawingContext. Например, в классе Ellipse метод OnRender может быть реализован следующим образом:

```
protected override void OnRender(DrawingContext drawingContext)
{
  Pen pen = ...;    // сконструировать перо на основе свойств StrokeXXX
  Rect rect = ...; // размер прямоугольника установит механизм компоновки
  drawingContext.DrawGeometry(this.Fill, pen, new EllipseGeometry(rect));
}
```

А в классе Rectangle реализация метода OnRender устроена примерно так:

```
protected override void OnRender(DrawingContext drawingContext)
{
  Pen pen = ...;   // сконструировать перо на основе свойств StrokeXXX
  Rect rect = ...; // размер прямоугольника установит механизм компоновки
  drawingContext.DrawRoundedRectangle(this.Fill, pen, rect,
    this.RadiusX, this.RadiusY);
}
```

Большая часть класса `Shape` – это служебный код, необходимый для взаимодействия с системой компоновки. Мы рассмотрим эти вопросы в главе 21.

Класс Line

В классе `Line` определены четыре свойства типа `double`, которые представляют отрезок, соединяющий точки (*x1,y1*) и (*x2,y2*). Эти свойства называются `X1`, `Y1`, `X2`, `Y2`. Координаты определены как четыре разных свойства, а не два свойства типа `Point` (как в классе `LineGeometry`), потому что так удобнее для привязки к данным.

Координаты, заданные в свойствах объекта `Line`, не абсолютные, а вычислены относительно системы координат, назначенной элементу `Line` системой компоновки. Например, следующая панель `StackPanel` содержит три отрезка, показанных на рис. 15.17:

```
<StackPanel>
  <Line X1="0" Y1="0" X2="100" Y2="100" Stroke="Black"
    StrokeThickness="10" Margin="4"/>
  <Line X1="0" Y1="0" X2="100" Y2="0" Stroke="Black"
    StrokeThickness="10" Margin="4"/>
  <Line X1="0" Y1="100" X2="100" Y2="0" Stroke="Black"
    StrokeThickness="10" Margin="4"/>
</StackPanel>
```

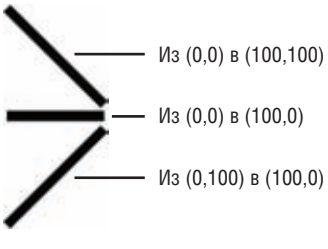


***Рис. 15.17.** На примере трех отрезков на панели StackPanel видно, что координаты относительные*

Отметим, что каждому отрезку выделяется место под его ограничивающий прямоугольник, поэтому горизонтальный отрезок получает только 10 единиц (исходя из толщины `Stroke`) плюс заданное поле `Margin`. Класс `Line` насле-

дует от Shape свойство Fill, но в данном случае оно бессмысленно, так как нечего заливать.

Класс Polyline

Класс Polyline (ломаная) представляет последовательность отрезков, определенных его свойством Points (коллекция объектов Point). На рис. 15.18 показаны четыре элемента Polyline:

```
<StackPanel>
  <Polyline Points="0,0 100,100" Stroke="Black" StrokeThickness="10" Margin="4"/>
  <Polyline Points="0,0 100,100 200,0" Stroke="Black"
    StrokeThickness="10" Margin="4"/>
  <Polyline Points="0,0 100,100 200,0 300,100"
    Stroke="Black" StrokeThickness="10" Margin="4"/>
  <Polyline Points="0,0 100,100 200,0 300,100 100,100"
     Stroke="Black" StrokeThickness="10" Margin="4"/>
</StackPanel>
```

Конвертер типа позволяет задавать точки Point в виде простого списка чередующихся значений *x* и *y*. Запятые необязательны и служат только для удобства. Их можно помещать между всеми соседними значениями или опустить вовсе.



***Рис. 15.18.** Четыре ломаных с числом вершин от 2 до 5*



***Рис. 15.19.** Те же ломаные, что на рис. 15.18, но с явно заданным свойством Fill*

На рис. 15.19 показано, что установка свойства Fill элемента Polyline приводит к такой же заливке, как для незамкнутого пути PathGeometry, – в предположении, что существует дополнительный отрезок, соединяющий первую точку с последней. Так происходит потому, что на уровне реализации Polyline *действительно* использует PathGeometry! Рисунок 15.19 был получен добавлением свойства Fill="Orange" в элементы Polyline на рис. 15.18.

Класс Polygon

Если класс Rectangle делает излишним класс Ellipse, то Polyline устраняет необходимость в классе Polygon. Единственное различие между Polyline и Polygon заключается в том, что Polygon автоматически добавляет отрезок, соединяющий первую точку с последней. (Иными словами, устанавливает значение true для свойства IsClosed объекта PathFigure, являющегося частью внутреннего объекта PathGeometry.)

Если взять все элементы Polyline, изображенные на рис. 15.19, и просто поменять их имена на Polygon, то получится результат, показанный на рис. 15.20. Обратите внимание, что начальный отрезок в первом и последнем многоугольниках получился заметно длиннее, чем на рис. 15.19. А все из-за типа сочленения Miter в точках, где начальный отрезок соединяется с конечным (в данном случае они совпадают). Поскольку угол между отрезками равен 0°, то сочленение оказалось бы бесконечно длинным, если бы свойство StrokeMiterLimit по умолчанию не было равно 10 единицам.

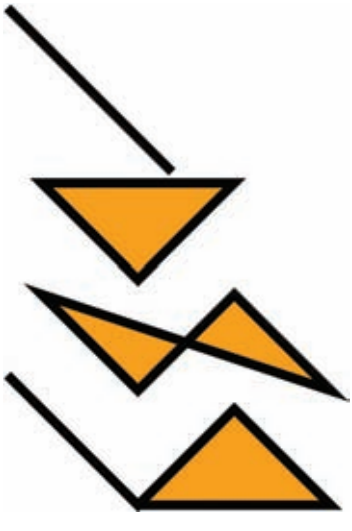


Рис. 15.20. *Многоугольники Polygon очень похожи на ломаные Polyline, только всегда образуют замкнутую фигуру*

Оба класса, Polygon и Polyline, раскрывают свойство FillRule внутреннего объекта PathGeometry в виде своего собственного свойства FillRule.

Класс Path

Как вы, наверное, догадались, любую фигуру Shape можно представить в терминах общего класса Path – точно так же, как все основные геометрические объекты являются частными случаями PathGeometry. Класс Path добавляет к классу Shape только свойство Data, значением которого может быть любой

геометрический объект. Следовательно, Path оказывается простейшим (и вместе с тем самым функционально насыщенным) способом включить произвольный геометрический объект непосредственно в пользовательский интерфейс. Отпадает необходимость в явном объекте Drawing или в низкоуровневой технике с применением DrawingContext; достаточно просто установить свойства Data, Fill и все, относящиеся к Stroke.

Следующий элемент Path дает точно те же перекрывающиеся треугольники, что изображены на рис. 15.6:

```
<Path Fill="Orange" Stroke="Black" StrokeThickness="10">
<Path.Data>
  <PathGeometry>
    <!-- Треугольник 1 -->
    <PathFigure IsClosed="True">
      <LineSegment Point="0,100"/>
      <LineSegment Point="100,100"/>
    </PathFigure>
    <!-- Треугольник 2 -->
    <PathFigure StartPoint="70,0" IsClosed="True">
      <LineSegment Point="0,100"/>
      <LineSegment Point="100,100"/>
    </PathFigure>
  </PathGeometry>
</Path.Data>
</Path>
```

Или, воспользовавшись конвертером типа Geometry, можно записать эту разметку короче:

```
<Path Fill="Orange" Stroke="Black" StrokeThickness="10"
  Data="M 0,0 L 0,100 L 100,100 Z M 70,0 L 0,100 L 100,100 Z"/>
```

Изображение, составленное из объектов Shape

Вернемся к изображению привидения, которое в листинге 15.1 было представлено в виде объекта DrawingImage, а в листингах с 15.2 по 15.4 – в виде последовательности команд DrawingContext. В листинге 15.5 все фрагменты привидения, которые теперь являются независимыми фигурами Shape, помещены на панель Canvas. Результат внешне ничем не отличается от визуализации объекта DrawingImage, вложенного в Image, как показано на рис. 15.13.

Листинг 15.5. Привидение, составленное из четырех независимых фигур

```
<Canvas xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Path Fill="Blue" Data="M 240,250
    C 200,375 200,250 175,200
    C 100,400 100,250 100,200
    C 0,350 0,250 30,130
    C 75,0 100,0 150,0
    C 200,0 250,0 250,150 Z"/>
  <Ellipse Fill="Black" Stroke="White" StrokeThickness="10"
```

```
      Width="40" Height="40" Canvas.Left="75" Canvas.Top="75"/>
    <Ellipse Fill="Black" Stroke="White" StrokeThickness="10"
      Width="40" Height="40" Canvas.Left="150" Canvas.Top="85"/>
    <Line X1="75" Y1="160" X2="175" Y2="150" StrokeStartLineCap="Round"
      StrokeEndLineCap="Round" Stroke="Black" StrokeThickness="10"/>
  </Canvas>
```

Числовые данные в элементах `Path` (тело) и `Line` (рот) точно такие же, как в исходном объекте `DrawingImage`. Но вот параметры обоих эллипсов при переходе от объектов `EllipseGeometry` к `Ellipse` пришлось немного изменить. Первоначально глаза представлялись кругами радиусом `15` с толщиной пера `10`. Поскольку середина контура совпадает с границей геометрического объекта, то полный радиус составлял только `20` единиц. Именно поэтому в листинге 15.5 для обоих элементов `Ellipse` заданы `Height` и `Width`, равные `40` (радиус, умноженный на 2). В данном случае вся фигура `Shape` вместе с контуром помещается в ограничивающую область. Что же касается значений, выбранных для `Canvas.Left` и `Canvas.Top`, то это оригинальные значения `EllipseGeometry Center` за вычетом полного радиуса 20.

В отличие от предыдущих реализаций привидения, эта поддерживает *проверку попадания при вводе* независимо для всех четырех частей (даже каждый глаз анализируется по отдельности!), поскольку все они являются производными от `UIElement`. Проверка попадания при вводе отличается от проверки попадания в `Visual`: первая в большей степени ориентирована на то, во что пользователь может физически попасть указателем мыши, пальцем или стилусом. Поддерживается только попадание в элемент, находящийся в самом верхнем слое, причем попадание регистрируется лишь в случае, когда свойства `IsEnabled` и `IsVisible` (определенные в классе `UIElement`) равны `true`. (Кроме того, поддерживается лишь проверка попадания одной точки, а не пересечения с произвольным геометрическим объектом, но это просто искусственное ограничение, а не значимое различие.)

КОПНЕМ ГЛУБЖЕ

Маленький секрет проверки попадания при вводе

Вообще-то, проверка попадания при вводе – всего лишь частный случай проверки попадания в `Visual`. На самом деле метод `InputHitTest` просто вызывает `VisualTreeHelper.HitTest`, передавая свои собственные делегаты для фильтрации и обработки результатов! Делегат фильтрации отсекает все ветви визуального дерева, исходящие из неактивных и невидимых элементов `UIElement`, а делегат обработки останавливает обход дерева, как только обнаружит первый подходящий элемент.

Чтобы проверить попадание при вводе, следует вызвать метод `InputHitTest` того объекта `UIElement`, чье визуальное дерево мы проверяем. Этому методу передается объект `Point`, а последний возвращает объект типа `IInputElement` (интерфейс, реализованный классами `UIElement` и `ContentElement`). Но проверка

попадания при вводе редко производится напрямую, потому что во всех подклассах UIElement уже определен целый ряд событий, генерируемых при взаимодействиях с ними (нажатии, щелчке мышью и прочих): GotKeyboardFocus, KeyDown, KeyUp, GotMouseCapture, MouseEnter, MouseLeave, MouseMove, MouseWheel, GotStylusCapture, StylusEnter, StylusLeave, StylusInAirMove и т. д. А если политика проверки попадания при вводе слишком ограничительна для ваших целей, то для любого объекта Shape можно выполнить и проверку попадания в Visual.

Кисти

Когда программируешь WPF на уровне XAML, это не так очевидно, но на самом деле элементы WPF почти никогда не взаимодействуют с цветами напрямую. Как правило, работа с цветами инкапсулирована в объекты, называемыми кистями (Brush). Это решение обладает необычайной выразительной мощью, потому что WPF содержит семь видов кистей, с помощью которых можно сделать почти все что угодно. Есть три *цветных кисти*, три *мозаичных* и одна специальная, рассматриваемая в конце главы (BitmapCacheBrush). Хотя в этом разделе использование кистей демонстрируется главным образом в применении к объектам Drawing или Window, имейте в виду, что кисти могут применяться для рисования фона, текста или контура практически любого объекта, который можно поместить на экран.

Цветные кисти

Три имеющиеся в WPF цветные кисти называются SolidColorBrush, LinearGradientBrush и RadialGradientBrush. Возможно, вы думаете, что, ознакомившись с несколькими примерами этих кистей в предыдущих главах, вы уже знаете о них все. Однако они обладают куда более гибкими возможностями, чем многие себе представляют.

Класс SolidColorBrush

Класс SolidColorBrush, который до сих пор неявно использовался в примерах книги, закрашивает область одним цветом. У него есть свойство Color типа System.Windows.Media.Color. Благодаря конвертеру типа, который преобразует строки вида «Blue» или «#FFFFFF» в объекты SolidColorBrush, такие кисти в XAML-разметке неотличимы от самих цветов Color.

Структура Color обладает большей функциональностью, чем может показаться на первый взгляд. Она поддерживает два цветовых пространства:

- **sRGB** – стандартное цветовое пространство RGB, рассчитанное на ЭЛТ-мониторы и знакомое большинству программистов и веб-дизайнеров. Красная, зеленая и синяя компоненты кодируются одним байтом, что дает возможность представить 256 цветов.
- **scRGB** – расширенное цветовое пространство RGB, в котором красная, зеленая и синяя компоненты кодируются числами с плавающей точкой. Это позволяет точно представить гораздо более широкую цветовую гамму. Если

все три компоненты равны 0.0, то получается черный цвет; если все равны 1.0, то белый. Однако scRGB допускает также значения вне этого диапазона, поэтому информация не потеряется, если в процессе применяемых к цветам преобразований какое-то промежуточное значение выйдет за пределы естественного диапазона. Схема scRGB обеспечивает повышенную точность, так как является линейным пространством цветов.

В классе `Color` имеются наборы свойств (по одному на канал) для каждого из двух цветовых пространств: `A`, `R`, `G`, `B` типа `Byte` – для хорошо знакомого пространства sRGB, а `ScA`, `ScR`, `ScG`, `ScB` типа `Single` – для более гибкого пространства scRGB. (`A` и `ScA` представляют альфа-канал для описания переменной прозрачности.) При установке любого из этих свойств согласованно обновляются оба внутренних представления в классе `Color`. Таким образом, для одного и того же экземпляра `Color` можно манипулировать свойствами из обоих наборов и синхронизация будет поддерживаться автоматически. Кстати, этой особенностью можно воспользоваться для преобразования цветов из sRGB в scRGB и обратно.

СОВЕТ

Обычно эффективнее использовать цвета, в которых степень прозрачности задана с помощью альфа-канала, чем применять свойство `Opacity`, чтобы сделать полупрозрачным сплошной цвет.

В классе `Color` определены также операторы, позволяющие складывать, вычитать и перемножать объекты и сравнивать их на равенство. Однако, поскольку в представлении scRGB используются числа с плавающей точкой (которые проверять на строгое равенство никогда не следует), то определен также статический метод `AreClose`, который принимает два цвета и возвращает `true`, если все их каналы отличаются на очень небольшую величину.

Конвертер типа `Color` поддерживает несколько строковых представлений цвета:

- По названию, например `Red`, `Khaki` или `DodgerBlue` (все распознаваемые названия перечислены в статических свойствах класса `Colors`).
- По sRGB-представлению в формате `#argb`, где `a`, `r`, `g` и `b` – шестнадцатеричные значения свойства `A`, `R`, `G`, `B`. Например, непрозрачному красному цвету соответствует строка `#FFFF0000` или просто `#FF0000` (поскольку по умолчанию предполагается, что свойство `A` равно максимальному значению `255`).
- По scRGB-представлению в формате `sc#a`, где `r g b`, `a`, `r`, `g` и `b` – десятичные значения свойств `ScA`, `ScR`, `ScG`, `ScB`. В этом представлении непрозрачный красный цвет записывается в виде `sc#1.0 1.0 0.0 0.0` или просто `sc#1.0 0.0 0.0`. Между любыми значениями можно поставить запятую, хотя это и необязательно.

КОПНЕМ ГЛУБЖЕ

Нестандартные цветовые профили

Продвинутые разработчики или дизайнеры могут определять цвета, основываясь на нестандартном ICC-профиле. (ICC (International Color Consortium) – это международная организация, которая стандартизировала кроссплатформенный формат профиля.) Из программы такой объект `Color` можно получить, обратившись к статическому методу `Color.FromValues`, который принимает массив значений типа `Single` и объект `Uri`, указывающий на файл профиля. В XAML можно воспользоваться конвертером типа `Color`, который принимает строку вида `ContextColor` *`Uri Значения`*.

Например, следующая кисть `SolidColorBrush` закрашивает кнопку красным фоном, обращаясь к файлу с sRGB-профилем, который должен находиться в подкаталоге *system32\spool\drivers\color* того каталога, куда установлена Windows:

```
<Button>
<Button.Background>
  <SolidColorBrush Color="ContextColor
file://C:/Windows/system32/spool/drivers/color/sRGB%20Color%20Space%20Profile.icm
      1.0,1.0,0.0,0.0"/>
</Button.Background>
</Button>
```

Нестандартные профили могут негативно сказаться на производительности, потому что обычно влекут за собой конвертацию цветов. Особенно это относится к растровым изображениям. Чтобы этим можно было как-то управлять, в классе `BitmapSource` поддерживается режим `BitmapCreateOptions.IgnoreColorProfile`, который ускоряет работу за счет игнорирования нестандартного профиля.

Класс LinearGradientBrush

Класс `LinearGradientBrush`, который уже несколько раз встречался в этой книге, закрашивает область градиентным цветом. Последний определяется двумя цветами в конечных точках воображаемого отрезка прямой с линейной интерполяцией между этими точками.

В классе `LinearGradientBrush` имеется свойство содержимого `GradientStops`, в котором хранится коллекция объектов `GradientStop`. Каждый из них обладает свойствами `Color` и `Offset`. Смещение `Offset` – это число типа `double`; оно измеряется относительно ограничивающего прямоугольника закрашиваемой области, причем `0` соответствует началу, а `1` – концу. Таким образом, если применить следующую кисть `LinearGradientBrush` к любому варианту нашего изображения с привидением, то получится результат, показанный на рис. 15.21:

```
<LinearGradientBrush>
  <GradientStop Offset="0" Color="Blue"/>
  <GradientStop Offset="1" Color="Red"/>
</LinearGradientBrush>
```



***Рис. 15.21.** Привидение закрашено простой линейно-градиентной кистью с переходом от синего к красному (см. также цветную вклейку)*

По умолчанию градиент начинается в левом верхнем углу ограничивающего прямоугольника и заканчивается в правом нижнем углу. Но эти точки можно изменить с помощью свойств `StartPoint` и `EndPoint` класса `LinearGradientBrush`. Координаты точек берутся относительно ограничивающего прямоугольника – так же как `Offset` в каждом объекте `GradientStop`. Таким образом, по умолчанию координаты точек `StartPoint` и `EndPoint` равны `(0,0)` и `(1,1)` соответственно.

Если требуется задавать размеры в абсолютных, а не относительных единицах, то можно присвоить свойству `MappingMode` значение `Absolute` (вместо подразумеваемого по умолчанию `RelativeToBoundingBox`). Отметим, что этот режим распространяется только на задание точек `StartPoint` и `EndPoint`; значения `Offset` в объектах `GradientStop` всегда измеряются относительно.

На рис. 15.22 показано, что получается при разных значениях `StartPoint` и `EndPoint` кисти `LinearGradientBrush`, использованной для создания изображения на рис. 15.21 (всюду оставлен относительный режим `MappingMode`). Отметим, что относительные значения не ограничены диапазоном от 0 до 1. Можно задавать как меньшие, так и большие значения, тогда градиент логически будет простираться *за пределы* ограничивающего прямоугольника. (Это в равной мере относится и к свойству `GradientStop Offset`.)

По умолчанию интерполяция цветов производится в пространстве sRGB, но можно перейти к пространству scRGB, присвоив свойству `ColorInterpolationMode` значение `ScRgbLinearInterpolation`. В результате градиент получится гораздо более плавным (рис. 15.23).

И последнее свойство для управления кистью `LinearGradientBrush` – `SpreadMethod`; оно определяет, как заполнять оставшуюся область, не покрытую градиентом. Это имеет смысл только в случае, когда для `LinearGradientBrush` явно указано, что *не* следует покрывать весь ограничивающий прямоугольник. По умолчанию подразумевается значение `Pad` (определенное в перечислении `GradientSpreadMethod`); это означает, что оставшееся место следует закрасить цветом, установленным для конечной точки. Альтернативы – `Repeat` и `Reflect`. В обоих режимах градиент бесконечно повторяется, но в режиме `Reflect` каждый второй градиент меняет направление, чтобы обеспечить плавность перехода. На рис. 15.24 демонстрируются все три значения `SpreadMethod` для линейно-

градиентной кисти, в которой градиент покрывает только средние 10% ограничивающего прямоугольника:

```
<LinearGradientBrush StartPoint=".45,.45" EndPoint=".55,.55" SpreadMethod="XXX">
  <GradientStop Offset="0" Color="Blue"/>
  <GradientStop Offset="1" Color="Red"/>
</LinearGradientBrush>
```

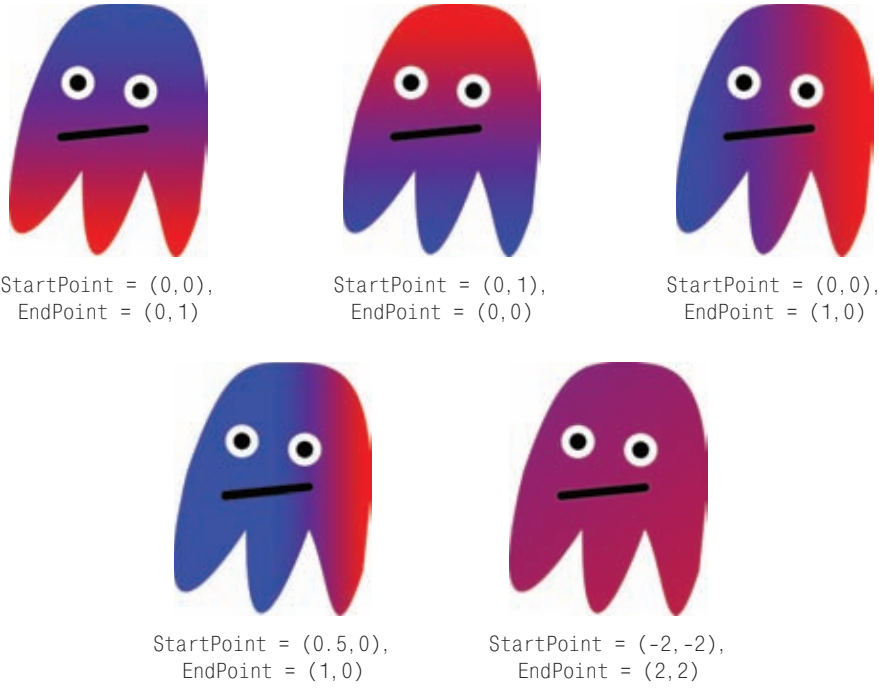


Рис. 15.22. *Результаты различного задания свойств StartPoint и EndPoint (см. также цветную вклейку)*

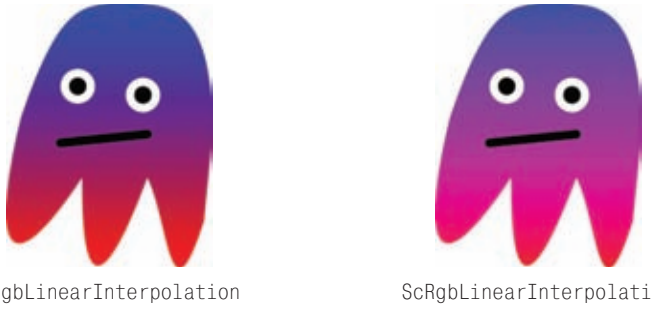


Рис. 15.23. *Режим ColorInterpolationMode влияет на внешний вид градиента (см. также цветную вклейку)*



Рис. 15.24. *При разных значениях SpreadMethod могут получаться совершенно непохожие результаты (см. также цветную вклейку)*

И не забывайте – поскольку в перьях тоже используются кисти, а не просто цвет, то объекты классов `Drawing`, `Shape`, `Control` и многих других могут иметь контуры со сложной заливкой. На рис. 15.25 показан вариант привидения, в котором для объекта `GeometryDrawing`, определяющего тело, задано такое перо:

```
<Pen Thickness="20">
<Pen.Brush>
  <LinearGradientBrush>
    <GradientStop Offset="0" Color="Red"/>
    <GradientStop Offset="0.2" Color="Orange"/>
    <GradientStop Offset="0.4" Color="Yellow"/>
    <GradientStop Offset="0.6" Color="Green"/>
    <GradientStop Offset="0.8" Color="Blue"/>
    <GradientStop Offset="1" Color="Purple"/>
  </LinearGradientBrush>
</Pen.Brush>
</Pen>
```

Отметим, что для кисти, связанной с пером, определено не две, а шесть точек смены градиента `GradientStop`, равномерно распределенных вдоль пути градиента.



Рис. 15.25. *Оконтуривание привидения пером с линейно-градиентной кистью (см. также цветную вклейку)*

СОВЕТ

Чтобы внутри градиентной кисти получить четкую линию, достаточно добавить две точки `GradientStop` с одним и тем же смещением `Offset`, но разными цветами `Color`. В следующем примере мы проделали это для смещений 0.2 *и* 0.6 и получили две четких линии, ограничивающих область, которая закрашена цветом `DarkBlue`:

```
<LinearGradientBrush EndPoint="0,1">
  <GradientStop Offset="0" Color="Aqua"/>
  <GradientStop Offset="0.2" Color="Blue"/>
  <GradientStop Offset="0.2" Color="DarkBlue"/>
  <GradientStop Offset="0.6" Color="DarkBlue"/>
  <GradientStop Offset="0.6" Color="Blue"/>
  <GradientStop Offset="1" Color="Aqua"/>
</LinearGradientBrush>
```

На рис. 15.26 показан результат применения этой кисти к телу привидения.



***Рис. 15.26.** Две четких линии внутри градиентной заливки, полученные в результате дублирования Offset (см. также цветную вклейку)*

Класс RadialGradientBrush

Класс `RadialGradientBrush` работает аналогично `LinearGradientBrush`, только в нем есть всего одна начальная точка, а градиент распространяется от нее по концентрическим эллипсам. Оба класса наследуют общему базовому классу `GradientBrush`, в котором определены свойства `GradientStops`, `SpreadMethod`, `ColorInterpolationMode` и `MappingMode`, рассмотренные выше при обсуждении `LinearGradientBrush`.

На рис. 15.27 показан результат применения к привидению следующей простой радиально-градиентной кисти:

```
<RadialGradientBrush>
  <GradientStop Offset="0" Color="Blue"/>
  <GradientStop Offset="1" Color="Red"/>
</RadialGradientBrush>
```



Рис. 15.27. *Раскраска привидения радиально-градиентной кистью с переходом от синего к красному (см. также цветную вклейку)*

По умолчанию воображаемый эллипс, управляющий распространением градиента, вписан в ограничивающий прямоугольник. Это будет отчетливо видно, если при раскраске привидения задать режим `SpreadMethod`, равный `Repeat` (рис. 15.28).



Рис. 15.28. *Если задать SpreadMethod равным Repeat, то становятся отчетливо видны границы эллипса (см. также цветную вклейку)*

Для изменения размеров и положения воображаемого эллипса в классе `RadialGradientBrush` определены свойства `Center`, `RadiusX` и `RadiusY`. По умолчанию они равны `(0.5,0.5)`, `0.5` и `0.5` соответственно, поскольку выражены в системе координат ограничивающего прямоугольника. Подразумеваемый по умолчанию размер эллипса часто оказывается недостаточным для закрашивания углов области (как на рис. 15.28), и самый простой способ закрасить всю область, не прибегая к свойству `SpreadMethod`, – увеличить радиусы эллипса.

В классе `RadialGradientBrush` имеется также свойство `GradientOrigin`, которое позволяет задать начальную точку градиенту независимо от определяющего эллипса. Чтобы не получить странные результаты, располагайте эту точку внутри определяющего эллипса. По умолчанию она имеет координаты `(0.5,0.5)`, то есть находится в центре эллипса, но на рис. 15.29 показано, что произойдет, если переместить ее в точку `(0,0)`:

```
<RadialGradientBrush GradientOrigin="0,0" SpreadMethod="Repeat">
  <GradientStop Offset="0" Color="Blue"/>
```

```
    <GradientStop Offset="1" Color="Red"/>
  </RadialGradientBrush>
```



Рис. 15.29. *Сдвиг начальной точки градиента в пределах эллипса с помощью свойства GradientOrigin (см. также цветную вклейку)*

Если установлен режим `MappingMode`, равный `Absolute`, то значения всех четырех свойств радиально-градиентной кисти (`Center`, `RadiusX`, `RadiusY`, `GradientOrigin`) считаются выраженными в абсолютных координатах, а не относительно ограничивающего прямоугольника.

Поскольку у каждого цвета `Color` есть альфа-канал, то любую градиентную кисть можно сделать полупрозрачной, изменив альфа-канал в цвете любой точки `GradientStop`. Так, в показанной ниже радиально-градиентной кисти использованы два синих цвета с разными значениями альфа-канала:

```
<RadialGradientBrush RadiusX="0.7" RadiusY="0.7">
  <GradientStop Offset="0" Color="#990000FF"/>
  <GradientStop Offset="1" Color="#000000FF"/>
</RadialGradientBrush>
```

На рис. 15.30 представлен результат применения этой кисти (очень показательный!) к рисунку привидения, помещенному поверх фотографии, чтобы эффект прозрачности был отчетливо виден.



Рис. 15.30. *Полупрозрачное привидение, полученное путем использования цветов с частично прозрачным альфа-каналом (см. также цветную вклейку)*

ПРЕДУПРЕЖДЕНИЕ

Когда речь заходит о градиентах, не все прозрачные цвета одинаковы!

Обратите внимание, что во второй точке `GradientStop` в разметке для рис. 15.30 задан «прозрачный синий» цвет, а не просто `Transparent`. Дело в том, что цвет `Transparent` определен как белый с нулевым альфа-каналом (`#00FFFFFF`). Хотя оба цвета невидимы, переход к каждому из них осуществляется по-разному. Если бы во второй точке `GradientStop` мы задали цвет `Transparent`, то наблюдалось бы не только постепенное изменение альфа от `0x99` до `0`, но также и нарастание красного и зеленого компонентов от `0` до `0xFF`. В результате кисть получилась бы более серой.

Мозаичные кисти

Помимо цветных кистей в WPF определены три *мозаичных кисти* – классы, производные от абстрактного базового класса `TileBrush`. Мозаичная кисть заполняет область повторяющимся образцом. В зависимости от вида кисти источником образца может быть произвольный объект `Drawing`, `Image` или `Visual`.

Все три мозаичных кисти работают одинаково, различие лишь в типе источника. Поэтому в следующем разделе мы рассмотрим основную функциональность всех мозаичных кистей на примере кисти `DrawingBrush`. А затем коротко обсудим две другие кисти: `ImageBrush` и `VisualBrush`.

Класс DrawingBrush

Вложение объекта `Drawing` в `DrawingBrush` аналогично вложению такого же объекта в `DrawingImage`. В следующей XAML-разметке мы взяли описывающий привидение элемент `DrawingGroup` из листинга 15.1 и сделали его свойством `Drawing` элемента `DrawingImage`, который будет использован для закрашивания фона окна:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Title="DrawingBrush as the Background">
<Window.Background>
  <DrawingBrush>
  <DrawingBrush.Drawing>
    <DrawingGroup>
      Три элемента GeometryDrawing из листинга 15.1...
    </DrawingGroup>
  </DrawingBrush.Drawing>
  </DrawingBrush>
</Window.Background>
</Window>
```

На рис. 15.31 показан результат визуализации. В отличие от `DrawingImage`, в классе `DrawingBrush` по умолчанию подразумевается черный, а не белый цвет фона.



Рис. 15.31. *Окно, закрашенное кистью DrawingBrush, содержащей рисунок привидения*

По умолчанию рисунок растягивается и заполняет всю область (или ограничивающий ее прямоугольник, если сама область непрямоугольная), но этим поведением можно управлять с помощью свойства `Stretch`, которое может принимать значения, определенные в перечислении `Stretch` (мы рассматривали их в главе 5 «Компоновка с помощью панелей» при описании элемента `Viewbox`). На рис. 15.32 показан результат установки каждого из возможных значений.

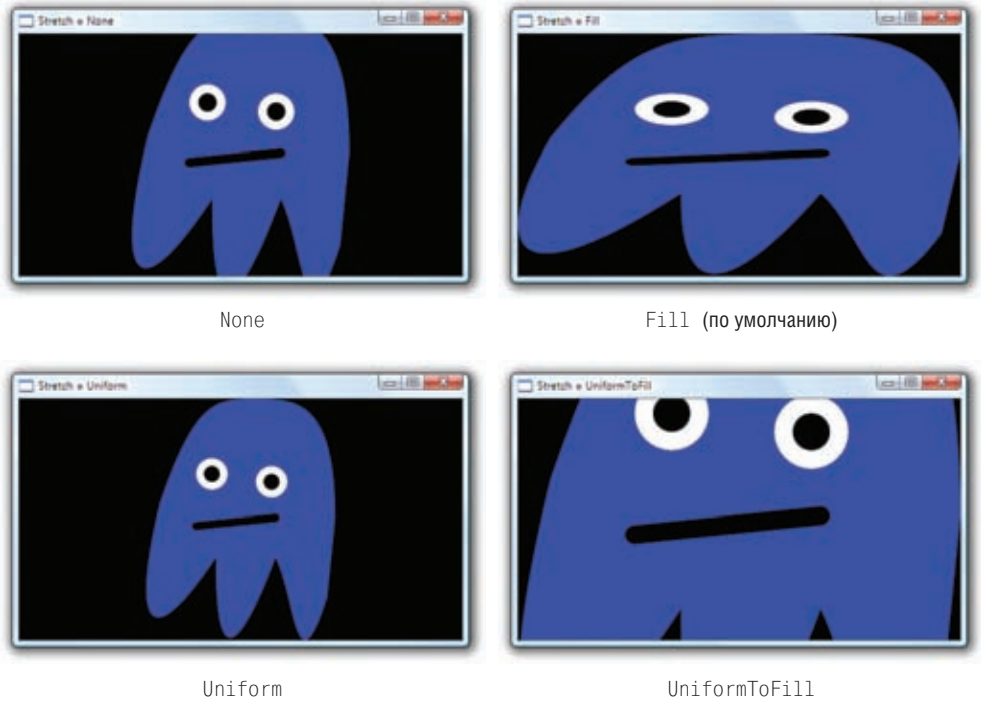


Рис. 15.32. *Кисть DrawingBrush с различными значениями Stretch*

Если свойство Stretch отлично от Fill, то рисунок Drawing центрируется по горизонтали и по вертикали. Но и это можно регулировать установкой для свойства AlignmentX значения Left, Center или Right и для свойства AlignmentY – Top, Center или Bottom.

Самая интересная часть класса DrawingBrush и причина, по которой эта кисть называется мозаичной, – свойство TileMode. Если ему присвоено значение Tile вместо подразумеваемого по умолчанию None, то Drawing будет повторяться в обоих направлениях до бесконечности. Однако для этого еще необходимо определить прямоугольник Rect, который будет занимать первая «плитка». Делается это с помощью свойства Viewport класса DrawingBrush. На рис. 15.33 показано, что получается, когда для Viewport задано несколько разных значений (в синтаксисе *x*, *y*, *ширина*, *высота*, который поддерживает конвертер типа Rect). Самое поразительное, что третье окно на рис. 15.33 можно масштабировать с помощью преобразования ScaleTransform и восстановить рисунок привидения во всех деталях!

Как и для некоторых свойств градиентных кистей, единицы во Viewport по умолчанию измеряются относительно ограничивающего прямоугольника. Это позволяет указать, сколько плиток должно уместиться по вертикали, а сколько – по горизонтали. Но можно также перейти в режим абсолютных координат, изменив значение свойства ViewportUnits (уже знакомого нам типа BrushMappingMode).

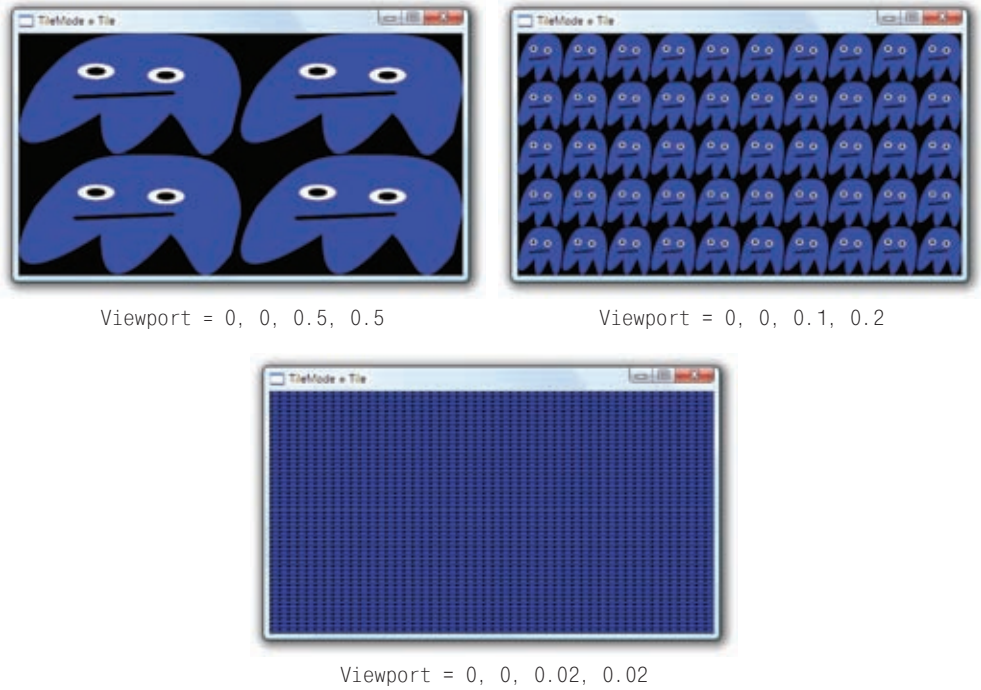


Рис. 15.33. *Различные значения Viewport в режиме TileMode=Tile и Stretch=Fill*

В перечислении TileMode, из которого берутся значения свойства TileMode, определены не только значения Tile и None. Есть еще три значения, позволяющие различными способами переворачивать плитки:

- FlipX – отражать плитку в каждом втором столбце относительно вертикальной оси.
- FlipY – отражать плитку в каждой второй строке относительно горизонтальной оси.
- FlipXY – и то и другое.

На рис. 15.34 показаны все три варианта. Хотя для рисунка привидения это не очень интересно, применение данных режимов к некоторым другим рисункам помогает создать иллюзию непрерывного заполнения.

И последнее, что можно настроить, – это свойство Viewbox, которое позволяет задать в качестве источника каждой плитки (или всей кисти, если TileMode равно None) часть рисунка Drawing. Значением Viewbox является прямоугольник, по умолчанию заданный в единицах измерения относительно ограничивающего прямоугольника, – как и Viewport. А с помощью свойства ViewboxUnits можно перейти к заданию абсолютных координат для Viewbox независимо от текущего значения ViewportUnits.

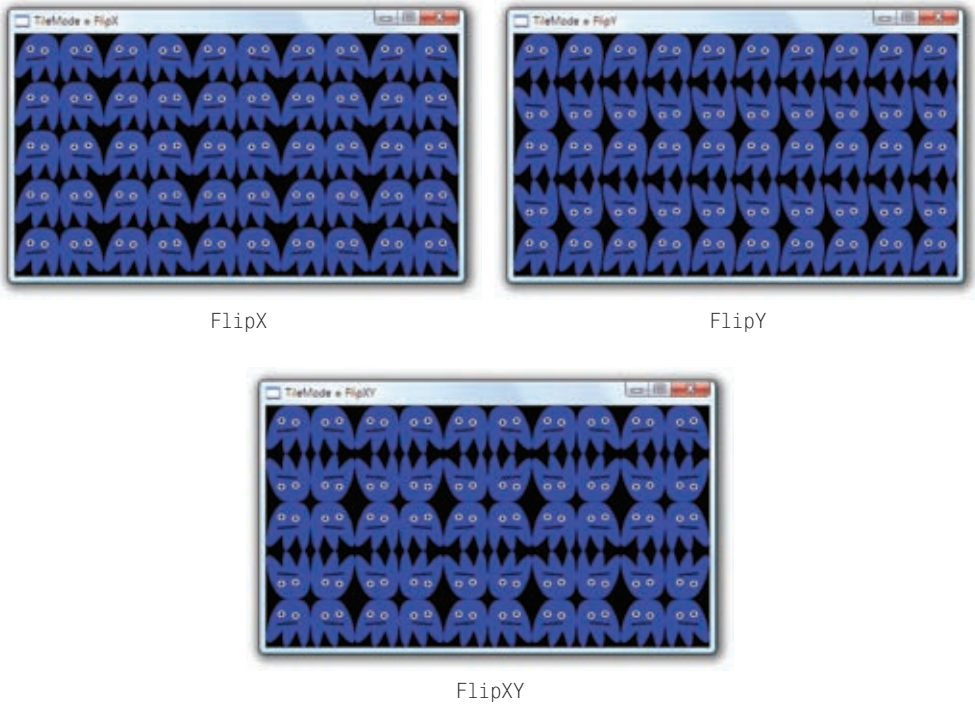
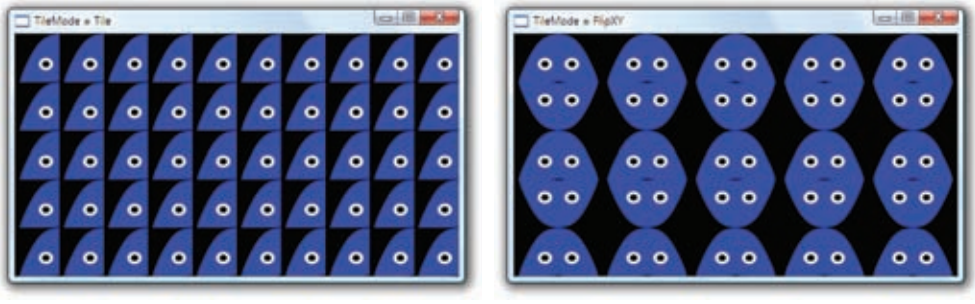


***Рис. 15.34.** Три способа перевертывания плиток, определенные в перечислении TileMode*

На рис. 15.35 в свойство Viewbox записан левый верхний квадрант рисунка привидения Drawing, то есть прямоугольник Rect с параметрами 0, 0, 0.5, 0.5. Затем эта настройка комбинируется с двумя различными значениями TileMode.



Tile FlipXY

Рис. 15.35. *Свойство Viewbox настроено так, что для формирования плитки берется только левый верхний квадрант рисунка. Показан результат в двух режимах TileMode*

И последнее замечание о кисти DrawingBrush: помните, что инкапсулированный в нее объект Drawing необязательно должен иметь тип GeometryDrawing. Это может быть, к примеру, и VideoDrawing!

Класс ImageBrush

Класс ImageBrush идентичен DrawingBrush во всем, кроме одного: в нем определено свойство ImageSource типа ImageSource, а не свойство Drawing типа Drawing. В нем должно храниться растровое, а не векторное изображение. (Но не стоит забывать, что наличие классов DrawingImage и ImageDrawing, обсуждавшихся выше, позволяет записать в кисть DrawingBrush растровое содержимое, а в кисть ImageBrush — векторное!)

В следующей XAML-разметке кисть ImageBrush применяется для закрашивания фона окна Window. Растровое изображение берется из файла *Winter Leaves.jpg*, который входит в состав дистрибутива Windows Vista:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="ImageBrush with TileMode = FlipXY">
<Window.Background>
  <ImageBrush TileMode="FlipXY" Viewport="0,0,0.1,0.2">
  <ImageBrush.ImageSource>
    <BitmapImage UriSource="C:\Users\Public\Pictures\Sample Pictures\Winter
      Leaves.jpg"/>
  </ImageBrush.ImageSource>
  </ImageBrush>
</Window.Background>
</Window>
```

На рис. 15.36 показано получившееся окно.



Рис. 15.36. *Применение кисти ImageBrush в режиме TileMode=FlipXY для создания интересного орнамента*

Класс VisualBrush

Класс `VisualBrush` также идентичен `DrawingBrush` во всем, кроме того, что вместо свойства `Drawing` типа `Drawing` в нем определено свойство `Visual` типа `Visual`. Однако возможность закрашивать с помощью любых объектов `Visual`, даже таких производных от `FrameworkElement`, как `Button` и `TextBox`, делает кисть `VisualBrush` поистине уникальной.

В следующей XAML-разметке фон окна закрашивается кистью `VisualBrush`, содержащей простую кнопку `Button`. На рис. 15.37 показан результат визуализации этой разметки.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="ImageBrush with TileMode = FlipXY">
<Window.Background>
  <VisualBrush TileMode="FlipXY" Viewport="0,0,0.5,0.5">
  <VisualBrush.Visual>
    <Button>OK</Button>
  </VisualBrush.Visual>
  </VisualBrush>
</Window.Background>
</Window>
```

Отметим, что кнопку `Button` внутри кисти `VisualBrush` нельзя нажать. `VisualBrush` лишь воспроизводит внешний вид объектов `Visual`, не обеспечивая никакой интерактивности в закрашенной области.

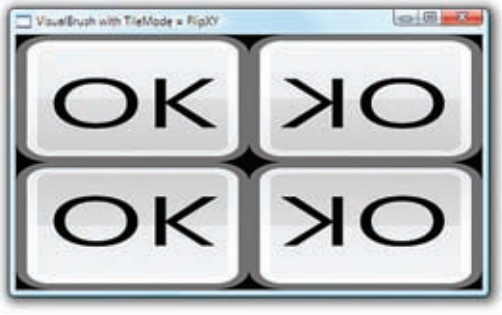


Рис. 15.37. *Фон окна закрашен кистью VisualBrush на основе кнопки*

Вместо того чтобы погружать элементы непосредственно в `VisualBrush`, чаще присваивают свойству `Visual` ссылку на экземпляр `UIElement`, уже присутствующий на экране и доступный для взаимодействия с пользователем. Это можно сделать как в процедурном коде, так и с помощью простой привязки, как показано в следующем примере:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="VisualBrush with TileMode = FlipXY">
  <DockPanel>
    <StackPanel Margin="10" x:Name="stackPanel">
      <Button>Button</Button>
      <CheckBox>CheckBox</CheckBox>
    </StackPanel>
    <Rectangle>
    <Rectangle.Fill>
      <VisualBrush TileMode="FlipXY" Viewport="0,0,0.5,0.5"
        Visual="{Binding ElementName=stackPanel}"/>
    </Rectangle.Fill>
    </Rectangle>
  </DockPanel>
</Window>
```

На рис. 15.38 показан результат визуализации этой разметки. В качестве свойства `Visual` кисти `VisualBrush` используется вся панель `StackPanel`, пристыкованная к левому краю. Кисть `VisualBrush` применяется для закрашивания прямоугольника `Rectangle`, занимающего оставшуюся часть окна. «Настоящие» кнопка и флажок, находящиеся слева, поддерживают интерактивность, а их визуальные копии – нет. Однако в визуальных копиях отражаются все изменения, происходящие с элементами `Button` и `CheckBox`, которые легли в их основу.

Наверное, эти примеры мало кого убедили в том, что для необычных кистей найдется действительно достойное применение! Но такие применения все же существуют. Приложение может воспользоваться кистью `VisualBrush`, чтобы

показать «динамически изменяющееся» внутреннее содержимое (возможно, документов) в компактной форме. Internet Explorer (начиная с версии 7) так и поступает в режиме Быстрые вкладки (Quick Tabs). Кроме того, в Windows технология, лежащая в основе VisualBrush, применяется для показа динамического эскиза окна, когда пользователь задерживает указатель мыши над его значком на панели задач, а также при переключении между окнами нажатием сочетания клавиш Alt+Tab или Windows+Tab.



***Рис. 15.38.** Копирование элемента Visual в кисть VisualBrush*

Еще одно популярное применение кисти VisualBrush – создание эффекта динамического отражения. В следующем окне под полем ввода TextBox создается его отражение – по существу, с помощью той же техники, что и в предыдущем фрагменте XAML:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="TextBox with Reflection" Width="500" Height="200" Background="DarkGreen">
  <StackPanel Margin="40">
    <TextBox x:Name="textBox" FontSize="30"/>
    <Rectangle Height="{Binding ElementName=textBox, Path=ActualHeight}"
      Width="{Binding ElementName=textBox, Path=ActualWidth}">
    <Rectangle.Fill>
      <VisualBrush Visual="{Binding ElementName=textBox}"/>
    </Rectangle.Fill>
    <Rectangle.LayoutTransform>
      <ScaleTransform ScaleY="-0.75"/>
    </Rectangle.LayoutTransform>
    </Rectangle>
  </StackPanel>
</Window>
```

Прямоугольник Rectangle, содержащий отражающую кисть VisualBrush, перевернут с помощью преобразования ScaleTransform. Но коэффициент масштабирования ScaleY равен не -1, а -0.75, чтобы добавить еще небольшую перспективу. Результат изображен на рис. 15.39.

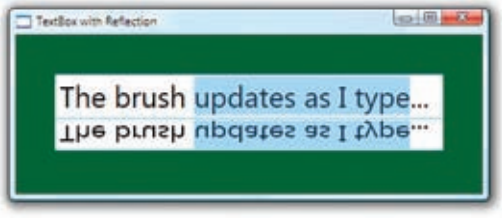


Рис. 15.39. *Простой эффект динамического отражения*

Однако эффект получился не вполне удовлетворительный, потому что отражение слишком четкое. Это можно исправить, воспользовавшись маской непрозрачности, которая обсуждается в следующем разделе.

Кисти как маски непрозрачности

Во всех подклассах `Visual` (и в классе `DrawingGroup`) имеется свойство `Opacity`, действие которого равномерно распространяется на весь объект. Но, кроме того, есть еще и свойство `OpacityMask`, с помощью которого можно создавать различные эффекты, связанные с прозрачностью. Свойство `OpacityMask` можно установить для любой кисти, и тогда альфа-канал этой кисти будет использоваться для того, чтобы определить, какие части объекта должны быть непрозрачными, какие – полностью прозрачными, а какие – занимать промежуточное положение.

Альфа-канал, используемый в сочетании с `OpacityMask`, может браться из цветов в случае цветной кисти, из рисунков – в случае кисти `DrawingBrush`, из изображений – в случае кисти `ImageBrush` (например, из прозрачности PNG-изображения) и т. д. В следующем окне `Window` линейно-градиентная кисть используется в качестве свойства `OpacityMask` для создания довольно-таки уродливой кнопки, представленной на рис. 15.40:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="LinearGradientBrush OpacityMask">
<Window.Background>
  <LinearGradientBrush>
    <GradientStop Offset="0" Color="Orange"/>
    <GradientStop Offset="1" Color="Brown"/>
  </LinearGradientBrush>
</Window.Background>
  <Button Margin="40" FontSize="80">OK
  <Button.OpacityMask>
    <LinearGradientBrush EndPoint="0.1,0.1" SpreadMethod="Reflect">
      <GradientStop Offset="0" Color="Blue"/>
      <GradientStop Offset="1" Color="Transparent"/>
    </LinearGradientBrush>
  </Button.OpacityMask>
  </Button>
</Window>
```



Рис. 15.40. *Кнопка с полосатой маской непрозрачности, созданной благодаря кисти LinearGradientBrush*

В линейно-градиентной кисти, послужившей источником маски непрозрачности, определен повторяющийся градиентный переход от синего к прозрачному, но синий цвет в данном случае неважен, потому что его никогда не видно. Важно лишь, что он полностью непрозрачен.

На рис. 15.41 показано, как выглядела бы та же кнопка, если бы свойству `OpacityMask` была присвоена кисть `DrawingBrush`, содержащая уже знакомый нам рисунок привидения. Слева тело привидения закрашено абсолютно непрозрачным цветом. Результат получается такой же, как если обрезать кнопку `Button` по контуру объекта `Geometry`, составляющего тело привидения. Справа тело привидения закрашено полупрозрачным цветом, но глаза и рот оставлены непрозрачными. Такого результата не удастся добиться только за счет обрезки.



Рис. 15.41. *Кисть DrawingBrush с рисунком привидения в качестве маски OpacityMask; тело закрашено двумя разными цветами*

Имея в своем распоряжении средства для создания приложений в стиле гаджетов (установка для свойства `AllowsTransparency` значения `true` и прочее), описанные в главе 7 «Структурирование и развертывание приложения», мы даже можем применять маску `OpacityMask` к окну верхнего уровня!

Как и было обещано, покажем, каким образом использовать OpacityMask, чтобы улучшить эффект отражения, представленный на рис. 15.39:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBox with Reflection" Width="500" Height="200"
        Background="DarkGreen">
  <StackPanel Margin="40">
    <TextBox x:Name="textBox" FontSize="30"/>
    <Rectangle Height="{Binding ElementName=textBox, Path=ActualHeight}"
      Width="{Binding ElementName=textBox, Path=ActualWidth}">
    <Rectangle.Fill>
      <VisualBrush Visual="{Binding ElementName=textBox}"/>
    </Rectangle.Fill>
    <Rectangle.LayoutTransform>
      <ScaleTransform ScaleY="-0.75"/>
    </Rectangle.LayoutTransform>
    <Rectangle.OpacityMask>
      <LinearGradientBrush EndPoint="0,1">
        <GradientStop Offset="0" Color="Transparent"/>
        <GradientStop Offset="1" Color="#77000000"/>
      </LinearGradientBrush>
    </Rectangle.OpacityMask>
    </Rectangle>
  </StackPanel>
</Window>
```

На рис. 15.42 показан результат этого изменения. Безусловно, этот пример применения OpacityMask отличается большим вкусом, чем все остальные в этой главе.

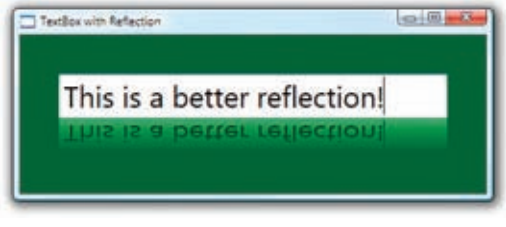


Рис. 15.42. Эффект динамического отражения, улучшенный за счет применения маски непрозрачности

Эффекты

В WPF есть два специальных визуальных эффекта, применимых к любому объекту Visual. Их код находится в пространстве имен System.Windows.Media.Effects. Называются эти эффекты DropShadowEffect и BlurEffect, и оба наследуют

абстрактному классу Effect. На рис. 15.43 показан результат их применения к кнопке Button. WPF применяет эти эффекты к уже сформированному растровому изображению на этапе постобработки.



***Рис. 15.43.** Применение двух встроенных эффектов к кнопке*

Класс Visual раскрывает эту функциональность с помощью защищенного свойства VisualEffect, но все его подклассы (в частности, UIElement) имеют открытое свойство Effect. Чтобы применить эффект к объекту, достаточно записать в свойство Effect последнего ссылку на экземпляр любого подкласса Effect. Например, первая кнопка на рис. 15.43 была создана следующим образом:

```
<Button Width="200">
  DropShadowEffect
<Button.Effect>
  <DropShadowEffect/>
</Button.Effect>
</Button>
```

ПРЕДУПРЕЖДЕНИЕ

Не пользуйтесь свойством BitmapEffect!

Первая версия WPF поставлялась с несколько иными классами эффектов, производными от BitmapEffect. В каждом классе, обладающем свойством Effect, определено также свойство BitmapEffect, в которое можно записать экземпляр класса BitmapEffect. Однако эти эффекты объявлены нерекомендуемыми, поэтому установка упомянутого свойства ровным счетом ничего не дает. Самое существенное отличие новых классов Effect от устаревших BitmapEffect заключается в том, что Effect в общем случае подвергается аппаратному ускорению, тогда как BitmapEffect никогда не был на это рассчитан.

Если у вас есть код, в котором используются старые классы BitmapEffect, то перейти на новые эффекты BlurEffect или DropShadowEffect будет несложно. Увы, для остальных трех BitmapEffect – BevelBitmapEffect, EmbossBitmapEffect и OuterGlowBitmapEffect – встроенных замен пока нет.

Конечно, можно и оставить BitmapEffect в программе, работающей со старой версией WPF. Они, правда, способны понизить производительность приложения, но при умеренном использовании в подходящих случаях особого вреда не нанесут. К тому же тот факт, что новые эффекты аппаратно ускоряются, еще не значит, что их можно использовать, когда вздумается. Все равно следует принимать во внимание соображения производительности и действовать осмотрительно.

На рис. 15.43 продемонстрировано действие обоих эффектов с подразумеваемыми по умолчанию настройками. Однако в каждом классе есть целый ряд дополнительных свойств. Эти свойства и принимаемые ими значения сведены в табл. 15.3.

Наиболее впечатляющими в WPF являются не только эти два встроенных эффекта, но и третий – `ShaderEffect`, подкласс класса `Effect`, который позволяет без особого труда добавлять собственные эффекты. (Устаревшие растровые эффекты, производные от класса `BitmapEffect`, не предоставляли такой возможности расширения без написания кода на C++ COM.) Унаследовав абстрактному классу `ShaderEffect`, вы сможете применить произвольный *пиксельный построитель текстуры* к любому объекту, обладающему свойством `Effect`. При этом используется поддержка построителей текстур в DirectX, а это означает, что сами построители должны быть написаны на языке High Level Shader Language (HLSL).

СОВЕТ

Многочисленные эффекты, построенные на базе класса `ShaderEffect`, имеются в библиотеке WPF Pixel Shader Effects Library по адресу *http://wpffx.codeplex.com*. Там вы найдете следующие эффекты, применяемые к одному объекту:

```
BandedSwirlEffect
BloomEffect
BrightExtractEffect
ColorKeyAlphaEffect
ColorToneEffect
ContrastAdjustEffect
DirectionalBlurEffect
EmbossedEffect
GloomEffect
GrowablePoissonDiskEffect
InvertColorEffect
LightStreakEffect
MagnifyEffect
MonochromeEffect
PinchEffect
PixelateEffect
RippleEffect
SharpenEffect
SmoothMagnifyEffect
SwirlEffect
ToneEffect
ToonEffect
ZoomBlurEffect
```

А также эффекты перехода от одного объекта к другому:

```
BandedSwirlTransitionEffect
BlindsTransitionEffect
BloodTransitionEffect
CircleRevealTransitionEffect
CircleStretchTransitionEffect
CircularBlurTransitionEffect
CloudReveralTransitionEffect
CloudyTransitionEffect
CrumbleTransitionEffect
DissolveTransitionEffect
DropFadeTransitionEffect
FadeTransitionEffect
LeastBrightTransitionEffect
LineRevealTransitionEffect
MostBrightTransitionEffect
PixelateInTransitionEffect
PixelateOutTransitionEffect
PixelateTransitionEffect
RadialBlurTransitionEffect
RadialWiggleTransitionEffect
RandomCircleRevealTransitionEffect
RippleTransitionEffect
RotateTransitionEffect
SaturateTransitionEffect
ShrinkTransitionEffect
SlideInTransitionEffect
SmoothSwirlTransitionEffect
SwirlTransitionEffect
WaterTransitionEffect
WaveTransitionEffect
```

Таблица 15.3. Свойства эффектов

Эффект	Свойство	Значение по умолчанию
DropShadowEffect	RenderingBias (чему отдать предпочтение при визуализации): Performance (производительности) или Quality (качеству)	Performance
	BlurRadius (радиус размытия): неотрицательное значение типа double	5
	Color: произвольный цвет (даже с полупрозрачным альфа-каналом)	Black
	Direction: число типа double, представляющее угол в градусах	315
	Opacity: число типа double от 0 (полностью прозрачный) до 1 (полностью непрозрачный)	1
	ShadowDepth (глубина тени): неотрицательное число типа double	5
BlurEffect	RenderingBias: Performance или Quality	Performance
	Radius: неотрицательное значение типа double	5
	KernelType (тип ядра свертки): Box или Gaussian	Gaussian

Повышение производительности визуализации

У векторной графики много преимуществ по сравнению с растровой, но им неизменно сопутствуют проблемы масштабирования. Даже при использовании наименее ресурсоемкого подхода к рисованию (класса DrawingContext, обсуждавшегося в разделе «Класс Visual»), перерисовка сложных рисунков может оказаться дорогостоящим делом. А в случаях когда требуется быстро генерировать последовательность рисунков, например при анимированном увеличении размера, затраты на визуализацию могут негативно повлиять на мнение пользователя о приложении.

Поэтому разработчики часто ищут любые способы избежать перерисовки. У WPF на этот случай припасено два интересных средства. Первое – класс RenderTargetBitmap, входивший уже в первую версию WPF. Второе – класс BitmapCache и дополняющий его класс BitmapCacheBrush – вместе они называются *механизмом кэширования композиции*. Этот механизм – самое существенное новшество в области двумерной графики, появившееся в версии WPF 4.

Класс RenderTargetBitmap

С помощью класса RenderTargetBitmap, производного от BitmapSource (который, в свою очередь, наследует ImageSource), можно нарисовать объект Visual на

растре, а затем отобразить этот растр вместо исходного Visual. Перерисовка растрового изображения производится куда быстрее, чем объекта Visual.

Ниже представлен типичный подход к созданию объекта RenderTargetBitmap, заполненного содержимым Visual:

```
private static ImageSource ProduceImageSourceForVisual(
  Visual source, double dpiX, double dpiY)
{
  if (source == null)
    return null;

  Rect bounds = VisualTreeHelper.GetDescendantBounds(source);

  RenderTargetBitmap bitmap = new RenderTargetBitmap(
    (int)(bounds.Width * dpiX / 96), (int)(bounds.Height * dpiY / 96),
    dpiX, dpiY, PixelFormats.Pbgra32);

  DrawingVisual drawingVisual = new DrawingVisual();
  using (DrawingContext ctx = drawingVisual.RenderOpen())
  {
    ctx.DrawRectangle(new VisualBrush(source), null,
      new Rect(new Point(), bounds.Size));
  }
  bitmap.Render(drawingVisual);
  return bitmap;
}
```

Обертывание исходного объекта Visual кистью VisualBrush – прием, позволяющий учитывать требования компоновки. Если содержимое Visual не требует от своего родителя никакого поведения в части компоновки, то обертывание можно опустить.

Класс BitmapCache

Класс RenderTargetBitmap способен повысить производительность рендеринга, но обладает некоторыми недостатками:

- Рендеринг производится программно
- Класс работает синхронно в потоке ГИП
- Он доступен только из процедурного кода
- Он отделен от дерева элементов

Новый механизм кэширования композиции в WPF 4 решает все эти проблемы, и к тому же он гораздо проще в использовании, чем RenderTargetBitmap! Класс BitmapCache позволяет автоматически кэшировать любой элемент UIElement вместе со всем деревом его подэлементов в виде растрового изображения в видеопамяти. Он обеспечивает аппаратный рендеринг дерева элементов в отдельном потоке рендеринга и легко доступен из XAML-кода. По существу, это аппаратно ускоренный вариант RenderTargetBitmap, хотя в нем отсутствует возможность доступа к отдельным битам растра.

Чтобы воспользоваться этим механизмом, достаточно установить свойство CacheMode для любого элемента UIElement, который вы хотели бы кэшировать. Типом свойства CacheMode является абстрактный класс CacheMode, хотя на данный момент в WPF есть лишь один его конкретный подкласс – BitmapCache. Например, в сетке Grid это можно записать следующим образом:

```
<Grid ...>
<Grid.CacheMode>
  <BitmapCache/>
</Grid.CacheMode>
  ...
</Grid>
```

Когда кэшированный элемент (или любой из его потомков) обновляется, BitmapCache автоматически обновляет только изменившуюся область. Кэш не становится недействительным в результате смены родителей, или изменения преобразований, или уровня прозрачности элементов! Более того, WPF автоматически использует сам элемент там, где это необходимо для обеспечения интерактивности.

Поведение объекта BitmapCache управляется тремя свойствами:

- RenderAtScale – число типа double, по умолчанию равное 1. Определяет, в каком масштабе элемент рисуется в кэшированном растре. Это свойство особенно интересно, если вы планируете изменять размер элемента. Если заранее нарисовать элемент в увеличенном масштабе, то, когда настанет время показать результат на экране, искажения не возникнет. Присваивание RenderAtScale значения, меньшего 1, повышает производительность за счет качества.
- SnapsToDevicePixels – булевское значение; если равно true, то для растрового изображения включается режим привязки пикселов.
- EnableClearType – булевское значение; если равно true, то включается режим рендеринга ClearType вместо полутонального сглаживания. Если это свойство равно true, то следует установить значение true также для свойства SnapsToDevicePixels, иначе рендеринг будет произведен некорректно.

Изменение любого из этих свойств делает кэш недействительным.

На рис. 15.44 показан результат применения свойства RenderAtScale к нескольким кнопкам, созданным следующим XAML-кодом:

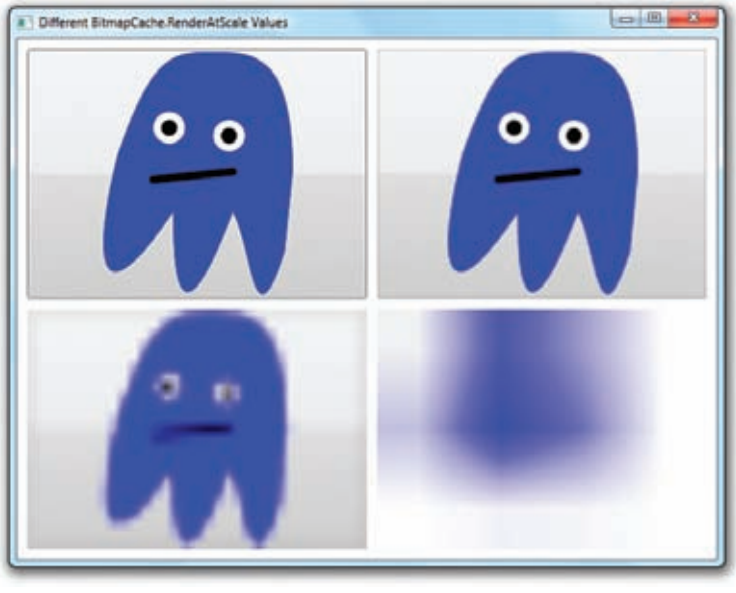
```
<Button>
<Button.CacheMode>
  <BitmapCache RenderAtScale="..."/>
</Button.CacheMode>
  ...
</Button>
```

Отметим, что кэширование применяется ко всему элементу Button, а не только к его содержимому, так что на обрамлении Button при малых значениях RenderAtScale также становятся заметны отдельные пикселы.

СОВЕТ

Класс `BitmapCache` переходит в режим программного рендеринга, если аппаратное ускорение не поддерживается. Однако в этом режиме максимально допустимый размер кэшированного растрового изображения составляет 2048×2048 пикселов.

`RenderAtScale`=1 `RenderAtScale`=0.5



`RenderAtScale`=0.1 `RenderAtScale`=0.01

Рис. 15.44. *Использование свойства RenderAtScale для понижения разрешения кэшированного растрового изображения*

СОВЕТ

Класс `BitmapCache` лучше всего подходит для статического содержимого, которое подвергается анимации или прокрутке, чтобы избежать образования узких мест в конвейере визуализации WPF из-за большого объема вычислений, связанных с повторной тесселяцией и растеризацией каждого кадра. Но необходим разумный баланс. Чем больше кэш, тем выше потребление памяти графического процессора.

Класс BitmapCacheBrush

Класс BitmapCacheBrush позволяет многократно использовать одно и то же кэшированное растровое изображение в тех местах, где можно применить кисть. Запишите в свойство Target объекта BitmapCacheBrush ссылку на любой объект Visual — и BitmapCacheBrush будет использовать кэшированный растр, если в этом Visual установлено свойство CacheMode. Но, даже если свойство CacheMode не установлено, управлять кэшированием можно непосредственно на уровне BitmapCacheBrush. (В классе BitmapCacheBrush нет свойства CacheMode, но есть свойство BitmapCache типа BitmapCache, которое выполняет те же функции.)

Таким образом, BitmapCacheBrush — более эффективная версия класса VisualBrush. Увеличение производительности обусловлено не только использованием кэшированного растрового изображения, но и интеллектуальной обработкой изменившихся областей.

ПРЕДУПРЕЖДЕНИЕ

Класс BitmapCacheBrush игнорирует привязку пикселов!

Класс BitmapCacheBrush игнорирует значение свойства SnapsToDevicePixels, считая, что оно равно false. Поэтому не устанавливайте свойство EnableClearType равным true для объекта BitmapCache, сохраненного в BitmapCacheBrush, или для свойства CacheMode того объекта Visual, который записан в свойство Target объекта BitmapCacheBrush.

Резюме

Возможно, поначалу вам показалось натяжкой включение главы о двумерной графике в часть, посвященную мультимедийным средствам. Но теперь, надеюсь, вы понимаете, насколько развита поддержка двумерной графики в WPF. В отличие от прежних технологий Windows, WPF предоставляет вам всю мощь DirectX в сочетании с простотой использования системы, работающей в режиме запоминания.

В этой главе мы уделили особое внимание векторной графике, но также постарались показать, как в этот контекст органично вписывается растровая графика. Вы также получили первое представление о поддержке видео; эта тема подробно рассматривается в главе 18.

Как часто бывает в WPF, своей мощью двумерная графика в немалой степени обязана тесной интеграции с другими компонентами. Для рисования линий, фигур и привидений применяются те же самые графические примитивы, что и для рисования кнопок, меню и списков. В следующей главе мы посмотрим, как WPF распространяет эту поддержку на третье измерение.

- Введение в трехмерную графику
- Камеры и системы координат
- Класс Transform3D
- Класс Model3D
- Класс Visual3D
- Класс Viewport3D
- Преобразование двумерных и трехмерных систем координат

16

Трехмерная графика

При проектировании трехмерных API в технологии Windows Presentation Foundation ставилась задача сделать их такими же доступными и простыми в использовании, как и все остальные части каркаса .NET Framework. Поскольку трехмерная графика (или 3D-графика) полностью интегрирована в платформу WPF, многие встречающиеся в ней концепции пересекаются с двумерной графикой и другими компонентами. Это существенно облегчает изучение новой темы для разработчиков, знакомых с двумерной графикой, потому что они постоянно сталкиваются с уже известными принципами и соглашениями. Таким образом, WPF – прекрасное средство для изучения 3D-графики.

В этой главе мы уделим основное внимание тем аспектам API, которые уникальны для 3D-графики, однако не следует забывать, что своей мощью все эти механизмы в значительной степени обязаны глубокой интеграции с прочими частями платформы. Эта интеграция пронизывает все: удаленное взаимодействие с пользовательским интерфейсом, печать, работу в контексте вебприложений с частичным доверием.

Как и в случае двумерной графики, возможности 3D доступны как из процедурного кода, так и из XAML-разметки. Чтобы отобразить в WPF трехмерную сцену, необходимо построить граф объектов, а делается это примерно так же, как создание изображения из объектов `Shape` или `Drawing`. После того как сцена описана, система берет на себя ответственность за ее визуализацию и перерисовку в нужные моменты времени. Все средства работы со свойствами, в частности привязка к данным и анимация, в полной мере применимы к 3D-объектам.

Трехмерное содержимое не отделено от остального мира непроницаемой стеной. Сцены, вложенные в элемент `Viewport3D`, органично сочетаются с другими элементами `UIElement`, их можно включать в шаблоны и в элементы `ItemsControl`. А на поверхностях 3D-моделей можно отображать двумерное содержимое, например видео, векторные рисунки `Drawing` и объекты `Visual`. Такие

службы, как проверка попадания, автоматически действуют и в трехмерных участках визуального дерева.

У этой главы три цели. Во-первых, дать введение в трехмерную графику для разработчиков, которые раньше не имели с ней дела. Во-вторых, служить справочником по трехмерным API, имеющимся в WPF. В-третьих, стать путеводителем для опытных разработчиков, знакомых с другими платформами, например DirectX, или пытающихся написать инструментальные средства, способные взаимодействовать с WPF.

Введение в трехмерную графику

Задача трехмерной графики – создание по трехмерным моделям двумерных изображений, которые можно было бы отобразить на некоем устройстве вывода, например на экране монитора. Создание изображений по 3D-модели – это новая парадигма, отличающаяся от всего, к чему привыкли разработчики, имевшие дело с двумерной графикой. При работе с двумя измерениями мы обычно рисуем точно ту фигуру, которую хотим изобразить, применяя абсолютные координаты. Если нужен прямоугольник с левым верхним углом в точке (50,75) шириной 100 единиц и высотой 30 единиц, то мы, как правило, берем элемент `Rectangle` (или `GeometryDrawing` с вложенным в него объектом `RectangleGeometry` с соответствующими размерами). Взгляните на листинг 16.1, где с помощью двумерных объектов `Drawing` представлен домик. Результат визуализации показан на рис. 16.1.

Листинг 16.1. Рисование домика с помощью двумерных объектов Drawing

```
<Page Background="Black"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Image>
  <Image.Source>
    <DrawingImage>
    <DrawingImage.Drawing>
      <DrawingGroup x:Name="House">
        <GeometryDrawing x:Name="Front" Brush="Red"
          Geometry="M0,260 L0,600 L110,670 L110,500 L190,550
            L190,710 L300,775 L300,430 L150,175"/>
        <GeometryDrawing x:Name="Side" Brush="Green"
          Geometry="M300,430 L300,775 L600,600 L600,260"/>
        <GeometryDrawing x:Name="Roof" Brush="Blue"
          Geometry="M150,175 L300,430 L600,260 L450,0"/>
      </DrawingGroup>
    </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
  </Image>
</Page>
```



Рис. 16.1. *Домик, нарисованный с помощью двумерных объектов Drawing*

Хотя нарисованный домик вполне выглядит трехмерным, данные, на основе которых построено изображение, двумерные. С точки зрения системы оно состоит из плоских двумерных многоугольников. Многоугольники можно вращать на плоскости, но вы не можете ни повернуть дом, чтобы увидеть заднюю сторону, ни сгенерировать изображение внутренней планировки. Нет никакой информации о тех частях дома, которые отсутствуют на рисунке. Если требуется получать изображения дома с разных точек обзора (не создавая независимые двумерные рисунки для каждого вида), то системе необходимо сообщить больше информации.

В листинге 16.2 показано, как то же самое изображение можно было бы получить, используя класс `Model3D` вместо двумерных рисунков `Drawing`. Код в листинге 16.2 длиннее своего двумерного аналога, зато обеспечивает большую гибкость в работе с домиком. Имея 3D-модель, можно генерировать двумерные изображения с любой точки обзора, изменив лишь несколько свойств; на рис. 16.2 это четко видно.

Листинг 16.2. Домик, нарисованный с помощью 3D-модели

```
<Page Background="Black"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Viewport3D>
  <Viewport3D.Camera>
    <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
  </Viewport3D.Camera>
  <Viewport3D.Children>
    <ModelVisual3D x:Name="Light">
    <ModelVisual3D.Content>
```

```
      <AmbientLight/>
    </ModelVisual3D.Content>
    </ModelVisual3D>
    <ModelVisual3D>
    <ModelVisual3D.Content>
       <Model3DGroup x:Name="House">

        <GeometryModel3D x:Name="Roof">
          <GeometryModel3D.Material>
            <DiffuseMaterial Brush="Blue"/>
          </GeometryModel3D.Material>
          <GeometryModel3D.Geometry>
            <MeshGeometry3D Positions="-1,1,1 0,2,1 0,2,-1 -1,1,
                                       -1 0,2,1 1,1,1 1,1,-1 0,2,-1"
                            TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7"/>
          </GeometryModel3D.Geometry>
        </GeometryModel3D>

        <GeometryModel3D x:Name="Sides">
        <GeometryModel3D.Material>
          <DiffuseMaterial Brush="Green"/>
        </GeometryModel3D.Material>
        <GeometryModel3D.Geometry>
          <MeshGeometry3D Positions="-1,1,1 -1,1,-1 -1,-1,
                                     -1 -1,-1,1 1,1,-1 1,1,1 1,-1,1 1,-1,-1"
                          TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7"/>
          </GeometryModel3D.Geometry>
        </GeometryModel3D>

        <GeometryModel3D x:Name="Ends">
        <GeometryModel3D.Material>
          <DiffuseMaterial Brush="Red"/>
        </GeometryModel3D.Material>
        <GeometryModel3D.Geometry>
          <MeshGeometry3D
            Positions="-0.25,0,1 -1,1,1 -1,-1,1 -0.25,-1,1 -0.25,0,1
                       -1,-1,1 0.25,0,1 1,-1,1 1,1,1 0.25,0,1 0.25,-1,
                       1 1,-1,1 1,1,1 0,2,1 -1,  1,1 -1,1,1 -0.25,0,
                       1 0.25,0,1 1,1,1 1,1,-1 1,-1,-1 -1,-1,-1 -1,
                       1,-1 1,1,-1 -1,1,-1 0,2,-1"
            TriangleIndices="0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
                             15 17 18 19 20 21 19 21 22 23 24 25"/>
        </GeometryModel3D.Geometry>
        </GeometryModel3D>

       </Model3DGroup>
    </ModelVisual3D.Content>
    </ModelVisual3D>
  </Viewport3D.Children>
  </Viewport3D>
</Page>
```



***Рис. 16.2.** Несколько видов дома*

В листинге 16.2 встречаются почти все объекты, которые мы будем обсуждать в этой главе. Хотя многие классы новые, они являются прямыми обобщениями типов, рассмотренных в главе 15 «Двумерная графика». В табл. 16.1 приведено сопоставление некоторых трехмерных типов их ближайшим двумерным аналогам.

Таблица 16.1. Сопоставление двумерных и трехмерных типов

Двумерный тип	Трехмерный тип	Описание
`Drawing`	`Model3D`	Объекты `Drawing` представляют части двумерного содержимого, например изображение, которое можно визуализировать с помощью объекта `Visual`. Объекты `Model3D` представляют части трехмерных моделей, которые можно визуализировать с помощью объекта `Visual3D`.
`Geometry`	`Geometry3D`	Объект `Geometry` представляет двумерную фигуру. Такие объекты умеют отвечать на вопросы, касающиеся занимаемой ими области и пересечения с другими объектами. Сам по себе объект `Geometry` невозможно визуализировать. Класс `GeometryDrawing` объединяет объект `Geometry` с кистью `Brush` и тем самым наделяет его визуальным представлением. Объект `Geometry3D` представляет трехмерную поверхность. Чтобы визуализировать `Geometry3D`, необходимо объединить его с материалом `Material` с помощью объекта `GeometryModel3D`.
`Visual`	`Visual3D`	`Visual` – базовый класс элементов, которые визуализируют двумерное содержимое. К ним относятся `DrawingVisual` и все классы, производные от `FrameworkElement`, например `Control` и `Shape`. `Visual3D` – базовый класс элементов, которые визуализируют трехмерное содержимое. `ModelVisual3D` – конкретный подкласс `Visual3D`, который визуализирует трехмерное содержимое, представленное в виде объектов `Model3D`.

Двумерный тип	Трехмерный тип	Описание
`UIElement`	`UIElement3D`	`UIElement` – подкласс `Visual`, который добавляет функциональность, ассоциируемую со многими основополагающими для WPF концепциями. Часто говорят, что класс `UIElement` привносит «жизнь»[1] (LIFE – layout, input, focus, eventing – компоновка, ввод, фокус, события) в иерархию двумерных классов. `UIElement3D` – трехмерный аналог класса `UIElement`, добавляет IFE (input, focus, eventing – ввод, фокус, события) в мир 3D. Он позволяет трехмерным объектам вносить свой вклад в поведение приложения, а не просто служить визуальным представлением трехмерного содержимого.
`Transform`	`Transform3D`	Подклассы `Transform` применяются для позиционирования, поворота и изменения размеров двумерных объектов `Drawing` и `Visual`. В листинге 16.2 нет объектов `Transform3D`, но когда ниже в этой главе мы будем говорить о трехмерных преобразованиях, то увидим, что они играют аналогичную роль для объектов `Model3D` и `Visual3D`.

Хотя большинство 3D-классов – прямые обобщения двумерного API, есть два понятия, не имеющих аналогов в двумерном мире; в листинге 16.2 они также присутствуют:

- **Камеры** `Camera` – для генерации изображения 3D-модели необходимо поместить на сцену виртуальную камеру. Как и в случае реальной камеры, ее положение, ориентация и другие свойства определяют вид сцены.
- **Материалы** `Material` **и источники света** `Light` – в двумерной графике визуальное представление закрашенного геометрического объекта обеспечивает кисть. В трехмерной графике кисти тоже используются, но вид трехмерной поверхности зависит еще и от освещения.

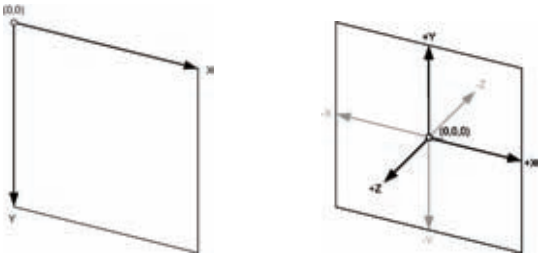
Ниже вы убедитесь, что камера, материалы и источники света играют важную роль в визуализации динамических 3D-сцен.

Камеры и системы координат

В реальном мире наблюдаемая картина зависит от того, где вы стоите, куда смотрите, как повернули голову и т. д. В WPF для управления тем, что появится в объекте `Viewport3D`, необходимо поместить на 3D-сцену виртуальную

[1] Здесь игра слов: с аббревиатурой LIFE совпадает слово life – жизнь. – *Прим. перев.*

камеру (объект Camera). Для этого следует позиционировать и ориентировать камеру в мировой системе координат (иногда для краткости ее называют *мировым пространством*). На рис. 16.3 показаны двумерная и трехмерная системы координат, применяемые в WPF.



2D 3D

Рис. 16.3. *Двумерная и трехмерная системы координат*

Помимо дополнительной оси *z* между двумерной и трехмерной системами координат существуют еще два различия.

В трехмерной системе ось *y* обычно направлена вверх, а не вниз. Кроме того, отрицательные координаты, довольно редко используемые в двумерной графике, – обычное дело в мире 3D. Поэтому начало координат обычно рассматривается как центр пространства, а не его левый верхний угол. Конечно, это всего лишь соглашения, и с помощью преобразований вы вольны отобразить эту систему координат на любую удобную для вас.

Два наиболее употребительных подкласса Camera – OrthographicCamera и PerspectiveCamera – обладают рядом свойств для задания позиции и ориентации камеры в мировой системе координат. В последующих разделах мы рассмотрим эти свойства и покажем, как с их помощью управлять тем, какая часть 3D-сцены видна.

Свойство Position

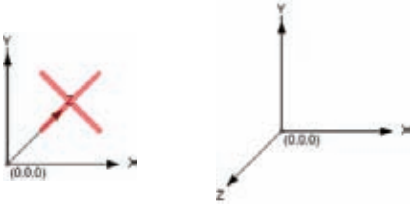
Свойство Position управляет расположением камеры в пространстве. Перемещая камеру, можно менять вид сцены. Свойство Position имеет тип Point3D. Объекты типа Point3D содержат координаты *x*, *y*, *z* и определяют точку в системе координат. При визуализации модели дома в листинге 16.2 мы задали позицию (5,5,5):

```
<Viewport3D.Camera>
  <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
</Viewport3D.Camera>
```

КОПНЕМ ГЛУБЖЕ

В WPF используется правосторонняя система координат

Под *направленностью* системы координат понимается соотношение между осью *z* и плоскостью *xy*. Если положительные направления осей *x* и *y* определены так, как показано на рис. 16.4, то направление оси *z* можно выбрать двумя способами. В левосторонней системе координат ось *z* направлена в сторону от наблюдателя, как показано на левом рисунке. В правосторонней системе координат ось *z* направлена в сторону к наблюдателю, как изображено на правом рисунке.



Левосторонняя Правосторонняя

Рис. 16.4. *Левосторонняя и правосторонняя системы координат*

В WPF стандартно используется правосторонняя система координат. Название «правосторонняя» происходит от *правила правой руки*: если указательный палец направлен в сторону положительного направления оси *x*, а средний – в сторону положительного направления оси *y*, то большой палец укажет положительное направление оси *z* (рис. 16.5).

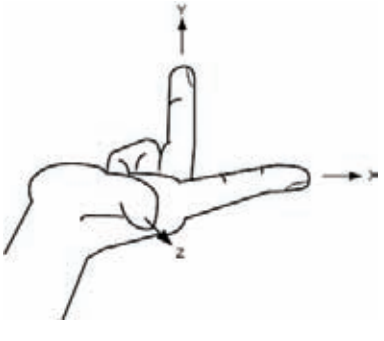


Рис. 16.5. *Правило правой руки*

Это простой, хотя и примитивный способ запомнить соотношения между осями. Ниже в этой главе мы расскажем еще об одном варианте правила правой руки для запоминания порядка обхода треугольников в классе `MeshGeometry3D`.

Это означает, что камера смещена от начала координат на пять единиц вправо по оси *x*, на пять единиц вверх по оси *y* и на пять единиц вперед по оси *z*. На рис. 16.6 видно, что камера расположена над домом и повернута к его юго-восточной стороне. (Не существует стандартного согласования между направлениями осей и сторонами света, но для удобства вы можете определить его в своем приложении.)

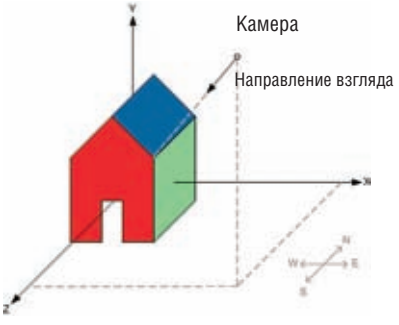


Рис. 16.6. *Камера расположена так, чтобы можно было видеть юго-восточную сторону дома*

Чтобы увидеть юго-западную сторону дома, можно попробовать расположить камеру в точке (-5,5,5):

```
<Viewport3D.Camera>
  <OrthographicCamera Position="-5,5,5" LookDirection="-1,-1,-1" Width="5"/>
</Viewport3D.Camera>
```

Новая позиция изображена на рис. 16.7.

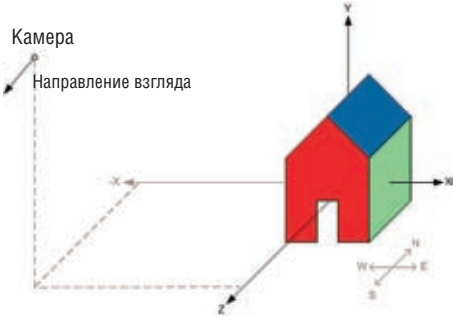


Рис. 16.7. *Камера расположена так, чтобы можно было видеть юго-западную сторону дома*

Но одного лишь перемещения камеры в новую позицию недостаточно, нужно еще изменить направление взгляда. Если прибегнуть к физической аналогии,

то представьте, что вы смотрите на приятеля в видоискатель, а потом делаете 10 больших шагов влево. Если не повернуть к нему лицо, то теперь вы будете снимать стену. Свойство `LookDirection` как раз и управляет тем, в каком направлении повернута камера.

Свойство LookDirection

Свойство `LookDirection` определяет, в каком направлении повернута камера. Его тип – `Vector3D`. Как и `Point3D`, объект типа `Vector3D` содержит координаты x, y, z, но определяет не точку в пространстве, а направление и абсолютную величину вектора. Абсолютная величина вектора `Vector3D` называется его длиной `Length` и вычисляется по следующей формуле:

$$\sqrt{x^2 + y^2 + z^2}.$$

ПРЕДУПРЕЖДЕНИЕ

Не забывайте про мертвую зону!

Поверхности, расположенные к камере ближе, чем на расстоянии `NearPlaneDistance`, отсекаются. При задании свойства `Position` следует внимательно следить за тем, чтобы интересующие вас объекты находились от камеры на расстоянии не меньше, чем значение `NearPlaneDistance`, в направлении взгляда. На рис. 16.8 показано, что произойдет, если придвинуть камеру слишком близко к модели дома.

Смысл свойства `NearPlaneDistance` класса `Camera` в том, чтобы обойти проблему ограниченной точности вычислений с плавающей точкой в Z-буфере графического процессора. Если точности Z-буфера не хватает, то возникает феномен, получивший название *коллизии в Z-буфере* (*Z-fighting*), когда графический процессор не в состоянии определить, какие поверхности ближе к камере. На рис. 16.9 приведен пример того, что может произойти при рендеринге в условиях коллизии. Структура таких артефактов изменяется в зависимости от угла обзора.

Причиной коллизии в Z-буфере обычно является попытка визуализировать объекты, расположенные слишком близко к камере. Свойство `NearPlaneDistance` класса `Camera` позволяет устранить коллизии в Z-буфере, просто отсекая объекты, находящиеся ближе определенного порога. По умолчанию `NearPlaneDistance` равно 0.125, что дает хорошие результаты.

Существуют и другие, не столь распространенные, причины появления коллизий в Z-буфере. Одна из них – попытка визуализировать объекты, находящиеся *очень* далеко от камеры. Имеется соответствующее свойство `FarPlaneDistance`, которое помогает бороться с этой проблемой, но поскольку она возникает редко, то по умолчанию значением этого свойства является положительная бесконечность.

Наконец, коллизии в Z-буфере могут возникать, когда визуализируются две поверхности, расположенные почти, но не совсем одна над другой. Единственный способ решить проблему в этом случае – развести поверхности на достаточное расстояние, чтобы было понятно, какая ближе к камере, а какая дальше. Но если две поверхности расположены точно одна над другой, то порядок визуализации детерминирован и коллизий не возникает. В этом случае поверхность, визуализируемая второй, всегда оказывается сверху.



Рис. 16.8. *Из-за порога близости часть дома отсечена камерой*



Рис. 16.9. *Артефакты феномена коллизии в Z-буфере*

Для элемента `Camera` в листинге 16.2 свойство `LookDirection` равно `<-1,-1,-1>`:

```
<Viewport3D.Camera>
  <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
</Viewport3D.Camera>
```

Координаты *x*, *y*, *z* этого вектора означают, что камера должна смотреть вниз, на северо-запад, как показано на рис. 16.10.

Выше уже отмечалось, что если перенести камеру в точку `(-5,5,5)`, то дом станет не виден. На рис. 16.11 показано почему. Само перемещение камеры не изменяет свойства `LookDirection`, так что в новом положении камера больше не направлена на дом.

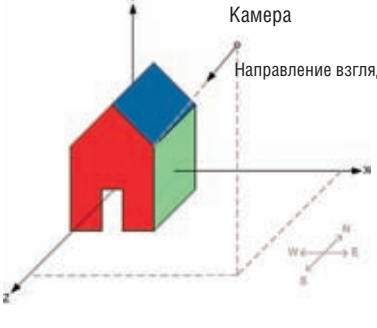


Рис. 16.10. *Камера смотрит вниз на северо-запад*

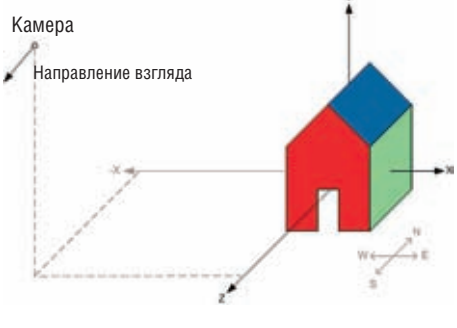


Рис. 16.11. *Перемещение камеры не изменяет направления взгляда*

Простой способ вычислить нужное значение `LookDirection` состоит в том, чтобы найти в пространстве какую-нибудь точку, которую вы хотите видеть, и вычесть ее из свойства `Position` объекта `Camera`. В нашем случае модель дома расположена вокруг начала координат `(0,0,0)`. Вычитание точки `(-5,5,5)` из `(0,0,0)` дает вектор `<5,-5,-5>`, как показано на рис. 16.12.

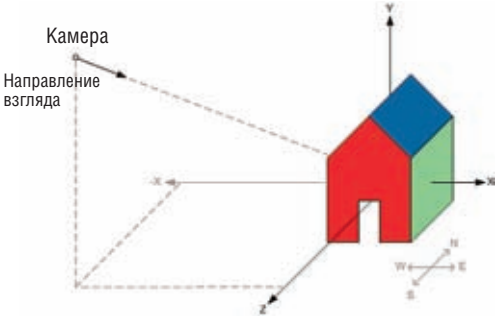


***Рис. 16.12.** Новое направления взгляда*

При таком значении `LookDirection` генерируется изображение, показанное на рис. 16.13:

```
<Viewport3D.Camera>
  <OrthographicCamera Position="-5,5,5" LookDirection="5,-5,-5" Width="5"/>
</Viewport3D.Camera>
```



***Рис. 16.13.** Вид дома с другой стороны*

СОВЕТ

Для методов, которые принимают объект `Vector3D` для задания направления, важно только направление вектора, но не его длина. Если задать в качестве `LookDirection` вектор `<1,-1,-1>`, то получится точно такое же изображение, как на рис. 16.13. Если для внутренних вычислений длину вектора необходимо нормализовать, WPF сделает это автоматически.

В общем случае вы должны позаботиться только о том, чтобы `Vector3D` определял какое-нибудь направление (то есть был отличен от нулевого вектора `<0,0,0>`), если только не собираетесь использовать его для вычисления точек `Point3D`. Если вектор `Vector3D` складывается с точкой `Point3D`, то в результате получается новая точка `Point3D`, и тогда длина вектора (`Length`) определяет, как далеко новая точка отстоит от исходной. Следует также иметь в виду, что длины векторов определяют направление при выполнении линейной интерполяции. Точнее, класс `Vector3DAnimation` не нормализует векторы до начала анимации.

Если камера перемещается часто, то имеет смысл написать коротенький служебный метод, который будет вычислять новое значение `LookDirection` в зависимости от позиции камеры и точки, на которую она должна быть направлена:

```
private void LookAt(ProjectionCamera camera, Point3D lookAtPoint)
{
  camera.LookDirection = lookAtPoint - camera.Position;
}
```

Свойство UpDirection

Свойство `LookDirection` определяет, в каком направлении камера смотрит, но его одного для полной установки ориентации камеры недостаточно. При фиксированном направлении взгляда камеру тем не менее можно поворачивать в пространстве, как показано на рис. 16.14. Именно так вы поступаете с настоящей камерой при переходе от ландшафтной съемки к портретной. Чтобы устранить эту последнюю неоднозначность ориентации, существует свойство `UpDirection` (направление вверх).

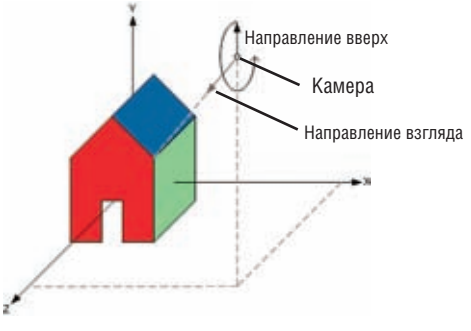


***Рис. 16.14.** Свойство UpDirection*

По умолчанию UpDirection равно <0,1,0>. Задавая другое направление, например <1,0,0>, вы, по существу, поворачиваете камеру на бок. На рис. 16.15 показано изображение, полученное при таком значении UpDirection.

```
<Viewport3D.Camera>
  <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1"
    UpDirection="1,0,0" Width="5"/>
</Viewport3D.Camera>
```



Рис. 16.15. *Задание положительного направления оси x в виде свойства UpDirection*

В этом разделе мы манипулировали обозревающей сцену камерой с помощью свойств Position, UpDirection и LookDirection. Часто это самый удобный способ расположить на сцене статическую камеру, но во многих ситуациях, когда требуется перемещать или поворачивать камеру, проще применить свойство Camera.Transform.

СОВЕТ

Свойство Camera.Transform особенно полезно, когда требуется, чтобы камера следовала за объектом, перемещающимся по сцене, поскольку в этом случае к объекту и к камере достаточно применить одно и то же преобразование Transform3D.

Основное преимущество свойства Camera.Transform заключается в том, что оно позволяет позиционировать и анимировать камеру так же, как все остальные 3D-объекты на сцене. Помните об этом, когда ниже мы будем обсуждать класс Transform3D.

ПРЕДУПРЕЖДЕНИЕ

Не забывайте преобразовывать UpDirection!

Если во время вращения камеры вокруг объекта изображение после прохождения определенной точки внезапно переворачивается на 180˚, значит, вы, скорее всего, забыли изменить свойство `UpDirection`. Проблема возникает, когда направление `LookDirection` совмещается с направлением `UpDirection`, как показано на рис. 16.16.

Когда камера приближается к дому, направление взгляда `LookDirection` корректируется так, чтобы камера смотрела вниз. После того как камера пройдет крышу, вы ожидаете, что она будет обводить дальнюю сторону дома снизу вверх. Но поскольку `UpDirection` по-прежнему указывает на положительное направление оси *y*, то камера вместо этого проворачивается на месте как раз в момент прохождения над коньком крыши. Хуже того, в момент, когда ваш взгляд находится точно над коньком, `LookDirection` и `UpDirection` направлены вдоль одной прямой и результат вообще не определен. При таком перемещении камеры следует поворачивать одновременно `UpDirection` и `LookDirection`, как показано на рис. 16.17.

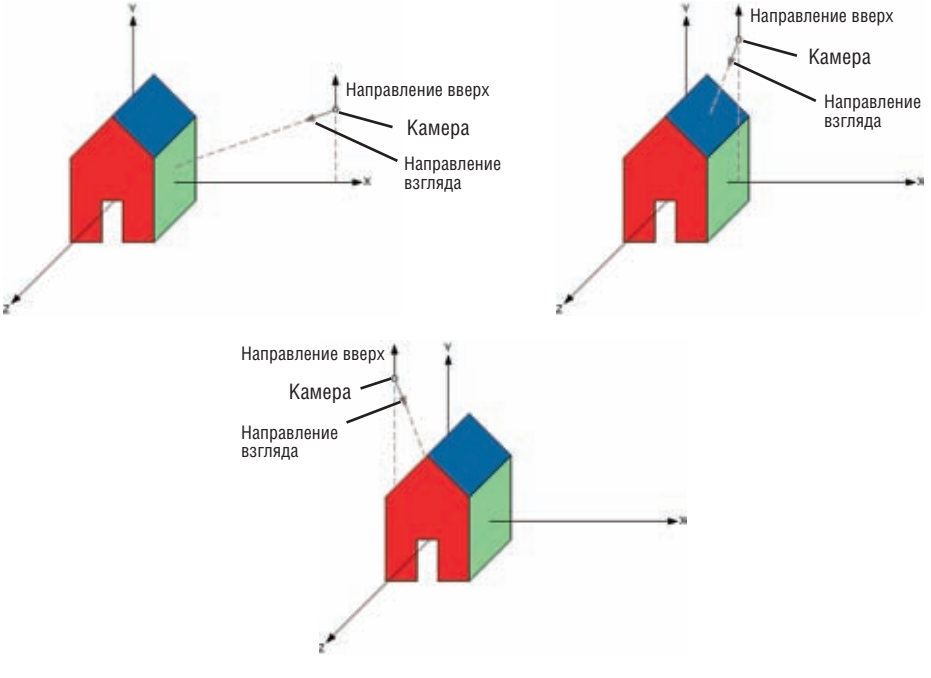


Рис. 16.16. *Неправильный обвод дома камерой*

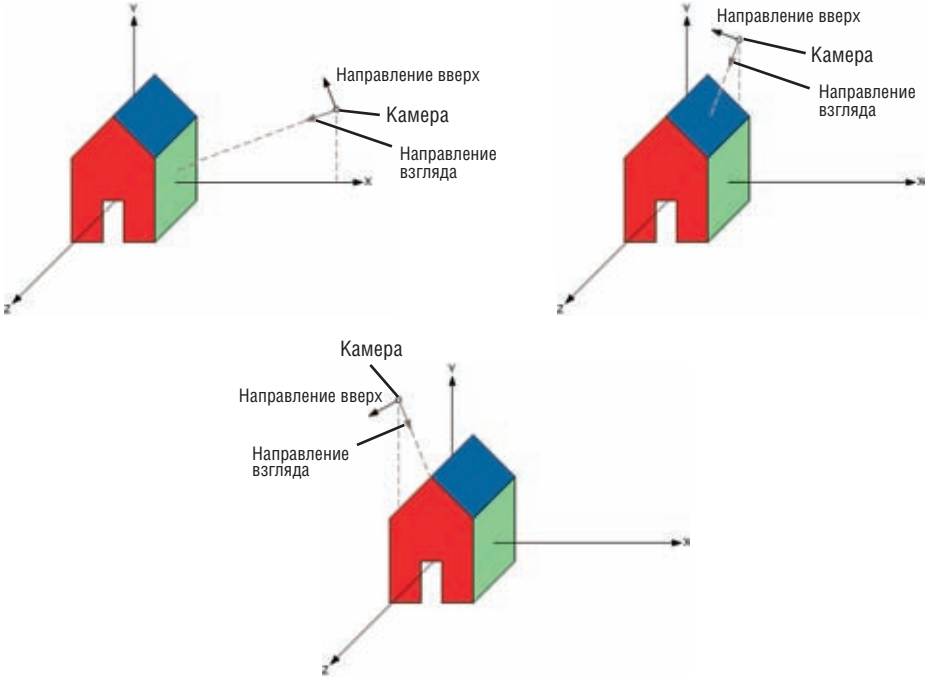


Рис. 16.17. *Направление вверх корректируется по мере обвода дома камерой*

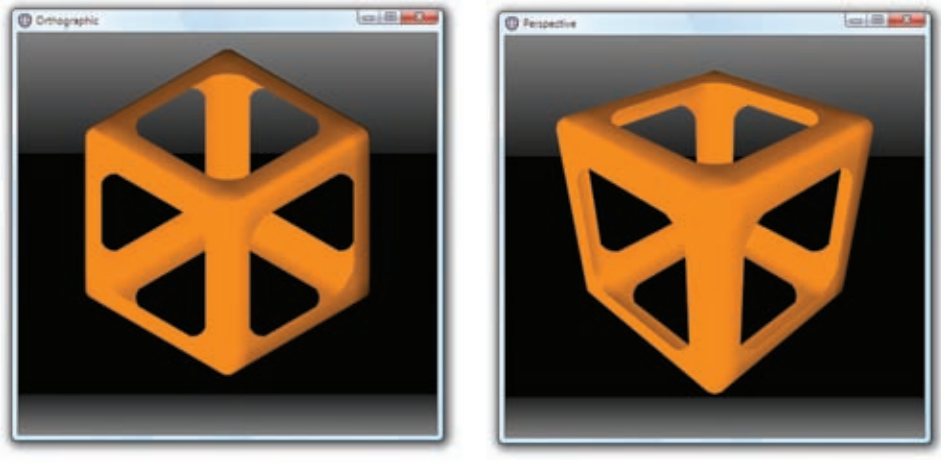
Классы OrthographicCamera и PerspectiveCamera

В WPF есть два типа камер, достаточных для большинства приложений. Перспективная камера `PerspectiveCamera` создает реалистичное изображение, на котором объект, отстоящий от камеры дальше, кажется меньше, чем такой же объект, расположенный ближе. Именно так человек воспринимает реальный мир. Другой тип камеры – ортографическая, `OrthographicCamera`, – более полезен для инструментов редактирования и некоторых визуализаций, поскольку видимый размер объекта не зависит от расстояния до камеры, что упрощает точные измерения и анализ. Технические и технологические чертежи часто выполняются в ортографической проекции. На рис. 16.18 показана одна и та же модель, визуализированная ортографической и перспективной камерами.

Принцип работы всех камер одинаков – спроецировать находящиеся на сцене 3D-модели на плоскость, получив тем самым изображение, которое затем показывается пользователю. В случае ортографической камеры каждая точка на плоскости проекции показывает, что находится в точности за ней, как видно на рис. 16.19. Это позволяет рассматривать участок пространства в форме

прямоугольной призмы. Ширина видимой области определяется свойством `OrthographicCamera.Width`. Высота вычисляется автоматически на основе ограничивающего прямоугольника `Viewport3D` так, чтобы сохранить пропорции 1:1. Вот пример использования ортографической камеры:

```
<Viewport3D.Camera>
  <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
</Viewport3D.Camera>
```



Пример ортографической камеры Пример перспективной камеры

Рис. 16.18. *Примеры ортографической и перспективной камер*

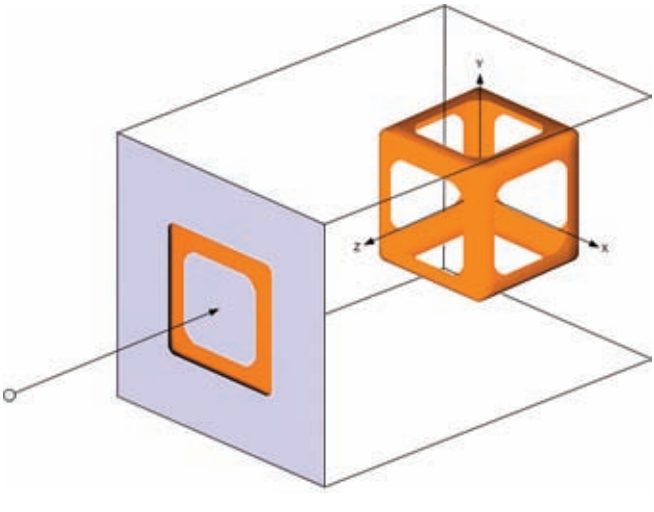


Рис. 16.19. *Ортографическая проекция*

В случае перспективной камеры ширина обозреваемой области непостоянна. Чем дальше расстояние от камеры, тем большая часть трехмерного мира в нее попадает. Это позволяет снять область сцены в виде усеченной пирамиды, как показано на рис. 16.20. Поскольку видимая область расширяется по мере удаления от камеры, далеко отстоящие объекты в перспективной проекции кажутся меньше. Степень расширения можно контролировать с помощью свойства `FieldOfView`. В WPF это свойство задает угол расширения сектора обзора по горизонтали. Вот пример использования перспективной камеры:

```
<Viewport3D.Camera>
  <PerspectiveCamera Position="5,5,5" LookDirection="-1,-1,-1" FieldOfView="45""/>
</Viewport3D.Camera>
```

Свойство `FieldOfView` можно сравнить с объективом с переменным фокусным расстоянием в настоящей камере. В ортографической камере его аналогом является свойство `Width`. Чем ниже значения `Width` и `FieldOfView`, тем меньшая часть 3D-объекта попадает в кадр. Чем выше значения `Width` и `FieldOfView`, тем большая часть сцены видна.

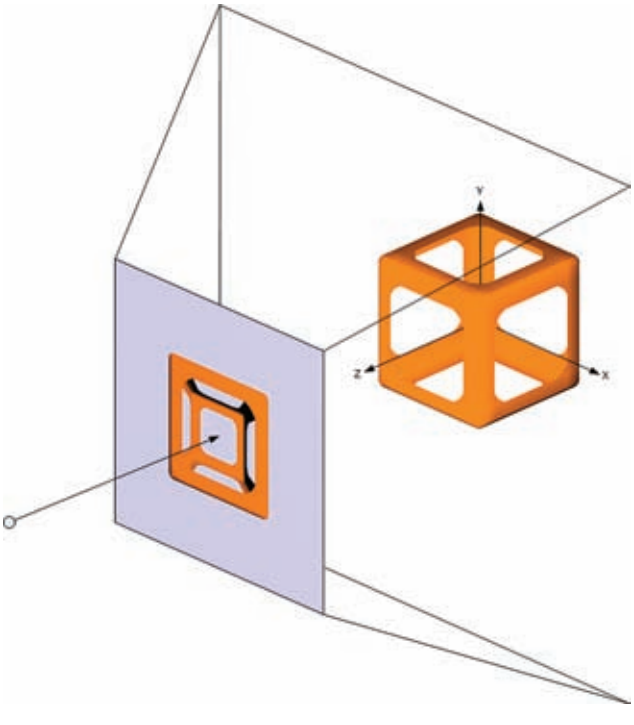


Рис. 16.20. *Перспективная проекция*

КОПНЕМ ГЛУБЖЕ

Класс MatrixCamera

WPF поддерживает еще и третий тип камеры, `MatrixCamera`, которая позволяет задавать преобразования представления и проекции в виде объектов `Matrix3D`. Математика проективных преобразований – очень увлекательная тема, но она выходит за рамки данной главы.

Класс `MatrixCamera` помогает переносить код, написанный для других платформ, например Direct3D. Опытный пользователь может с помощью класса `MatrixCamera` создавать камеры, которые WPF напрямую не поддерживает, например с сектором обзора в виде усеченной пирамиды.

Структура матриц, применяемых в классе `MatrixCamera`, такая же как в Direct3D. Это позволяет без особого труда переносить код методов для конструирования матриц представления и проекции из таких библиотек, как D3DX. Эти матрицы хорошо документированы в DirectX SDK.

Класс Transform3D

Как и класс `Transform` в двумерной графике, класс `Transform3D` позволяет позиционировать, поворачивать и изменять размеры 3D-объектов. Преобразования `Transform3D` можно применять к объектам `Model3D`, `ModelVisual3D` и `Camera`. Для этого следует установить их свойство `Transform`. При определении свойства `Transform` 3D-объекта вы устанавливаете соответствие между системой координат объекта и новой системой координат. По существу, то же самое происходит, когда вы позиционируете элемент на двумерной плоскости с помощью его свойств `Canvas.Left` и `Canvas.Top`.

На рис. 16.21 показано двумерное изображение привидения из главы 15. Все команды рисования привидения заданы в локальной системе координат привидения. С помощью преобразования `TranslateTransform` можно сдвинуть систему координат привидения, так что ее начало больше не будет совпадать с началом системы координат контейнера. Это показано в правой части рис. 16.21.

Преобразование `TranslateTransform` перемещает привидение и все его дочерние объекты `Visual` по экрану, но само привидение на это никак не реагирует. Ни одна из команд его рисования не модифицируется, меняется лишь точка отсчета. Именно так панель `Canvas` и перемещает расположенные на ней элементы – конструируя преобразование `TranslateTransform` и применяя его к объектам `Visual`.

Все то же самое относится и к трехмерным преобразованиям. В 3D-графике имеется мировая система координат верхнего уровня. Для того чтобы изменять положение, размер и ориентацию 3D-объектов в мировой системе координат, имеется пять подклассов класса `Transform3D`:

- `TranslateTransform3D` – параллельный перенос 3D-объекта относительно его контейнера.



Системы координат привидения и контейнера совпадают

Система координат привидения сдвинута относительно системы координат контейнера

Рис. 16.21. *Системы координат привидения и контейнера*

- `ScaleTransform3D` – масштабирование 3D-объекта относительно его контейнера.
- `RotateTransform3D` – поворот 3D-объекта относительно его контейнера.
- `MatrixTransform3D` – преобразование 3D-объекта, описываемое матрицей `Matrix3D`.
- `Transform3DGroup` – содержит коллекцию объектов `Transform3D`. Объект `Transform3DGroup` сам является преобразованием `Transform3D` и служит для применения нескольких преобразований к одному 3D-объекту.

В этом разделе мы применим эти преобразования к модели домика, приведенной в начале главы. В листинге 16.3 содержится та же XAML-разметка, что и раньше, с двумя выделенными изменениями. Во-первых, мы добавили преобразование (пока что тождественное, которое ничего не делает), а во-вторых, увеличили свойство `Width` камеры, чтобы был виден эффект применения различных преобразований.

Листинг 16.3. Модифицированная модель домика

```
<Page Background="Black"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Viewport3D>
  <Viewport3D.Camera>
    <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="10"/>
  </Viewport3D.Camera>
  <Viewport3D.Children>
    <ModelVisual3D x:Name="Light">
    <ModelVisual3D.Content>
      <AmbientLight/>
    </ModelVisual3D.Content>
    </ModelVisual3D>
    <ModelVisual3D>
    <ModelVisual3D.Transform>
```

```
      <x:Static Member="Transform3D.Identity"/>
    </ModelVisual3D.Transform>
    <ModelVisual3D.Content>
     <Model3DGroup x:Name="House">

      <GeometryModel3D x:Name="Roof">
      <GeometryModel3D.Material>
        <DiffuseMaterial Brush="Blue"/>
      </GeometryModel3D.Material>
      <GeometryModel3D.Geometry>
        <MeshGeometry3D Positions="-1,1,1 0,2,1 0,2,-1 -1,1,
                                   -1 0,2,1 1,1,1 1,1,-1 0,2,-1"
                        TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7"/>
      </GeometryModel3D.Geometry>
      </GeometryModel3D>

      <GeometryModel3D x:Name="Sides">
      <GeometryModel3D.Material>
        <DiffuseMaterial Brush="Green"/>
      </GeometryModel3D.Material>
      <GeometryModel3D.Geometry>
        <MeshGeometry3D Positions="-1,1,1 -1,1,-1 -1,-1,
                                   -1 -1,-1,1 1,1,-1 1,1,1 1,-1,1 1,-1,-1"
                        TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7"/>
      </GeometryModel3D.Geometry>
      </GeometryModel3D>

      <GeometryModel3D x:Name="Ends">
      <GeometryModel3D.Material>
        <DiffuseMaterial Brush="Red"/>
      </GeometryModel3D.Material>
      <GeometryModel3D.Geometry>
        <MeshGeometry3D
          Positions="-0.25,0,1 -1,1,1 -1,-1,1 -0.25,-1,
                     1 -0.25,0,1 -1,-1,1 0.25,0,1 1,-1,1 1,1,
                     1 0.25,0,1 0.25,-1,1 1,-1,1 1,1,1 0,2,1 -1,
                     1,1 -1,1,1 -0.25,0,1 0.25,0,1 1,1,1 1,1,
                    -1 1,-1,-1 -1,-1,-1 -1,1,-1 1,1,-1 -1,1,
                    -1 0,2,-1"
          TriangleIndices="0 1 2 3 4 5 6 7 8 9 10 11 12 13
                           14 15 16 17 15 17 18 19 20 21 19 21 22 23 24 25"/>
      </GeometryModel3D.Geometry>
      </GeometryModel3D>
     </Model3DGroup>
    </ModelVisual3D.Content>
    </ModelVisual3D>
  </Viewport3D.Children>
  </Viewport3D>
</Page>
```

Преобразование TranslateTransform3D

Преобразование `TranslateTransform3D` сдвигает объект на заданную величину относительно контейнера. Смещение определяется свойствами `OffsetX`, `OffsetY` и `OffsetZ`. Например, если задать `OffsetZ = 3`, то домик сместится вперед по оси *z* на три единицы, как показано на рис. 16.22:

```
<ModelVisual3D.Transform>
  <TranslateTransform3D OffsetZ="3"/>
</ModelVisual3D.Transform>
```

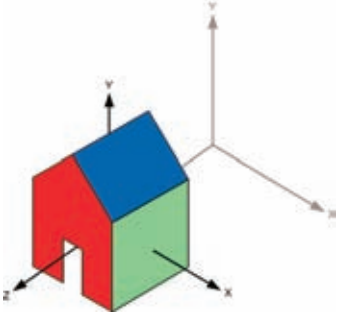


Рис. 16.22. *Параллельный перенос домика вперед вдоль оси z на три единицы*

Отметим, что позиционировать 3D-объекты можно и проще – конструируя модели так, чтобы начало координат находилось в удобной точке. Например, сейчас начало координат находится примерно в центре домика. Чтобы переместить центр домика в точку (3,2,1), можно было бы задать такое преобразование переноса:

```
<ModelVisual3D.Transform>
  <TranslateTransform3D OffsetX="3" OffsetY="2" OffsetZ="1"/>
</ModelVisual3D.Transform>
```

Преобразование ScaleTransform3D

Преобразования `ScaleTransform3D` применяются, когда нужно изменить размеры 3D-объектов. Коэффициенты масштабирования по каждой оси задаются с помощью свойств `ScaleX`, `ScaleY` и `ScaleZ`. Поскольку коэффициенты по разным осям могут быть различными, то преобразование `ScaleTransform3D` позволяет растягивать объекты. Например, следующее преобразование удваивает ширину домика по оси *x* (рис. 16.23):

```
<ModelVisual3D.Transform>
  <ScaleTransform3D ScaleX="2"/>
</ModelVisual3D.Transform>
```

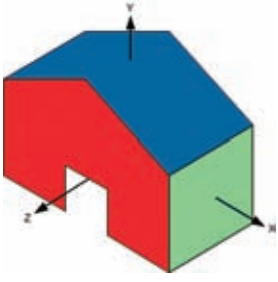


Рис. 16.23. *Масштабирование домика вдоль оси x*

Чтобы изменить размер 3D-объекта с сохранением пропорций, задавайте `ScaleX`, `ScaleY` и `ScaleZ` одинаковыми. Это называется *равномерным масштабированием*. Если коэффициент равномерного масштабирования равен 2, то размер объект удваивается, а если 0.5 – то уменьшается в два раза.

ПРЕДУПРЕЖДЕНИЕ

Чтобы сохранить исходный размер, задавайте коэффициент 1, а не 0!

Чтобы сохранить исходный размер объекта, его следует масштабировать в отношении 1:1, а не 1:0. Если задать `ScaleX`, `ScaleY` или `ScaleZ`, равным `0`, то объект просто сплющится в одном или нескольких направлениях. Сплющивание в одном направлении иногда бывает полезно – например, чтобы превратить шар в диск. Но сплющивание в двух направлениях превращает трехмерный объект в невидимую линию, а во всех трех – в невидимую точку!

Применяя масштабирование, вы растягиваете или сжимаете пространство. В результате смещаются все точки, кроме центра масштабирования. По умолчанию этот центр совпадает с началом координат. На рис. 16.23 домик остался на месте, потому что его центр как раз и находился в начале координат. Если бы мы перенесли домик так, чтобы его центр оказался в точке (3,0,0), а затем подвергли масштабированию с коэффициентом 0.5, то центр домика переместился бы в точку (1.5,0,0), поскольку пространство сжалось в направлении к началу координат. Это показано в приведенном ниже XAML-коде и на рис. 16.24:

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <TranslateTransform3D OffsetX="3"/>
    <ScaleTransform3D ScaleX="0.5" ScaleY="0.5" ScaleZ="0.5"/>
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

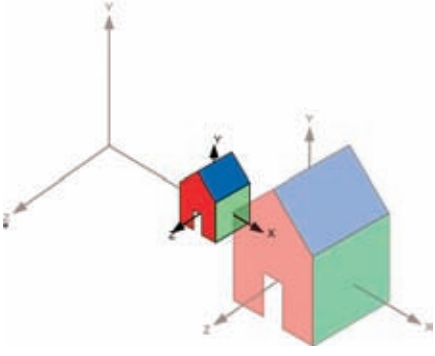


Рис. 16.24. *Домик перемещается, когда пространство сжимается в направлении начала координат*

Один из способов предотвратить смещение домика в результате масштабирования состоит в том, чтобы задать в качестве центра масштабирования другую точку. Для этого служат свойства `CenterX`, `CenterY` и `CenterZ`. В следующем XAML-коде мы располагаем центр масштабирования в новом центре домика.

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <TranslateTransform3D OffsetX="3"/>
    <ScaleTransform3D ScaleX="0.5" ScaleY="0.5" ScaleZ="0.5" CenterX="3"/>
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

В результате домик сжимается «на месте», как показано на рис. 16.25.

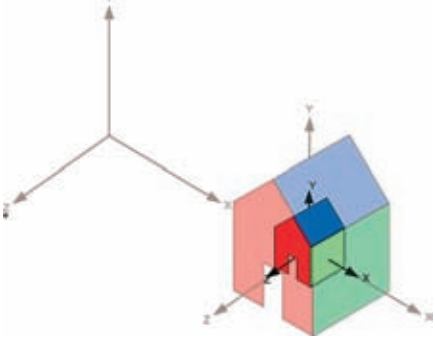


Рис. 16.25. *Центр масштабирования совмещен с центром домика*

Другой способ оставить домик на месте – поменять местами преобразования параллельного переноса и масштабирования:

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <ScaleTransform3D ScaleX="0.5" ScaleY="0.5" ScaleZ="0.5"/>
    <TranslateTransform3D OffsetX="3"/>
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

Если сначала выполнить масштабирование, а потом параллельный перенос, то масштабирование не окажет влияния на величину смещения при переносе, поскольку домик сжимается, все еще находясь в начале координат. А после того как он уменьшен до нужного размера, мы переносим его на три единицы вдоль оси *x*.

Глядя на рис. 16.24, вы, возможно, обратили внимание, что с приближением коэффициента масштабирования к нулю домик сдвигается к центру масштабирования. Возникает вопрос, что произойдет при переходе коэффициента масштабирования через нуль в область отрицательных чисел. Это приведет к зеркальному отражению объекта. На рис. 16.26 показано, что при отрицательном коэффициенте `ScaleZ` домик зеркально отразился относительно плоскости *xy*:

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <TranslateTransform3D OffsetZ="3"/>
    <ScaleTransform3D ScaleZ="-1"/>
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

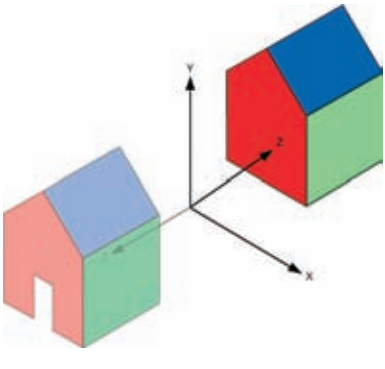


Рис. 16.26. *Зеркальное отражение вдоль оси z*

Отметим, что при отражении изменяется направление оси *z*. Если после такого масштабирования применить параллельный перенос, то свойство `OffsetZ` будет перемещать объект в противоположном направлении.

КОПНЕМ ГЛУБЖЕ

Масштабирование вдоль неглавной оси

Внутренний алгоритм реализации свойств `CenterX`, `CenterY` и `CenterZ` сначала параллельно переносит объект так, чтобы указанная точка оказалась в начале координат. Затем применяется масштабирование и объект параллельно переносится обратно, так чтобы центр масштабирования оказался в исходной точке.

Аналогичную технику можно применять для масштабирования объекта в направлении, отличном от осей x, y, z. Сначала с помощью преобразования `RotationTransform3D` поворачиваем объект так, чтобы нужное направление масштабирования совпало с одной из главных осей. Затем выполняем масштабирование и применяем обратный поворот, чтобы масштабированный объект оказался ориентирован в пространстве так же, как и раньше.

Преобразование RotateTransform3D

Преобразования `RotateTransform3D` применяются для поворота 3D-объектов в пространстве. Сам поворот описывается объектом класса `Rotation3D`. Это абстрактный класс с двумя конкретными реализациями:

- `AxisAngleRotation3D` – поворот вокруг заданной оси `Axis` на угол `Angle`, выраженный в градусах. Обычно это самый удобный и понятный человеку способ описания поворотов в трехмерном пространстве.
- `QuaternionRotation3D` – поворот задается в виде объекта `Quaternion`. Кватернионы представляют собой хитроумный способ кодирования оси и угла поворота и обладают рядом удобных свойств, что делает их весьма популярным средством в системах трехмерной графики.

FAQ

Почему WPF не ограничивается каким-то одним способом задания поворотов?

В первых версиях WPF поддерживались только кватернионы, но этот способ оказался трудным для разработчиков, впервые переходящих от двумерной к трехмерной графике. Типичной ошибкой было создание поворота от 0 до 360°, в результате чего во время анимации типа `Rotation3DAnimation` не происходило вообще никакого перемещения, так как начальная и конечная ориентации объекта совпадали. Для поворота объекта на угол, больший 179,9999...°, требовалось либо создавать кумулятивные анимации, либо использовать несколько опорных кадров.

Позже была добавлена возможность задания поворота с помощью пары свойств `Axis/Angle`, чтобы облегчить новичкам в 3D-графике описание тривиальной анимации с вращением на месте. Для создания такого вращения достаточно анимировать свойство `Angle` с помощью класса `DoubleAnimation`. Однако поддержка

класса Quaternion оставлена для тех, кто пишет программы экспорта данных для пакетов моделирования, в которых повороты чаще всего представляются в виде кватернионов.

Чтобы поддержать анимированный переход от одной компоновки к другой, когда одна конфигурация описывается с помощью свойств Axis/Angle, а другая – посредством кватернионов, классы AxisAngleRotation3D и QuaternionRotation3D наследуют общему базовому классу Rotation3D. Переход между любыми двумя углами поворота Rotation3D можно анимировать с помощью класса Rotation3DAnimation, который всегда выбирает кратчайший путь между двумя ориентациями.

Чего WPF напрямую не поддерживает, так это описания поворота с помощью углов Эйлера. В этой схеме поворот объекта описывается углами поворота относительно трех осей. Что это за оси, в каком порядке производятся повороты и в каком направлении, не стандартизовано.

В WPF нет класса EulerAngleRotation3D, но вы можете сконструировать эквивалентное ему преобразование Transform3D, поместив в группу Transform3DGroup три преобразования RotateTransform3D, как показано в следующем XAML-коде:

```
<Transform3DGroup>
  <RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="RotateX" Axis="1,0,0" Angle="0"/>
  </RotateTransform3D.Rotation>
  </RotateTransform3D>
  <RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="RotateY" Axis="0,1,0" Angle="0"/>
  </RotateTransform3D.Rotation>
  </RotateTransform3D>
  <RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="RotateZ" Axis="0,0,1" Angle="0"/>
  </RotateTransform3D.Rotation>
  </RotateTransform3D>
</Transform3DGroup>
```

Отметим, что для получения желаемого результата оси, возможно, придется подправить.

На рис. 16.27 показан результат поворота домика на 45° вокруг оси *y*:

```
<ModelVisual3D.Transform>
  <RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D Axis="0,1,0" Angle="45"/>
  </RotateTransform3D.Rotation>
  </RotateTransform3D>
</ModelVisual3D.Transform>
```

В правосторонней системе координат поворот на положительный угол означает вращение против часовой стрелки.

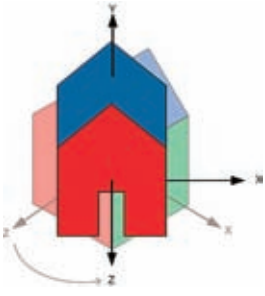


Рис. 16.27. *Поворот на 45° вокруг оси y*

Отметим, что после поворота оси *x* и *z* направлены иначе. Если бы до поворота мы применили параллельный перенос, как в коде ниже, то наблюдалось бы такое же поведение, как в случае преобразования `ScaleTransform3D`:

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <TranslateTransform3D OffsetZ="3"/>
    <RotateTransform3D>
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D Axis="0,1,0" Angle="45"/>
    </RotateTransform3D.Rotation>
    </RotateTransform3D>
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

Преобразование поворота приводит к перемещению пространства вокруг некоторой точки. По умолчанию эта точка – начало координат. Если центр модели не совпадает с центром поворота, то модель сместится, как показано на рис. 16.28.

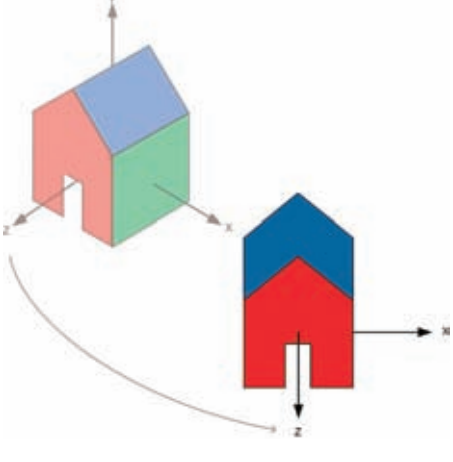


Рис. 16.28. *Побочный эффект поворота*

Как и раньше, если требуется повернуть домик «на месте», то один из вариантов – изменить центр поворота с помощью свойств CenterX, CenterY и CenterZ:

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <TranslateTransform3D OffsetZ="3"/>
    <RotateTransform3D CenterZ="3">
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D Axis="0,1,0" Angle="45"/>
    </RotateTransform3D.Rotation>
    </RotateTransform3D>
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

Другой способ – поменять местами преобразования поворота и параллельного переноса. Если выполнить параллельный перенос после поворота, то поворот не окажет влияния на вектор смещения.

```
<ModelVisual3D.Transform>
  <Transform3DGroup>
    <RotateTransform3D>
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D Axis="0,1,0" Angle="45"/>
    </RotateTransform3D.Rotation>
    </RotateTransform3D>
    <TranslateTransform3D OffsetZ="3"/>
  </Transform3DGroup>
</ModelVisual3D.Transform>
```

Комбинирование преобразований Transform3D

В отличие от двумерной графики, где чаще всего встречается простейший параллельный перенос, в 3D-графике обычно применяется комбинация трех преобразований: масштабирование, поворот и перенос (как правило, именно в таком порядке). Для применения нескольких преобразований предназначен класс Transform3DGroup. В следующем XAML-коде показано типичное использование элемента Transform3DGroup:

```
<Transform3DGroup>
  <ScaleTransform3D x:Name="Size" ScaleX="1" ScaleY="1" ScaleZ="1"/>
  <RotateTransform3D>
  <RotateTransform3D.Rotation>
    <AxisAngleRotation3D x:Name="Orientation" Axis="0,1,0" Angle="0"/>
  </RotateTransform3D.Rotation>
  </RotateTransform3D>
  <TranslateTransform3D x:Name="Position" OffsetX="0" OffsetY="0" OffsetZ="0"/>
</Transform3DGroup>
```

КОПНЕМ ГЛУБЖЕ

Класс MatrixTransform3D

WPF поддерживает еще и пятый тип преобразований `Transform3D`: `MatrixTransform3D`. Этот класс позволяет описать произвольное проективное преобразование трехмерного пространства в виде объекта `Matrix3D`. Говоря *«проективное»*, мы здесь имеем в виду, что `Matrix3D` – полная матрица 4×4. Это вовсе не означает, что вы обязаны задавать матрицу проекций, как для камеры, хотя это и возможно. Объекты `MatrixTransform3D` полезны для определения преобразований, которые нельзя выразить в виде других подклассов `Transform3D`, и для переноса кода, в котором преобразования представляются в виде матриц.

Стоит отметить, что для любого объекта `Transform3D` можно получить соответствующий ему объект `Matrix3D` с помощью свойства `Value`. Поскольку это относится и к объектам `Transform3DGroup`, то получается, что любую последовательность преобразований `Transform3D` можно свернуть в одно преобразование `MatrixTransform3D`.

Класс Model3D

Объекты `Model3D` – это те кирпичики, из которых строится трехмерная модель сцены. Класс `Model3D` – аналог класса `Drawing` в двумерной графике. Но если в двумерной графике класс `Drawing` вместе со своими подклассами – лишь один из многих способов добавить графическое содержимое в WPF-приложение, то класс `Model3D` – единственный способ описать 3D-содержимое в WPF.

В состав WPF входят три подкласса `Model3D`:

- `Light` – имеет несколько подклассов, вводящих в сцену свет. Часто забывают о том, что источники света `Light` – на самом деле объекты `Model3D`, а это очень удобно, например, чтобы снабдить автомобиль фарами с помощью класса `Model3DGroup`.

СОВЕТ

Вводить XAML-код вручную очень полезно в учебных целях и даже для создания простеньких моделей, но как долговременная стратегия создания 3D-моделей этот подход не выдерживает критики.

Так же как большинство растровых изображений создается в графических редакторах, трехмерные модели обычно конструируются в специальных программах моделирования. А те, что не созданы в такой программе, обычно генерируются процедурно.

Когда необходимы формы более сложные, чем плоскость или куб, следует обратиться к программе 3D-моделирования, имеющей средства экспорта в формате XAML. Для большинства популярных программ имеются многочисленные сторонние модули экспорта, в том числе бесплатные. Существуют также программы 3D-моделирования, например ZAM 3D от компании Electric Rain, которые явно ориентированы на WPF и имеют встроенную поддержку XAML.

- `GeometryModel3D` – визуализирует поверхность (описанную в виде объекта `Geometry3D`) с заданным материалом `Material`. Класс `GeometryModel3D` аналогичен классу `GeometryDrawing` в двумерной графике.
- `Model3DGroup` – содержит коллекцию объектов `Model3D`. Объект `Model3DGroup` сам является частным случаем `Model3D` и потому часто применяется для группировки нескольких объектов `GeometryModel3D` и `Light` в единую 3D-модель.

Все эти классы уже встречались в листингах 16.2 и 16.3, описывающих простой домик.

Класс Light

Освещение – концепция, которая в WPF свойственна только трехмерной графике. В двумерной графике цвета, появляющиеся на экране, обычно поступают непосредственно от используемой кисти `Brush` или пера `Pen`. В мире 3D есть дополнительный этап освещения, на котором динамически вычисляются отбрасываемые объектами тени в зависимости от их близости к расположенным на сцене источникам света. Динамическое освещение позволяет гораздо проще создавать и анимировать реалистично выглядящие сцены.

Освещение складывается из трех основных компонентов: объекты `Light`, являющиеся источниками света; объекты `Material`, то есть материалы, по-разному отражающие свет в камеру; и геометрия (`Geometry`) модели, определяющая углы падения и отражения света. В этом разделе мы рассмотрим различные источники освещения, поддерживаемые WPF:

- `DirectionalLight` (направленный источник света) – расположен в бесконечности, освещает сцену параллельными лучами света. Этот класс аппроксимирует удаленный источник света, например Солнце.
- `PointLight` (точечный источник) – излучает свет равномерно во всех направлениях. Яркость света убывает с увеличением расстояния от источника. Класс `PointLight` аппроксимирует нефокусированные источники света, например электрические лампочки.
- `SpotLight` (прожектор) – испускает конус света. Как и в случае `PointLight`, яркость убывает с увеличением расстояния от источника. Класс `SpotLight` аппроксимирует фокусированные источники света, например луч фонаря.
- `AmbientLight` (рассеянный свет) – освещает все поверхности равномерно. Яркий источник рассеянного света создает плоские изображения из-за отсутствия теней. Однако не слишком яркий источник аппроксимирует эффект рассеянного освещения, созданного диффузно отражающими поверхностями на сцене.

Возможно, вы обратили внимание, что в каждом описании употребляется слово *«аппроксимирует»*. Важно понимать, что цель освещения в графических системах реального времени, к каковым относится и WPF, – не в том, чтобы создать точную физическую модель поведения света в реальном мире. Чтобы уложиться в жесткие временные рамки, графические системы применяют хитроумные приемы и грубые оценки. Назовем два общеупотребительных

приближения: поверхности не препятствуют прохождению света (то есть не отбрасывают тени); освещенность вычисляется только в вершинах сетки, а затем интерполируется на всю поверхность. В WPF используются оба эти приближения.

Создание реалистично освещенной сцены – своего рода искусство. Чтобы добиться желаемого эффекта, приходится прибегать к нереалистичным трюкам, например добавлять лишние источники света, включать эффекты освещения в используемые материалы и т. д. Не считайте это чем-то зазорным. Хотя в API для работы с освещением и материалами применяются модели из реального мира, на самом деле это всего лишь инструменты.

Класс DirectionalLight

Класс `DirectionalLight` аппроксимирует источник света, который расположен так далеко, что исходящие от него лучи можно считать параллельными. Например, свет Солнца, достигший Земли. На рис. 16.29 иллюстрируется эффект следующего направленного источника, освещающего сферу:

```
<DirectionalLight Direction="1,-1,-0.5" Color="White"/>
```

Направление света, падающего на сцену, определяется свойством `Direction`. Разумеется, свойство `Transform`, унаследованное от `Model3D`, также влияет на направление света. Цвет источника регулируется свойством `Color`.

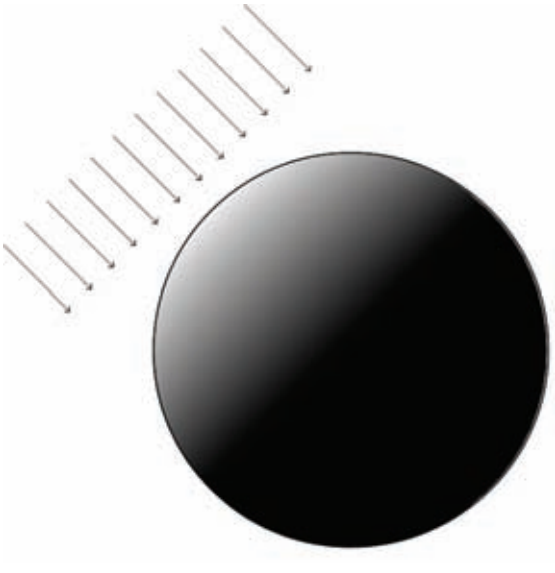


Рис. 16.29. *Источник DirectionalLight, освещающий сферу*

Изображения с одним источником `DirectionalLight` часто получаются неестественными – и не без причины. В реальном мире, даже когда свет падает на

сцену с одного направления (как солнечные лучи), он обычно отражается от различных находящихся на сцене объектов, что создает игру света. Аппроксимировать это можно, в частности, добавив неяркий источник AmbientLight, о котором речь пойдет ниже.

СОВЕТ

С помощью свойства Color можно управлять яркостью источника. Например, #FFFFFF – максимально яркий белый свет. #808080 – белый свет половинной яркости. Альфа-компонента цвета источника ни на что не влияет.

Источники света складываются. Например, два одинаковых (в одной и той же точке Position, одного и того же направления Direction и т. д.) источника света половинной яркости дают тот же эффект, что один источник максимальной яркости.

Класс PointLight

Класс PointLight аппроксимирует источник, равномерно излучающий свет во всех направлениях из одной точки, как, например, ничем не прикрытая электрическая лампочка. В отличие от DirectionalLight, яркость света, излучаемого источником PointLight, уменьшается с увеличением расстояния от него. На рис. 16.30 изображен эффект освещения сферы близко расположенным источником PointLight, который описывается следующим образом:

```
<PointLight Color="White" Position="2,2,2"
  ConstantAttenuation="0"
  LinearAttenuation="0"
  QuadraticAttenuation="0.125"/>
```

Местоположение источника PointLight описывается его свойством Position. Уменьшение яркости света с возрастанием расстояния описывается следующей формулой, включающей свойства ConstantAttenuation (постоянное затухание), LinearAttenuation (линейное затухание) и QuadraticAttenuation (квадратичное затухание):

$$Затухание = \frac{1}{\max(1, C + Ld + Qd^2)}$$

где C, L, Q – соответственно ConstantAttenuation, LinearAttenuation и QuadraticAttenuation, а d – расстояние между источником света (его свойством Position) и освещаемой точкой. Из этой формулы можно извлечь кое-какую полезную информацию. Например, если $C=1$, $L=0$, $Q=0$, то получается точечный источник света с постоянной яркостью, не зависящей от расстояния. Однако обычно значения этих свойств устанавливаются методом проб и ошибок.

В классе PointLight есть также свойство Range, задающее радиус области, вне которой точечный источник сразу и резко перестает быть виден. Свойство Range не связано со свойствами затухания, поскольку оно не влияет на яркость

света вне области отсечения. По умолчанию значением Range является положительная бесконечность.

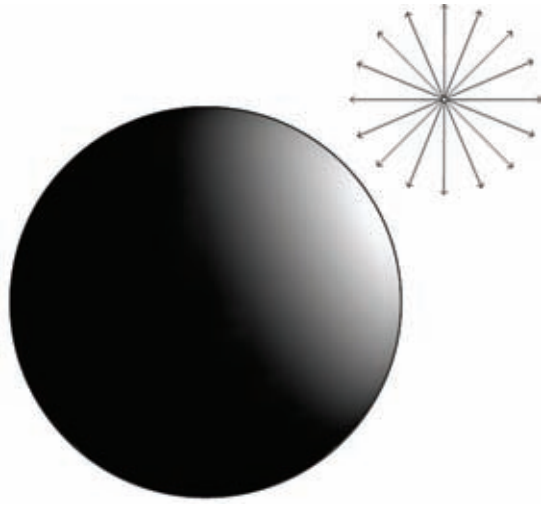


Рис. 16.30. *Источник PointLight, освещающий сферу*

Класс SpotLight

SpotLight – этот точечный источник (PointLight), сфокусированный в луч. В реальном мире для фокусировки применяются линзы и отражатели. В современной компьютерной графике аппроксимация достигается путем ограничения излучаемого света конусом. На рис. 16.31 показано, что SpotLight – это действительно PointLight, лучи от которого распространяются только внутри телесного угла.

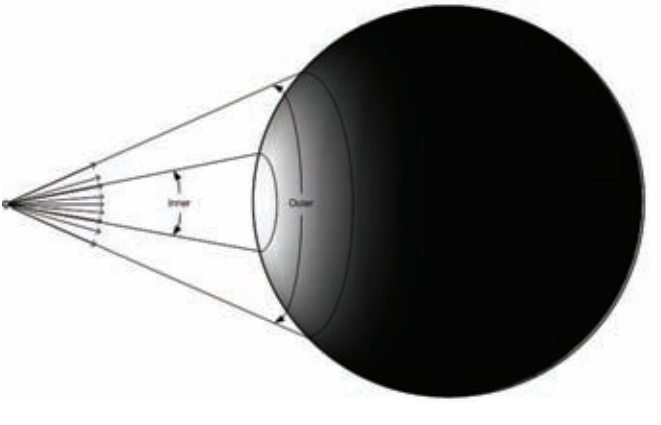


Рис. 16.31. *Источник SpotLight, освещающий сферу*

FAQ

Почему созданный мною источник SpotLight или PointLight не освещает мою модель?

В начале этого раздела было сказано, что для имитации освещения в режиме реального времени используются разнообразные хитроумные уловки и грубые приближения. Одно из таких приближений заключается в том, что WPF вычисляет яркость света от источников только в узлах.

Иногда такое приближение приводит к странным результатам, когда дело касается источников типа `PointLight` и `SpotLight`. Такая ситуация изображена на рис. 16.32. Окружность показывает, где свет пересекает четырехугольник. Это может быть конус от источника `SpotLight` или сфера, освещенная источником `PointLight` с конечным значением `Range`. Поверхность осталась неосвещенной, потому что свет не дошел до узлов, расположенных в углах.

Чтобы справиться с этой проблемой, поверхность следует покрыть сеткой более мелких четырехугольников, как показано на рис. 16.33. Добавление вершин увеличивает количество точек, в которых вычисляется освещенность, и эффект источника `SpotLight` становится виден.

Чем мельче ячейки сетки, тем выше детализация. Однако для создания идеального круга света сетку пришлось бы измельчать до тех пор, пока в каждом пикселе не оказался бы узел сетки.

Если вас устраивает довольно протяженная область постепенного затухания, как на рис. 16.31, то обычно достаточно небольшого измельчения. Если же необходима резкая граница между освещенной и неосвещенной областями, то, пожалуй, лучше встроить подсветку в материал. Особенно хорошо этот прием будет работать, если освещение сцены статическое.

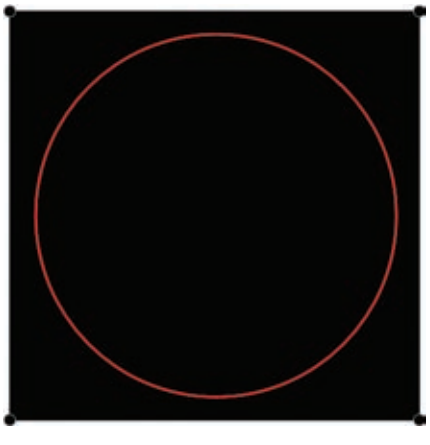


Рис. 16.32. *Свет внутри четырехугольника*

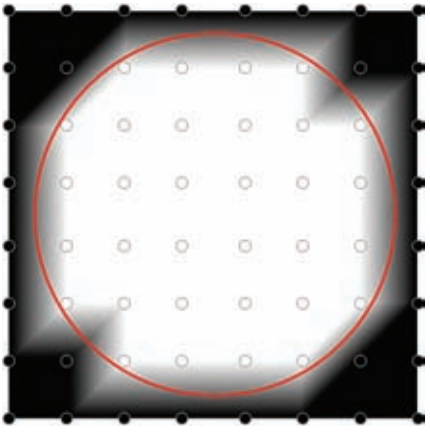


Рис. 16.33. *Свет внутри более мелкой сетки*

Свойство Direction определяет направление конуса. Форма конуса регулируется свойствами OuterConeAngle и InnerConeAngle. Например:

```
<SpotLight Color="White" Position="2,2,2"
  Direction="-1,-1,-1"
  InnerConeAngle="45"
  OuterConeAngle="90"/>
```

Область внутри телесного угла InnerConeAngle получает свет того цвета и яркости, который задан в свойстве Color. В области между InnerConeAngle и OuterConeAngle яркость постепенно уменьшается. Варьируя разность между InnerConeAngle и OuterConeAngle, можно изменять размер области затухания. Если сделать угол InnerConeAngle большим или равным OuterConeAngle, то получится источник SpotLight, не имеющий области затухания.

Класс AmbientLight

Класс AmbientLight обычно применяют для аппроксимации света, рассеянного многочисленными диффузно отражающими поверхностями, присутствующими на сцене. Лучи от источника типа AmbientLight падают на все поверхности со всех направлений, как показано на рис. 16.34.

В классе AmbientLight есть всего одно интересное свойство, Color, которое определяет яркость и цвет излучаемого света. Свойство Transform, унаследованное от класса Model3D, в классе AmbientLight ничего не делает.

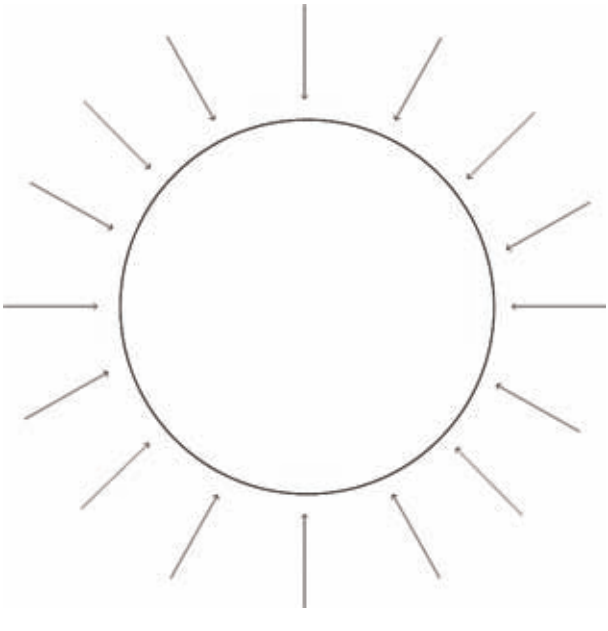


Рис. 16.34. *Источник AmbientLight, освещающий сферу*

Добавление на сцену источника AmbientLight максимальной яркости (см. код ниже) обычно приводит к плоскому изображению, пример которого показан на рис. 16.35:

```
<AmbientLight Color="White"/>
```



Рис. 16.35. *Источник AmbientLight максимальной яркости*

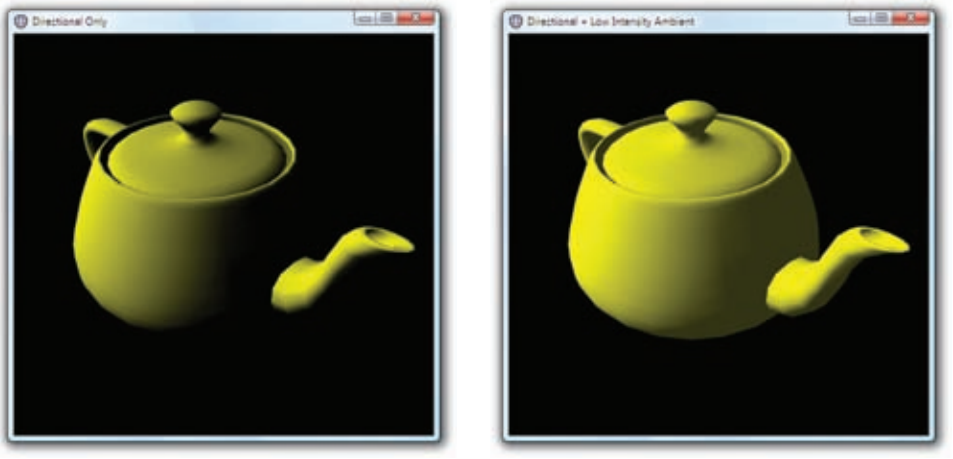
Однако неяркий источник AmbientLight проявляет неосвещенные области сцены, в результате чего получается более мягкое изображение, близкое к естественно освещенной сцене. На рис. 16.36 показана освещенная сцена с источником AmbientLight и без него:

```
<Model3DGroup>
  <DirectionalLight Direction="1,-1,-1" Color="White"/>
  <AmbientLight Color="#FF333333"/>
</Model3DGroup>
```

СОВЕТ

Есть хорошее эвристическое правило, позволяющее предотвратить появление плоских сцен: помещайте на сцену только один источник AmbientLight яркостью не более одной трети белого (то есть #555555 *или ниже*).

Для управления тем, как сильно источник AmbientLight влияет на конкретные объекты на сцене, используйте свойство DiffuseMaterial.AmbientColor. Например, если задать черный цвет AmbientColor, то все модели, состоящие из такого материала DiffuseMaterial, вообще не будут подвержены влиянию источников AmbientLight.



Только источник DirectionalLight После добавления неяркого источника AmbientLight

Рис. 16.36. *Освещение сцены с использованием источника AmbientLight и без него*

Класс GeometryModel3D

Форма объектов, видимых на 3D-сцене, определяется их геометрией. В WPF геометрия трехмерного тела задается с помощью объектов Geometry3D. Однако сам по себе объект Geometry3D определяет лишь трехмерную поверхность без какого бы то ни было визуального представления. Чтобы эту поверхность увидеть, необходимо добавить еще материал – объект Material. Класс GeometryModel3D, производный от Model3D, как раз и объединяет оба аспекта с помощью свойств Geometry и Material.

Ниже приведен пример элемента GeometryModel3D, представляющего квадрат (описанный в виде MeshGeometry3D) из синего материала DiffuseMaterial:

```
<GeometryModel3D>
<GeometryModel3D.Material>
  <DiffuseMaterial Brush="Blue"/>
</GeometryModel3D.Material>
<GeometryModel3D.Geometry>
  <MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 1,1,0"
                  TriangleIndices="0 1 2, 0 2 3"/>
</GeometryModel3D.Geometry>
</GeometryModel3D>
```

В этом разделе мы сначала рассмотрим различные типы материалов, а затем класс MeshGeometry3D.

Класс Material

Как уже упоминалось выше, свойства объектов типа Light определяют направление и цвет лучей, освещающих сцену. Свойства материалов говорят о том, какие лучи отражаются к наблюдателю, создавая тем самым видимое изобра-

жение. В реальном мире материалы поглощают свет с одной длиной волны и отражают – с другой. Яблоко кажется человеку красным, потому что кожура плода отражает красный свет и поглощает свет с другой длиной волны. В WPF именно тип и свойства объектов `Material` определяют, какие цвета отражают назад в камеру и создают изображение. В этом разделе мы обсудим различные типы материалов, поддерживаемые WPF:

- `DiffuseMaterial` (диффузный материал) – рассеивает падающий свет во всех направлениях, что создает плоский матовый внешний вид, как у газетной бумаги.
- `SpecularMaterial` (зеркальный материал) – отражает свет под тем же углом, под которым он падает. Зеркальные материалы применяются для создания эффекта блеска, характерного для гладких поверхностей из металла или пластика.
- `EmissiveMaterial` (излучающий материал) – аппроксимирует поверхность, излучающую свет. Излучающий материал всегда кажется освещенным, независимо от присутствия на сцене источников света; однако он не освещает другие объекты. Часто для достижения этого эффекта излучающие материалы используют в сочетании с источниками света. Кроме того, они применяются для создания объектов, которые всегда должны быть ярко освещены и для которых нежелательно наличие тени, как, например, во многих пользовательских интерфейсах.
- `MaterialGroup` – применяет к модели несколько материалов. Каждый материал визуализируется по очереди, причем последний в группе оказывается сверху.

Материал `DiffuseMaterial`. Это наиболее употребительный тип материала. Свет, падающий на объект, изготовленный из диффузного материала, рассеивается во всех направлениях, что создает впечатление матовой поверхности. На рис. 16.37 визуализирована модель чайника из диффузного материала.



Рис. 16.37. *Модель чайника из диффузного материала (см. также цветную вклейку)*

Рассеивание равномерно и не зависит от угла обзора камеры. Однако угол между падающим светом и поверхностью влияет на яркость отраженного света, как видно на рис. 16.38. Луч, падающий перпендикулярно поверхности, отражается с максимальной яркостью. Яркость отражения снижается по мере уменьшения угла падения света. Именно поэтому части чайника, обращенные к свету, кажутся ярко освещенными, тогда как повернутые в сторону от источника света остаются неосвещенными.

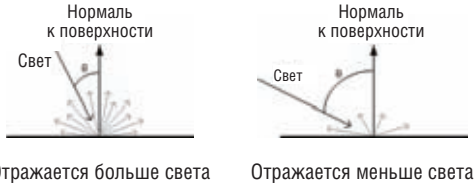


***Рис. 16.38**. Яркость отраженного света*

Цвет, отражаемый материалом, регулируется его свойством `Brush`. Изображение красного чайника на рис. 16.37 создано путем освещения белым светом диффузного материала, отражающего только красный свет.

```
<DiffuseMaterial Brush="Red"/>
```

Варьировать цвета, отражаемые материалом, который покрывает поверхность объекта, можно с помощью несплошных цветных кистей. Например, в левой части рис. 16.39 показан тот же чайник с полосатым рисунком, который был создан применением такой кисти `ImageBrush`:

```
<DiffuseMaterial>
<DiffuseMaterial.Brush>
  <ImageBrush ImageSource="C:\ZebraStripes.png" ViewportUnits="Absolute"/>
</DiffuseMaterial.Brush>
</DiffuseMaterial>
```

ПРЕДУПРЕЖДЕНИЕ

Если вы пользуетесь кистью, отличной от SolidColorBrush, то координаты текстуры необходимы!

При попытке воспользоваться кистью типа `GradientBrush`, `ImageBrush`, `DrawingBrush` или `VisualBrush`, не задавая координаты текстуры, модель не будет визуализирована. Без координат текстуры невозможно установить соответствие между точками на поверхности и цветами, присутствующими в кисти. Для сплошной кисти `SolidColorBrush` это не проблема, потому что всем точкам поверхности сопоставляется один и тот же цвет.

Отсутствие или неправильное задание координат текстуры обычно легко диагностируется. Если после смены кисти на `SolidColorBrush` модель появляется, значит, для геометрического объекта, скорее всего, не заданы координаты текстуры.



ImageBrush на белом материале ImageBrush на оранжевом материале

Рис. 16.39. *Кисть ImageBrush, нанесенная на белый и окрашенный материалы (см. также цветную вклейку)*

СОВЕТ

Имеющаяся в WPF возможность использовать кисти, а не просто статические изображения в качестве источника текстуры делает наложение текстур гораздо более выразительным. Привязывать к данным и анимировать можно не только сами 3D-модели, но и содержимое заданных для них кистей, в качестве которых можно выбирать анимированные двумерные рисунки `Drawing`, видео и даже двумерные элементы управления, например `DocumentViewer`!

Какая часть кисти на какой части трехмерной поверхности видна, определяется координатами текстуры (иногда их называют UV-коодинатами) объекта `Geometry`. Координаты текстуры подробно обсуждаются ниже.

В правой части рис. 16.39 показана та же кисть `ImageBrush` оранжевого тона, так что теперь полоски выглядят, как на шкуре тигра, а не зебры. Добиться этого эффекта можно тремя способами.

- Изменить изображение, лежащее в основе кисти `ImageBrush`.
- Сменить цвет `Color` источников света на оранжевый. Белые области диффузного материала отражают свет любого цвета. Если на сцене присутствует только оранжевый источник, то будет отражаться оранжевый свет.
- Сменить свойство `Color` материала на оранжевый. По существу, это эквивалентно смене цвета источника, но распространяется только на данный конкретный материал, а не на все материалы на сцене.

КОПНЕМ ГЛУБЖЕ

Вычисление результирующего отраженного цвета

Результирующий цвет, отраженный в камеру, вычисляется по формуле:

$$\left(\sum_{i=0}^{n} Lc_i Mc_i\right) Mb$$

где Lc – свойство `Color` каждого источника света, Mc – свойство `Color` каждого материала, а Mb – образец цвета из кисти материала. Альфа-компоненты свойства `Color` материала и образца цвета из кисти перемножаются. Альфа-компонент цвета источника игнорируется.

Обычно для изменения яркости света на сцене используют свойство `Color` источников света. Менять оттенок источников имеет смысл также для создания нужного цвета окружающей среды, например зеленый свет – в лесу, синий – под водой и т. д.

Свойство `Color` материала полезно, когда требуется профильтровать свет, отражаемый конкретными объектами. Можно, скажем, поиграть с освещением сцены, затемняя некоторые объекты. Другое применение этого свойства – придать новое качество источнику изображения `ImageSource`, изменив его цвет; на рис. 16.39 мы воспользовались этим, превратив текстуру зебры в тигриные полоски:

```
<DiffuseMaterial Color="Orange">
<DiffuseMaterial.Brush>
  <ImageBrush ImageSource="C:\ZebraStripes.png" ViewportUnits="Absolute"/>
</DiffuseMaterial.Brush>
</DiffuseMaterial>
```

Эта техника может оказаться особенно полезной, когда нужно, чтобы пользователь имел возможность сам выбирать цвет 3D-модели, например автомобиля.

FAQ

Почему полупрозрачные диффузные материалы не всегда просвечивают?

Есть несколько способов создать полупрозрачный диффузный материал: использовать источник изображения `ImageSource` с альфа-каналом, использовать свойство кисти `Brush.Opacity` или задать альфа-канал в цвете `DiffuseMaterial.Color`. Если вы создадите полупрозрачный диффузный материал, то с удивлением обнаружите, что иногда находящиеся за ним объекты не видны.

Это результат того, как WPF обрабатывает перекрывающиеся поверхности. Дабы гарантировать, что поверхность, ближайшая к камере, визуализируется последней (то есть оказывается сверху), WPF не сортирует все присутствующие на сцене треугольники для того, чтобы потом вывести их в порядке от задних к передним, а пользуется буфером глубины. Использовать буфер глубины гораздо быстрее, чем сортировать сцену (с потенциальным подразбиением взаимопроникающих объектов), но у этой технологии есть побочный эффект: после того как ближняя к камере поверхность визуализирована, поверхности, отстоящие дальше, пропускаются. Неприятность возникает, когда ближняя поверхность полупрозрачна.

Чтобы объекты из полупрозрачных диффузных материалов визуализировались, как задумано, необходимо внимательно отнестись к конструированию сцены. Как и в двумерной графике, объекты на 3D-сцене визуализируются в том порядке, в каком они помещены в свойство `Children`. Помещая полупрозрачные объекты в конец коллекции `Children`, можно гарантировать, что находящиеся за ними объекты будут нарисованы раньше.

Другая возможность создать эффект полупрозрачности – использовать материалы типа `EmissiveMaterial`. Излучающие материалы аддитивно смешиваются, поэтому буфер глубины при их визуализации не используется. Мы рассмотрим этот вопрос в следующем разделе.

FAQ

А где в WPF класс AmbientMaterial?

Те, кто переносит код или импортирует файлы в формате, основанном на конвейере визуализации с фиксированными функциями, который применяется в Direct3D и на других 3D-платформах, возможно, недоумевают, почему в WPF нет класса `AmbientMaterial`. В традиционном подходе к освещению на основе фиксированных функций пользователю разрешено задавать четыре цвета материала – рассеянный (ambient), диффузный, излучающий и зеркальный, – которые впоследствии используются для вычисления вклада источников света в каждом узле.

Рассеянный и диффузный цвета похожи, но задаются порознь, чтобы пользователь мог ограничить вклад вездесущего `AmbientLight` в освещенность отдельных частей сцены. Пусть, например, сцена на открытом воздухе должна быть ярко освещена источником `AmbientLight`, но в пещеру внутри холма его свет проникать не должен. Тогда вы можете сделать рассеянный цвет материала внутри пещеры черным и тем самым подавить все источники `AmbientLight`.

Для этой цели в WPF имеется свойство `AmbientColor` в классе `DiffuseMaterial`. Обычное свойство `Color` управляет тем, как диффузный материал отражает свет от источников всех типов, кроме `AmbientLight`. А свойство `AmbientColor` ограничивает цвет отраженного света только от источников типа `AmbientLight`.

Таким образом, диффузный, зеркальный и излучающий цвета материалов в традиционном конвейере отображаются на свойства `Color` классов `DiffuseMaterial`, `SpecularMaterial` и `EmissiveMaterial` соответственно. А рассеянный цвет отображается на свойство `AmbientColor` класса `DiffuseMaterial`.

Материал `EmissiveMaterial`. Такой материал всегда излучает свет, видимый камере. Однако он не освещает другие поверхности на сцене как источник света. В левой части рис. 16.40 показано влияние следующего излучающего материала на модель чайника:

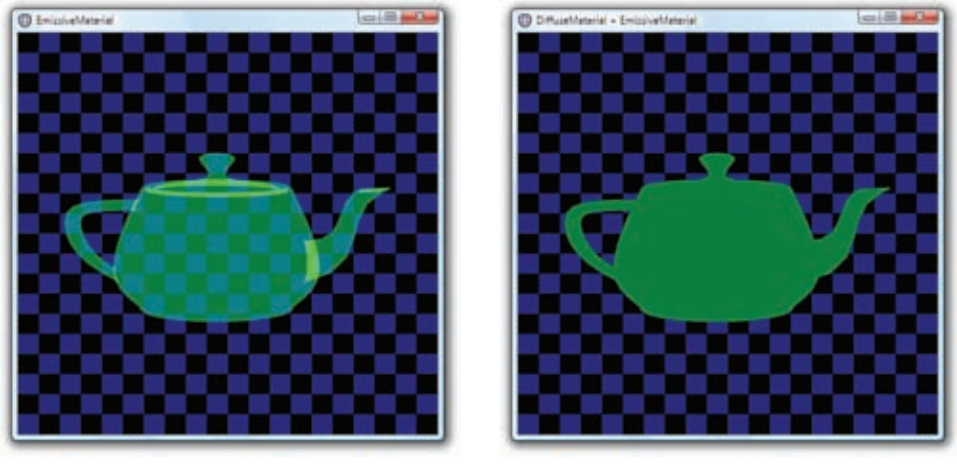
```
<EmissiveMaterial Brush="Green"/>
```

Излучающие материалы аддитивно подмешиваются к изображению. При аддитивном смешении свет складывается с изображением, но не преграждает путь свету, исходящему от объектов, которые находятся позади материала. Именно поэтому в левой части рис. 16.40 сквозь чайник просвечивает фон в клеточку. Ярко-зеленые области – это следы перекрытых геометрических объектов, которые вообще-то не должны быть видны (ободок крышки, ручка и носик немного вдаются в корпус чайника).

Чтобы предотвратить такой взгляд сквозь модель, можно комбинировать `EmissiveMaterial` с `DiffuseMaterial`, воспользовавшись элементом `MaterialGroup`:

```
<MaterialGroup>
  <DiffuseMaterial Brush="Black"/>
  <EmissiveMaterial Brush="Green"/>
</MaterialGroup>
```

Результат показан в правой части рис. 16.40.



`EmissiveMaterial` `EmissiveMaterial` поверх черного `DiffuseMaterial`

Рис. 16.40. *Модель чайника из излучающего материала (см. также цветную вклейку)*

В данном случае излучающий материал, как и раньше, аддитивно смешивается с изображением. Но, поскольку под ним находится черный чайник, результатом оказывается сумма черного и излучающего цветов, то есть только излучающий цвет. Отметим также, что теперь скрытые детали внутри чайника не видны. Это объясняется тем, что ближняя сторона черного чайника

не дает проникнуть взгляду сквозь модель и увидеть скрытые крышку, ручку и носик.

Материал `SpecularMaterial`. Зеркальный материал отражает свет обратно к наблюдателю, когда камера расположена под углом, близким к углу отражения света от поверхности. Он также аддитивно смешивается и сам по себе выглядит, как стекло, что видно в левой части рис. 16.41:

```
<SpecularMaterial Brush="White" SpecularPower="10"/>
```

Часто `SpecularMaterial` сочетают с `DiffuseMaterial`, чтобы добавить яркости твердым блестящим поверхностям (правая часть рис. 16.41):

```
<MaterialGroup>
  <DiffuseMaterial Brush="Red"/>
  <SpecularMaterial Brush="White" SpecularPower="40"/>
</MaterialGroup>
```

Сравните этот рисунок с изображением красного чайника из одного лишь диффузного материала на рис. 16.37.

«Твердость» поверхности регулируется свойством `SpecularPower`. Чем выше значение этого свойства, тем более фокусированным выглядит зеркальное пятно.



`SpecularMaterial` сам по себе



`SpecularMaterial` поверх красного `DiffuseMaterial`

Рис. 16.41. *Модель чайника из зеркального материала (см. также цветную вклейку)*

СОВЕТ

Чтобы поверхность выглядела как пластмассовая, можно сочетать яркий диффузный материал с белым зеркальным. Для получения металлической поверхности попробуйте сочетание темного диффузного материала с ярким зеркальным того же оттенка.

В отличие от диффузного материала, который рассеивает свет равномерно, зеркальный материал отражает свет в направлении, противоположном направлению падения. Как показано на рис. 16.42, свет отражается от SpecularMaterial, как от зеркала, и виден лишь в том случае, когда камера находится близко к траектории отраженного луча.



***Рис. 16.42.** Отражение света от зеркального материала*

Отметим, что, поскольку источники типа AmbientLight ненаправленные, они не оказывают влияния на зеркальные материалы.

Как и в случае диффузного материала, результирующий цвет, отраженный к наблюдателю, является комбинацией свойств Color присутствующих на сцене источников света – Brush объекта SpecularMaterial и Color материала. Точная формула вычисления результирующего цвета приведена в разделе «DiffuseMaterial» выше.

СОВЕТ

В отличие от традиционного освещения с фиксированными функциями, когда для зеркального выделения разрешается задавать только цвет, WPF позволяет использовать произвольную кисть Brush. Применяя альфа-канал в изображении, можно создать материал, в котором зеркальные характеристики поверхности изменяются. Эту технику, которая называется *картированием глянца* (*gloss mapping*), можно использовать для придания блеска только металлическим частям текстуры автомобиля, создания отпечатков пальцев на стекле и т. д.

Комбинирование материалов. Выше уже было продемонстрировано, что класс MaterialGroup позволяет применять к поверхности несколько материалов. Чаще всего EmissiveMaterial или SpecularMaterial располагают поверх DiffuseMaterial, как мы и делали для создания материала чайника.

Класс Geometry3D

По аналогии с двумерным классом Geometry класс Geometry3D применяется для определения формы трехмерных объектов. Сами по себе объекты Geometry3D не имеют никакого визуального представления. Для получения объекта Model3D, который можно визуализировать, их необходимо объединить с материалами, используя класс GeometryModel3D. У класса Geometry3D есть всего один конкретный подкласс: MeshGeometry3D.

Класс MeshGeometry3D представляет набор трехмерных поверхностей, заданный в виде списка треугольников. В этом классе определены следующие свойства:

- Positions – определяет вершины треугольников, входящих в сетку (узлы сетки).
- TriangleIndices – описывает, как из узлов составляются треугольники. Если свойство TriangleIndices не задано, подразумевается, что узлы связываются в порядке следования 0 1 2, затем 3 4 5 и т. д.
- Normals – позволяет дополнительно настроить освещение сетки.
- TextureCoordinates – описывает отображение трехмерной поверхности на двумерную плоскость для каждого узла, использованного в материалах.

Каждое из свойств Positions, Normals и TextureCoordinates – это коллекция, содержащая по одному элементу для всех узлов сетки. Например, позиция узла 0 задается нулевым элементом в коллекции Positions, нормаль к поверхности в узле 0 – нулевым элементом в коллекции Normals и т. д.

Свойство Positions. Треугольники, образующие сетку, определяются заданием трехмерных координат узлов в их вершинах. Координаты хранятся в коллекции Positions объекта MeshGeometry3D. По умолчанию каждая группа из трех точек Point3D в коллекции Positions рисуется в виде треугольника. Так, следующий фрагмент создает треугольник, изображенный на рис. 16.43.

```
<MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0"/>
```

Добавив второй треугольник, можно создать квадрат (рис. 16.44).

```
<MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 -1,1,0 1,-1,0 1,1,0"/>
```

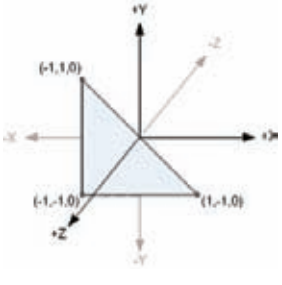


***Рис. 16.43.** Треугольник, описываемый элементом MeshGeometry3D*

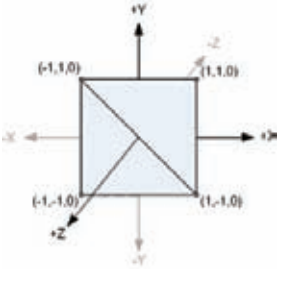
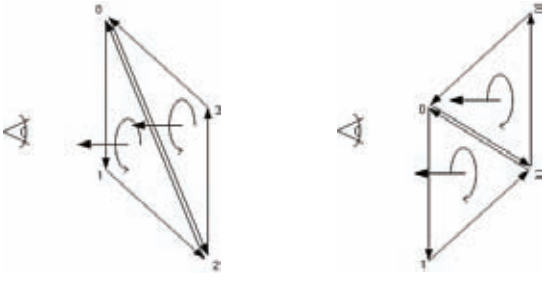


***Рис. 16.44.** Квадрат, описываемый элементом MeshGeometry3D*

Передняя и задняя стороны. Одна из особенностей трехмерной графики, которая часто удивляет разработчиков, привыкших к геометрии на плоскости, заключается в том, что у треугольников в сетке MeshGeometry3D передняя и задняя стороны различаются. С каждой стороной может быть ассоциирован свой материал. Можно также вообще не визуализировать какую-то сторону, оставив свойство Material равным null. Какая сторона считается передней, зависит

от порядка обхода вершин. На рис. 16.45 показан порядок обхода треугольников, составляющих квадрат, при взгляде спереди и сзади.

Порядок обхода определяется порядком соединения вершин треугольника. Так, если точка 0 соединена с точкой 1, то создается ориентированное ребро, начинающееся в точке 0 и заканчивающееся в точке 1. Направления ребер определяют порядок обхода против часовой стрелки, если смотреть спереди, как показано в левой части рис. 16.45.



Вид квадрата спереди Вид квадрата сзади

Рис. 16.45. *Вид на квадрат с двух разных точек обзора*

КОПНЕМ ГЛУБЖЕ

Порядок обхода и правило правой руки

В разделе о камерах мы ознакомились с правилом правой руки для запоминания того, куда направлена ось *z* в правосторонней системе координат. Есть и второе правило правой руки, которое позволяет определить, какая сторона треугольника передняя. Направление, в котором согнуты пальцы, когда большой палец указывает вам в лицо, и есть направление обхода. Как показано на рис. 16.46, в правосторонней системе координат это направление против часовой стрелки.



Рис. 16.46. *Второе правило правой руки*

Это правило полезно также для запоминания положительного направления вращения в правосторонней системе координат.

СОВЕТ

Если вы подозреваете, что с направлением обхода сетки есть какие-то проблемы, попробуйте задать свойства `Material` и `BackMaterial` так, чтобы треугольник был виден вне зависимости от того, с какой стороны на него смотреть.

Если допустимо задать одинаковый материал спереди и сзади, то направление обхода можно вообще не принимать во внимание. Но иногда бывает полезно задать различные материалы для разных сторон. Кроме того, визуализация будет происходить быстрее, если избежать прорисовки материала `BackMaterial`, когда он не виден на сцене.

Свойство `TriangleIndices`. Сетке придается нужная форма за счет добавления треугольников. Даже искривленные поверхности аппроксимируются большим числом маленьких треугольников. По мере увеличения количества треугольников в сетке растет и число общих ребер.

Свойство `TriangleIndices` позволяет задать общие для разных треугольников узлы сетки. Если коллекция `TriangleIndices` пуста, то считается, что все точки нужно соединять в том порядке, в котором они представлены в коллекции `Positions`. В противном случае точки соединяются в группы по три, как описаны в `TriangleIndices`. Например, квадрат, показанный на рис. 16.47, можно создать, задав всего четыре уникальных точки:

```
<MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 1,1,0"
                TriangleIndices="0 1 2, 0 2 3"/>
```

Обобществление одной точки несколькими треугольниками отличается по семантике от объявления одной точки много раз. Если точка является общей, то треугольники считаются частями одной непрерывной поверхности. Если же точки объявлены отдельно, то треугольники принадлежат разным граничащим поверхностям, у которых могут быть разные нормали или координаты текстур.

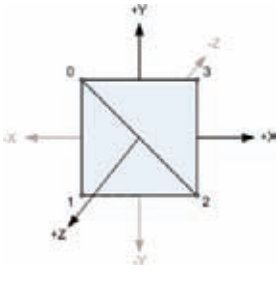


Рис. 16.47. *Индексы вершин*

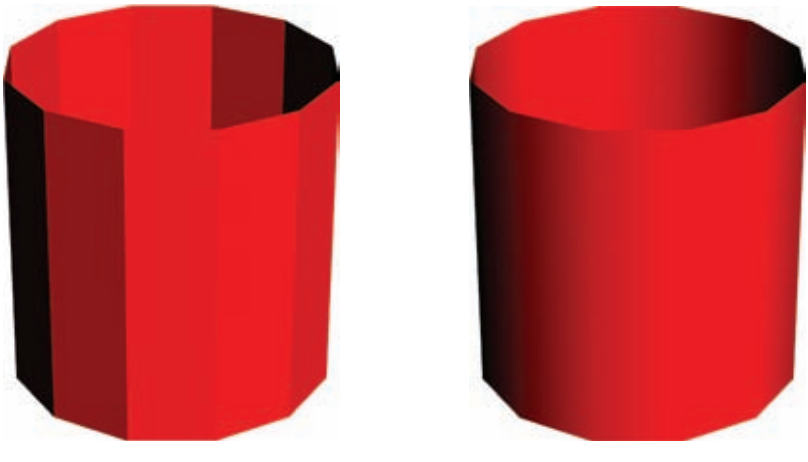
СОВЕТ

Неважно, используете вы коллекцию `TriangleIndices` или нет, тревожиться по поводу появления зазоров между треугольниками, объявленными в одном элементе `MeshGeometry3D`, нет нужды. В WPF определены строгие правила визуализации, гарантирующие, что треугольники с общими точками визуализируются как смежные, без зазоров.

Однако же вы *должны* знать, что преобразования не всегда точны. Если к двум элементам `MeshGeometry3D` применяются разные преобразования, чтобы сделать их смежными, то может случиться, что из-за ошибок при вычислениях с плавающей точкой образуется небольшой зазор между сетками.

Иногда ошибку можно обойти, слегка изменив преобразование и создав тем самым небольшое перекрытие. А порой приходится конструировать элементы `MeshGeometry3D` так, чтобы они имели смежные точки, а не добиваться их слитности с помощью преобразований.

Свойство `Normals`. *Нормалью* называется вектор, перпендикулярный поверхности в данной точке. Нормали в узлах задаются для того, чтобы система знала, какие треугольники представляют плоские поверхности, а какие аппроксимируют искривленные. На рис. 16.48 показана разница между тенями на плоской и плавно изогнутой трубе, причем обе аппроксимированы 12 четырехугольниками.

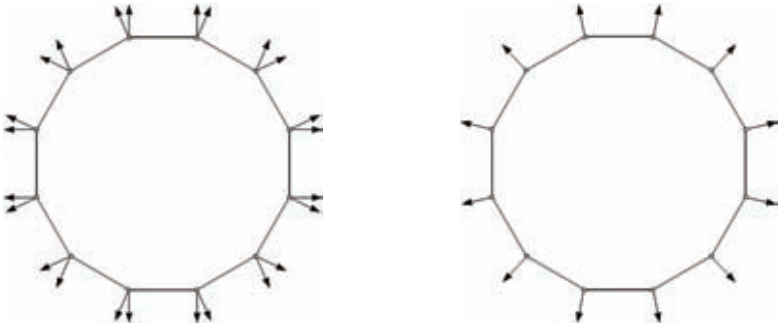


Труба с плоскими тенями Труба с плавными тенями

Рис. 16.48. *Две трубы, аппроксимированных 12 четырехугольниками*

Когда нормали в каждой вершине треугольника параллельны, как показано на сечении трубы в левой части рис. 16.49, визуализированная поверхность

кажется плоской. Если же нормали направлены в разные стороны, то затенение плавно интерполируется на всю поверхность треугольника. Чтобы создать гладкую поверхность, как справа на рис. 16.49, нормали к соседним треугольникам должны быть одинаковы во избежание складок.



Сечение плоской трубы Сечение гладкой трубы

Рис. 16.49. *Сечения труб, показанных на рис. 16.48*

СОВЕТ

Если вы не зададите нормали, то система сгенерирует их самостоятельно, усреднив нормали к плоскостям всех треугольников, сходящихся в каждом узле. Если у треугольников нет общих узлов, то результатом будет нормаль к плоскости треугольника, что дает картину плоских теней, показанную в левой части рис. 16.48. Если из коллекции `TriangleIndices` следует, что узлы являются общими для нескольких смежных треугольников, то усреднение дает более плавную картину теней, показанную на рис. 16.48 справа.

Рассмотрим простую сетку, описывающую квадрат. Если требуется, чтобы квадрат выглядел плоским, то все нормали должны быть перпендикулярны поверхности, как показано на рис. 16.50 слева:

```
<MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 1,1,0"
                TriangleIndices="0 1 2, 0 2 3"
                Normals="0,0,1 0,0,1 0,0,1 0,0,1"/>
```

Если же нужно, чтобы квадрат освещался так, будто он является аппроксимацией слегка изогнутой поверхности, то нормали должны соответствовать кривизне поверхности, как показано на рис. 16.50 справа:

```
<MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 1,1,0"
                TriangleIndices="0 1 2, 0 2 3"
                Normals="-0.25,0.25,1 -0.25,-0.25,1 0.25,-0.25,1 0.25,0.25,1"/>
```

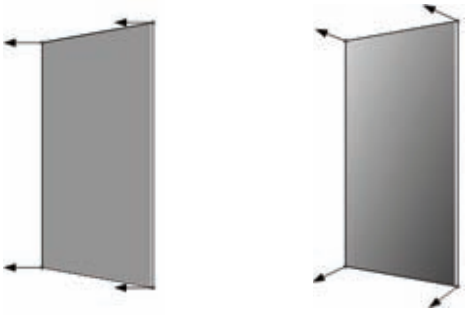


Рис. 16.50. *Результат применения двух разных наборов нормалей*

Свойство `TextureCoordinates`. При установке свойства `Fill` двумерного объекта `Shape` предполагается, что кисть нужно отобразить на ограничивающий прямоугольник области, занимаемой фигурой. В трехмерной графике задача отображения возлагается на вас. Каждый элемент коллекции `TextureCoordinates` – это двумерная точка в пространстве кисти. Эти точки устанавливают соответствие между треугольниками в трехмерном пространстве и треугольниками в пространстве кисти. Треугольники в пространстве кисти дают цвета материалов на этапе визуализации поверхности. На рис. 16.51 показано, как сопоставить вершины квадрата, чтобы нанести на него изображение:

```
<MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 1,1,0"
                TriangleIndices="0 1 2  0 2 3"
                TextureCoordinates="0,0 0,1 1,1 1,0"/>
```

Не забывайте, что в WPF применяется соглашение, согласно которому начало двумерной системы координат находится в левом верхнем углу, а ось *y* направлена вниз. Кроме того, по соглашению изображение-источник обычно занимает квадрат с диагональю от (0,0) до (1,1).

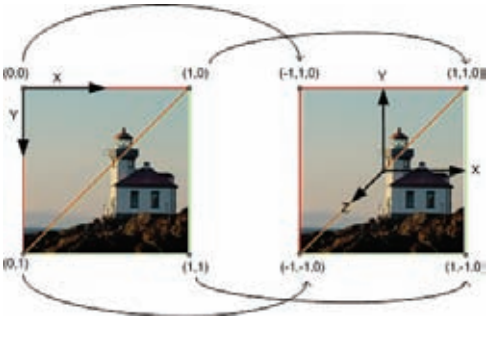


Рис. 16.51. *Отображение между двумерным пространством кисти и трехмерной поверхностью*

ПРЕДУПРЕЖДЕНИЕ

По умолчанию координаты текстур в WPF интерпретируются иначе, чем вы, возможно, ожидаете!

Следует знать о двух странностях в способе интерпретации координат текстур в WPF. По умолчанию поведение координат текстуры очень похоже на поведение двумерных геометрических объектов, что весьма удобно для интеграции 2D- и 3D-графики. Однако когда вы пытаетесь использовать сетку с координатами текстуры, сгенерированными для другой системы, некоторые параметры кисти необходимо изменить.

Во-первых, по умолчанию пространство кисти отображается на всю область, описываемую координатами текстуры. Это означает, что такое определение не покажет левый верхний квадрант кисти, на что вы, возможно, рассчитываете:

```
<MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 1,1,0"
                TriangleIndices="0 1 2, 0 2 3"
                TextureCoordinates="0,0 0,0.5 0.5,0.5 0.5,0"/>
```

На самом деле прямоугольник (0,0) – (0.5,0.5) становится относительным ограничивающим прямоугольником источника и отображается вся кисть. Чтобы избежать этого, следует присвоить свойству `ViewportUnits` кисти значение `Absolute`:

```
<ImageBrush ViewportUnits="Absolute" .../>
```

Это желательно почти всегда при применении кистей к трехмерным сеткам. Поведение по умолчанию полезно в двумерной графике для отображения пространства кисти на ограничивающий прямоугольник закрашиваемого двумерного геометрического объекта. В трехмерном случае это требуется редко.

Во-вторых, нужно иметь в виду, что в некоторых системах двумерной графики ось y направлена вверх, а не вниз, как в WPF. Если используются координаты текстуры, сгенерированные для такой системы, то кисть будет применяться в направлении, противоположном желаемому. Это можно исправить простым преобразованием кисти:

```
<ImageBrush ViewportUnits="Absolute" Transform="1,0,0,-1,0,1" .../>
```

Наконец, если для вашей сетки координаты текстуры выходят за пределы диапазона от 0 до 1, то, вероятно, предполагалось использовать мозаичный способ. В WPF режим `TileMode` следует включать явно:

```
<ImageBrush ViewportUnits="Absolute" Transform="1,0,0,-1,0,1" TileMode="Tile" .../>
```

В качестве примера взята кисть `ImageBrush`, но все сказанное относится к любым кистям, для которых заданы координаты текстуры.

Класс Model3DGroup

Класс `Model3DGroup` наследует `Model3D`. Такие объекты используются для группировки набора объектов `Model3D` в единую модель. Группировка нескольких `GeometryModel3D` с помощью `Model3DGroup` – это способ построить модель, в которой применяется несколько материалов. В листинге 16.4 показано, как из шести элементов `GeometryModel3D` можно образовать модель куба. Альтернативный

подход – создать один элемент MeshGeometry3D, в котором для достижения того же эффекта используется MaterialGroup.

СОВЕТ

В некоторых системах трехмерной графики «сеткой» (mesh) называется объект, который содержит не только геометрическую информацию, но и материалы. Иногда в сетке разрешается даже присутствие нескольких материалов. В WPF этому соответствует объект Model3DGroup, содержащий несколько объектов GeometryModel3D, по одному для каждого материала.

Листинг 16.4. Куб

```
<Model3DGroup x:Name="Cube">

  <GeometryModel3D x:Name="Front">
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="Orange"/>
  </GeometryModel3D.Material>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="1,1,1 -1,1,1 -1,-1,1 1,-1,1"
                    TextureCoordinates="1,1 0,1 0,0 1,0"
                    TriangleIndices="0 1 2 0 2 3"/>
  </GeometryModel3D.Geometry>
  </GeometryModel3D>

  <GeometryModel3D x:Name="Right">
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="Yellow"/>
  </GeometryModel3D.Material>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="1,1,-1 -1,1,-1 -1,1,1 1,1,1"
                    TextureCoordinates="0,0 1,0 1,1 0,1"
                    TriangleIndices="0 1 2 0 2 3"/>
  </GeometryModel3D.Geometry>
  </GeometryModel3D>

  <GeometryModel3D x:Name="Back">
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="Red"/>
  </GeometryModel3D.Material>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="-1,-1,-1 -1,1,-1 1,1,-1 1,-1,-1"
                    TextureCoordinates="1,0 1,1 0,1 0,0"
                    TriangleIndices="0 1 2 0 2 3"/>
  </GeometryModel3D.Geometry>
  </GeometryModel3D>

  <GeometryModel3D x:Name="Left">
  <GeometryModel3D.Material>
```

```
    <DiffuseMaterial Brush="Blue"/>
  </GeometryModel3D.Material>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="-1,1,1 -1,1,-1 -1,-1,-1 -1,-1,1"
                    TextureCoordinates="1,1 0,1 0,0 1,0"
                    TriangleIndices="0 1 2 0 2 3"/>
  </GeometryModel3D.Geometry>
  </GeometryModel3D>

  <GeometryModel3D x:Name="Top">
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="Green"/>
  </GeometryModel3D.Material>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="1,-1,1 1,-1,-1 1,1,-1 1,1,1"
                    TextureCoordinates="1,1 0,1 0,0 1,0"
                    TriangleIndices="0 1 2 0 2 3"/>
  </GeometryModel3D.Geometry>
  </GeometryModel3D>

  <GeometryModel3D x:Name="Bottom">
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="Purple"/>
  </GeometryModel3D.Material>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D Positions="-1,-1,1 -1,-1,-1 1,-1,-1 1,-1,1"
                    TextureCoordinates="0,1 0,0 1,0 1,1"
                    TriangleIndices="0 1 2 0 2 3"/>
  </GeometryModel3D.Geometry>
  </GeometryModel3D>

</Model3DGroup>
```

Класс Visual3D

Все элементы, которые рисуют двумерное содержимое на экране, наследуют возможность визуализации от базового класса `Visual`. Аналогично объекты `Visual3D` – это узлы визуального дерева, способные отображать трехмерное содержимое. Службы `Visual` – проверка попадания, построение ограничивающего прямоугольника и прочие – также обобщаются на `Visual3D` и доступны через класс `VisualTreeHelper`.

В состав WPF входят три непосредственных подкласса `Visual3D`: `ModelVisual3D`, `UIElement3D` и `Viewport2DVisual3D`. Мы рассмотрим их в этом разделе.

Класс ModelVisual3D

Класс `ModelVisual3D` аналогичен двумерному классу `DrawingVisual` и входил в WPF еще со времен первой версии. Для задания содержимого `ModelVisual3D` предназначено свойство `Content`:

```
<Viewport3D>
<Viewport3D.Camera>
  <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
</Viewport3D.Camera>
<Viewport3D.Children>
  <ModelVisual3D Content="{StaticResource CubeModel}"/>
</Viewport3D.Children>
</Viewport3D>
```

В классе `ModelVisual3D` есть также свойство `Children`. Поэтому элементы `ModelVisual3D` применяются для композиции нескольких моделей на сцене внутри элемента `Viewport3D`:

```
<Viewport3D>
<Viewport3D.Camera>
  <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
</Viewport3D.Camera>
<Viewport3D.Children>
  <ModelVisual3D Transform="{DynamicResource SquadronTransform}">
    <ModelVisual3D Content="{StaticResource AirplaneModel}"
      Transform="{DynamicResource PlaneTransform1}"/>
    <ModelVisual3D Content="{StaticResource AirplaneModel}"
      Transform="{DynamicResource PlaneTransform2}"/>
  </ModelVisual3D>
</Viewport3D.Children>
</Viewport3D>
```

FAQ

В каких случаях лучше использовать Model3DGroup, а в каких – ModelVisual3D?

Хотя, в принципе, всю сцену можно собрать в единой группе `Model3DGroup`, отображаемой в одном элементе `ModelVisual3D` (или `ModelUIElement3D`, о котором речь пойдет ниже), при этом вы лишаете себя важных оптимизаций производительности. Классы `ModelVisual3D` и `ModelUIElement3D` оптимизированы для работы в качестве узлов сцены. Они кэшируют ограничивающие параллелепипеды и другую информацию, чего облегченные элементы `Model3DGroup` делать не умеют.

Другая крайность – использовать `ModelVisual3D` вообще для всех геометрических объектов `GeometryModel3D` на сцене. Но и так поступать не рекомендуется, потому что при этом без нужды увеличивается рабочее множество приложения. Объекты `Model3DGroup` – это облегченные конструкции, предназначенные для группировки нескольких элементов `GeometryModel3D` в единую модель.

В общем случае элементы `Model3DGroup` лучше использовать для объединения нескольких частей одной модели (например, шины, лобового стекла и корпуса автомобиля), а `ModelVisual3D` (или `ModelUIElement3D`) – для отображения экземпляров 3D-моделей (скажем, для каждого нового автомобиля).

Свойством содержимого в классе ModelVisual3D является Children (а не Content!), поэтому в показанном выше XAML-коде оба дочерних элемента добавляются непосредственно в родительский элемент. Имейте в виду, что объекты Model3D можно повторно использовать в нескольких ModelVisual3D.

Класс UIElement3D

Появившийся в версии WPF 3.5 абстрактный класс UIElement3D и его подклассы – следующий по сравнению с Visual3D шаг в деле внедрения принципов двумерной графической системы WPF в мир 3D. Как уже отмечалось выше в этой главе, про двумерный класс WPF UIElement часто говорят, что он привносит поддержку LIFE (компоновка, ввод, фокус и события). В трехмерной графике нет компоновки, но поддержку IFE (ввод, фокус и события) класс UIElement3D обеспечивает. Это существенно упрощает такие задачи, как присоединение обработчиков событий мыши к 3D-элементам на сцене. Вместо того чтобы обрабатывать каждый щелчок мышью в объекте Viewport3D, а затем в муках выяснять, по какой из 3D-моделей пользователь щелкнул, можно просто присоединить обработчики событий прямо к отдельным элементам UIElement3D.

В состав WPF входят два класса, производных от UIElement3D: ModelUIElement3D и ContainerUIElement3D. Если вы еще не забыли о принципе составления имен *FooBar*, описанном в предыдущей главе, то не удивитесь тому, что ModelUIElement3D – это UIElement3D, содержащий Model, а ContainerUIElement3D – это UIElement3D, который ведет себя как контейнер.

Класс ModelUIElement3D

В листингах 16.5 и 16.6 элемент ModelUIElement3D используется для создания куба, который меняет цвет при каждом щелчке по нему. У элемента ModelUIElement3D есть своя модель Model3D, но нет дочерних элементов. Обратите внимание, что обработчик события MouseDown присоединен к элементу ModelUIElement3D, а не к Viewport3D.

Листинг 16.5. MainWindow.xaml – куб, по которому можно щелкать мышью

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <Viewport3D>
    <Viewport3D.Camera>
      <PerspectiveCamera Position="3,3,4" LookDirection="-1,-1,-1"
                         FieldOfView="60"/>
    </Viewport3D.Camera>
    <Viewport3D.Children>
      <ModelVisual3D>
      <ModelVisual3D.Content>
        <DirectionalLight Direction="-0.3,-0.4,-0.5" />
      </ModelVisual3D.Content>
      </ModelVisual3D>
      <ModelUIElement3D MouseDown="Cube_MouseDown">
```

```
        <ModelUIElement3D.Model>
          <GeometryModel3D>
          <GeometryModel3D.Material>
            <DiffuseMaterial>
            <DiffuseMaterial.Brush>
              <SolidColorBrush Color="Purple" x:Name="CubeBrush"/>
            </DiffuseMaterial.Brush>
            </DiffuseMaterial>
          </GeometryModel3D.Material>
          <GeometryModel3D.Geometry>
            <MeshGeometry3D
              Positions="1,1,-1 1,-1,-1 -1,-1,-1 -1,1,-1 1,1,1 -1,1,1 -1,-1,1 1,-1,1
                         1,1,-1 1,1,1 1,-1,1 1,-1,-1 1,-1,-1 -1,-1,-1 -1,1,-1 -1,-1,1 -1,-1,-1
                        -1,-1,-1 -1,-1,1 -1,1,1 -1,1,-1 1,1,1 1,1,-1 -1,1,-1 -1,1,1"
              TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12 13 14 12 14
                               15 16 17 18 16 18 19 20 21 22 20 22 23"
              TextureCoordinates="0,1 0,0 1,0 1,1 1,1 -0,1 0,-0 1,0 1,1 -0,1 0,-0
                                  1,0 1,0 1,1 -0,1 0,-0 -0,0 1,-0 1,1 0,1 1,-0 1,1
                                  0,1 -0,0" />
          </GeometryModel3D.Geometry>
          </GeometryModel3D>
        </ModelUIElement3D.Model>
        </ModelUIElement3D>
      </Viewport3D.Children>
      </Viewport3D>
    </Grid>
  </Window>
```

Листинг 16.6. MainWindow.xaml.cs – застраничный код для куба, допускающего щелчки мышью

```
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

public partial class MainWindow : Window
{
  static Random r;

  public MainWindow()
  {
    InitializeComponent();
    r = new Random();
  }

  private void Cube_MouseDown(object sender, MouseButtonEventArgs e)
  {
    // Выбрать случайный цвет
    CubeBrush.Color = Color.FromRgb((byte)r.Next(), (byte)r.Next(), (byte)r.Next());
  }
}
```

Класс ContainerUIElement3D

Класс ContainerUIElement3D – это простой контейнер для одного или нескольких элементов ModelUIElement3D:

```
<Viewport3D>
<Viewport3D.Children>
  <ContainerUIElement3D>
    <ModelUIElement3D ...>
    <ModelUIElement3D ...>
  </ContainerUIElement3D>
</Viewport3D.Children>
</Viewport3D>
```

В самом классе ContainerUIElement3D нет модели Model3D, а имеется только коллекция Children типа Visual3DCollection в качестве свойства содержимого. Этот простой класс уместнее было бы назвать ModelUIElement3DGroup.

Отметим разницу между классом ModelVisual3D и двумя подклассами UIElement3D. В классе ModelVisual3D имеется как свойство Model3D, так и коллекция типа Visual3DCollection. В классе UIElement3D функциональность контейнера и модели разделена между подклассами ContainerUIElement3D и ModelUIElement3D соответственно.

Класс Viewport2DVisual3D

Появившийся в версии WPF 3.5 класс Viewport2DVisual3D позволяет отобразить динамическое интерактивное 2D-содержимое непосредственно на трехмерную поверхность. Раньше динамическое 2D-содержимое можно было отображать на трехмерную поверхность с помощью кистей VisualBrush и DrawingBrush. Но поскольку это всего лишь кисти, то интерактивность не обеспечивалась. Кнопку Button, нарисованную на трехмерной сфере с помощью кисти VisualBrush, было невозможно нажать. Класс Viewport2DVisual3D преодолел этот барьер: теперь кнопка, отображенная на сферу, отвечает на щелчки мышью. А в поле TextBox, нарисованном на поверхности куба, можно вводить текст с помощью клавиатуры и мыши. Мультисенсорные устройства ввода позволяют манипулировать двумерными элементами, нарисованными в трехмерном пространстве. Работают даже контекстные меню для таких средств, как проверка правописания в поле TextBox!

Имя Viewport2DVisual3D звучит странно, но, по существу, оно не отклоняется от соглашения об именовании, принятого в WPF. Это *действительно* объект Visual3D, который *ведет себя как* двумерный порт просмотра. Правда, смущает тот факт, что в WPF нет класса с именем Viewport2D.

В листинге 16.7 демонстрируется использование Viewport2DVisual3D в сочетании с кнопкой Button. В элементе Viewport2DVisual3D задается сетка MeshGeometry3D, материал Material и конечный объект Visual. Для интерактивного материала необходимо присвоить присоединенному свойству Viewport2DVisual3D.IsVisualHostMaterial значение true. Если с интерактивным материалом связана кисть Brush, она игнорируется. Цвет, ассоциированный с материалом,

модулируется цветом конечного объекта `Visual`, как обычно. Результат представлен на рис. 16.52.

Листинг 16.7. Интерактивная кнопка в трехмерном пространстве

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Grid>
    <Viewport3D>
    <Viewport3D.Camera>
      <PerspectiveCamera Position="0.2,0.1,1" LookDirection="-0.2,-.1,-1"
                         FieldOfView="120"/>
    </Viewport3D.Camera>
    <Viewport3D.Children>
      <ModelVisual3D>
      <ModelVisual3D.Content>
        <DirectionalLight Direction="-0.3,-0.4,-0.5" />
      </ModelVisual3D.Content>
      </ModelVisual3D>
      <Viewport2DVisual3D>
      <Viewport2DVisual3D.Geometry>
        <MeshGeometry3D Positions="-1,1,0 -1,-1,0 1,-1,0 1,1,0"
                        TextureCoordinates="0,0 0,1 1,1 1,0"
                        TriangleIndices="0 1 2 0 2 3" />
      </Viewport2DVisual3D.Geometry>
      <Viewport2DVisual3D.Material>
        <DiffuseMaterial Viewport2DVisual3D.IsVisualHostMaterial="True" />
      </Viewport2DVisual3D.Material>
        <Button>Hello, 3D</Button>
      </Viewport2DVisual3D>
    </Viewport3D.Children>
    </Viewport3D>
  </Grid>
</Window>
```



***Рис. 16.52.** Интерактивная двумерная кнопка в трехмерном пространстве*

СОВЕТ

Класс Viewport2DVisual3D поддерживает кэширование композиции. В нем имеется свойство CacheMode, работающее точно так же, как одноименное свойство в классе UIElement.

Проверка попадания в трехмерном пространстве

Выше уже отмечалось, что самый простой способ проверить попадание в область, занимаемую 3D-моделью, – создать объект ModelUIElement3D и связать с ним обработчик события MouseDown. Но наличие ModelUIElement3D не является обязательным условием проверки попадания в 3D-модель.

Как и в случае двумерных объектов Visual, их трехмерные аналоги Visual3D принимают участие в проверке попадания в Visual. Чтобы провести такую проверку для Visual3D, нужно сначала получить событие проверки в двумерном элементе UIElement, содержащем 3D-сцену, например в родительском элементе Viewport3D:

```
<Viewport3D MouseDown="MouseDownHandler">
```

При вызове обработчика события можно будет произвести проверку попадания в точке события:

```
private void MouseDownHandler(object sender, MouseButtonEventArgs e)
{
  base.OnMouseLeftButtonDown(e);

  Viewport3D viewport = (Viewport3D)sender;
  Point location = e.GetPosition(viewport);

  HitTestResult result = VisualTreeHelper.HitTest(viewport, location);

  if (result != null && result.VisualHit is Visual3D)
  {
    MessageBox.Show("Hit Visual3D!");
  }
}
```

Разумеется, для Visual3D работает и обсуждавшийся в главе 15 перегруженный вариант метода VisualTreeHelper.HitTest, который вызывает делегаты обратного вызова для извещения о нескольких попаданиях. Чтобы запустить проверку попадания в область 3D-сцены, можно воспользоваться перегруженным вариантом VisualTreeHelper.HitTest, который принимает объекты Visual3D и HitTestParameters3D.

КОПНЕМ ГЛУБЖЕ

Получение детальной информации о попадании

`HitTestResult` – базовый класс, у которого есть несколько подклассов, например `PointHitTestResult` и `GeometryHitTestResult`. Конкретный тип возвращаемого значения `HitTestResult` зависит от того, как именно вы запускали проверку на попадание и чем она закончилась. Если проверялось попадание точки и выяснилось, что она принадлежит области, занимаемой 3D-сеткой, то можете привести `HitTestResult` к типу `RayMeshGeometry3DHitTestResult`. Этот тип содержит разнообразную детальную информацию о попадании.

Класс Viewport3D

Класс `Viewport3D` предлагает функциональность, противоположную `Viewport2DVisual3D`. Если `Viewport2DVisual3D` – это частный случай `Visual3D`, который дает возможность вкладывать элементы 2D внутрь 3D, то `Viewport3D` – подкласс двумерного класса `FrameworkElement`, позволяющий вкладывать элементы 3D в 2D.

Родителем `Viewport3D` всегда является двумерный элемент, например `Window` или `Grid`. Дочерними элементами `Viewport3D` являются объекты `Visual3D`. 3D-сцена, описанная с помощью потомков типа `Visual3D`, рисуется внутри прямоугольника, ограничивающего `Viewport3D`. Вид на 3D-сцену внутри `Viewport3D` регулируется свойством `Camera` элемента `Viewport3D`.

СОВЕТ

Размеры многих контейнероподобных элементов обычно устанавливаются на одном из этапов работы системы компоновки – этапе измерения (см. главу 21 «Компоновка с помощью нестандартных панелей») – так, чтобы они соответствовали размеру содержимого. Например, размер кнопки `Button` обычно выбирается исходя из размеров текста или другого отображаемого в ней содержимого. Элементы `Viewport3D` работают прямо противоположным образом: `Viewport3D` изменяет вид 3D-сцены так, чтобы она уместилась в его ограничивающий прямоугольник. По умолчанию свойство `ClipToBounds` объекта `Viewport3D` равно `false`, то есть его 3D-содержимое может выходить *за пределы* ограничивающего прямоугольника. Если требуется, чтобы содержимое `Viewport3D` оставалось в пределах ограничивающего прямоугольника, присвойте этому свойству значение `true`.

По этой причине необходимо задавать свойства `Width` и `Height` элемента `Viewport3D`, если только он уже не растянут на всю область системой компоновки. Если этого не сделать, то по умолчанию элемент `Viewport3D` получит размеры 0×0 и 3D-сцена не появится на экране.

Замечательным следствием того факта, что `Viewport3D` – полноценный элемент `FrameworkElement`, участвующий в компоновке, является возможность включить 3D-элементы практически в любое место приложения. На самом деле

дизайнеры могут с помощью стилей и шаблонов даже изменить подразумеваемый по умолчанию внешний вид элементов управления с интерактивным 3D-содержимым. На рис. 16.53 показан результат применения такого стиля к программе Photo Gallery из главы 7 «Структурирование и развертывание приложения». Отметим, что содержимое и фон на гранях кубов привязаны к шаблону кнопки `Button`. Если изменить содержимое или фон кнопки, то куб обновится в реальном времени! При нажатии на кнопку куб поворачивается.



Рис. 16.53. *Применение кнопок с кубическим стилем в программе Photo Gallery*

Листинг 16.8. Стиль кубической кнопки

```
<!-- Этот стиль изменяет внешний вид всех кнопок, делая их
     трехмерными кубами. Поскольку у элемента Viewport3D
     нет "естественного размера", то для кнопок необходимо
     задавать свойства Width и Height, если только
     они не растягиваются на весь объемлющий контейнер. -->
<Style TargetType="{x:Type Button}">
  <Setter Property="Template">
  <Setter.Value>
    <ControlTemplate>
    <ControlTemplate.Triggers>
      <!-- При нажатии кнопки повернуть куб -->
      <Trigger Property="Button.IsPressed" Value="true">
      <Trigger.EnterActions>
        <BeginStoryboard>
          <Storyboard TargetName="RotateY" TargetProperty="Angle">
            <DoubleAnimation Duration="0:0:1" From="0" To="360"
                             DecelerationRatio="1.0"/>
          </Storyboard>
        </BeginStoryboard>
      </Trigger.EnterActions>
      </Trigger>
    </ControlTemplate.Triggers>
    <Viewport3D>
    <Viewport3D.Camera>
      <PerspectiveCamera Position="2.9,2.65,2.9" LookDirection="-1,-1,-1"/>
```

```
 </Viewport3D.Camera>
<Viewport3D.Children>
   <ModelVisual3D x:Name="Light">
   <ModelVisual3D.Content>
     <DirectionalLight Direction="-0.3,-0.4,-0.5"/>
   </ModelVisual3D.Content>
   </ModelVisual3D>
   <ModelVisual3D x:Name="Cube">
   <ModelVisual3D.Transform>
     <RotateTransform3D>
       <RotateTransform3D.Rotation>
         <AxisAngleRotation3D x:Name="RotateY" Axis="0,1,0" Angle="0"/>
       </RotateTransform3D.Rotation>
     </RotateTransform3D>
   </ModelVisual3D.Transform>
   <ModelVisual3D.Content>
     <GeometryModel3D>
     <GeometryModel3D.Material>
       <DiffuseMaterial>
       <DiffuseMaterial.Brush>
         <!-- Используем VisualBrush для отображения исходных Background
              и Content кнопки на гранях куба -->
         <VisualBrush ViewportUnits="Absolute" Transform="1,0,0,-1,0,1">
         <VisualBrush.Visual>
           <Border Background="{Binding Path=Background,
             RelativeSource={RelativeSource TemplatedParent}}">
             <Label Content="{Binding Path=Content,
               RelativeSource={RelativeSource TemplatedParent}}"/>
           </Border>
         </VisualBrush.Visual>
         </VisualBrush>
       </DiffuseMaterial.Brush>
       </DiffuseMaterial>
     </GeometryModel3D.Material>
     <GeometryModel3D.Geometry>
       <MeshGeometry3D
         Positions="1,1,-1 1,-1,-1 -1,-1,-1 -1,1,-1 1,1,1 -1,1,1 -1,-1,1
            1,-1,1 1,1,-1 1,1,1 1,-1,1 1,-1,-1 1,-1,-1 1,-1,1 -1,-1,1 -1,-1,-1
            -1,-1,-1 -1,-1,1 -1,1,1 -1,1,-1 1,1,1 1,1,-1 -1,1,-1 -1,1,1"
         TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12
                          13 14 12 14 15 16 17 18 16 18 19 20 21 22 20 22 23"
         TextureCoordinates="0,1 0,0 1,0 1,1 1,1 0,1 0,-0 1,0 1,1
                             0,1 0,-0 1,0 1,0 1,1 0,1 0,-0 0,0 1,-0 1,1 0,1 1,-0
                             1,1 0,1 0,0"/>
     </GeometryModel3D.Geometry>
     </GeometryModel3D>
   </ModelVisual3D.Content>
   </ModelVisual3D>
 </Viewport3D.Children>
 </Viewport3D>
 </ControlTemplate>
```

```
    </Setter.Value>
    </Setter>
  </Style>
```

КОПНЕМ ГЛУБЖЕ

Класс Viewport3DVisual

В реализации класса `Viewport3D` используется класс `Viewport3DVisual`, чтобы перекинуть мост от двумерного к трехмерному визуальному дереву. Класс `Viewport3DVisual` – это в общем-то деталь реализации, но если вы захотите программировать на уровне `Visual`, а не `FrameworkElement`, то `Viewport3DVisual` – как раз тот подкласс двумерного `Visual`, который необходим для подключения к дереву `Visual3D`. В классе `Viewport3DVisual` определены такие же свойства, как в классе `Viewport3D`, и еще добавлено свойство `Viewport`, которое служит для задания границ области, где должна отображаться 3D-сцена, поскольку на уровне `Visual` понятия компоновки нет.

Преобразование двумерных и трехмерных систем координат

WPF предоставляет ряд служб для отображения трехмерных точек в двумерное пространство и наоборот. Это весьма полезно в приложениях, где необходимо взаимодействие между двумерным и трехмерным содержимым. Представьте, к примеру, программу для просмотра трехмерных молекул, в которых отдельные атомы снабжены двумерными текстовыми метками. Хотелось бы, чтобы метки рисовались поверх трехмерного содержимого, но при этом текст должен следовать за атомами при вращении модели. Службы преобразования системы координат как раз и позволяют добиться такого результата. Ниже мы рассмотрим API трехмерных преобразований и объясним, как с ним работать.

Метод Visual.TransformToAncestor

В классе `Visual` имеется метод `TransformToAncestor`, который возвращает объект `GeneralTransform2DTo3D`. Он полезен, когда объект `Visual` вложен в `Viewport2DVisual3D`. Возвращенный объект преобразует двумерные координаты вложенного объекта `Visual` в трехмерные координаты объекта `Visual3D`.

В листингах 16.9 и 16.10 точка (0,0), принадлежащая вложенной в `Viewport2DVisual3D` кнопке `Button`, отображается в трехмерное пространство, а в том месте, где в пространстве находится соответствующая точка `Point3D`, нарисован фиолетовый кубик. Когда больший куб поворачивается, меньший повторяет его вращение, потому что по мере поворачивания изменяется объект `GeneralTransform2DTo3D`. Результат показан на рис. 16.54.



***Рис. 16.54.** Отображение точки (0,0), начала Viewport2DVisual3D, в трехмерное пространство*

***Листинг 16.9.** MainWindow.xaml – большой куб с кнопками и маленьким фиолетовым кубиком*

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <Viewport3D Panel.ZIndex="0">
    <Viewport3D.Camera>
      <PerspectiveCamera Position="3,3,4" LookDirection="-1,-1,-1"
                         FieldOfView="60" />
    </Viewport3D.Camera>
    <Viewport3D.Children>
      <ModelVisual3D>
      <ModelVisual3D.Content>
        <DirectionalLight Direction="-0.3,-0.4,-0.5" />
      </ModelVisual3D.Content>
      </ModelVisual3D>
      <ModelVisual3D x:Name="Container">
        <Viewport2DVisual3D>
        <Viewport2DVisual3D.Transform>
          <Transform3DGroup>
            <TranslateTransform3D OffsetX="1.5" />
            <RotateTransform3D>
            <RotateTransform3D.Rotation>
              <AxisAngleRotation3D x:Name="rotationY" Axis="0,1,0" Angle="0" />
            </RotateTransform3D.Rotation>
            </RotateTransform3D>
          </Transform3DGroup>
        </Viewport2DVisual3D.Transform>
        <Viewport2DVisual3D.Geometry>
```

```
    <MeshGeometry3D Positions="1,1,-1 1,-1,-1 -1,-1,-1 -1,1,-1 1,1,1 -1,1,1
                               -1,-1,1 1,-1,1 1,1,1 -1,1,1 1,-1,1 1,-1,-1
                                1,-1,-1 1,-1,1 -1,-1,1 -1,-1,-1 -1,-1,-1
                               -1,-1,1 -1,1,1 -1,1,-1 1,1,1 1,1,-1 -1,1,-1
                               -1,1,1"
      TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12 13 14 12
                       14 15 16 17 18 16 18 19 20 21 22 20 22 23"
      TextureCoordinates="0,1 0,0 1,0 1,1 1,1 -0,1 0,-0 1,0 1,1 -0,1 0,-0
                          1,0 1,0 1,1 -0,1 0,-0 -0,0 1,-0 1,1 0,1 1,-0 1,1
                          0,1 -0,0" />
  </Viewport2DVisual3D.Geometry>
  <Viewport2DVisual3D.Material>
    <DiffuseMaterial Viewport2DVisual3D.IsVisualHostMaterial="True" />
  </Viewport2DVisual3D.Material>
    <Button Name="TestButton">
    <Button.RenderTransform>
      <ScaleTransform ScaleY="-1" />
    </Button.RenderTransform>
      Hello, 3D
    </Button>
  </Viewport2DVisual3D>
</ModelVisual3D>
<ModelUIElement3D>
<ModelUIElement3D.Transform>
  <Transform3DGroup>
    <ScaleTransform3D ScaleX="0.2" ScaleY="0.2" ScaleZ="0.2" />
    <TranslateTransform3D x:Name="cube_translation" />
  </Transform3DGroup>
</ModelUIElement3D.Transform>
<ModelUIElement3D.Model>
  <GeometryModel3D>
  <GeometryModel3D.Material>
    <DiffuseMaterial>
      <DiffuseMaterial.Brush>
        <SolidColorBrush Color="Purple" />
      </DiffuseMaterial.Brush>
    </DiffuseMaterial>
  </GeometryModel3D.Material>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D
      Positions="1,1,-1 1,-1,-1 -1,-1,-1 -1,1,-1 1,1,1 -1,1,1 -1,-1,1 1,-1,1
                 1,1,-1 1,1,1 1,-1,1 1,-1,-1 1,-1,-1 -1,-1,1 -1,-1,-1 -1,1,-1
                -1,-1,-1 -1,-1,1 -1,1,1 -1,1,-1 1,1,1 1,1,-1 -1,1,-1  -1,1,1"
      TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12 13 14 12
        14 15 16 17 18 16 18 19 20 21 22 20 22 23"
      TextureCoordinates="0,1 0,0 1,0 1,1 1,1 -0,1 0,-0 1,0 1,1 -0,1 0,-0
                          1,0 1,0 1,1 -0,1 0,-0 -0,0 1,-0 1,1 0,1 1,-0 1,1
                          0,1 -0,0" />
    </GeometryModel3D.Geometry>
    </GeometryModel3D>
</ModelUIElement3D.Model>
</ModelUIElement3D>
```

```
      </Viewport3D.Children>
      </Viewport3D>
    </Grid>

  <Window.Triggers>
    <EventTrigger RoutedEvent="Window.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetName="rotationY"
            Storyboard.TargetProperty="Angle"
            From="0" To="360" Duration="0:0:12"
            RepeatBehavior="Forever" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Window.Triggers>
  </Window>
```

Листинг 16.10. MainWindow.xaml.cs – застраничный код, который приводит маленький фиолетовый кубик в нужное положение

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Media.Media3D;

public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    CompositionTarget.Rendering += CompositionTarget_Rendering;
  }

  static TimeSpan lastRenderTime = new TimeSpan();

  void CompositionTarget_Rendering(object sender, EventArgs e)
  {
    // Это нужно делать только один раз в каждом кадре
    if (lastRenderTime == ((RenderingEventArgs)e).RenderingTime)
      return;

    lastRenderTime = ((RenderingEventArgs)e).RenderingTime;

    GeneralTransform2DTo3D transform =
      TestButton.TransformToAncestor(Container);
    Point3D point = transform.Transform(new Point(0, 0));

    cube_translation.OffsetX = point.X;
    cube_translation.OffsetY = point.Y;
    cube_translation.OffsetZ = point.Z;
  }
}
```

В листинге 16.10 событие CompositionTarget.Rendering применяется для того, чтобы выполнять преобразование координат только один раз в каждом кадре. Будьте внимательны при работе с этим событием, поскольку из-за структурных изменений на сцене оно может генерироваться несколько раз в одном кадре. В приведенном коде гарантия однократной обработки связана с тем фактом, что переданный экземпляр EventArgs на самом деле является объектом класса RenderingEventArgs, в котором имеется свойство RenderingTime (время начала визуализации).

Методы Visual3D.TransformToAncestor и Visual3D.TransformToDescendant

В классе Visual3D есть методы для обратного отображения трехмерного пространства на двумерное. Объект типа GeneralTransform3DTo2D, возвращаемый методом Visual3D.TransformToAncestor, преобразует координаты трехмерного пространства объекта Visual3D в координаты некоторого его двумерного родителя. Это особенно полезно, когда приложение отслеживает трехмерную точку на экране и рисует двумерное содержимое, позиция которого привязана к отслеживаемой точке.

В листингах 16.11 и 16.12 метод TransformToAncestor применяется для того, чтобы элементы TextBlock следовали за вершинами вращающегося куба, как показано на рис. 16.55.

Листинг 16.11. MainWindow.xaml – куб и текстовые блоки

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid Name="myGrid">
    <TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
   <TextBlock.RenderTransform>
     <TranslateTransform x:Name="t_000" />
   </TextBlock.RenderTransform>
     (-1,-1,-1)
    </TextBlock>
    <TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
    <TextBlock.RenderTransform>
      <TranslateTransform x:Name="t_001" />
    </TextBlock.RenderTransform>
      (-1,-1,1)
    </TextBlock>
    <TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
    <TextBlock.RenderTransform>
      <TranslateTransform x:Name="t_010" />
    </TextBlock.RenderTransform>
      (-1,1,-1)
    </TextBlock>
    <TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
    <TextBlock.RenderTransform>
      <TranslateTransform x:Name="t_011" />
```

```
</TextBlock.RenderTransform>
  (-1,1,1)
</TextBlock>
<TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
<TextBlock.RenderTransform>
  <TranslateTransform x:Name="t_100" />
</TextBlock.RenderTransform>
  (1,-1,-1)
</TextBlock>
<TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
<TextBlock.RenderTransform>
  <TranslateTransform x:Name="t_101" />
</TextBlock.RenderTransform>
  (1,-1,1)
</TextBlock>
<TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
<TextBlock.RenderTransform>
  <TranslateTransform x:Name="t_110" />
</TextBlock.RenderTransform>
  (1,1,-1)
</TextBlock>
<TextBlock Panel.ZIndex="1" IsHitTestVisible="False">
<TextBlock.RenderTransform>
  <TranslateTransform x:Name="t_111" />
</TextBlock.RenderTransform>
  (1,1,1)
</TextBlock>
<Viewport3D Panel.ZIndex="0">
<Viewport3D.Camera>
  <PerspectiveCamera Position="3,3,4" LookDirection="-1,-1,-1"
                     FieldOfView="60"/>
</Viewport3D.Camera>
<Viewport3D.Children>
  <ModelVisual3D>
  <ModelVisual3D.Content>
    <DirectionalLight Direction="-0.3,-0.4,-0.5" />
  </ModelVisual3D.Content>
  </ModelVisual3D>
  <ModelUIElement3D x:Name="Cube">
  <ModelUIElement3D.Transform>
    <RotateTransform3D>
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D x:Name="rotationY" Axis="0,1,0" Angle="0" />
    </RotateTransform3D.Rotation>
    </RotateTransform3D>
  </ModelUIElement3D.Transform>
  <ModelUIElement3D.Model>
    <GeometryModel3D x:Name="OB_Cube">
    <GeometryModel3D.Material>
      <DiffuseMaterial>
        <DiffuseMaterial.Brush>
          <SolidColorBrush Color="Orange" x:Name="CubeBrush" />
```

```
            </DiffuseMaterial.Brush>
          </DiffuseMaterial>
        </GeometryModel3D.Material>
        <GeometryModel3D.Geometry>
          <MeshGeometry3D x:Name="ME_Cube2"
            Positions="1,1,-1 1,-1,-1 -1,-1,-1 -1,1,-1 1,1,1 -1,1,1 -1,-1,1
                       1,-1,1 1,1,-1 1,1,1 1,-1,1 1,-1,-1 1,-1,-1 1,-1,1
                      -1,-1,1 -1,-1,-1 -1,-1,-1 -1,-1,1 -1,1,1 -1,1,-1 1,1,1
                       1,1,-1 -1,1,-1 -1,1,1"
            TriangleIndices="0 1 2 0 2 3 4 5 6 4 6 7 8 9 10 8 10 11 12 13 14
                             12 14 15 16 17 18 16 18 19 20 21 22 20 22 23"
            TextureCoordinates="0,1 0,0 1,0 1,1 1,1 -0,1 0,-0 1,0 1,1 -0,1
                                0,-0 1,0 1,0 1,1 -0,1 0,-0 -0,0 1,-0 1,1 0,1
                                1,-0 1,1 0,1 -0,0"/>
        </GeometryModel3D.Geometry>
        </GeometryModel3D>
      </ModelUIElement3D.Model>
      </ModelUIElement3D>
    </Viewport3D.Children>
    </Viewport3D>
  </Grid>

<Window.Triggers>
  <EventTrigger RoutedEvent="Window.Loaded">
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetName="rotationY"
          Storyboard.TargetProperty="Angle"
          From="0" To="360" Duration="0:0:12"
          RepeatBehavior="Forever" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Window.Triggers>
</Window>
```

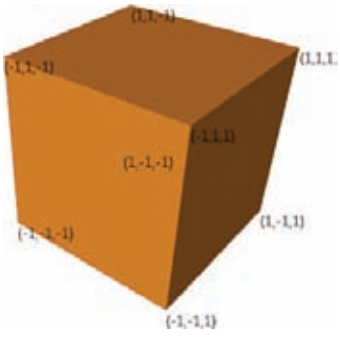


Рис. 16.55. *Отображение трехмерных точек в вершинах куба в двумерное пространство*

Листинг 16.12. MainWindow.xaml.cs – застраничный код, который обновляет положение текстовых блоков

```
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Media3D;

public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    CompositionTarget.Rendering += CompositionTarget_Rendering;
  }

  static TimeSpan lastRenderTime = new TimeSpan();

  void CompositionTarget_Rendering(object sender, EventArgs e)
  {
    // Это нужно делать только один раз в каждом кадре
    if (lastRenderTime == ((RenderingEventArgs)e).RenderingTime)
      return;
    lastRenderTime = ((RenderingEventArgs)e).RenderingTime;

    GeneralTransform3DTo2D transform = Cube.TransformToAncestor(myGrid);

    Point p = transform.Transform(new Point3D(-1, -1, -1));
    t_000.X = p.X; t_000.Y = p.Y;

    p = transform.Transform(new Point3D(-1, -1, 1));
    t_001.X = p.X; t_001.Y = p.Y;

    p = transform.Transform(new Point3D(-1, 1, -1));
    t_010.X = p.X; t_010.Y = p.Y;

    p = transform.Transform(new Point3D(-1, 1, 1));
    t_011.X = p.X; t_011.Y = p.Y;

    p = transform.Transform(new Point3D(1, -1, -1));
    t_100.X = p.X; t_100.Y = p.Y;

    p = transform.Transform(new Point3D(1, -1, 1));
    t_101.X = p.X; t_101.Y = p.Y;

    p = transform.Transform(new Point3D(1, 1, -1));
    t_110.X = p.X; t_110.Y = p.Y;

    p = transform.Transform(new Point3D(1, 1, 1));
    t_111.X = p.X; t_111.Y = p.Y;
  }
}
```

В каждом кадре код получает объект GeneralTransform3DTo2D, описывающий переход от координат куба к координатам его родительской сетки Grid. Этот объект используется для преобразования позиций всех восьми вершин куба в координаты экрана. Затем текстовые блоки размещаются в двумерном пространстве так, чтобы их позиции соответствовали преобразованным вершинам куба. Как и раньше, преобразование производится в обработчике события CompositionTarget.Rendering.

Нерассмотренными остались определенные в классе Visual3D метод преобразования TransformToDescendant и еще один перегруженный вариант метода TransformToAncestor: они возвращают объекты GeneralTransform3D, с помощью которых можно произвести преобразования между различными объектами Visual3D в иерархии трехмерных объектов.

Резюме

К этому моменту вы, вероятно, понимаете, что трехмерные API в WPF – это прямое обобщение хорошо знакомых двумерных API. Как показано в табл. 16.1 в начале главы, большинство 3D-типов естественно вытекают из классов, рассмотренных в предыдущих главах. Поэтому WPF является идеальной платформой для приложений, в которых необходимо сочетать трехмерную графику с двумерным пользовательским интерфейсом.

Хотя на первый взгляд 3D-средства, включенные в WPF, могут показаться элементарными, в них скрыта немалая мощь, обусловленная тесной интеграцией с другими частями платформы. 3D-преобразования в WPF могут быть привязаны к данным. На поверхности трехмерных объектов можно отображать видео, рисунки и даже двумерные элементы управления. Целую 3D-сцену можно использовать в качестве шаблона данных (DataTemplate) или элемента управления (ControlTemplate). И все это работает совместно с печатью, удаленным доступом и в контексте приложения с частичным доверием.

В этой главе мы говорили об API, специфичных для трехмерной графики, но это еще не все. В главе 19 «Интероперабельность с другими технологиями» мы рассмотрим класс D3DImage, обеспечивающий интероперабельность с Direct3D. Кроме того, многие из наиболее интересных средств 3D являются в то же время общими возможностями платформы. Когда в следующих главах мы будем обсуждать анимацию и мультимедиа, не забывайте, что все это применимо и к трехмерным объектам.

- Анимация в процедурном коде
- Анимация в XAML-коде
- Анимация с опорными кадрами
- Переходные функции
- Анимация и менеджер визуальных состояний

17

Анимация

Средства анимации в WPF позволяют безо всякого труда включать динамические эффекты в приложения и компоненты. Однако это тот механизм, которым легко злоупотребить! Поэтому оставьте мысли о будущих приложениях, изобилующих прыгающими кнопками и вращающимися меню, а подумайте лучше, как применить анимацию с пользой. Без сомнения, вам попадались сайты с продуманной анимацией на базе Adobe Flash, оставляющие приятное впечатление, и приходилось смотреть по телевизору репортажи со спортивных матчей либо новости с бегущей строкой или анимированными переходами. В пользовательские интерфейсы iPhone, iPad, Windows-телефонов, Xbox и Windows 7 (и это только малая часть списка) анимация вплетена очень органично. Конечно, анимация подходит не для любой программы, но во многих случаях ее разумное применение может оказаться успешным.

При проектировании с помощью таких инструментов, как Microsoft Expression Blend, поддержка анимации в WPF может дать возможности, сравнимые с Adobe Flash. Но, поскольку это часть платформы WPF и применяемые API сравнительно просты, можно без труда создать целый спектр анимаций и без помощи инструментов. В этой главе мы продемонстрируем несколько разных подходов к анимации, для которых не требуется ничего, кроме коротеньких фрагментов кода на C# или XAML.

Мы начнем эту главу с изучения базовых классов анимации в WPF и способов их использования в процедурном коде. Затем посмотрим, как те же самые классы применяются в XAML, и попутно ознакомимся с дополнительными концепциями. Рассмотрев оба подхода, мы перейдем к более действенным формам анимации с использованием опорных кадров и/или переходных функций. И наконец, более подробно остановимся на использовании анимаций совместно с менеджером визуальных состояний (VSM).

Анимация в процедурном коде

Большинство людей, думая об анимации, представляют себе механизм мультипликации, когда движение имитируется быстрой сменой статических изображений. В WPF у анимации есть более точное определение: изменение значения свойства во времени. Это может быть как-то связано с движением, например, когда мы заставляем элемент вырастать, увеличивая его ширину `Width`, но динамическое изменение цвета – тоже анимация.

Реализовать анимацию можно с помощью специальной поддержки, обсуждаемой в этой главе, и даже без особых усилий – благодаря модели графики с сохранением, принятой в WPF. В этом разделе мы для начала рассмотрим имеющиеся средства для выполнения анимации вручную. А затем ознакомимся с классами WPF, которые почти всю работу берут на себя.

Выполнение анимации «вручную»

Классический способ реализации анимации – сконфигурировать таймер и создать функцию обратного вызова, которая будет периодически вызываться при каждом срабатывании таймера. Внутри этой функции можно вручную обновить целевое свойство (проделав несложные вычисления для получения нового значения в зависимости от истекшего времени) – и так до тех пор, пока не будет достигнуто нужное конечное значение. В этот момент мы останавливаем таймер и/или удаляем обработчик события.

Разумеется, ничто не мешает вам применить этот классический подход и в WPF. В состав WPF даже входит класс `DispatcherTimer`, удобный для реализации подобной схемы. Частота срабатывания таймера задается путем установки свойства `Interval`, а обработчик присоединяется к событию `Tick`.

Хотя этот подход может быть хорошо знаком программистам Windows-приложений, выполнять анимацию с помощью таймера не рекомендуется. Таймеры не синхронизированы ни с частотой вертикальной развертки монитора, ни с механизмом визуализации WPF.

Вместо того чтобы реализовывать анимацию на базе таймера, можно организовать покадровую анимацию, присоединив обработчик к статическому событию `Rendering` из класса `System.Windows.Media.CompositionTarget`. Оно генерируется не с запрограммированным интервалом, а *один раз в кадре* в момент между завершением компоновки и началом визуализации. (Это событие, аналогичное событию `enterFrame` в анимациях Adobe Flash, использовалось в двух примерах в конце предыдущей главы.)

Применение покадровой анимации на основе события `Rendering` рекомендуется как альтернатива не только подходу на основе таймера, но даже классам анимации, составляющим основной предмет рассмотрения в этой главе, в тех случаях, когда имеются сотни объектов, нуждающихся в точной анимации. Например, для обнаружения столкновений и других анимаций в физических задачах следует применять именно эту схему. Она же обычно используется

в задачах анимирования перехода элементов панели из одной компоновки в другую. Событие Rendering в общем случае дает наилучшую производительность и допускает максимальную свободу настройки (поскольку в его обработчике можно написать произвольный код), хотя приходится идти на некоторые компромиссы. При обычных условиях WPF визуализирует кадры, только когда становится недействительной какая-то часть пользовательского интерфейса. Но если к событию Rendering присоединен обработчик, то кадры визуализируются непрерывно. Поэтому применять событие Rendering лучше всего для коротких анимаций.

КОПНЕМ ГЛУБЖЕ

Разница между DispatcherTimer и другими таймерами в .NET

Основное различие между классом DispatcherTimer и другими таймерами, например System.Threading.Timer или System.Timers.Timer, заключается в том, что обработчики событий от DispatcherTimer вызываются в потоке пользовательского интерфейса. Для WPF-приложений это существенно, потому что позволяет манипулировать элементами UIElement внутри обработчика, не заботясь о многопоточности. При работе с другими таймерами логику обновления элементов необходимо выносить в отдельную функцию и вызывать ее в потоке ГИП с помощью класса Dispatcher. Например:

```
void Callback(object sender, EventArgs e)
{
  // Вызываем метод DoTheRealWork в потоке ГИП:
  this.Dispatcher.Invoke(DispatcherPriority.Normal,
    new TimerDispatcherDelegate(DoTheRealWork));
}
```

Метод Dispatcher.Invoke выполняет синхронный вызов. Можно произвести и асинхронный вызов, воспользовавшись методом Dispatcher.BeginInvoke. А вот при работе с таймером DispatcherTimer метод обратного вызова может выглядеть так:

```
void Callback(object sender, EventArgs e)
{
  // Обновляем свойство непосредственно в потоке ГИП
}
```

По умолчанию обратные вызовы DispatcherTimer обрабатываются с приоритетом DispatcherPriority, равным Background, но при конструировании объекта DispatcherTimer можно задать и явное значение DispatcherPriority.

Введение в классы анимации

Хотя анимация на основе события CompositionTarget.Rendering – вполне разумный и приемлемый подход, проектировщики WPF хотели сделать процесс анимации более простым и декларативным. Поэтому в пространстве имен System.Windows.Media.Animation имеется немало классов, позволяющих описать

и применить анимацию вообще без написания кода вручную. Они чрезвычайно полезны, когда вы заранее знаете, как должна вести себя анимация на протяжении длительного промежутка времени.

У этих классов анимации есть два существенных аспекта:

- **Они могут изменять только значение свойства зависимости.** Поэтому определение анимации в WPF оказывается чуть более ограничительным, чем было сформулировано выше, если только не применяется подход на основе таймера `DispatcherTimer` или события `Rendering`.
- **Они выполняют анимацию, «не зависящую от разрешающей способности аппаратуры формирования времени».** Как графика в WPF не зависит от разрешающей способности устройства вывода, так и анимация на основе классов анимации в WPF не ускоряется при увеличении тактовой частоты кварцевого генератора; она просто работает более плавно! WPF может изменять частоту кадров исходя из различных условий, а вам как разработчику анимации до этого нет дела.

В пространстве имен `System.Windows.Media.Animation` имеется много похожих классов, потому что различные типы данных анимируются разными классами. Например, если требуется изменять во времени значение свойства зависимости элемента, имеющее тип `double`, то можно воспользоваться объектом класса `DoubleAnimation`. А если нужно изменять значение свойства зависимости `Thickness`, то понадобится класс `ThicknessAnimation`. В WPF встроены классы анимации для 22 разных типов данных; все они перечислены в табл. 17.1.

Таблица 17.1. Типы данных, для которых имеются встроенные классы анимации

Примитивные типы данных .NET	Типы данных WPF
`Boolean`	`Thickness`
`Byte`	`Color`
`Char`	`Size`
`Decimal`	`Rect`
`Int16`	`Point`
`Int32`	`Point3D`
`Int64`	`Vector`
`Single`	`Vector3D`
`Double`	`Rotation3D`
`String`	`Matrix`
`Object`	`Quaternion`

КОПНЕМ ГЛУБЖЕ

Классы анимации и отсутствие универсальных классов

В пространстве имен `System.Windows.Media.Animation` есть следующие классы:

- 22 класса с именами вида *XXX*`AnimationBase`
- 17 классов с именами вида *XXX*`Animation`
- 22 класса с именами вида *XXX*`AnimationUsingKeyFrames`
- 22 класса с именами вида *XXX*`KeyFrameCollection`
- 22 класса с именами вида *XXX*`KeyFrame`
- 22 класса с именами вида `Discrete`*XXX*`KeyFrame`
- 17 классов с именами вида `Linear`*XXX*`KeyFrame`
- 17 классов с именами вида `Spline`*XXX*`KeyFrame`
- 17 классов с именами вида `Easing`*XXX*`KeyFrame`
- 3 класса с именами вида *XXX*`AnimationUsingPath`

(Здесь *XXX* представляет тип данных из табл. 17.1.)

Классы в каждой из десяти групп почти идентичны друг другу, а в целом 10 разных концепций вылились аж в 181 класс! Когда человек, знакомый с .NET, сталкивается с подобной архитектурой, первая реакция часто оказывается такой: «А почему разработчики WPF не воспользовались универсальными типами?» Иначе говоря, почему бы не завести один класс `Animation<T>`, который позволит анимировать значения типа `double` с помощью конкретизации `Animation<double>`, значения типа `Thickness` – с помощью конкретизации `Animation<Thickness>` и т. д.?

Очевидная (но не слишком убедительная) причина заключается в том, что до появления версии XAML2009 полноценная поддержка универсальных типов в языке XAML еще отсутствовала. Но, даже если бы универсальные типы поддерживались в полной мере, все равно у этих классов есть особенности, мешающие их универсализации. Например, наличие класса `Animation<T>` означало бы, что его можно конкретизировать любым типом данных, в частности построить тип `Animation<Window>`. Но такая анимация не поддерживается и не существует ограничения, которое позволило бы эффективно выразить, какие именно типы-параметры поддерживаются данным универсальным типом.

Использование анимации

Чтобы понять, как работают классы анимации, рассмотрим тип данных `double`. Понять принцип анимирования `double` довольно просто, и это весьма популярный подход, поскольку в разных элементах встречается немало полезных свойств зависимости типа `double`.

Допустим, нам требуется увеличить свойство `Width` кнопки `Button` с `50` до `100`. Для целей демонстрации мы можем поместить кнопку в простое окно с панелью `Canvas`:

```
<Window x:Class="Window1" Title="Animation" Width="300" Height="300"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Canvas>
    <Button x:Name="b">OK</Button>
  </Canvas>
</Window>
```

В застраничном файле идею анимации свойства `Width` от `50` до `100` очень легко выразить с помощью класса `DoubleAnimation`:

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Animation;

public partial class Window1 : Window
{
  public Window1()
  {
    InitializeComponent();

    // Определяем анимацию
    DoubleAnimation a = new DoubleAnimation();
    a.From = 50;
    a.To = 100;

    // Начинаем анимацию
    b.BeginAnimation(Button.WidthProperty, a);
  }
}
```

Объект `DoubleAnimation` содержит начальное и конечное значения свойства типа `double` — *любого* свойства типа `double`. Затем вызывается метод `BeginAnimation` класса `Button`, который ассоциирует анимацию со свойством зависимости `Width` *и* запускает ее. Если вы сейчас откомпилируете и выполните этот код, то увидите, что ширина кнопки плавно увеличивается от `50` до `100` в течение одной секунды.

В классах анимации помимо `From` и `To` есть и много других свойств, которые позволяют разными интересными способами настраивать поведение. Мы рассмотрим эти свойства в данном разделе. Есть в классах анимации и ряд простых событий, например событие `Completed`, которое генерируется, как только целевое свойство достигнет конечного значения.

Линейная интерполяция

Важно понимать, что значение свойства типа `double` плавно изменяется во времени благодаря механизму *линейной интерполяции*. (В противном случае анимация ничем не отличалась бы от простой установки свойства!) Иначе говоря, в примере односекундной анимации значение `Width` становится равным `55` по истечении 0,1 с (5% выполнения как по времени, так и по значению), `75` по истечении 0,5 с (50% выполнения) и т. д. На внутреннем уровне имеется функ-

ция, которая вызывается с регулярными интервалами и выполняет вычисления, которые производили бы и вы сами, если бы реализовывали анимацию менее продвинутым способом. Именно поэтому большая часть типов данных в табл. 17.1 – числовые. (Нечисловые типы данных, такие как `Boolean` и `String`, рассматриваются в этой главе ниже.)

Для того чтобы понять, как получить с помощью анимации желаемый результат, требуется немного попрактиковаться. Приведем несколько примеров.

- Если требуется организовать постепенное появление элемента на экране, то не имеет смысла анимировать его свойство `Visibility`, потому что между `Hidden` и `Visible` нет промежуточных значений. Вместо этого следует анимировать свойство `Opacity` типа `double`, переходя от `0` к `1`.
- Если требуется, чтобы элемент, расположенный внутри сетки `Grid`, скользил по экрану, то *можно было бы* анимировать его присоединенное свойство `Grid.Column` с помощью класса `Int32Animation`, но переходы из одного столбца в другой оказались бы прерывистыми. Вместо этого следует ассоциировать с элементом преобразование `TranslateTransform` в режиме `RenderTransform`, а затем анимировать его свойство `X` (типа `double`) с помощью класса `DoubleAnimation`.
- Анимация свойства `Width` столбца сетки `Grid` (используемая в примере раздела «А теперь все вместе: создание сворачиваемой, стыкуемой, изменяющей размер панели, как в Visual Studio» в конце главы 5 «Компоновка с помощью панелей») не вполне очевидна, потому что свойство `ColumnDefinition.Width` определено как структура `GridLength`, для которой нет встроенного класса анимации. Вместо этого можно анимировать свойства `MinWidth` и/или `MaxWidth` объекта `ColumnDefinition`, оба типа `double`, или присвоить свойству `Width` этого объекта значение `Auto`, а потом вставить в этот столбец элемент, чье свойство `Width` вы собираетесь анимировать.

Повторное использование анимаций

В показанном выше коде мы присоединили анимацию к кнопке с помощью вызова метода `BeginAnimation`. Метод `BeginAnimation` можно вызывать многократно с целью применить одну и ту же анимацию к разным элементам или даже к различным свойствам одного элемента. Например, если добавить следующую строку в предыдущий застраничный код, то свойство `Height` окна `Window` будет изменяться синхронно со свойством `Width` кнопки:

```
this.BeginAnimation(Window.HeightProperty, a);
```

Результат такой модификации показан на рис. 17.1. (Прежде чем фыркать при мысли о расширяющемся окне, подумайте о том, что для такого механизма вполне могут найтись полезные применения – например, чтобы увеличить размер диалогового окна, когда пользователь раскрывает находящийся в нем элемент `Expander`. А простенькая анимация выглядит приятнее внезапного перехода к новому размеру.)

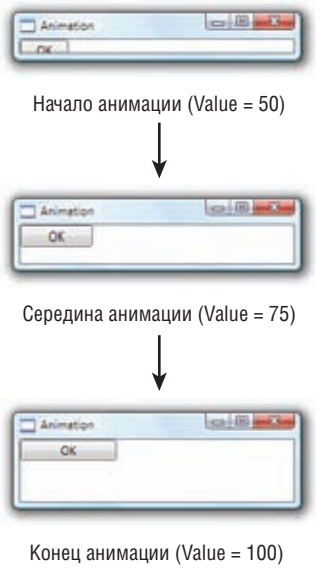


Рис. 17.1. *Одна и та же анимация DoubleAnimation одновременно увеличивает ширину кнопки и высоту окна с 50 до 100*

Управление продолжительностью

Продолжительность рассматривавшейся до сих пор простой анимации `DoubleAnimation` по умолчанию составляла 1 с, но ее можно изменить с помощью свойства `Duration`:

```
DoubleAnimation a = new DoubleAnimation();
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

ПРЕДУПРЕЖДЕНИЕ

Будьте внимательны, когда задаете Duration или TimeSpan в виде строки!

Метод `TimeSpan.Parse`, который используется также конвертером типа `Duration` в интересах XAML, принимает и сокращенные формы записи, так что задавать все компоненты строки *дни.часы:минуты:секунды.доли_секунды* необязательно. Однако поведение отличается от интуитивно ожидаемого. Строка `"2"` означает 2 *дня*, а не 2 секунды! Строка `"2.5"` – это 2 *дня* и 5 *часов*! А строка `"0:2"` означает 2 *минуты*. Учитывая, что анимация, как правило, продолжается не дольше нескольких секунд, обычно применяется синтаксис *часы:минуты:секунды* или *часы:минуты:секунды.доли_секунды*. Таким образом, 2 секунды можно выразить в виде `"0:0:2"`, а полсекунды – в виде `"0:0:0.5"` или `"0:0:.5"`.

Теперь анимация на рис. 17.1 занимает 5 с вместо одной. Обычно объект Duration создают с помощью стандартного класса TimeSpan, который входил в .NET Framework начиная с самой первой версии. Благодаря статическому методу TimeSpan.Parse длительность промежутка времени можно задавать в виде строки формата *дни.часы:минуты:секунды.доли_секунды*.

КОПНЕМ ГЛУБЖЕ

Разница между Duration и TimeSpan

Причина, по которой в WPF определен дополнительный тип Duration помимо TimeSpan, заключается в том, что в Duration есть два специальных значения, которые невозможно выразить с помощью TimeSpan: Duration.Automatic и Duration.Forever. Оба они предназначены для использования в таких более сложных классах, как Storyboard (см. ниже).

Automatic – значение по умолчанию для свойства Duration в любом классе анимации, оно эквивалентно промежутку TimeSpan продолжительностью 1 с. Значение Forever не имеет смысла для такой простой анимации, как DoubleAnimation, потому что в этом случае начальное значение оставалось бы неизменным бесконечно долго. В WPF нет способа интерполировать значения между текущим и заключительным моментами времени!

Гибкое задание From и To

Еще до того как анимация на рис. 17.1 изменит ширину Width кнопки и высоту Height окна от 50 до 100, эти свойства должны скачком принять значение 50 вместо текущего значения. Это никак не отражается на анимации, которая начинается в момент показа окна. Но если вызывать метод BeginAnimation в ответ на некоторое событие, то такой скачкообразный переход будет резать глаз.

Это можно исправить, задав в качестве To текущее значение Width/Height вместо 50, но тогда понадобилось бы расщепить анимацию на два разных объекта – один анимирует свойство ActualWidth кнопки Button до 100, другой – свойство ActualHeight окна Window до 100. К счастью, есть альтернатива. Задавать поле From объекта анимации необязательно. Если оно опущено, то анимация начинается с текущего значения целевого свойства, каким бы оно ни было. Например, вы можете попробовать изменить предыдущую анимацию, как показано ниже:

```
DoubleAnimation a = new DoubleAnimation();
// Закомментировано: a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

Возможно, вы думаете, что ширина Width кнопки при этом изменится от значения по умолчанию (столько, сколько необходимо для содержимого "OK", плюс небольшой отступ) до 100 за время 5 с. На самом деле возникает исключение AnimationException со следующим сообщением во внутреннем исключении:

```
'System.Windows.Media.Animation.DoubleAnimation' cannot use default
origin value of 'NaN'.
```

Поскольку свойство Width не было установлено, оно по умолчанию имеет значение NaN. А анимация не может интерполировать значения между NaN и 100! Более того, применение анимации к свойству ActualWidth (которое равно истинной ширине кнопки, а не NaN) вместо Width тоже не проходит, потому что это свойство допускает только чтение и не является свойством зависимости. Таким образом, необходимо где-то предварительно явно установить ширину Width целевой кнопки. Например:

```
<!-- Теперь анимация может увеличить ширину этой кнопки
     без задания From: -->
<Button x:Name="b" Width="20">OK</Button>
```

Анимация окна Window на рис. 17.1 работает без задания From, не требуя дополнительных ухищрений, потому что для его высоты Height уже задано значение 300. Однако отметим, что та же самая анимация теперь увеличивает ширину кнопки от 20 до 100, зато высоту окна *уменьшает* от 300 до 100! Аналогично, если бы ширина Width кнопки была задана большей 100, то в результате анимации она уменьшилась бы до 100.

СОВЕТ

Опускать явное задание From важно для получения плавной анимации, особенно когда анимация начинается в ответ на повторяющееся действие пользователя. Например, если анимация приводит к увеличению ширины кнопки от 50 до 100 при нажатии кнопки, то при серии быстрых нажатий ширина каждый раз возвращалась бы к начальному значению 50. Если же задание From опустить, то последующее нажатие просто продолжило бы анимацию, начиная с текущего значения, и визуально все происходило бы плавно. Аналогично, если элемент должен увеличиваться по событию MouseEnter и уменьшаться по событию MouseLeave, то, опустив задание From в обоих анимациях, мы предотвратим скачкообразное изменение размера элемента в случае, когда указатель мыши покидает его еще до завершения увеличения или повторно входит до завершения уменьшения.

На самом деле даже поле To задавать необязательно! Если к предыдущей кнопке применить показанную ниже анимацию, то ширина изменится от 50 до 20 (явно заданной при определении) за 5 с:

```
DoubleAnimation a = new DoubleAnimation();
a.From = 50;
// Закомментировано: a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

В каждом классе анимации имеется поле By, которое можно задавать вместо поля To. Следующая запись означает «анимировать значение *на* 100 (до 150)», а не «анимировать значение *до* 100»:

```
DoubleAnimation a = new DoubleAnimation();
a.From = 50;
a.By = 100; // Эквивалентно a.To = 50 + 100;
```

Использование By без From – это гибкий способ выразить такую идею: «анимировать значение от текущего до большего на 100 единиц».

```
DoubleAnimation a = new DoubleAnimation();
a.By = 100; // Эквивалентно a.To = currentValue + 100;
```

Отрицательные величины также поддерживаются и означают уменьшение текущего значения:

```
DoubleAnimation a = new DoubleAnimation();
a.By = -100; // Эквивалентно a.To = currentValue - 100;
```

Простые приемы работы с анимацией

До сих пор мы видели лишь основные свойства классов анимации: From, To, Duration, By. Но есть и много других свойств, позволяющих изменить поведение анимации разными интересными способами.

Подобно свойству By, какие-то из них могут показаться нелепыми излишествами, которые можно было бы тривиально имитировать, написав простенький код. Это так, но основная цель всех этих свойств – дать возможность выполнять такие приемы прямо из XAML.

Свойство BeginTime

Если вы не хотите, чтобы анимация начиналась сразу после вызова метода BeginAnimation, то можете вставить интервал задержки, присвоив свойству BeginTime объект типа TimeSpan:

```
DoubleAnimation a = new DoubleAnimation();
// Задержать анимацию на 5 секунд:
a.BeginTime = TimeSpan.Parse("0:0:5");
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

Это свойство в основном применяется в последовательности анимаций, следующих одна за другой. Можно даже присвоить BeginTime отрицательное значение:

```
DoubleAnimation a = new DoubleAnimation();
// Начать анимацию с середины:
a.BeginTime = TimeSpan.Parse("-0:0:2.5");
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

В этом случае анимация начинается немедленно, но с момента 2,5 с на временной шкале (как если бы начало анимации было отодвинуто на 2,5 с в про-

шлое). Поэтому эта анимация эквивалентна такой, где From равно 75, To — 100, а Duration составляет 2,5 с.

Отметим, что свойство BeginTime имеет тип Nullable<TimeSpan>, а не Duration, поскольку дополнительные выразительные возможности Duration здесь не нужны. (Какой смысл задавать для BeginTime значение Forever?!)

СОВЕТ

В коде этого раздела используется метод TimeSpan.Parse, поскольку он поддерживает тот же синтаксис, что и конвертер типа TimeSpan (и, следовательно, тот же, который применяется в XAML). В процедурном коде можно использовать и другие способы инициализации TimeSpan, например статические методы FromSeconds или FromMilliseconds.

Свойство SpeedRatio

Свойство SpeedRatio — это коэффициент, применяемый к Duration. По умолчанию оно имеет значение 1, но может быть равно любому числу типа double, большему 0:

```
DoubleAnimation a = new DoubleAnimation();
a.BeginTime = TimeSpan.Parse("0:0:5");
// Удвоить скорость анимации:
a.SpeedRatio = 2;
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

Значение, меньшее 1, означает замедление анимации, а большее 1 — ускорение. SpeedRatio не влияет на BeginTime; показанная выше анимация все равно начинается с 5-секундной задержкой, но переход от 50 к 100 занимает всего 2,5 с вместо 5.

Свойство AutoReverse

Если свойство AutoReverse равно true, то анимация «воспроизводится в обратном направлении» сразу по завершении. Обратная анимация занимает столько же времени, сколько прямая. Например, следующая анимация изменяет значение от 50 до 100 в первые 5 с и от 100 до 50 в последующие 5 с:

```
DoubleAnimation a = new DoubleAnimation();
a.AutoReverse = true;
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

Свойство SpeedRatio влияет на скорость анимации в *обоих* направлениях. Поэтому если в предыдущей анимации установить SpeedRatio = 2, то вся анимация займет 5 с, а если сделать SpeedRatio = 0.5, то на анимацию уйдет 20 с.

Отметим, что задержка, заданная с помощью BeginTime, не приводит к задержке обратной анимации; она всегда начинается сразу же после завершения прямой.

Свойство RepeatBehavior

Свойство RepeatBehavior позволяет установить одно из трех поведений:

- Автоматически повторять анимацию заданное число раз вне зависимости от ее продолжительности
- Автоматически повторять анимацию, пока не истечет заданное время
- Прервать анимацию досрочно

Для повторения анимации заданное число раз следует сконструировать объект RepeatBehavior с параметром типа double:

```
DoubleAnimation a = new DoubleAnimation();
// Выполнить анимацию два раза подряд:
a.RepeatBehavior = new RepeatBehavior(2);
a.AutoReverse = true;
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

Если AutoReverse равно true, то будет повторена и обратная анимация. Так что показанная выше анимация в течение 20 с изменяет значение следующим образом: 50 — 100 — 50 — 100 — 50. Если BeginTime вводит задержку, то задержка *не* повторяется. Поскольку количество итераций задается числом типа double, то можно воспроизводить анимацию неполное число раз.

СОВЕТ

Анимацию можно повторять до бесконечности, если присвоить свойству RepeatBehavior значение RepeatBehavior.Forever:

```
a.RepeatBehavior = RepeatBehavior.Forever;
```

Чтобы повторять анимацию до истечения заданного времени, следует сконструировать объект RepeatBehavior с параметром типа TimeSpan, а не double. Следующая анимация эквивалентна предыдущей:

```
DoubleAnimation a = new DoubleAnimation();
// Выполнить анимацию два раза подряд:
a.RepeatBehavior = new RepeatBehavior(TimeSpan.Parse("0:0:20"));
a.AutoReverse = true;
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

Для завершения двух полных циклов анимации необходимо 20 с, поскольку свойство AutoReverse равно true. Отметим, что при задании SpeedRatio проме-

жуток времени TimeSpan, указанный для поведения RepeatBehavior, не масштабируется; если в предыдущем примере установить SpeedRatio = 2, то за те же 20 с будет выполнено четыре цикла анимации вместо двух.

Чтобы использовать свойство RepeatBehavior как способ досрочного прекращения анимации, достаточно сконструировать объект, указав в качестве параметра TimeSpan длительность, меньшую естественной длительности анимации. В следующем примере анимация изменяет значение от 50 до 75 за 2,5 с:

```
DoubleAnimation a = new DoubleAnimation();
// Остановить анимацию посередине:
a.RepeatBehavior = new RepeatBehavior(TimeSpan.Parse("0:0:2.5"));
a.From = 50;
a.To = 100;
a.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

КОПНЕМ ГЛУБЖЕ

Полная продолжительность анимации на временной шкале

При том разнообразии корректировок, которые можно применить к анимации с помощью свойств BeginTime, SpeedRatio, AutoReverse и RepeatBehavior, бывает трудно понять, сколько же времени займет анимация с момента запуска. Свойство Duration уж точно не годится для получения истинной продолжительности! На самом деле истинная продолжительность анимации описывается следующей формулой:

$$\text{Полная продолжительность на временной шкале} = \text{BeginTime} + \left(\frac{\text{Duration} * (\text{AutoReverse} \ ? \ 2 : 1)}{\text{SpeedRatio}} * \text{RepeatBehavior} \right)$$

Она применима, если объект RepeatBehavior сконструирован с параметром типа double (или оставлено подразумеваемое по умолчанию значение этого свойства – 1). Если объект RepeatBehavior сконструирован с параметром типа TimeSpan, то полная продолжительность анимации на временной шкале просто равна величине RepeatBehavior плюс значение BeginTime.

Свойства AccelerationRatio, DecelerationRatio и EasingFunction

По умолчанию анимация обновляет целевое значение линейно. Если прошло 25% времени, отведенного на анимацию, значит, значение стало на 25% ближе к конечному. Но свойства AccelerationRatio и DecelerationRatio позволяют сделать интерполяцию нелинейной. Это распространенный способ заставить элемент «притянуться» к конечному значению, как на пружине, сделав анимацию более жизненной.

Обоим свойствам можно присвоить значение типа double из диапазона от 0 до 1 (по умолчанию подразумевается 0). Значение AccelerationRatio представляет процентную долю времени, в течение которого целевое значение должно ускоряться от нулевой до максимальной скорости. Аналогично DecelerationRatio

представляет процентную долю времени, в течение которого целевое значение должно замедляться от максимальной до нулевой скорости. Таким образом, сумма обоих свойств должна быть меньше или равна единице (100%).

На рис. 17.2 показано, что означают различные величины AccelerationRatio и DecelerationRatio на практике.

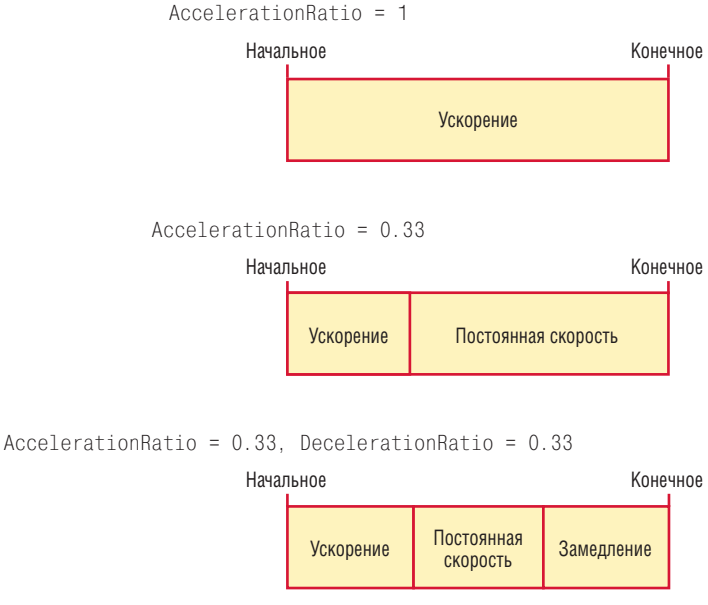


***Рис. 17.2.** Как AccelerationRatio и DecelerationRatio влияют на скорость изменения значения на пути от начального до конечного*

Начиная с версии WPF 4 у анимации появилось также свойство EasingFunction, значением которого может быть любой объект, реализующий интерфейс IEasingFunction. Такие объекты управляют ускорением и замедлением произвольно сложным способом. В состав WPF входит 11 классов, реализующих IEasingFunction, и написать свой собственный не составляет труда (если вы понимаете, как математически описывается желаемый эффект). Дополнительные сведения см. в разделе «Переходные функции» ниже.

Свойства IsAdditive и IsCumulative

Если свойство IsAdditive равно true, то текущее (получающееся после анимации) значение целевого свойства неявно прибавляется к свойствам From и To анимации. Это не отражается на повторении анимации с помощью свойства RepeatBehavior, а применяется к анимации, вручную запущенной впоследствии. По существу, это означает, что анимация будет продолжена с того значения свойства зависимости, которое оказалось после анимации, а не с того, какое оно имело до ее начала.

Свойство IsCumulative похоже на IsAdditive, но работает совместно с RepeatBehavior (и *только* с RepeatBehavior). Например, если RepeatBehavior трижды повторяет анимацию от 50 до 100, то по умолчанию значение изменится от 50 до 100, затем скачком вернется к 50, снова изменится до 100, еще раз перейдет к 50 и наконец остановится на 100. Если же IsCumulative равно true, то на протяжении того же времени значение будет плавно изменяться от 50 до 200. Если взять ту же самую анимацию и установить для AutoReverse значение true, то вы увидите, что значение изменяется в следующем порядке: 50 – 100 – 50 – скачкообразный переход к 100 – 150 – 100 – скачкообразный переход к 150 – 200 – 150.

Свойство FillBehavior

По умолчанию, когда анимация завершается, целевое свойство сохраняет значение, достигнутое в конце анимации, если оно не будет изменено каким-то другим механизмом. Обычно это именно то, что нужно, но если требуется, чтобы целевое свойство скачком вернулось к значению до начала анимации, то можно присвоить свойству FillBehavior значение Stop (вместо подразумеваемого по умолчанию HoldEnd).

Анимация в XAML-коде

В классах анимации есть уйма полезных свойств, и легко представить себе их задание в XAML-разметке. Например:

```
<DoubleAnimation From="50" To="100" Duration="0:0:5" AutoReverse="True"/>
```

Но куда поместить такой объект? Один из вариантов – определить его в качестве ресурса, чтобы к нему можно было обратиться из процедурного кода и в нужный момент вызвать метод BeginAnimation. Можно будет даже подкорректировать некоторые свойства анимации, чтобы получать разные эффекты в зависимости от условий, складывающихся в приложении.

Неудивительно, однако, что WPF поддерживает запуск анимации на чистом XAML. Ключ к этому дает менеджер визуальных состояний, а также триггеры, которые могут содержать не только элементы Setter, но и *действия*.

Действия доступны триггерам всех трех типов, но в этой главе мы сосредоточимся только на триггерах событий, потому что они ничего *кроме* действий содержать и не могут. Менеджер визуальных состояний мы рассмотрим в конце главы.

Триггеры событий и раскадровки

В главе 3 «Основные принципы WPF» отмечалось, что триггер события (представленный классом EventTrigger) активируется в момент генерации маршрутизируемого события. Это событие задается с помощью свойства триггера RoutedEvent, а сам триггер может содержать одно или несколько действий (объектов, производных от абстрактного класса TriggerAction) в коллекции Actions. Классы анимации, например DoubleAnimation, сами действиями не являются, поэтому поместить их непосредственно в коллекцию Actions триггера нельзя.

Вместо этого анимации помещаются в объект под названием Storyboard (раскадровка), обертываемый действием BeginStoryboard.

Таким образом, ассоциирование рассмотренной выше анимации DoubleAnimation с триггером события, активируемым по нажатию кнопки, могло бы выглядеть так:

```
<Button>
  OK
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard TargetProperty="Width">
        <DoubleAnimation From="50" To="100"
          Duration="0:0:5" AutoReverse="True"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
  </EventTrigger>
</Button.Triggers>
</Button>
```

Эти два дополнительных элемента играют ту же роль, что и метод BeginAnimation в процедурном коде: Storyboard с помощью своего свойства TargetProperty определяет свойство зависимости, к которому применяется анимация, а BeginStoryboard задает момент начала анимации, присоединяя Storyboard к триггеру.

Может показаться, что объект BeginStoryboard избыточен, но в состав WPF входят и другие классы, производные от TriggerAction. Есть действие для воспроизведения звуков и еще несколько действий, работающих в сочетании с BeginStoryboard и позволяющих декларативно приостанавливать раскадровку, смещаться по ней, останавливать и т. д. (Они называются PauseStoryboard, SeekStoryboard и т. д.)

СОВЕТ

В XAML нельзя начать анимацию иначе, как поместив ее в элемент Storyboard.

Задание целевого свойства

В приведенной выше XAML-разметке свойству TargetProperty элемента Storyboard присвоено имя свойства (Width) самого целевого объекта. Однако же типом TargetProperty является PropertyPath, а значит, поддерживаются и более сложные выражения (примеры мы видели в предыдущих главах), скажем свойство с цепочкой субсвойств.

Ниже показана кнопка Button, в которой в качестве фона Background задана линейно-градиентная кисть LinearGradientBrush с тремя точками смены градиен-

та GradientStop. Мы используем анимацию ColorAnimation, чтобы заставить средний цвет Color изменяться от черного к белому и обратно. (Идея анимировать цвет может показаться странной, но внутреннее представление типа Color состоит из трех значений с плавающей точкой, соответствующих компонентам ScA, ScR, ScB и ScG, поэтому ColorAnimation может их интерполировать точно так же, как DoubleAnimation интерполирует одно значение.) Для анимации среднего цвета линейно-градиентной кисти в раскадровке Storyboard должно присутствовать составное выражение TargetProperty:

```
<Button Padding="30">
  OK
<Button.Background>
  <LinearGradientBrush>
    <GradientStop Color="Blue" Offset="0"/>
    <GradientStop Color="Black" Offset="0.5"/>
    <GradientStop Color="Blue" Offset="1"/>
  </LinearGradientBrush>
</Button.Background>
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Loaded">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard TargetProperty="Background.GradientStops[1].Color">
        <ColorAnimation From="Black" To="White" Duration="0:0:2"
          AutoReverse="True" RepeatBehavior="Forever"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
  </EventTrigger>
</Button.Triggers>
</Button>
```

Синтаксис TargetProperty соответствует тому, что вы сами написали бы на C# для доступа к этому свойству, только без приведения типов. В этой раскадровке предполагается, что в качестве фона Background кнопки задан некий объект, обладающий свойством GradientStops, которое можно индексировать, причем в индексированной коллекции есть хотя бы два элемента и у второго имеется свойство Color типа Color. Если хотя бы одно из этих предположений не выполнено, анимация не произойдет. Разумеется, в данном случае все предположения выполнены, поэтому кнопка успешно анимируется, как показано на рис. 17.3.



***Рис. 17.3.** Анимация среднего цвета в линейно-градиентной кисти*

Можно было бы присоединить DoubleAnimation к целевому свойству Background.GradientStops[1].Offset и создать эффект анимированной кисти, заставив перемещаться темную область. Если же вы хотите анимировать *одновременно* Color и Offset в ответ на одно и то же событие Loaded, то можете добавить в триггер два действия BeginStoryboard:

```
<EventTrigger RoutedEvent="Button.Loaded">
<EventTrigger.Actions>
  <BeginStoryboard>
    <Storyboard TargetProperty="Background.GradientStops[1].Color">
      <ColorAnimation From="Black" To="White" Duration="0:0:2"
        AutoReverse="True" RepeatBehavior="Forever"/>
    </Storyboard>
  </BeginStoryboard>
  <BeginStoryboard>
    <Storyboard TargetProperty="Background.GradientStops[1].Offset">
      <DoubleAnimation From="0" To="1" Duration="0:0:2"
        AutoReverse="True" RepeatBehavior="Forever"/>
    </Storyboard>
  </BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
```

К счастью, WPF предлагает механизм анимирования разных свойств в одной и той же раскадровке Storyboard. Во-первых, объект Storyboard сам может содержать несколько анимаций. Свойством содержимого Storyboard является Children, коллекция объектов типа Timeline (базовый класс всех классов анимации). Во-вторых, TargetProperty – не просто обычное свойство зависимости, но также и присоединенное свойство, которое можно ассоциировать с потомками Storyboard! Поэтому предыдущий XAML-код можно переписать и в таком виде:

```
<EventTrigger RoutedEvent="Button.Loaded">
<EventTrigger.Actions>
  <BeginStoryboard>
    <Storyboard>
      <ColorAnimation From="Black" To="White" Duration="0:0:2"
        Storyboard.TargetProperty="Background.GradientStops[1].Color"
        AutoReverse="True" RepeatBehavior="Forever"/>
      <DoubleAnimation From="0" To="1" Duration="0:0:2"
        Storyboard.TargetProperty="Background.GradientStops[1].Offset"
        AutoReverse="True" RepeatBehavior="Forever"/>
    </Storyboard>
  </BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
```

Здесь в одной раскадровке находятся две анимации, причем для каждой задано свое свойство целевого объекта. Обе анимации начинаются одновременно, но, если нужно, чтобы они начинались в разное время, достаточно задать для них различные значения свойства BeginTime.

Задание целевого объекта

В показанных до сих пор раскадровках целевой объект, содержащий целевое свойство, определялся неявно. По умолчанию это объект, в который вложены триггеры, или – в случае стиля `Style` – шаблон-родитель. Но можно задать и другой целевой объект, воспользовавшись свойством `TargetName` класса `Storyboard`. Как и `TargetProperty`, свойство `TargetName` можно применить либо непосредственно к `Storyboard`, либо к отдельным его потомкам в виде присоединенного свойства.

Ниже приведен забавный пример использования `TargetName` для трансформации одного изображения в другое путем анимации непрозрачности второго изображения, наложенного на первое:

```
<Grid xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Grid.Triggers>
  <EventTrigger RoutedEvent="Grid.Loaded">
    <BeginStoryboard>
      <Storyboard TargetName="jim2" TargetProperty="Opacity">
        <DoubleAnimation From="1" To="0" Duration="0:0:4"
          AutoReverse="True" RepeatBehavior="Forever"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Grid.Triggers>
  <Image Name="jim1" Source="jim1.gif"/>
  <Image Name="jim2" Source="jim2.gif"/>
</Grid>
```

Изображенный здесь Джим сбрил свою роскошную бороду и коротко подстригся, сделав практически одинаковые по композиции фотографии до и после этого. Результат применения этой анимации показан на рис. 17.4.

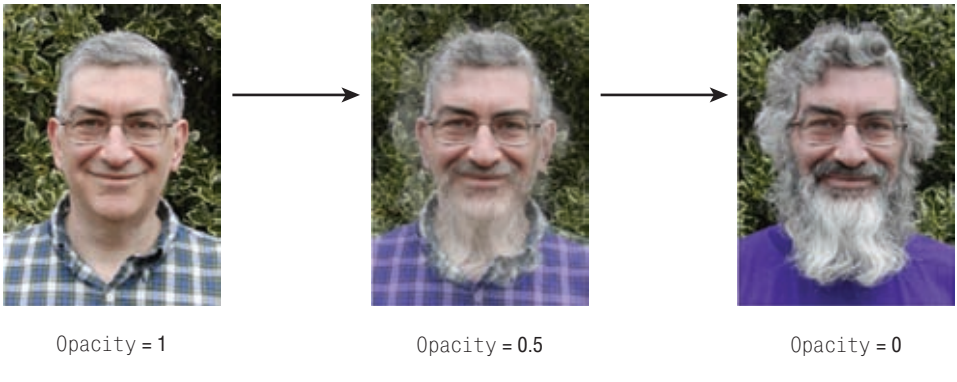


***Рис. 17.4.** Анимация свойства Opacity объекта Image для трансформации двух похожих фотографий*

Этот пример использования TargetName несколько натянутый, потому что триггер события можно было бы связать непосредственно с jim2, а не с родительской сеткой Grid. Но в более крупных приложениях (например, в слайд-шоу изображений Image) желательно собрать все анимации в одном месте с одним триггером события, быть может, даже в одной раскадровке Storyboard. И добиться этого можно, используя TargetName в качестве присоединенного свойства каждой анимации.

Триггеры событий внутри стиля

Во всех примерах XAML-кода в этом разделе триггер события добавлялся непосредственно в элементы. Но чаще триггеры событий включаются в стиль Style. В листинге 17.1 к восьми кнопкам на панели StackPanel применен стиль с анимациями. Эти анимации вдвое увеличивают размер кнопки по событию MouseEnter и возвращают нормальный размер по событию MouseLeave, что создает упрощенный вариант эффекта «рыбьего глаза». Результат показан на рис. 17.5.

Листинг 17.1. Применение к кнопкам стиля со встроенной анимацией

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="Animation">
<Window.Resources>
  <Style TargetType="{x:Type Button}">
    <Setter Property="VerticalAlignment" Value="Bottom"/>
    <Setter Property="LayoutTransform">
    <Setter.Value>
      <ScaleTransform/>
    </Setter.Value>
    </Setter>
  <Style.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
    <EventTrigger.Actions>
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleX"
            To="2" Duration="0:0:0.25"/>
          <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleY"
            To="2" Duration="0:0:0.25"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.MouseLeave">
    <EventTrigger.Actions>
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleX"
            To="1" Duration="0:0:0.25"/>
          <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleY"
            To="1" Duration="0:0:0.25"/>
```

```
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
    </EventTrigger>
  </Style.Triggers>
  </Style>
</Window.Resources>
  <StackPanel Orientation="Horizontal">
    <Button>1</Button>
    <Button>2</Button>
    <Button>3</Button>
    <Button>4</Button>
    <Button>5</Button>
    <Button>6</Button>
    <Button>7</Button>
    <Button>8</Button>
  </StackPanel>
</Window>
```



***Рис. 17.5.** Ко всем кнопкам применен стиль с анимациями увеличения и уменьшения размера*

В этом листинге `TargetProperty` использовано в качестве присоединенного свойства для анимации `ScaleX` и `ScaleY` в одной и той же раскадровке. В обеих анимациях предполагается, что в свойство `LayoutTransform` записан экземпляр преобразования `ScaleTransform`. Если бы на самом деле `LayoutTransform` содержало группу `TransformGroup`, в которой `ScaleTransform` было бы первым дочерним элементом, то в анимациях для доступа к нужным свойствам следовало бы указать выражения `LayoutTransform.Children[0].ScaleX` и `LayoutTransform.Children[0].ScaleY`.

СОВЕТ

Самый лучший способ анимировать размер и положение элемента – присоединить преобразование `ScaleTransform` и/или `TranslateTransform` и анимировать его свойства. Анимация свойств `ScaleX` и `ScaleY` преобразования `ScaleTransform` обычно полезнее, чем анимация свойств `Width` и `Height`, потому что позволяет изменять размер элемента в процентах, а не на абсолютное число единиц. А анимация преобразования `TranslateTransform` лучше, чем анимация чего-то вроде `Canvas.Left` и `Canvas.Top`, потому что работает вне зависимости от того, на какую панель помещен элемент.

СОВЕТ

Для типа RepeatBehavior, как и для Duration, определен конвертер, упрощающий его использование в XAML. Фиксированное время можно задавать с помощью строки, отформатированной как TimeSpan. Для задания поведения RepeatBehavior.Forever подойдет строка "Forever", а число, за которым следует суффикс "x" (например, "2x" или "3x"), трактуется как коэффициент.

Чтобы анимировать каждую кнопку с помощью ScaleTransform, не требуя явно задавать для всех кнопок преобразование ScaleTransform, в листинге 17.1 в свойство LayoutTransform записан экземпляр ScaleTransform прямо в стиле Style. (Разумеется, такая схема перестает работать, если в какой-то кнопке свойство LayoutTransform задано явно.) Во всех анимациях свойство From опущено, чтобы эффект оставался плавным. Продолжительность Duration задана с помощью простой строки благодаря конвертеру типа, понимающему формат TimeSpan.Parse (а также строк "Automatic" и "Forever").

КОПНЕМ ГЛУБЖЕ

Запуск анимации из триггера свойства

Коллекцию Style.Triggers в листинге 17.1 можно заменить эквивалентной, в которой есть всего один триггер свойства IsMouseOver:

```
<Style.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
  <Trigger.EnterActions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleX"
          To="2" Duration="0:0:0.25"/>
        <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleY"
          To="2" Duration="0:0:0.25"/>
      </Storyboard>
    </BeginStoryboard>
  </Trigger.EnterActions>
  <Trigger.ExitActions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleX"
          To="1" Duration="0:0:0.25"/>
        <DoubleAnimation Storyboard.TargetProperty="LayoutTransform.ScaleY"
          To="1" Duration="0:0:0.25"/>
      </Storyboard>
    </BeginStoryboard>
  </Trigger.ExitActions>
  </Trigger>
</Style.Triggers>
```

Вместо одной коллекции Actions в триггере свойства может быть задано две коллекции: EnterActions и ExitActions. Действия, содержащиеся в EnterActions, активируются, когда включается сам триггер (то есть в момент, когда применяются все элементы Setter), а действия из коллекции ExitActions – когда триггер деактивируется (если операции, выполненные элементами Setter, отменяются). В данном примере желаемого эффекта можно достичь с помощью как триггеров событий, так и триггера свойства. На чем остановиться – дело вкуса.

Использование раскадровки как временной шкалы

Объект Storyboard – не просто контейнер, содержащий одну или несколько анимаций, в которых установлены целевые объекты и свойства. Класс Storyboard наследует Timeline, базовому классу, общему для всех классов анимаций (DoubleAnimation, ColorAnimation и т. д.). Это означает, что у Storyboard есть много свойств и событий, которые обсуждались выше в этой главе: Duration, BeginTime, SpeedRatio, AutoReverse, RepeatBehavior, AccelerationRatio, DecelerationRatio, FillBehavior и прочие.

В листинге 17.2 показана раскадровка, которая постепенно проявляет и стирает текстовый блок TextBlock, создавая эффект сменяющихся титров в конце фильма. Для самого элемента Storyboard задано свойство RepeatBehavior, чтобы вся последовательность анимации повторялась бесконечно. На рис. 17.6 показаны три момента визуализации этой разметки:

Листинг 17.2. Раскадровка, содержащая несколько анимаций

```
<Grid xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      Background="Black" TextBlock.Foreground="White" TextBlock.FontSize="30">
<Grid.Triggers>
  <EventTrigger RoutedEvent="Grid.Loaded">
    <BeginStoryboard>
      <Storyboard TargetProperty="Opacity" RepeatBehavior="Forever">
        <DoubleAnimation Storyboard.TargetName="title1" BeginTime="0:0:2"
          From="0" To="1" Duration="0:0:2" AutoReverse="True"/>
        <DoubleAnimation Storyboard.TargetName="title2" BeginTime="0:0:6"
          From="0" To="1" Duration="0:0:2" AutoReverse="True"/>
        <DoubleAnimation Storyboard.TargetName="title3" BeginTime="0:0:10"
          From="0" To="1" Duration="0:0:2" AutoReverse="True"/>
        <DoubleAnimation Storyboard.TargetName="title4" BeginTime="0:0:14"
          From="0" To="1" Duration="0:0:2" AutoReverse="True"/>
        <DoubleAnimation Storyboard.TargetName="title5" BeginTime="0:0:18"
          From="0" To="1" Duration="0:0:2" AutoReverse="True"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Grid.Triggers>
  <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" Opacity="0"
    Name="title1">In a world</TextBlock>
  <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" Opacity="0"
```

```
      Name="title2">where user interfaces need to be created</TextBlock>
    <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" Opacity="0"
      Name="title3">one book</TextBlock>
    <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" Opacity="0"
      Name="title4">will explain it all...</TextBlock>
    <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" Opacity="0"
      Name="title5">WPF 4 Unleashed</TextBlock>
  </Grid>
```



Рис. 17.6. *Несколько моментов последовательности сменяющихся титров*

Задание для раскадровки свойств, унаследованных от класса `Timeline`, распространяется на весь набор дочерних анимаций, хотя и не совсем так, как если бы те же свойства были заданы для отдельных анимаций. Например, если в листинге 17.2 задать `RepeatBehavior="Forever"` для каждой дочерней анимации, а не для `Storyboard` в целом, то получится хаос. Первый титр будет появляться и исчезать, как положено, но затем на шестой секунде начнут появляться и исчезать *оба* титра, `title1` и `title2`. На десятой секунде будут появляться и исчезать уже три титра, `title1`, `title2` и `title3`, и т. д.

Аналогично, если для каждой анимации `DoubleAnimation` задать `SpeedRatio="2"`, то на каждое появление и исчезновение будет затрачиваться одна секунда вместо двух, но заключительная анимация все равно начнется через 18 с после начала последовательности. С другой стороны, задание `SpeedRatio="2"` для раскадровки ускорит всю анимацию в целом, включая каждый отрезок `BeginTime`, вдвое. Следовательно, заключительная анимация начнется через 9 с после начала последовательности. Задание `AccelerationRatio="1"` для раскадровки приведет к тому, что каждая анимация (и промежуток времени между ними) будет быстрее предыдущей. Если присвоить `Duration` значение, меньшее естественной продолжительности, то вся последовательность анимаций прервется досрочно.

Анимация с опорными кадрами

Обычные классы анимации поддерживают только линейную интерполяцию при переходе от одного значения к другому (или ограниченные формы нелинейной интерполяции при использовании свойств `AccelerationRatio` и `DecelerationRatio`), если только не заданы переходные функции. Чтобы декларативно задать нестандартную, более сложную анимацию, можно определить *опорные кадры* и указать в них конкретные значения в определенные моменты времени. Для этого необходим класс анимации с поддержкой опорных кадров. Например, классу `DoubleAnimation` соответствует класс `DoubleAnimationUsingKeyFrames`; такие же аналоги имеются для всех классов вида *XXX*`Animation`.

Классы анимации с опорными кадрами обладают теми же свойствами и событиями, что и их обычные аналоги, за исключением свойств From, To и By. Вместо этого у них есть коллекция KeyFrames, в которой хранятся ключевые кадры, специфичные для анимируемого типа. Всего в WPF есть четыре типа опорных кадров, которые мы рассмотрим в следующем разделе.

Линейные опорные кадры

В листинге 17.3 класс DoubleAnimationUsingKeyFrames используется для того, чтобы смоделировать зигзагообразный полет мухи (рис. 17.7). Поскольку изображение Image находится внутри Canvas, движение обеспечивается за счет анимации присоединенных свойств Canvas.Left и Canvas.Top, а не более гибкой анимацией преобразования TranslateTransform.

Листинг 17.3. Зигзагообразная анимация, изображенная на рис. 17.7

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="Animation Using Keyframes" Height="300" Width="580">
  <Canvas>
    <Image Source="fly.png">
    <Image.Triggers>
      <EventTrigger RoutedEvent="Image.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation Storyboard.TargetProperty="(Canvas.Left)"
              From="0" To="500" Duration="0:0:3"/>
            <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="(Canvas.Top)"
              Duration="0:0:3">
              <LinearDoubleKeyFrame Value="0" KeyTime="0:0:0"/>
              <LinearDoubleKeyFrame Value="200" KeyTime="0:0:1"/>
              <LinearDoubleKeyFrame Value="0" KeyTime="0:0:2"/>
              <LinearDoubleKeyFrame Value="200" KeyTime="0:0:3"/>
            </DoubleAnimationUsingKeyFrames>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
      </EventTrigger>
    </Image.Triggers>
    </Image>
  </Canvas>
</Window>
```

Движение мухи состоит их двух анимаций, которые начинаются одновременно после загрузки изображения. Одна – простая DoubleAnimation, которая линейно увеличивает горизонтальную координату от 0 до 500. Другая – анимация с опорными кадрами, в которой вертикальная координата совершает колебательное движение от 0 до 200 и обратно.

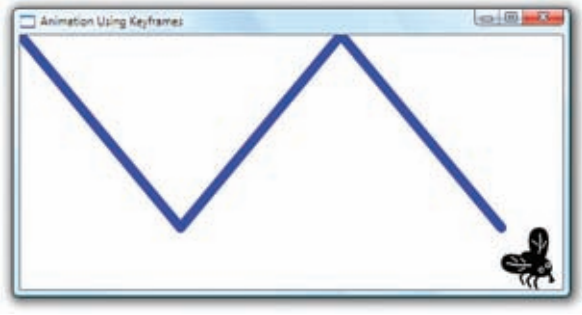


Рис. 17.7. *Зигзагообразное движение легко создать за счет анимации с опорными кадрами*

КОПНЕМ ГЛУБЖЕ

Анимация и привязка к данным

Чтобы не усложнять обсуждение, в листинге 17.3 начальные и конечные значения анимируемых свойств `Canvas.Left` и `Canvas.Top` зашиты. Но можно было бы воспользоваться привязкой к данным и установить значения свойства `To` в соответствии с размерами `Window` или `Canvas`. Например:

```
<DoubleAnimation Storyboard.TargetProperty="(Canvas.Left)" From="0"
  To="{Binding RelativeSource={RelativeSource FindAncestor,
  AncestorType={x:Type Canvas}}, Path=ActualWidth}"
  Duration="0:0:3"/>
```

К сожалению, такую анимацию нельзя выполнить в триггере по событию `Image.Loaded`, потому что это событие генерируется до того, как становится известна фактическая высота `ActualHeight` окна `Window` или панели `Canvas`. (До этого момента `ActualHeight` равно `NaN`, что приводит к исключению `AnimationException`.) Впрочем, никто не мешает произвести такую привязку к данным в анимациях, ассоциированных с более поздними событиями.

ПРЕДУПРЕЖДЕНИЕ

Когда присоединенное свойство задается в качестве TargetProperty, его необходимо заключать в круглые скобки!

Обратите внимание, что в листинге 17.3 в тех случаях, когда свойства `Canvas.Left` и `Canvas.Top` используются в роли значения свойства `TargetProperty` элемента `Storyboard`, они заключаются в скобки. Это требование относится к любому присоединенному свойству, которое встречается в составе пути к свойству. Не будь скобок, анимация искала бы свойство `Canvas` объекта `Image` (ожидая, что оно вернет объект, обладающий свойствами `Left` и `Top`) и возбудила бы исключение, потому что такого свойства не существует.

Каждый экземпляр опорного кадра (LinearDoubleKeyFrame) в листинге 17.3 содержит конкретное значение и время, когда это значение должно быть присвоено свойству. Впрочем, атрибут KeyTime необязателен. Если он опущен, то WPF предполагает, что опорный кадр находится на полпути между окружающими его кадрами. Если атрибут KeyTime отсутствует во всех опорных кадрах, то они равномерно распределяются по всей продолжительности анимации. (Этот режим можно задать и явно, присвоив свойству KeyTime значение KeyTimeType.Uniform или просто "Uniform" в XAML.)

Хотя в опорных кадрах в листинге 17.3 вертикальное положение мухи в моменты 0, 1, 2 и 3 с задано точно, WPF все равно должна вычислить промежуточные значения между этими «опорными моментами». Поскольку каждый опорный кадр представлен объектом типа LinearDoubleKeyFrame, то промежуточные значения получаются простой линейной интерполяцией. Например, в моменты 0,5, 1,5 и 2,5 с вычисленное значение равно 100.

Но в классе DoubleAnimationUsingKeyFrames свойство KeyFrames – это коллекция объектов абстрактного класса DoubleKeyFrame, которая может быть заполнена объектами, представляющими опорные кадры других типов. Помимо LinearDoubleKeyFrame у класса DoubleKeyFrame есть еще три подкласса: SplineDoubleKeyFrame, DiscreteDoubleKeyFrame и EasingDoubleKeyFrame.

СОВЕТ

Величину KeyTime можно задать в виде процента, а не значения типа TimeSpan. Это удобно, когда временные параметры опорного кадра желательно выразить независимо от продолжительности анимации. Например, элементы DoubleAnimationUsingKeyFrame в листинге 17.3 можно заменить следующими и результат будет точно таким же:

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="(Canvas.Top)"
  Duration="0:0:3">
  <LinearDoubleKeyFrame Value="0" KeyTime="0%"/>
  <LinearDoubleKeyFrame Value="200" KeyTime="33.3%"/>
  <LinearDoubleKeyFrame Value="0" KeyTime="66.6%"/>
  <LinearDoubleKeyFrame Value="200" KeyTime="100%"/>
</DoubleAnimationUsingKeyFrames>
```

Свойству KeyTime можно также присвоить значение Paced, тогда опорные кадры разместятся так, чтобы целевое свойство изменялось с постоянной скоростью. Иными словами, два опорных кадра, между которыми значение изменяется от 0 до 200, отстоят друг от друга на время, вдвое большее, чем опорные кадры с изменением значения от 0 до 100.

Сплайновые опорные кадры

Каждому классу Linear*XXX*KeyFrame соответствует класс Spline*XXX*KeyFrame. Его можно использовать так же, как «линейную» версию, поэтому следующая модификация листинга 17.3 дает тот же результат, что и прежде:

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="(Canvas.Top)"
  Duration="0:0:3">
  <SplineDoubleKeyFrame Value="0" KeyTime="0:0:0"/>
  <SplineDoubleKeyFrame Value="200" KeyTime="0:0:1"/>
  <SplineDoubleKeyFrame Value="0" KeyTime="0:0:2"/>
  <SplineDoubleKeyFrame Value="200" KeyTime="0:0:3"/>
</DoubleAnimationUsingKeyFrames>
```

В классах сплайновых опорных кадров имеется дополнительное свойство KeySpline, отличающее их от линейных классов. Свойству KeySpline можно присвоить объект типа KeySpline, который описывает желаемое движение в виде кубической кривой Безье. В классе KeySpline имеется два свойства типа Point, которые представляют управляющие точки кривой. (Начальной точкой кривой всегда является 0, а конечной – 1.) Конвертер типа позволяет задать KeySpline на XAML в виде списка из двух точек. Например, после следующей модификации движение мухи из простого зигзага (см. рис. 17.7) становится более сложным (рис. 17.8):

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="(Canvas.Top)"
  Duration="0:0:3">
  <SplineDoubleKeyFrame KeySpline="0,1 1,0" Value="0" KeyTime="0:0:0"/>
  <SplineDoubleKeyFrame KeySpline="0,1 1,0" Value="200" KeyTime="0:0:1"/>
  <SplineDoubleKeyFrame KeySpline="0,1 1,0" Value="0" KeyTime="0:0:2"/>
  <SplineDoubleKeyFrame KeySpline="0,1 1,0" Value="200" KeyTime="0:0:3"/>
</DoubleAnimationUsingKeyFrames>
```

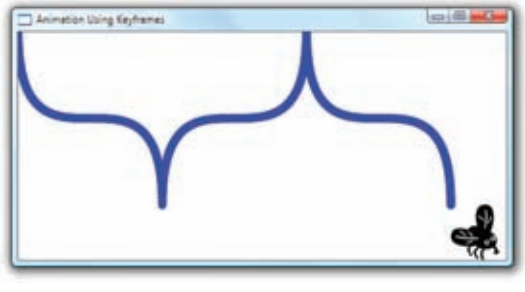


***Рис. 17.8.** Если задана сплайновая анимация, то интерполяция между опорными кадрами основана на кубических кривых Безье*

Отыскание того значения KeySpline, которое даст желаемый эффект, – дело непростое и почти наверняка потребует какого-нибудь инструмента конструирования, например Expression Blend. Но в Сети можно найти и несколько бесплатных инструментов, позволяющих визуализировать кривую Безье с заданными управляющими точками.

Дискретные опорные кадры

Дискретный опорный кадр просто означает, что переход из состояния, достигнутого после предыдущего опорного кадра, должен проводиться без ин-

терполяции. Замена элементов DoubleAnimationUsingKeyFrame в листинге 17.3 следующими дает движение, показанное на рис. 17.9:

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="(Canvas.Top)"
  Duration="0:0:3">
  <DiscreteDoubleKeyFrame Value="0" KeyTime="0:0:0"/>
  <DiscreteDoubleKeyFrame Value="200" KeyTime="0:0:1"/>
  <DiscreteDoubleKeyFrame Value="0" KeyTime="0:0:2"/>
  <DiscreteDoubleKeyFrame Value="200" KeyTime="0:0:3"/>
</DoubleAnimationUsingKeyFrames>
```



Рис. 17.9. *Наличие дискретного опорного кадра заставляет муху подпрыгнуть вверх по вертикали из предыдущего положения в следующее без интерполяции*

Разумеется, в составе одной и той же анимации могут быть опорные кадры разных типов. Следующий код заставляет муху летать по траектории, изображенной на рис. 17.10:

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="(Canvas.Top)"
  Duration="0:0:3">
  <DiscreteDoubleKeyFrame Value="0" KeyTime="0:0:0"/>
  <LinearDoubleKeyFrame Value="200" KeyTime="0:0:1"/>
  <DiscreteDoubleKeyFrame Value="0" KeyTime="0:0:2"/>
  <SplineDoubleKeyFrame KeySpline="0,1,1,0" Value="200" KeyTime="0:0:3"/>
</DoubleAnimationUsingKeyFrames>
```

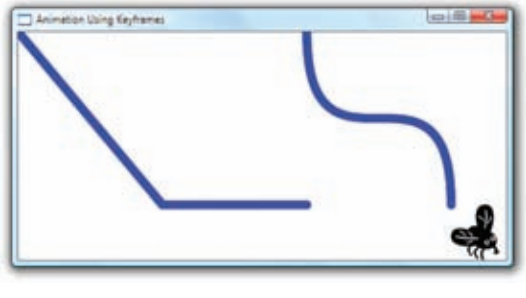


Рис. 17.10. *Сочетание опорных кадров трех типов в одной анимации*

Поскольку время в первом опорном кадре совпадает с начальным моментом, то его тип вообще не имеет значения, так как в каждом кадре указывается, как производить интерполяцию *перед* этим кадром.

Как и в случае `Spline`*`XXX`*`KeyFrame`, каждому классу `Linear`*`XXX`*`KeyFrame` соответствует класс `Discrete`*`XXX`*`KeyFrame`. Но в WPF есть еще пять дополнительных классов дискретных опорных кадров, не имеющих ни линейных, ни сплайновых аналогов. Эти классы позволяют анимировать объекты типа `Boolean`, `Char`, `Matrix`, `Object` и `String`. Для этих типов данных WPF поддерживает только анимацию с дискретными опорными кадрами, потому что интерполяция в этом случае бессмысленна (или даже невозможна, как в случае типа `Boolean`).

Вот, например, какую анимацию можно было бы применить к элементу `TextBlock` для постепенного перехода строчных букв набранного текста в прописные (в каждом опорном кадре по умолчанию подразумевается значение `Uniform` свойства `KeyTime`):

```
<StringAnimationUsingKeyFrames Storyboard.TargetProperty="Text" Duration="0:0:.5">
  <DiscreteStringKeyFrame Value="play"/>
  <DiscreteStringKeyFrame Value="Play"/>
  <DiscreteStringKeyFrame Value="PLay"/>
  <DiscreteStringKeyFrame Value="PLAy"/>
  <DiscreteStringKeyFrame Value="PLAY"/>
</StringAnimationUsingKeyFrames>
```

СОВЕТ

Если требуется просто установить значение свойства в триггере события, а не анимировать его в традиционном понимании, то можно воспользоваться анимацией с опорным кадром для моделирования элемента `Setter`. Например, следующая анимация заставляет кнопку мгновенно исчезнуть сразу после нажатия путем установки значения `0` для свойства `Opacity` в опорном кадре, расположенном в начале анимации, которая больше ничего не содержит:

```
<Button>
  Click Me Once
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Opacity">
          <DiscreteDoubleKeyFrame Value="0" KeyTime="0"/>
        </DoubleAnimationUsingKeyFrames>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
  </EventTrigger>
</Button.Triggers>
</Button>
```

Переходные опорные кадры

Начиная с версии WPF 4 каждому классу Linear*XXX*KeyFrame (и Spline*XXX*KeyFrame) соответствует также класс Easing*XXX*KeyFrame. У таких классов есть свойство EasingFunction, в которое можно записать ссылку на объект любого класса, реализующего интерфейс IEasingFunction. Как и свойство EasingFunction в классах анимации, оно дает максимальную гибкость интерполяции. А теперь самое время поговорить о том, что такое переходные функции.

КОПНЕМ ГЛУБЖЕ

Анимация вдоль пути

В WPF встроена еще одна возможность анимации типов Double, Point и Matrix. Классы DoubleAnimationUsingPath, PointAnimationUsingPath и MatrixAnimationUsingPath позволяют задать путь PathGeometry, который определяет способ изменения целевого значения (между точками пути производится линейная интерполяция). Хотя технически эти классы можно использовать вместе с любыми свойствами подходящего типа, предназначены они для анимации положения объектов, причем PathGeometry интерпретируется как «путь», вдоль которого перемещается объект. (В случае DoubleAnimationUsingPath берется две таких анимации. Одна применяет указанные в PathGeometry значения X для анимации абсциссы целевого объекта, а другая – значения Y для анимации ординаты.)

Переходные функции

В состав WPF входит 11 переходных функций – классов, реализующих интерфейс IEasingFunction, – которые можно применять к анимации или к опорному кадру. Каждый класс поддерживает три режима с помощью свойства EasingMode: EaseIn (по умолчанию), EaseOut и EaseInOut. Вот как применяется одна из переходных функций – QuadraticEase – к простейшей анимации DoubleAnimation:

```
<DoubleAnimation Storyboard.TargetProperty="(Canvas.Top)" From="200" To="0"
  Duration="0:0:3">
<DoubleAnimation.EasingFunction>
  <QuadraticEase/>
</DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

А вот как можно изменить режим EasingMode на другой, отличный от EaseIn:

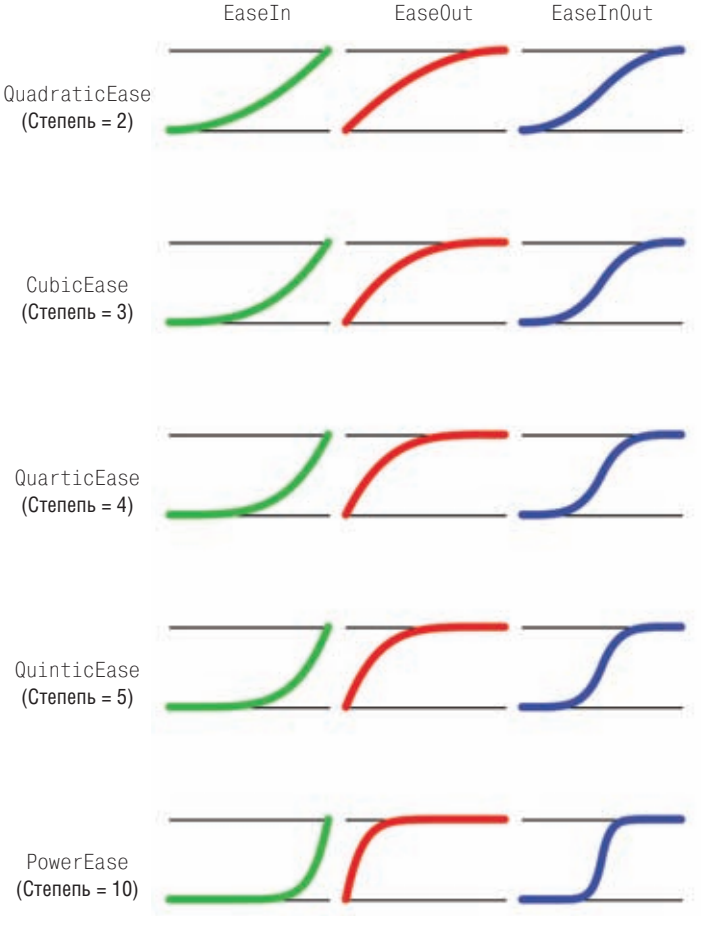
```
<DoubleAnimation Storyboard.TargetProperty="(Canvas.Top)" From="200" To="0"
  Duration="0:0:3">
<DoubleAnimation.EasingFunction>
  <QuadraticEase EasingMode="EaseOut"/>
</DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

Режим `EaseOut` инвертирует интерполяцию, произведенную с помощью `EaseIn`, а `EaseInOut` совпадает с `EaseIn` в течение первой половины анимации и с `EaseOut` — во второй половине.

Встроенные переходные функции

В табл. 17.2 иллюстрируется пять переходных функций во всех трех режимах: показано, по какому пути перемещается объект, если абсцисса интерполируется линейно, а ордината анимируется снизу вверх с помощью различных переходных функций.

Таблица 17.2. Пять степенных переходных функций



Все пять функций производят интерполяцию на основе простой степенной функции. В случае подразумеваемой по умолчанию линейной интерполяции

по истечении 50% (0,5) времени значение также изменяется на 50% (0,5). В случае квадратичной интерполяции значение изменяется на 25% ($0,5 \times 0,5 = 0,25$) по прошествии 50% времени. В случае кубической интерполяции значение изменяется на 12,5% ($0,5 \times 0,5 \times 0,5 = 0,125$) по прошествии 50% времени и т. д. Хотя в WPF есть четыре разных класса для степеней от 2 до 5, на самом деле необходим один общий класс PowerEase, который производит интерполяцию на основе значения своего свойства Power. По умолчанию Power равно 2 (сводя интерполяцию к квадратичной), но в табл. 17.2 продемонстрирована картина для случая, когда Power равно 10, просто чтобы показать, что с ростом Power переход становится резче. Применение класса PowerEase с Power, равным 10, выглядит следующим образом:

```
<DoubleAnimation Storyboard.TargetProperty="(Canvas.Top)" From="200" To="0"
  Duration="0:0:3">
<DoubleAnimation.EasingFunction>
  <PowerEase Power="10"/>
</DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

Другие встроенные переходные функции

В табл. 17.3 показаны остальные шесть переходных функций во всех трех режимах.

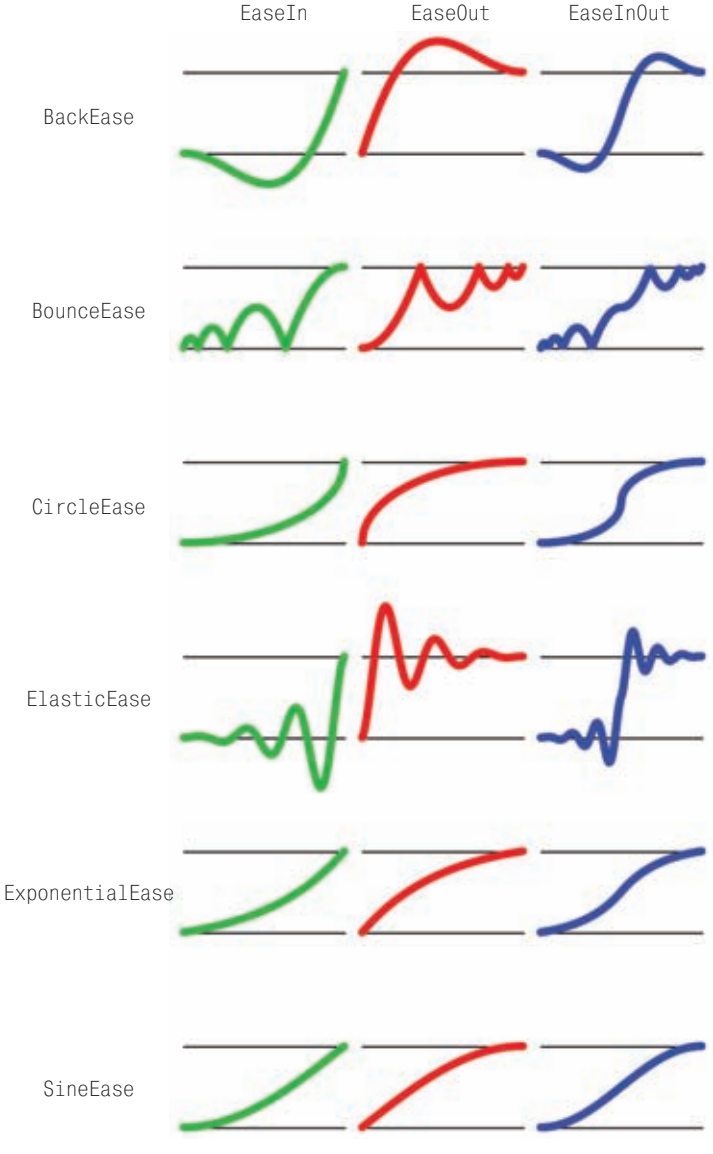
Каждая из этих шести функций обладает уникальным (и иногда конфигурируемым) поведением:

- BackEase – сначала слегка уменьшает анимируемое значение (отдаляясь от конечного значения), а затем начинает увеличивать. В классе BackEase имеется свойство Amplitude (по умолчанию равно 1), определяющее, как далеко назад отклоняется значение.
- BounceEase – описывает приближение с подскоком (по крайней мере, при анимации положения). В классе BounceEase есть два свойства, управляющих поведением. Свойство Bounces (по умолчанию равно 3) указывает, сколько подскоков произвести на протяжении анимации, а свойство Bounciness (по умолчанию равно 2) описывает отличие амплитуды следующего подскока по сравнению с предыдущим. В режиме EaseIn значение Bounciness=2 удваивает высоту каждого следующего подскока. В режиме EaseOut значение Bounciness=2 уменьшает высоту следующего подскока вдвое.
- CircleEase – ускоряет (в режиме EaseIn) или замедляет (в режиме EaseOut) анимацию в соответствии с «круговой функцией» $f(t) = 1 - \sqrt{1 - t^2}$.
- ElasticEase – колебательное приближение к конечному значению. Как и в случае BounceEase, поведение регулируется двумя свойствами. Свойство Oscillations (по умолчанию равно 3) указывает, сколько колебаний должно быть произведено за время анимации, а свойство Springiness (по умолчанию равно 3) управляет амплитудой колебаний. Интерпретация Springiness интуитивно неочевидна: чем выше значение, тем меньше амплитуда колебаний (как будто пружина жестче и меньше растягивается), а чем значе-

ние ниже, тем амплитуда больше (на мой взгляд, при этом движение оказывается *более*, а не *менее* пружинистым).

- `ExponentialEase` – интерполирует значение с помощью экспоненциальной функции, поведение определяется свойством `Exponent` (по умолчанию равно 2).
- `SineEase` – интерполирует значение синусоидой.

Таблица 17.3. Остальные шесть встроенных переходных функций



ПРЕДУПРЕЖДЕНИЕ

Функции BackEase и ElasticEase могут вычислять неожиданные отрицательные значения!

Поскольку BackEase и ElasticEase в процессе интерполяции вычисляют значения, выходящие за пределы диапазона от From до To, то всякая анимация, начинающаяся в нуле (в режимах EaseIn или EaseInOut) или заканчивающаяся в нуле (в режимах EaseOut или EaseInOut), почти наверняка заведет вас в область отрицательных чисел. Если такая анимация применяется к величине, которая не может быть отрицательной, например к ширине Width или высоте Height элемента, то возбуждается исключение.

Написание своей переходной функции

Написание собственной переходной функции сводится к написанию класса, реализующего интерфейс IEasingFunction. В интерфейсе IEasingFunction есть всего один метод Ease:

```
public double Ease(double normalizedTime)
{
  // Вернуть значение, приведенное к диапазону от 0 до 1
  ...
}
```

Метод Ease вызывается по ходу анимации и получает нормированное значение времени в диапазоне от 0 до 1. В ответ метод должен вернуть также значение, приведенное к диапазону от 0 до 1, отражающее ход выполнения. (Однако значение может и выходить за пределы этого диапазона, как в случае BackEase и ElasticEase.)

Таким образом, следующий класс реализует линейную переходную функцию (хотя особого смысла в этом не видно):

```
public class LinearEase : IEasingFunction
{
  public double Ease(double normalizedTime)
  {
    return normalizedTime; // Линейная интерполяция
  }
}
```

А следующий класс реализует квадратичную переходную функцию, как встроенный класс QuadraticEase:

```
public class SimpleQuadraticEase : IEasingFunction
{
  public double Ease(double normalizedTime)
  {
    // Реализован только режим EaseIn:
```

```
    return normalizedTime * normalizedTime; // Квадратичная интерполяция
  }
}
```

Чем класс SimpleQuadraticEase отличается от встроенного класса QuadraticEase, так это отсутствием поддержки EasingMode. К счастью, в WPF имеется абстрактный класс EasingFunctionBase (базовый для всех 11 встроенных переходных функций), который дает поведение EasingMode задаром.

В классе EasingFunctionBase определено свойство зависимости EasingMode и реализован интерфейс IEasingFunction. Реализация метода Ease вызывает абстрактный метод EaseInCore, который производные классы должны реализовать точно так же, как реализовали бы метод Ease (если в математических вычислениях учитывается только случай EaseIn). Но в зависимости от текущего режима EasingMode класс EasingFunctionBase модифицирует значение normalizedTime перед вызовом метода EaseInCore, а также возвращенное им значение. В результате таких преобразований логики, реализованной в режиме EaseIn, оказывается достаточно для реализации всех трех режимов. Все это делается прозрачно для производного класса, поэтому для реализации переходной функции с полной поддержкой EasingMode достаточно всего лишь изменить имя базового класса и переименовать метод Ease в EaseInCore:

```
public class CompleteQuadraticEase : EasingFunctionBase
{
  protected override double EaseInCore(double normalizedTime)
  {
    return normalizedTime * normalizedTime; // Квадратичная интерполяция
  }

  // Обязательно должно быть в любом подклассе EasingFunctionBase:
  protected override Freezable CreateInstanceCore()
  {
    return new CompleteQuadraticEase();
  }
}
```

Единственная сложность – необходимость реализовать метод CreateInstanceCore, определенный в абстрактном классе Freezable, которому наследует EasingFunctionBase. Теперь класс CompleteQuadraticEase ведет себя в точности, как встроенный QuadraticEase. С помощью подобной техники можно определять новые интересные переходные функции, например SexticEase (которой предшествовала бы QuinticEase):

```
public class SexticEase : EasingFunctionBase
{
  protected override double EaseInCore(double normalizedTime)
  {
    return normalizedTime * normalizedTime * normalizedTime
           * normalizedTime * normalizedTime * normalizedTime;
  }
```

```
    // Обязательно должно быть в любом подклассе EasingFunctionBase:
    protected override Freezable CreateInstanceCore()
    {
      return new SexticEase();
    }
  }
```

КОПНЕМ ГЛУБЖЕ

Что на самом деле означают режимы EaseOut and EaseInOut

Режим `EaseIn` понять легко, потому что он в точности соответствует логике, реализованной в методе `EaseInCore`, и представлению человека о том, как анимируемое значение изменяется во времени. А чтобы разобраться в режимах `EaseOut` и `EaseInOut`, давайте рассмотрим, какие преобразования выполняет метод `EasingFunctionBase.Ease` до и после вызова метода производного класса `EaseInCore`.

В режиме `EaseIn` метод `EaseInCore` многократно вызывается со значениями от `0` до `1`. В режиме же `EaseOut` метод `EaseInCore` вызывается со значениями от `1` до `0` (методу `EaseInCore` вместо `normalizedTime` передается `1-normalizedTime`). Затем значение `value`, возвращенное `EaseInCore`, снова инвертируется: вместо него возвращается `1-value`.

В режиме `EaseInOut` поведение в первой (`normalizedTime` изменяется от `0` до `0.5` исключительно) и во второй (`normalizedTime` изменяется от `0.5` до `1`) половине анимации различно. В первой половине значение `normalizedTime`, передаваемое `EaseInCore`, удваивается (то есть весь отрезок от `0` до `1` пробегается за половину времени анимации), но возвращаемое значение вдвое уменьшается. Во второй половине значение `normalizedTime`, передаваемое `EaseInCore`, удваивается и инвертируется (отрезок от `1` до `0` пробегается за половину времени анимации). Возвращенное значение уменьшается вдвое, инвертируется, а затем к нему добавляется `0.5` (поскольку это вторая половина пути к конечному значению). Именно поэтому анимация в режиме `EaseInOut` симметрична и всегда проходит половину разности между начальным и конечным значениями за половину времени.

Анимация и менеджер визуальных состояний

Если элемент управления пользуется менеджером визуальных состояний (см. главу 14 «Стили, шаблоны, обложки и темы»), то его шаблон может содержать сколько угодно визуальных состояний `VisualState`. Каждый объект `VisualState`, по существу, представляет собой коллекцию объектов `Storyboard`, которые обеспечивают переход допускающих анимацию свойств к значениям, требуемым в данном состоянии.

Теперь, когда вы все знаете об анимации, легко понять, насколько простым и мощным может быть такой переход. В листинге 17.4 приведен модифицированный шаблон кнопки `Button` из листинга 14.8, в котором триггеры заменены визуальными состояниями `VisualState` (и обработаны некоторые состояния, которые ранее не обрабатывались триггерами).

Листинг 17.4. Шаблон элемента управления Button с визуальными состояниями

```
<Style TargetType="{x:Type Button}">
  <Setter Property="FocusVisualStyle" Value="{x:Null}"/>
  <Setter Property="Background" Value="Black"/>
  <Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="{x:Type Button}">
      <Grid RenderTransformOrigin=".5,.5">
        <VisualStateManager.VisualStateGroups>
          <VisualStateGroup Name="CommonStates">
            <VisualState Name="Normal"/>
            <VisualState Name="MouseOver">
              <Storyboard>
                <ColorAnimation Storyboard.TargetName="outerCircle"
                                Storyboard.TargetProperty=
      "(Ellipse.Fill).(LinearGradientBrush.GradientStops)[1].(GradientStop.Color)"
                                To="Orange" Duration="0:0:.4"/>
              </Storyboard>
            </VisualState>
            <VisualState Name="Pressed">
              <Storyboard>
                <DoubleAnimation Storyboard.TargetName="scaleTransform"
                                 Storyboard.TargetProperty="ScaleX" To=".9"
                                 Duration="0"/>
                <DoubleAnimation Storyboard.TargetName="scaleTransform"
                                 Storyboard.TargetProperty="ScaleY" To=".9"
                                 Duration="0"/>
              </Storyboard>
            </VisualState>
            <VisualState Name="Disabled">
              <Storyboard>
                <ColorAnimation Storyboard.TargetName="outerCircle"
                                Storyboard.TargetProperty=
      "(Ellipse.Fill).(LinearGradientBrush.GradientStops)[1].(GradientStop.Color)"
                                To="Gray" Duration="0:0:.4"/>
              </Storyboard>
            </VisualState>
          </VisualStateGroup>
          <VisualStateGroup Name="FocusStates">
            <VisualState Name="Unfocused"/>
            <VisualState Name="Focused">
              <Storyboard>
                <DoubleAnimation Storyboard.TargetProperty=
      "(Grid.RenderTransform).(TransformGroup.Children)[1].(TranslateTransform.Y)"
                                To="-20" AutoReverse="True"
                                RepeatBehavior="Forever" Duration="0:0:.4">
                <DoubleAnimation.EasingFunction>
                  <QuadraticEase/>
                </DoubleAnimation.EasingFunction>
                </DoubleAnimation>
              </Storyboard>
```

```
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <Grid.RenderTransform>
      <TransformGroup>
        <ScaleTransform x:Name="scaleTransform"/>
        <TranslateTransform x:Name="translateTransform"/>
      </TransformGroup>
    </Grid.RenderTransform>
      <Ellipse x:Name="outerCircle">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0"
            Color="{Binding RelativeSource={RelativeSource TemplatedParent},
                            Path=Background.Color}"/>
          <GradientStop x:Name="highlightGradientStop" Offset="1" Color="Red"/>
        </LinearGradientBrush>
      </Ellipse.Fill>
      </Ellipse>
      <Ellipse RenderTransformOrigin=".5,.5">
      <Ellipse.RenderTransform>
        <ScaleTransform ScaleX=".8" ScaleY=".8"/>
      </Ellipse.RenderTransform>
    <Ellipse.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0" Color="White"/>
        <GradientStop Offset="1" Color="Transparent"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
    </Ellipse>
    <Viewbox>
      <ContentPresenter Margin="{TemplateBinding Padding}"/>
    </Viewbox>
  </Grid>
 </ControlTemplate>
</Setter.Value>
</Setter>
</Style>
```

На рис. 17.11 показаны результаты различных сочетаний состояний, которые могут встретиться при нормальном взаимодействии с пользователем. Состояние `Normal` в группе `CommonStates` не делает ничего; оно оставляет подразумеваемые по умолчанию визуальные объекты без изменений. В состоянии `MouseOver` цвет выделения анимируется до оранжевого, в состоянии `Pressed` все визуальное дерево масштабируется до 90%, а в состоянии `Disabled` цвет выделения анимируется до серого (`Gray`). В группе состояний `FocusStates` подразумеваемое по умолчанию состояние `Unfocused` не делает ничего, но в состоянии `Focused` используется переходная функция `QuadraticEase`, которая заставляет кнопку подпрыгивать все время, пока она находится в этом состоянии. (Для такого типа бесконечно повторяющейся анимации с автовозвратом функция `QuadraticEase` лучше подходит на роль имитатора подскоков, чем `BounceEase`!) В этом

стиле свойству FocusVisualStyle присваивается значение null, чтобы подавить появление пунктирного прямоугольника вокруг подпрыгивающей кнопки, когда она владеет фокусом клавиатуры.



Normal, Unfocused



MouseOver, Unfocused



Pressed, Focused



Disabled, Unfocused



Normal, Focused

Рис. 17.11. *Поведение визуальных состояний кнопки в сочетании с шаблоном в листинге 17.4 (см. также цветную вклейку)*

Поведения Focused и Disabled в главе 14 не упоминались, но можете сравнить состояния MouseOver и Pressed в этом листинге с триггерами IsMouseOver и IsPressed из той главы.

```
<ControlTemplate.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
    <Setter TargetName="outerCircle" Property="Fill" Value="Orange"/>
  </Trigger>
  <Trigger Property="IsPressed" Value="True">
    <Setter Property="RenderTransform">
    <Setter.Value>
      <ScaleTransform ScaleX=".9" ScaleY=".9"/>
    </Setter.Value>
  </Setter>
    <Setter Property="RenderTransformOrigin" Value=".5,.5"/>
  </Trigger>
</ControlTemplate.Triggers>
```

Раскадровки не могут устанавливать такие свойства, как RenderTransform или RenderTransformOrigin, поэтому два упомянутых свойства теперь устанавлива-

ются прямо внутри визуального дерева. Анимации в состоянии Pressed просто изменяют значения ScaleX и ScaleY существующего преобразования ScaleTransform.

Переходы

С состояниями, определенными в листинге 17.4, связана небольшая проблема. Переходы из одного состояния в другое получаются плавными при условии, что конечным состоянием не является Normal или Unfocused. Поскольку они оставлены пустыми, то происходит мгновенный переход к поведению, определенному в визуальном объекте по умолчанию. Эту проблему можно решить, добавив раскадровки Storyboard с явно заданными анимациями в значения по умолчанию, но это пришлось бы делать для каждого свойства, анимируемого любым другим состоянием в группе, чтобы учесть все возможные переходы.

На наше счастье, класс VisualStateGroup предлагает гораздо более изящное решение. В нем определено свойство-коллекция Transitions; в эту коллекцию можно поместить один или несколько объектов VisualTransition, которые умеют автоматически генерировать подходящие анимации для сглаживания переходов между любыми двумя состояниями. В классе VisualTransition имеются строковые свойства To и From, в которые можно записать имена исходного и конечного состояний. Если оба свойства опустить, то объект будет применяться ко всем переходам; можно задать только свойство To, тогда он станет применяться ко всем переходам в это состояние и т. д. При переходе из одного состояния в другое менеджер визуальных состояний выбирает объект VisualTransition, наиболее точно соответствующий данному переходу. Степень точности соответствия определяется следующим образом:

1. VisualTransition с точно указанными To и From.
2. VisualTransition с точно указанным To и без явно заданного From.
3. VisualTransition с точно указанным From и без явно заданного To.
4. VisualTransition, подразумеваемый по умолчанию, To и From явно не заданы.

Если свойство Transitions в объекте VisualStateGroup не установлено, то по умолчанию в качестве перехода между любыми двумя состояниями берется анимация нулевой продолжительности.

Чтобы задать характеристики объекта VisualTransition, можно установить свойство GeneratedDuration, управляющее длительностью сгенерированной линейной анимации. Если также задать свойство GeneratedEasingFunction, то получится нелинейный переход между двумя состояниями. Можно даже записать в свойство Storyboard объект, описывающий раскадровку с произвольными анимациями.

В листинге 17.5 элемент VisualStateGroup модифицирован по сравнению с предыдущим листингом – мы воспользовались элементами VisualTransition, чтобы исправить проблему мгновенного перехода в состояния Normal и Unfocused.

СОВЕТ

Простейший способ управления визуальными состояниями и переходами между ними состоит в том, чтобы для анимаций внутри каждого `VisualState` **установить продолжительность** `Duration`**, равную** `0`**, – тогда они будут похожи скорее на элементы** `Setter`**, чем на настоящие анимации, – и описать желаемые анимации между состояниями (с ненулевой продолжительностью) с помощью свойства** `VisualTransitions` **объекта** `VisualStateGroup`**. Исключение составляют состояния с непрерывными анимациями, например подпрыгивание в состоянии** `Focused` **в листингах 17.4 и 17.5.**

Листинг 17.5. Модифицированные по сравнению с листингом 17.4 элементы VisualStateGroup, в которых используются переходы

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="CommonStates">
  <VisualStateGroup.Transitions>
    <!-- Применить во всем переходам... -->
    <VisualTransition GeneratedDuration="0:0:.4"/>
    <!-- ...но переопределить для переходов в состояние Pressed и из него: -->
    <VisualTransition To="Pressed" GeneratedDuration="0"/>
    <VisualTransition From="Pressed" GeneratedDuration="0"/>
  </VisualStateGroup.Transitions>
    <VisualState Name="Normal"/>
    <VisualState Name="MouseOver">
      <Storyboard>
        <ColorAnimation Storyboard.TargetName="outerCircle"
                        Storyboard.TargetProperty=
    "(Ellipse.Fill).(LinearGradientBrush.GradientStops)[1].(GradientStop.Color)"
                        To="Orange" Duration="0"/>
      </Storyboard>
    </VisualState>
    <VisualState Name="Pressed">
      <Storyboard>
        <DoubleAnimation Storyboard.TargetName="scaleTransform"
                         Storyboard.TargetProperty="ScaleX" To=".9"
                         Duration="0"/>
        <DoubleAnimation Storyboard.TargetName="scaleTransform"
                         Storyboard.TargetProperty="ScaleY" To=".9"
                         Duration="0"/>
      </Storyboard>
    </VisualState>
    <VisualState Name="Disabled">
      <Storyboard>
        <ColorAnimation Storyboard.TargetName="outerCircle"
                        Storyboard.TargetProperty=
    "(Ellipse.Fill).(LinearGradientBrush.GradientStops)[1].(GradientStop.Color)"
                         To="Gray" Duration="0"/>
      </Storyboard>
```

```
  </VisualState>
  </VisualStateGroup>
  <VisualStateGroup Name="FocusStates">
  <VisualStateGroup.Transitions>
    <!-- Применить только в одном направлении: -->
    <VisualTransition To="Unfocused" GeneratedDuration="0:0:.4">
    <VisualTransition.GeneratedEasingFunction>
      <QuadraticEase/>
    </VisualTransition.GeneratedEasingFunction>
  </VisualTransition>
  </VisualStateGroup.Transitions>
    <VisualState Name="Unfocused"/>
    <VisualState Name="Focused">
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty=
     "(Grid.RenderTransform).(TransformGroup.Children)[1].(TranslateTransform.Y)"
                         To="-20" AutoReverse="True"
                         RepeatBehavior="Forever" Duration="0:0:.4">
        <DoubleAnimation.EasingFunction>
          <QuadraticEase/>
        </DoubleAnimation.EasingFunction>
        </DoubleAnimation>
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

ПРЕДУПРЕЖДЕНИЕ

Элементы VisualTransition не работают с анимациями, для которых целевой элемент отсутствует в дереве элементов!

Возможно, вы обратили внимание на длинные пути в свойстве `Storyboard.TargetProperty` в трех анимациях из листингов 17.4 и 17.5:

```
<ColorAnimation Storyboard.TargetName="outerCircle"
                Storyboard.TargetProperty=
      "(Ellipse.Fill).(LinearGradientBrush.GradientStops)[1].(GradientStop.Color)"
                To="Orange" .../>
...
<ColorAnimation Storyboard.TargetName="outerCircle"
                Storyboard.TargetProperty=
      "(Ellipse.Fill).(LinearGradientBrush.GradientStops)[1].(GradientStop.Color)"
                To="Gray" .../>
...
<DoubleAnimation Storyboard.TargetProperty=
      "(Grid.RenderTransform).(TransformGroup.Children)[1].(TranslateTransform.Y)"
                To="-20" AutoReverse="True"
                RepeatBehavior="Forever" Duration="0:0:.4">
...
</DoubleAnimation>
```

В черновом варианте этих листингов были прямые ссылки на GradientStop (в первых двух анимациях) и на TranslateTransform (в последней анимации) с помощью свойства Storyboard.TargetName, чтобы пути выглядели проще:

```
<ColorAnimation Storyboard.TargetName="highlightGradientStop"
                Storyboard.TargetProperty="Color"
                To="Orange" .../>
...
<ColorAnimation Storyboard.TargetName="highlightGradientStop"
                Storyboard.TargetProperty="Color"
                To="Gray" .../>
...
<DoubleAnimation Storyboard.TargetName="translateTransform"
                 Storyboard.TargetProperty="Y"
                 To="-20" AutoReverse="True"
                 RepeatBehavior="Forever" Duration="0:0:.4">
...
</DoubleAnimation>
```

Эти анимации имеют сходную семантику и работают точно так же, как анимации в листингах 17.4 и 17.5, пока мы не пытаемся воспользоваться объектами VisualTransition. Но сгенерированные анимации не работают с анимациями, для которых целевой элемент, поименованный в TargetName, не находится в дереве элементов. Обходных путей два: либо поместить все поведение внутрь элементов VisualState и не пользоваться объектами VisualTransition, либо гарантировать, что для всех интересующих нас анимаций в качестве цели указан элемент, присутствующий в дереве. В листинге 17.5 применен второй подход. (В анимации TranslateTransform целью неявно является корневая сетка Grid.)

Отметим, что анимации в состоянии Pressed все же работают непосредственно с преобразованием ScaleTransform. Они оставлены в таком виде, потому что переходы в это состояние и из него все равно совершаются мгновенно. При желании можно изменить листинг 17.5, чтобы добиться плавного перехода в состояние Pressed и из него. Нужно всего лишь изменить анимации в этом состоянии так, чтобы в качестве цели использовалась корневая сетка Grid, и задать для свойства TargetProperty значения

```
"(Grid.RenderTransform).(TransformGroup.Children)[0].(ScaleTransform.ScaleX)"
```

и

```
"(Grid.RenderTransform).(TransformGroup.Children)[0].(ScaleTransform.ScaleY)"
```

Пути к свойствам в этом разделе записаны с помощью максимально явного синтаксиса, который обычно применяется только к присоединенным свойствам, но вообще-то необязательно записывать их так длинно. Допустимы различные сокращения, например:

```
"Fill.GradientStops[1].Color"

"RenderTransform.Children[1].Y"

"RenderTransform.Children[0].ScaleX"

"RenderTransform.Children[0].ScaleY"
```

В коллекции Transitions для группы CommonStates элемент VisualTransition длительностью 0,4 с применяется ко всем переходам состояний. Два дополнительных элемента VisualTransition переопределяют это поведение для переходов в состояние Pressed и из него, чтобы сохранить мгновенность перехода при нажатии и отпускании кнопки. Поскольку новый элемент VisualTransition берет на себя заботу о плавности анимаций, продолжительности Duration анимаций, определенных для состояний MouseOver и Disabled, теперь равны 0.

Элемент VisualTransition, добавленный в группу FocusStates, применяется только к переходам в состояние Unfocused, чтобы не мешать анимации подпрыгивания в состоянии Focused. Чтобы этот переход не выглядел чужеродным при выходе из анимации подпрыгивания в состоянии Focused, с элементом VisualTransition связана переходная функция QuadraticEase, согласованная с анимацией, из которой производится переход.

Резюме

С помощью анимации можно делать очень многое: от простейшего эффекта наката (который теперь уже повсеместно применяется даже в стандартных пользовательских интерфейсах) до мультфильмов. Элементы Storyboard, необходимая составная часть анимации на чистом XAML, помогают организовать сложные последовательности анимаций и управлять ими.

Хотя излишеств следует избегать во всех сферах WPF, к анимации это относится особенно – злоупотребление ею может сделать приложение или компонент неудобным либо непригодным для людей с ограниченными возможностями. Не следует также забывать о влиянии анимации на производительность. Стоит переусердствовать с анимацией, как в общем-то полезное приложение может оказаться неработающим на не слишком мощном компьютере, например на нетбуке.

К счастью, WPF позволяет реализовать насыщенную анимацию (или иную функциональность) на мощном компьютере, откатившись к менее требовательным решениям, когда мощности системы не хватает. Ключом к этому механизму является класс RenderCapability в пространстве имен System.Windows.Media. В нем определено статическое свойство Tier и статическое событие TierChanged. Если вы работаете на компьютере уровня 0, то вся визуализация производится программно. На компьютере уровня 1 иногда применяется аппаратная визуализация. А на компьютере уровня 2 (самый высокий) аппаратная визуализация применяется всюду, где только *возможно*. Таким образом, в системе уровня 0 следует ограничить применение нескольких одновременных анимаций (равно как сложных градиентов и трехмерной графики). Помимо удаления анимаций есть еще один способ адаптироваться к работе на компьютерах нижнего уровня: уменьшить естественную частоту кадров (обычно равную 60 кадров/с) с помощью присоединенного свойства DesiredFrameRate элемента Storyboard. Это может сократить потребление процессорного времени.

СОВЕТ

Если вы ловите себя на том, что слишком увлеклись анимацией (или сложной статической графикой, все равно двумерной или трехмерной), пользуйтесь свойством `RenderCapability.Tier` для корректировки. Отметим, что, хотя свойство `Tier` – 32-разрядное целое, содержательная часть хранится в старшем слове. Поэтому, чтобы получить истинный уровень, необходимо сдвинуть значение вправо на 16 бит:

```
int tier = RenderCapability.Tier >> 16
```

Так было сделано для того, чтобы в будущем иметь возможность ввести подуровни, но в результате получилась сплошная путаница!

- Аудио
- Видео
- Речь

18

Аудио, видео и речь

В этой главе рассматриваются те области мультимедиа, которым в последние десять лет индустрия производства ПО уделяла особое внимание: аудио, видео и речь (последнее может считаться очень частным случаем аудио). Во всех трех областях Windows Presentation Foundation существенно снижает барьер сложности по сравнению с предшествующими технологиями. (Аудио, видео и речь схожи в том, что их проблематично продемонстрировать в книге со статическими изображениями!) Так что даже если вы не включали эти возможности в свои программы раньше, прочитав эту главу, вы, возможно, измените свое отношение к ним!

Аудио

В WPF с аудио очень легко работать. Но, в отличие от многих других аспектов, поддержка аудио не является революционной, не претендует на принадлежность к «следующему поколению» и даже не учитывает последних достижений в области оборудования. Это просто слой над функциональностью, уже существующей в Win32 и Windows Media Player, которая покрывает большинство потребностей в аудиосредствах. Правда, на основе WPF вам не удастся создать профессиональное аудиоприложение, зато вы сможете украсить свою программу музыкальными и звуковыми эффектами!

Как часто бывает в WPF, воспроизвести аудио можно разными способами, и у каждого есть свои плюсы и минусы. Средства работы с аудио представлены следующими классами:

- `SoundPlayer`
- `SoundPlayerAction`
- `MediaPlayer`
- `MediaElement` и `MediaTimeline`

Класс SoundPlayer

Простейший способ воспроизвести аудиофайл в WPF-приложении – воспользоваться механизмом, применяемым и в приложениях других типов: классом `System.Media.SoundPlayer`. Класс `SoundPlayer`, входящий в состав .NET Framework еще с версии 2.0, – это простая обертка вокруг PlaySound API на платформе Win32. Это означает, что ему присущ целый ряд ограничений:

- Поддерживаются только аудиофайлы в формате WAV.
- Не поддерживается одновременное воспроизведение нескольких звуков (любой новый звук прерывает уже звучащий).
- Не поддерживается переменная громкость звука.

Однако это наименее ресурсоемкий подход к воспроизведению звука, поэтому он вполне подходит для простых звуковых эффектов. В следующем фрагменте показано, как воспроизвести звук с помощью класса `SoundPlayer`:

```
SoundPlayer player = new SoundPlayer("tada.wav");
player.Play();
```

Переданная конструктору `SoundPlayer` строка может быть именем или URL-адресом файла. Начиная с версии .NET Framework 3.5 можно использовать абсолютные или относительные URI со схемой `pack` – точно так же, как в элементах управления, например `Image`. Поэтому звуковой файл можно включить в проект, как любой другой двоичный ресурс WPF (с действием при построении `Resource` или `Content`), или оставить автономным в первоисточнике.

Метод `Play` воспроизводит звук асинхронно, но можно вызвать метод `PlaySync` для синхронного воспроизведения в текущем потоке или метод `PlayLooping` для повторяющегося асинхронного воспроизведения до тех пор, пока не будет вызван метод `Stop` (либо не начнется воспроизведение другого звука в каком-нибудь экземпляре `SoundPlayer` или даже в результате прямого обращения к базовому Win32 API).

Из соображений производительности аудиофайл не загружается до момента первого воспроизведения звука. Но такое поведение может привести к нежелательной задержке, особенно если большой файл загружается по сети. Поэтому в классе `SoundPlayer` определены также методы `Load` и `LoadAsynch`, которые позволяют произвести загрузку еще до первого воспроизведения.

На тот случай, когда требуется воспроизвести знакомый системный звук, не заботясь о том, где находится соответствующий файл на целевом компьютере, в пространстве имен `System.Media` имеется также класс `SystemSounds`, обладающий статическими свойствами `Asterisk`, `Beep`, `Exclamation`, `Hand` и `Question`. Каждое свойство имеет тип класса `SystemSound`, в котором есть собственный метод `Play` (только для асинхронного неповторяющегося воспроизведения). Но лично я использовал бы звуки из этого класса лишь изредка (а то и не использовал бы вовсе), чтобы не смущать пользователей трелями, которые они ожидают услышать только от самой Windows!

Класс SoundPlayerAction

Если вы собираетесь использовать класс SoundPlayer для добавления простых звуковых эффектов к таким событиям пользовательского интерфейса, как наведение указателя мыши или нажатие кнопки, то можете без труда определить обработчик события, в котором будут обращения к SoundPlayer. Но в WPF имеется класс SoundPlayerAction (производный от TriggerAction), который позволяет использовать SoundPlayer вообще без написания процедурного кода.

В следующем фрагменте XAML прямо внутрь элемента Button вставляются триггеры EventTrigger, которые воспроизводят аудиофайл при нажатии кнопки или при наведении на нее указателя мыши:

```
<Button>
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
  <EventTrigger.Actions>
    <SoundPlayerAction Source="click.wav"/>
  </EventTrigger.Actions>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseEnter">
  <EventTrigger.Actions>
    <SoundPlayerAction Source="hover.wav"/>
  </EventTrigger.Actions>
  </EventTrigger>
</Button.Triggers>
</Button>
```

Класс SoundPlayerAction просто обертывает класс SoundPlayer таким образом, чтобы его было удобно использовать в триггерах, поэтому он подвержен всем вышеперечисленным ограничениям. На самом деле ограничений даже *больше*, потому что способ взаимодействия с SoundPlayer не настраивается. SoundPlayerAction сам конструирует объект SoundPlayer с тем именем файла, которое указано в его свойстве Source, и при активации соответствующего действия вызывает метод Play этого объекта. Нет возможности ни воспроизвести звук синхронно (да и зачем бы?), ни повторять его в цикле, ни загрузить аудиофайл предварительно.

Класс MediaPlayer

Если присущие классам SoundPlayer и SoundPlayerAction ограничения неприемлемы, то можно воспользоваться специфичным для WPF классом MediaPlayer из пространства имен System.Windows.Media. Он создан поверх Windows Media Player, поэтому поддерживает все те же аудиоформаты (WAV, WMA, MP3 и прочие). С его помощью можно воспроизводить несколько звуков одновременно (хотя и в разных экземплярах MediaPlayer) и регулировать громкость звука за счет свойства Volume, принимающего значения от 0 до 1 (по умолчанию подразумевается 0.5).

Но MediaPlayer предоставляет и массу других средств управления аудио:

- Можно приостанавливать воспроизведение методом `Pause` (если свойство `CanPause` возвращает `true`).
- Можно заглушить звук, присвоив значение `true` свойству `IsMuted`.
- Можно управлять балансом между левым и правым динамиками, присваивая свойству `Balance` значение от `-1` до `1`. Значение `-1` означает, что весь аудиопоток посылается в левый динамик, `0` (по умолчанию) – в оба динамика, а `1` – в правый динамик.
- Для тех аудиоформатов, которые это поддерживают, можно ускорять или замедлять воспроизведение (не меняя высоты звука), присваивая свойству `SpeedRatio` произвольное неотрицательное значение типа `double`. По умолчанию подразумевается `1.0`, поэтому величина, меньшая `1.0`, замедляет воспроизведение, а большая `1.0` – ускоряет.
- Можно узнать продолжительность аудиоклипа, опросив свойство `NaturalDuration` (которое не зависит от `SpeedRatio`), и текущую точку воспроизведения – с помощью свойства `Position`.
- Если аудиоформат поддерживает поиск, то свойство `Position` позволяет даже *установить* точку воспроизведения.

Ниже показан простейший способ использования `MediaPlayer` для воспроизведения аудиофайла:

```
MediaPlayer player = new MediaPlayer();
player.Open(new Uri("music.wma", UriKind.Relative));
player.Play();
```

Один экземпляр позволяет воспроизводить несколько аудиофайлов, но только по одному за раз. После открытия файла методом `Open` к этому файлу можно применять такие методы, как `Play`, `Pause` и `Stop`. С помощью метода `Close` можно освободить ресурсы, связанные с файлом (при этом воспроизведение останавливается). Файл всегда воспроизводится асинхронно, поэтому не стоит вызывать `Close` сразу после показанных выше трех строк, потому что в этом случае вы вообще ничего не услышите!

СОВЕТ

Дополнительные сведения о классе `MediaPlayer` и его особенностях изложены в разделе «Видео» ниже, поэтому прочитайте его, даже если не собираетесь пользоваться встроенной в WPF поддержкой видео.

Классы MediaElement и MediaTimeline

Класс `MediaPlayer` обладает большей гибкостью, чем `SoundPlayer`, но предназначен только для использования в процедурном коде. (Его основная функциональность раскрывается только через методы, свойства не являются свойствами зависимости, а события не маршрутизируются.) Но как `SoundPlayerAction`

обертывает SoundPlayer, делая возможным декларативное использование, так класс MediaElement обертывает с той же целью MediaPlayer.

MediaElement представляет собой полноценный элемент FrameworkElement и находится в пространстве имен System.Windows.Controls. Таким образом, его можно делать частью пользовательского интерфейса, он принимает участие в компоновке и т. д. (Это кажется странным, пока вы не осознаете, что MediaElement служит также для показа видео, о чем пойдет речь в следующем разделе.) Большинство свойств MediaElement являются свойствами зависимости, а его события маршрутизируются.

В свойство Source объекта MediaElement можно записать URI-адрес аудиофайла, но воспроизведение начнется только после того, как элемент будет загружен. Для декларативного воспроизведения в произвольный момент времени можно устанавливать свойство Source динамически, применяя анимацию с помощью класса MediaTimeline.

В следующем фрагменте XAML, напоминающем приведенный выше пример использования класса SoundPlayerAction, показано, как с помощью элементов MediaElement и MediaTimeline воспроизвести аудиофайл при нажатии кнопки или наведении на нее указателя мыши.

```
<MediaElement x:Name="audio"/>
...
<Button>
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <MediaTimeline Source="click.wma" Storyboard.TargetName="audio"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.MouseEnter">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <MediaTimeline Source="hover.wma" Storyboard.TargetName="audio"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
  </EventTrigger>
</Button.Triggers>
</Button>
```

Помимо действия BeginStoryboard ту же самую раскадровку Storyboard можно использовать в действиях PauseStoryboard, ResumeStoryboard, SeekStoryboard и StopStoryboard соответственно для приостановки, возобновления, подмотки и остановки воспроизведения аудиофайла.

СОВЕТ

Чтобы создать фоновое звуковое сопровождение, воспроизводящееся в цикле, можно присвоить свойству `RepeatBehavior` элемента `MediaTimeline` значение `Forever` и использовать этот элемент в триггере, срабатывающем по событию `Loaded` элемента `MediaElement`. Например:

```
<MediaElement x:Name="audio">
<MediaElement.Triggers>
  <EventTrigger RoutedEvent="MediaElement.Loaded">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <MediaTimeline Source="music.mp3" Storyboard.TargetName="audio"
          RepeatBehavior="Forever"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
  </EventTrigger>
</MediaElement.Triggers>
</MediaElement>
```

К сожалению, по достижении конца ролика и до начала его воспроизведения сначала может возникать небольшая пауза. Один (весьма странный) способ преодолеть эту проблему заключается в том, чтобы создать видео с нужным аудиороликом в качестве звуковой дорожки и записать в `Source` адрес этого видеофайла (а сам элемент `MediaElement` сделать невидимым). Это результат того, что WPF теснее интегрирована с видео и в данном случае поддерживает циклическое воспроизведение без паузы.

Видео

Поддержка видео в WPF основана на классе `MediaPlayer`, описанном в предыдущем разделе, а также на сопутствующих ему классах `MediaElement` и `MediaTimeline`. Таким образом, все форматы файлов, поддерживаемые Windows Media Player (WMV, AVI, MPG и прочие), поддерживаются и WPF-приложениями. Информация этого раздела относится также к воспроизведению аудио с помощью классов `MediaPlayer` и/или `MediaElement`.

ПРЕДУПРЕЖДЕНИЕ

Поддержка аудио и видео в WPF опирается на Windows Media Player версии 10 или выше!

Если на компьютере не установлен Windows Media Player хотя бы версии 10, то при попытке использовать класс `MediaPlayer` (и связанные с ним) возникает исключение. Это относится только к версиям операционной системы, предшествующим Windows Vista.

ПРЕДУПРЕЖДЕНИЕ

До Windows Vista существовала только 32-разрядная версия Windows Media Player!

В 64-разрядные версии Windows до выхода Windows Vista была включена только 32-разрядная версия Windows Media Player. Поскольку поддержка видео (и расширенных возможностей аудио) в WPF базируется на Windows Media Player, то ею невозможно воспользоваться в 64-разрядном приложении на одной из таких платформ. Необходимо, чтобы ваше приложение работало в 32-разрядном режиме. В этом случае оно автоматически будет пользоваться 32-разрядной версией .NET Framework (которая устанавливается вместе с 64-разрядной).

Управление визуальными аспектами класса MediaElement

Как и в классах `Viewbox` и `Image`, в классе `MediaElement` определены свойства `Stretch` и `StretchDirection`, которые управляют тем, как видео заполняет отведенное место. На рис. 18.1 представлены три значения `Stretch` в применении к элементу `MediaElement`, находящемуся непосредственно в окне `Window`:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <MediaElement Source="C:\Users\Public\Videos\Sample Videos\butterfly.wmv"
    Stretch="XXX"/>
</Window>
```



`Uniform` (по умолчанию) `Fill` `UniformToFill`

Рис. 18.1. *Элемент MediaElement при трех разных значениях свойства Stretch*

Конечно, замечательной особенностью класса `MediaElement` является то, что он, как и большинство других элементов, производных от `FrameworkElement`, позволяет манипулировать видео куда более разнообразно. Так, в следующем

XAML-коде, визуализированном на рис. 18.2, два экземпляра видео помещены один поверх другого, причем оба полупрозрачны, оба обрезаны по кругу и один повернут на 180°:

```
<Canvas>
  <MediaElement Source="C:\Users\Public\Videos\Sample Videos\butterfly.wmv"
    Opacity="0.5">
  <MediaElement.Clip>
    <EllipseGeometry Center="220,220" RadiusX="220" RadiusY="220"/>
  </MediaElement.Clip>
  <MediaElement.LayoutTransform>
    <RotateTransform Angle="180"/>
  </MediaElement.LayoutTransform>
  </MediaElement>

  <MediaElement Source="C:\Users\Public\Videos\Sample Videos\butterfly.wmv"
    Opacity="0.5">
  <MediaElement.Clip>
    <EllipseGeometry Center="220,220" RadiusX="220" RadiusY="220"/>
  </MediaElement.Clip>
  </MediaElement>
</Canvas>
```

Более того, поместив `MediaElement` внутрь кисти `VisualBrush`, вы сможете использовать видео практически всюду – в качестве фона списка `ListBox`, материала 3D-поверхности и т. д. Только прежде чем увлекаться визуальными кистями на основе видеороликов, не забывайте измерять производительность.



***Рис. 18.2.** Обрезанный, повернутый и полупрозрачный видеоролик в двух разных элементах MediaElement*

FAQ

Как получить отдельный кадр видео?

Чтобы сделать стоп-кадр, можно установить свойство `Position` видео. Но если требуется сохранить кадр в виде объекта `Image`, то следует вывести объект `MediaElement` в `RenderTargetBitmap` (как и любой другой объект `Visual`). Например:

```
MediaElement mediaElement = ...;
Size desiredSize = ...;
Size dpi = ...;
RenderTargetBitmap bitmap = new RenderTargetBitmap(desiredSize.Width,
  desiredSize.Height, dpi.Width, dpi.Height, PixelFormats.Pbgra32);
bitmap.Render(mediaElement);
Image image = new Image();
image.Source = BitmapFrame.Create(bitmap);
```

Если вы работаете с `MediaPlayer`, а не `MediaElement`, то можно создать объект `DrawingVisual`, который будет передан методу `Render` объекта `RenderTargetBitmap`:

```
DrawingVisual visual = new DrawingVisual();
MediaPlayer mediaPlayer = ...;
Size desiredSize = ...;
using (DrawingContext dc = visual.RenderOpen())
{
  dc.DrawVideo(mediaPlayer, new Rect(0, 0, desiredSize.Width,
    desiredSize.Height));
}
```

Самым важным в этом фрагменте является метод `DrawVideo` класса `DrawingContext`, который принимает экземпляр `MediaPlayer` и прямоугольник `Rect`. На самом деле, когда `MediaElement` визуализирует видео, он обращается к `DrawVideo` из своего метода `OnRender`!

Управление мультимедийным содержимым

В двух предыдущих фрагментах XAML мы поступали просто – устанавливали свойство `Source` элемента `MediaElement` напрямую. В результате мультимедийный файл начинает воспроизводиться сразу после загрузки элемента. Но более вероятно, что вы захотите воспроизводить, приостанавливать и останавливать видео в конкретные моменты времени. Это можно сделать так же, как в разделе «Аудио», то есть с помощью триггера, в котором используется элемент `MediaTimeline`. В приведенном ниже фрагменте также есть триггеры, в которых используются раскадровки `PauseStoryboard` и `ResumeStoryboard`, а все вместе дает реализацию простенького медиаплеера.

```
<Grid>
<Grid.Triggers>
  <EventTrigger RoutedEvent="Button.Click" SourceName="playButton">
```

```
    <EventTrigger.Actions>
      <BeginStoryboard Name="beginStoryboard">
        <Storyboard>
          <MediaTimeline Source="C:\Users\Public\Videos\Sample Videos\butterfly.wmv"
            Storyboard.TargetName="video"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="pauseButton">
    <EventTrigger.Actions>
      <PauseStoryboard BeginStoryboardName="beginStoryboard"/>
    </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="resumeButton">
    <EventTrigger.Actions>
      <ResumeStoryboard BeginStoryboardName="beginStoryboard"/>
    </EventTrigger.Actions>
    </EventTrigger>
  </Grid.Triggers>

    <MediaElement x:Name="video"/>
    <StackPanel Orientation="Horizontal" VerticalAlignment="Bottom">
      <Button x:Name="playButton" Background="#55FFFFFF" Height="40">Play</Button>
      <Button x:Name="pauseButton" Background="#55FFFFFF" Height="40">Pause</Button>
      <Button x:Name="resumeButton" Background="#55FFFFFF" Height="40">Resume
      </Button>
    </StackPanel>
  </Grid>
```

Пользовательский интерфейс состоит из трех полупрозрачных кнопок `Button`, управляющих воспроизведением видео под ними (рис. 18.3).



Рис. 18.3. *Простой видеоплеер с кнопками, которые управляют показом видео с помощью раскадровок*

Хотя по умолчанию мультимедийное содержимое, заданное в качестве значения свойства `Source` элемента `MediaElement`, начинает воспроизводиться сразу

после загрузки элемента, это поведение можно изменить с помощью свойств LoadedBehavior и UnloadedBehavior, оба типа MediaState. В перечислении MediaState определены следующие значения: Play (подразумевается по умолчанию для LoadedBehavior), Pause, Stop, Close (подразумевается по умолчанию для UnloadedBehavior) и Manual.

СОВЕТ

Когда элемент MediaTimeline комбинируется с другими анимациями внутри того же элемента Storyboard, иногда хочется изменить способ синхронизации анимаций. При воспроизведении мультимедиа часто возникает начальная задержка, обусловленная загрузкой и буферизацией данных, в результате чего видео отстает от других анимаций. А если назначить элементу Storyboard фиксированную продолжительность, то из-за таких задержек конец медиасодержимого может оказаться обрезан.

Чтобы изменить это поведение, можно присвоить свойству SlipBehavior элемента Storyboard значение Slip вместо подразумеваемого по умолчанию Grow. Тогда все анимации будут дожидаться готовности медиасодержимого и только потом продолжат выполнение.

На случай если понадобится управлять мультимедиа из процедурного кода, класс MediaElement раскрывает методы того объекта MediaPlayer, который обертывает (Play, Stop и т. д.), но вызывать эти методы можно только тогда, когда и LoadedBehavior, и UnloadedBehavior равны Manual. Кроме того, свойства Position и SpeedRatio тоже можно устанавливать, лишь если элемент находится в ручном режиме (Manual).

Отметим, что ручной режим применим только в случае, когда в триггерах, присоединенных к MediaElement, нет ни одного элемента MediaTimeline. Если MediaElement является целью анимации, то его поведение *всегда* определяется таймером анимации (раскрываемым в виде свойства Clock типа MediaClock) и не может быть изменено вручную иначе как путем взаимодействия с таймером.

СОВЕТ

Чтобы включить в приложение потоковое аудио или видео, достаточно записать в свойство Source URL-адрес потока. Будет работать любая кодировка, поддерживаемая Windows Media Player, например ASF-кодированные WMV-файлы. О том, как включить живое видео с локальной веб-камеры (у которой нет URL-адреса), см. главу 19 «Интероперабельность с другими технологиями».

ПРЕДУПРЕЖДЕНИЕ

Мультимедийные файлы не могут быть внедренными ресурсами!

URI-адреса, задаваемые в качестве значений свойства `Source` элементов `MediaPlayer`, `MediaElement` и `MediaTimeline`, не настолько универсальны, как URI, используемые в других сферах WPF. Это должны быть пути, которые понимает Windows Media Player, например абсолютные или относительные пути в файловой системе или URL-адреса. Это означает, что не существует встроенной поддержки для ссылки на мультимедийный файл, внедренный в качестве ресурса. По иронии судьбы единственный обсуждаемый в этой главе механизм, который поддерживает задание мультимедиа в виде произвольного потока, – это во всем остальном крайне ограниченные классы `SoundPlayer/SoundPlayerAction`!

Это также означает, что нельзя ссылаться на файлы в первоисточнике по адресу вида `pack://siteOfOrigin`. Вместо этого необходимо жестко зашивать путь или URL либо программно извлекать путь к первоисточнику с помощью свойства `ApplicationDeployment.CurrentDeployment.ActivationUri` (в пространстве имен `System.Deployment.Application`, определенном в сборке `System.Deployment.dll`), а затем дописывать его перед именем файла для образования полного URI.

СОВЕТ

Для диагностики ошибок при использовании `MediaPlayer` или `MediaElement` необходимо присоединить обработчик к событию `MediaFailed`, определенному в обоих классах. Это может выглядеть так:

```
<MediaElement Source="nonExistentFile.wmv" MediaFailed="OnMediaFailed"/>
```

Метод `OnMediaFailed` определен в застраничном коде следующим образом:

```
void OnMediaFailed(object o, ExceptionRoutedEventArgs e)
{
  MessageBox.Show(e.ErrorException.ToString());
}
```

Если файл, указанный в свойстве `Source`, не существует, то вместо «безмолвной» ошибки вы увидите такое исключение:

```
System.IO.FileNotFoundException: Cannot find the media file. --->
System.Runtime.InteropServices.COMException (0xC00D1197):
Exception from HRESULT: 0xC00D1197
```

Многие удивляются, узнав, что для получения этого исключения необходимо включать что-то явно. Но дело в том, что из-за асинхронной природы обработки мультимедиа возбуждаемое напрямую исключение не удастся перехватить нигде, кроме глобального обработчика.

FAQ

Как получить ассоциированные с аудио или видео метаданные, например имя исполнителя или жанр?

В WPF нет способа доступа к таким метаданным. Для получения этой информации необходимо воспользоваться неуправляемым API Windows Media Player.

Речь

Речевые API, находящиеся в пространстве имен `System.Speech`, позволяют без особого труда включать в приложение как распознавание, так и синтез речи. Они построены на базе Microsoft SAPI, а поскольку используют стандартизованные W3C грамматики синтеза и распознавания, то прекрасно интегрируются с другими существующими движками.

Хотя пространство имен `System.Speech` появилось вместе с WPF, находящиеся в нем классы *никак не связаны* с WPF; вы не найдете в них свойств зависимости, маршрутизируемых событий, встроенного механизма анимации голоса и прочего. Поэтому их можно спокойно использовать в любом приложении .NET – на базе WPF, Windows Forms и даже консольных.

Синтез речи

Синтез речи (или механизм *текст–речь*) – это процесс преобразования текста в слышимую речь. Для произнесения текста необходим «голос». В последних версиях Windows по умолчанию устанавливается очень приятный голос под названием Microsoft Anna. В комплекте Microsoft SAPI SDK (который можно бесплатно скачать по адресу *http://microsoft.com/speech*) есть как Microsoft Anna, так и другие образцы, например Microsoft Sam, по звучанию больше похожий на голос робота; их можно установить практически в любой версии Windows.

Озвучивание текста

Для синтеза речи понадобится добавить в проект ссылки на сборку `System.Speech.dll`. Соответствующие классы находятся в пространстве имен `System.Speech.Synthesis`. Получить подлежащий озвучиванию текст совсем просто:

```
SpeechSynthesizer synthesizer = new SpeechSynthesizer();
synthesizer.Speak("I love WPF!");
```

Текст произносится синхронно в соответствии с настройками голоса, скорости и громкости, выбранными в диалоговом окне Речь на Панели управления. Для асинхронного произнесения текста следует использовать метод `SpeakAsync` вместо `Speak`:

```
synthesizer.SpeakAsync("I love WPF!");
```

Скорость и громкость произнесения текста можно изменить с помощью свойств Rate и Volume объекта SpeechSynthesizer. И то и другое – целые числа, но Rate изменяется в диапазоне от -10 до 10, а Volume – от 0 до 100. Чтобы отменить еще не произнесенный текст, вызовите метод SpeakAsyncCancelAll.

Если установлено несколько голосов, то в любой момент голос можно сменить, вызвав метод SelectVoice:

```
synthesizer.SelectVoice("Microsoft Sam");
```

Перебрать установленные голоса позволяет метод GetInstalledVoices. Можно даже попытаться выбрать голос, соответствующий указанному полу и возрасту (правда, от выбранного в следующем предложении голоса почему-то мурашки по коже бегают):

```
synthesizer.SelectVoiceByHints(VoiceGender.Female, VoiceAge.Adult);
```

Еще можно направить выход в WAV-файл, а не в динамики, обратившись к методу SetOutputToWaveFile:

```
synthesizer.SetOutputToWaveFile("c:\Users\Adam\Documents\speech.wav");
```

Этот режим распространяется на все последующие вызовы Speak и SpeakAsync. Чтобы перенаправить синтезатор обратно в динамики, следует вызвать метод SetOutputToDefaultAudioDevice.

SSML и PromptBuilder

Передача простых строк объекту SpeechSynthesizer и настройки голоса, скорости, громкости и других параметров уже позволяют сделать довольно много. Но класс SpeechSynthesizer поддерживает также ввод на XML-языке, который называется Speech Synthesis Markup Language (SSML). Он позволяет инкапсулировать сложный текст в одном фрагменте и более точно управлять поведением синтезатора. Содержимое в формате SSML можно передать объекту SpeechSynthesizer напрямую с помощью одного из методов SpeakSsml или SpeakSsmlAsync, но, кроме того, имеются перегруженные варианты методов Speak и SpeakAsync, которые принимают экземпляр класса PromptBuilder.

PromptBuilder – это удобный класс для программного построения сложной речи. С его помощью можно сделать все, что выразимо на языке SSML, но с точки зрения изучения он, вообще говоря, проще.

СОВЕТ

Язык Speech Synthesis Markup Language (SSML) описан в рекомендации W3C, опубликованной по адресу *http://w3.org/TR/speech-synthesis*.

В следующем фрагменте с помощью PromptBuilder строится простой диалог, который затем воспроизводится с помощью передачи методу SpeakAsync:

```
SpeechSynthesizer synthesizer = new SpeechSynthesizer();
PromptBuilder promptBuilder = new PromptBuilder();

promptBuilder.AppendTextWithHint("WPF", SayAs.SpellOut);
promptBuilder.AppendText("sounds better than WPF.");

// Двухсекундная пауза
promptBuilder.AppendBreak(new TimeSpan(0, 0, 2));

promptBuilder.AppendText("The time is");
promptBuilder.AppendTextWithHint(DateTime.Now.ToString("hh:mm"), SayAs.Time);

// Двухсекундная пауза
promptBuilder.AppendBreak(new TimeSpan(0, 0, 2));

promptBuilder.AppendText("Hey Sam, can you spell queue?");

promptBuilder.StartVoice("Microsoft Sam");
promptBuilder.AppendTextWithHint("queue", SayAs.SpellOut);
promptBuilder.EndVoice();

promptBuilder.AppendText("Do it faster!");

promptBuilder.StartVoice("Microsoft Sam");
promptBuilder.StartStyle(new PromptStyle(PromptRate.ExtraFast));
promptBuilder.AppendTextWithHint("queue", SayAs.SpellOut);
promptBuilder.EndStyle();
promptBuilder.EndVoice();

// Произнести весь текст, хранящийся в PromptBuilder
synthesizer.SpeakAsync(promptBuilder);
```

Создав экземпляр `PromptBuilder`, мы добавляем в него содержимое разных типов. Метод `AppendTextWithHint` произносит слова по складам (в результате слово WPF звучит более отчетливо), а строки, представляющие время (например, «08:25») слышатся естественнее. Можно также окружить фрагмент содержимого обращениями к методам `Start`*XXX*/`End`*XXX*, которые изменяют голос или его параметры либо обозначают начало и конец предложений и абзацев. Такие фрагменты могут быть вложенными – точно так же, как элементы XML, соответствующие им в SSML-разметке.

КОПНЕМ ГЛУБЖЕ

Преобразование PromptBuilder в SSML

Чтобы получить SSML-представление объекта `PromptBuilder`, вызовите метод `ToXml` (он отработает нормально при условии, что в момент вызова объект корректно сформирован, например для каждого обращения к `Start`*XXX* имеется парное обращение к `End`*XXX*). Вот как выглядит результат вызова этого метода для объекта `PromptBuilder`, построенного выше (в 8:25 вечера):

```
<speak version="1.0" xmlns="http://www.w3.org/2001/10/synthesis"
  xml:lang="en-US">
  <say-as interpret-as="characters">WPF</say-as>
    sounds better than WPF
  <break time="2000ms"/>
  The time is
  <say-as interpret-as="time">08:25</say-as>
  <break time="2000ms"/>
  Hey Bob, can you spell queue?
  <voice name="Microsoft Sam">
    <say-as interpret-as="characters">queue</say-as>
  </voice>
  Do it faster!
  <voice name="Microsoft Sam">
    <prosody rate="x-fast">
      <say-as interpret-as="characters">queue</say-as>
    </prosody>
  </voice>
</speak>
```

Это удобно, когда нужно сохранить содержимое для последующего воспроизведения.

СОВЕТ

Класс `SpeechSynthesizer` поддерживает даже воспроизведение аудиофайлов в формате WAV! Сделать это можно двумя способами:

```
promptBuilder.AppendAudio("sound.wav");
```

(Можно также включить эквивалентную директиву в SSML-файл, который затем передать методу `SpeakSsml` или `SpeakSsmlAsync`.)

Другой способ – воспользоваться перегруженным вариантом метода `Speak` или `SpeakAsync`, который принимает экземпляр класса, производного от `Prompt`, например `FilePrompt`. Класс `FilePrompt` позволяет воспроизвести содержимое файла – текстового, SSML или WAV:

```
synthesizer.SpeakAsync(new FilePrompt("text.txt", SynthesisMediaType.Text));
synthesizer.SpeakAsync(new FilePrompt("content.ssml", SynthesisMediaType.Ssml));
synthesizer.SpeakAsync(new FilePrompt("sound.wav", SynthesisMediaType.WaveAudio));
```

Распознавание речи

Распознавание речи – процесс, прямо противоположный ее синтезу. Под этим понимается извлечение речевых звуков из аудиовхода с последующим преобразованием их в текст.

СОВЕТ

Чтобы распознавание речи заработало, необходимо установить и запустить соответствующий движок. Он входит в состав Windows Vista и более поздних версий ОС, а также в состав Office XP и более поздних версий. Его можно также бесплатно скачать по адресу *http://microsoft.com/speech*. Чтобы запустить встроенный в Windows движок распознавания речи, необходимо открыть меню Пуск и выбрать пункты Все программы→Стандартные→Специальные возможности→Распознавание речи.

Преобразование произнесенных слов в текст

Для использования средств распознавания речи необходимо включить в проект ссылку на сборку `System.Speech.dll` (ту же самую, что и для синтеза речи). Но сами классы находятся в пространстве имен `System.Speech.Recognition`. Простейшая форма распознавания речи демонстрируется на примере следующего кода, где создается объект `SpeechRecognizer`, загружается грамматика и присоединяется обработчик события `SpeechRecognized`:

```
SpeechRecognizer recognizer = new SpeechRecognizer();
recognizer.LoadGrammar(new DictationGrammar());
recognizer.SpeechRecognized +=
  new EventHandler<SpeechRecognizedEventArgs>(recognizer_SpeechRecognized);
```

Класс `DictationGrammar` – единственная грамматика, входящая в состав .NET Framework, – подходит для распознавания речи в общем случае. Событие `SpeechRecognized` генерируется всякий раз, когда произнесенные слова или фразы преобразуются в текст, поэтому простейший обработчик мог бы выглядеть так:

```
void recognizer_SpeechRecognized(object sender, SpeechRecognizedEventArgs e)
{
  if (e.Result != null)
    textBox.Text += e.Result.Text + " ";
}
```

Если предварительно распознавание речи не было настроено на Панели управления, то при создании объекта `SpeechRecognizer` откроется диалоговое окно, показанное на рис. 18.4. Вряд ли вы захотите увидеть нечто подобное в своем приложении! Даже после того, как распознавание речи сконфигурировано, создание `SpeechRecognizer` автоматически открывает программу распознавания речи, встроенную в Windows. Эта программа выводит небольшое окошко, показанное на рис. 18.5.

Такого взаимодействия с системой распознавания речи в Windows можно избежать, если воспользоваться классом `SpeechRecognitionEngine` вместо `SpeechRecognizer`. Пользователи, которые предварительно не сконфигурировали распознаватель речи, не заметят никакого вмешательства, а от вас потребуется два дополнительных шага:

```
SpeechRecognitionEngine engine = new SpeechRecognitionEngine();
engine.LoadGrammar(new DictationGrammar());
engine.SetInputToDefaultAudioDevice();

// Продолжать, пока не будет вызван метод RecognizeAsyncStop
// или RecognizeAsyncCancel:
engine.RecognizeAsync(RecognizeMode.Multiple);

// Можно использовать тот же обработчик события, что и раньше:
engine.SpeechRecognized +=
  new EventHandler<SpeechRecognizedEventArgs>(recognizer_SpeechRecognized);
```



Рис. 18.4. *При первом использовании распознавателя речи появляется мастер, помогающий настроить микрофон и обучить компьютер звукам вашего голоса*

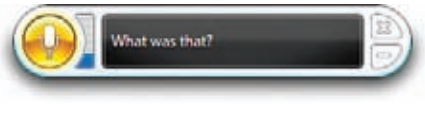


Рис. 18.5. *Встроенная в Windows программа распознавания речи открывает окно, которое может быть плавающим либо пристыкованным к верхнему или нижнему краю экрана, но обязательно должно быть открыто, иначе SpeechRecognizer работать не будет*

В классе `SpeechRecognitionEngine` определены многие члены, имеющиеся в `SpeechRecognizer`, плюс целый ряд дополнительных. При работе с ним необходимо вручную сконфигурировать источник ввода (аудиоустройство по умолчанию,

аудиопоток или WAV-файл на диске) и сообщить, когда начинать прослушивание, вызвав метод `Recognize` или `RecognizeAsync`. Если при вызове `RecognizeAsync` был задан параметр `RecognizeMode.Multiple`, то распознавание будет работать в фоновом режиме, пока вы не вызовете метод `RecognizeAsyncStop` или `RecognizeAsyncCancel`. Метод `RecognizeAsyncStop` завершает работу после окончания текущего действия распознавания, а метод `RecognizeAsyncCancel` — немедленно. Событие `SpeechRecognized` в классе `SpeechRecognitionEngine` работает точно так же, как одноименное событие в классе `SpeechRecognizer`, поэтому приведенный выше обработчик `recognizer_SpeechRecognized` можно не менять.

СОВЕТ

У класса `SpeechRecognitionEngine` есть еще одно преимущество по сравнению со `SpeechRecognizer`. Если на экране присутствует окно распознавателя речи Windows, то этот класс перехватывает хорошо известные устные команды, например «Start» для открытия меню Пуск и «File» для открытия меню Файл текущей программы (если у нее таковое имеется). При использовании класса `SpeechRecognitionEngine` никакие команды не перехватываются, так что эти слова можно обрабатывать, как любые другие.

Любой из описанных выше подходов в сочетании с обработчиком `recognizer_SpeechRecognized` годится для голосового ввода текста в поле `TextBox`. Но в Windows Vista и более поздних версиях это не нужно, поскольку такую возможность вы получаете даром! Например, если включить распознавание речи и передать фокус элементу WPF `TextBox`, то слова, произносимые в микрофон, будут автоматически появляться в поле, как показано на рис. 18.6. Так происходит потому, что система распознавания речи в Windows интегрирована с интерфейсами UI Automation, которые раскрывают элементы WPF. Можно даже производить определенные действия, например нажатие кнопки, произнося их имена автоматизации! (Это касается не только WPF, но также Windows Forms и любой другой технологии построения пользовательских интерфейсов, интегрированной со специальными возможностями Windows.)

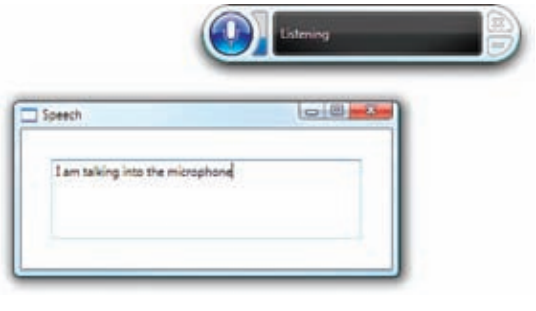


***Рис. 18.6.** Надиктовывание текста в элемент WPF TextBox с помощью программы Windows Speech Recognition*

Распознавание речи обычно применяется для того, чтобы включить в интерфейс программы голосовые команды, более сложные, чем обеспечивает стандартный механизм специальных возможностей. Такие команды, как правило, содержат несколько слов или фраз, заранее известных программе. Чтобы справиться с этой задачей, вы должны сообщить объекту SpeechRecognizer или SpeechRecognitionEngine дополнительную информацию о том, чего ожидаете. Для этого предназначен язык SRGS.

Описание грамматики на языке SRGS

Написать обработчик события SpeechRecognized, который будет программно реагировать на определенные слова или фразы, сложно, если не наложить ограничения на входные данные. Требуется игнорировать посторонние фразы и выбирать существенные слова из более крупных речевых конструкций, предсказать которые заранее невозможно. Например, если требуется реагировать на слово *go*, следует ли принимать также слово *goat*, предполагая, что распознаватель просто неправильно понял пользователя?

Чтобы избежать подобной игры в угадайку, классы SpeechRecognizer и SpeechRecognitionEngine поддерживают описание специальной грамматики на языке Speech Recognition Grammar Specification (SRGS). Имея грамматику, в которой отражены все допустимые входные данные, распознаватель может автоматически игнорировать бессмысленные фразы и тем самым улучшить точность распознавания.

СОВЕТ

Спецификация Speech Recognition Grammar Specification (SRGS) описана в рекомендации W3C, опубликованной по адресу *http://w3.org/TR/speechgrammar*.

Чтобы присоединить грамматику, следует вызвать тот же метод LoadGrammar, что был показан выше. SRGS-грамматики можно описывать на диалекте XML, так что приведенный ниже код загружает грамматику из XML-файла в текущем каталоге:

```
SpeechRecognitionEngine engine = new SpeechRecognitionEngine();
SrgsDocument doc = new SrgsDocument("grammar.xml");
engine.LoadGrammar(new Grammar(doc));
```

Класс SrgsDocument (и другие типы, относящиеся к SRGS) определен в пространстве имен System.Speech.Recognition.SrgsGrammar.

Объект SrgsDocument можно также строить в памяти, для чего существует достаточно развитый API. Ниже показан код, который строит грамматику, допускающую только две команды: stop и go:

```
SpeechRecognitionEngine engine = new SpeechRecognitionEngine();
SrgsDocument doc = new SrgsDocument();
SrgsRule command = new SrgsRule("command", new SrgsOneOf("stop", "go"));
```

```
doc.Rules.Add(command);
doc.Root = command;
engine.LoadGrammar(new Grammar(doc));
```

Но можно выразить и гораздо более сложные грамматики. Следующий пример мог бы встретиться в карточной игре, где допустимы команды вида *three of hearts* (тройка червей) или *ace of spades* (туз пик):

```
SpeechRecognitionEngine engine = new SpeechRecognitionEngine();
SrgsDocument doc = new SrgsDocument();
SrgsRule command = new SrgsRule("command");
SrgsRule rank = new SrgsRule("rank");
SrgsItem of = new SrgsItem("of");
SrgsRule suit = new SrgsRule("suit");
SrgsItem card = new SrgsItem(new SrgsRuleRef(rank), of, new SrgsRuleRef(suit));
command.Add(card);
rank.Add(new SrgsOneOf("two", "three", "four", "five", "six", "seven",
  "eight", "nine", "ten", "jack", "queen", "king", "ace"));
of.SetRepeat(0, 1);
suit.Add(new SrgsOneOf("clubs", "diamonds", "spades", "hearts"));
doc.Rules.Add(command, rank, suit);
doc.Root = command;
engine.LoadGrammar(new Grammar(doc));
```

В этой грамматике карта определяется фразой *rank of suit* (достоинство масти), где *rank* (достоинство) может принимать 13 значений, *suit* (масть) – 4 значения, а предлог *of* можно опускать (поэтому и вызывается метод `SetRepeat`, который устанавливает, что фраза может быть произнесена нуль или один раз).

Описание грамматики с помощью класса GrammarBuilder

Описывать грамматику с помощью API в пространстве имен `System.Speech.Recognition.SrgsGrammar` или на языке SRGS XML (синтаксис которого здесь не рассматривается) бывает довольно сложно. Поэтому в пространстве имен `System.Speech.Recognition` имеется также класс `GrammarBuilder`, который раскрывает наиболее употребительные средства грамматик распознавания в виде гораздо более простых API. В классе `Grammar` (экземпляр которого передается методу `LoadGrammar`) имеется перегруженный конструктор, который принимает объект типа `GrammarBuilder`, поэтому его легко использовать всюду, где допустимо употребление `SrgsDocument`.

Вот, например, как с помощью `GrammarBuilder` можно записать первую грамматику из предыдущего раздела:

```
SpeechRecognitionEngine engine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder(new Choices("stop", "go"));
engine.LoadGrammar(new Grammar(builder));
```

А вот другое определение грамматики карточной игры:

```
SpeechRecognitionEngine engine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder();
builder.Append(new Choices("two", "three", "four", "five", "six", "seven",
```

```
    "eight", "nine", "ten", "jack", "queen", "king", "ace"));
builder.Append("of", 0, 1);
builder.Append(new Choices("clubs", "diamonds", "spades", "hearts"));
engine.LoadGrammar(new Grammar(builder));
```

Класс `GrammarBuilder` не обладает всей мощью и гибкостью класса `SrgsDocument`, но зачастую его возможностей вполне хватает. В карточной игре пользователь может произнести «two clubs» (двойка треф) или «too uh cubs», а обработчик события `SpeechRecognized` должен получить каноническую строку «two of clubs». Можно усложнить грамматику, снабжая отдельные элементы семантическими метками, так чтобы обработчик события мог выделить понятия, например достоинство и масть, даже не разбирая каноническую строку.

Резюме

Поддержка аудио, видео и речи – венец встроенных в WPF богатых мультимедийных средств. Поддержка аудио ограниченна, но достаточна для большинства типичных задач. Поддержка видео составляет лишь подмножество функциональности Windows Media Player, но органичная интеграция с прочими средствами WPF (например, возможность преобразовывать или анимировать видео, как любое другое содержимое) делает ее чрезвычайно привлекательной. Основанная на стандартах поддержка синтеза и распознавания речи в WPF вполне соответствует современному техническому уровню, и этот механизм прост в использовании, хотя и является в основном оберткой вокруг неуправляемого программного интерфейса Microsoft SAPI.

VI

Дополнительные вопросы

Глава 19 «Интероперабельность с другими технологиями»

Глава 20 «Пользовательские и нестандартные элементы управления»

Глава 21 «Компоновка с помощью нестандартных панелей»

- Встраивание элементов управления Win32 в WPF-приложения
- Встраивание элементов управления WPF в Win32-приложения
- Встраивание элементов управления Windows Forms в WPF-приложения
- Встраивание элементов управления WPF в приложения Windows Forms
- Сочетание содержимого DirectX с содержимым WPF
- Встраивание элементов управления ActiveX в WPF-приложения

19

Интероперабельность с другими технологиями

Несмотря на необъятность Windows Presentation Foundation, в ней не хватает кое-каких возможностей, имеющихся в других технологиях. А при создании пользовательского интерфейса на базе WPF иногда хочется эти возможности использовать. Например, в четвертом выпуске WPF *все еще* нет тех стандартных элементов управления, которые вот уже почти десять лет как имеются в Windows Forms: `NumericUpDown`, `NotifyIcon` и др. Кроме того, Windows Forms поддерживает многодокументный интерфейс (MDI) управления окнами, обертки вокруг дополнительных стандартных диалоговых окон и API Win32 и различные удобные классы и методы, например `Screen.AllScreens` (возвращает массив экранов с информацией о физических размерах каждого). В Win32 есть такие элементы управления, как поле ввода IP-адреса (`SysIPAddress32`), у которого нет эквивалентов ни в Windows Forms, ни в WPF. В составе Windows имеется множество элементов пользовательского интерфейса на базе Win32, не поддерживаемых по-настоящему в WPF, скажем эффекты «стекла», диалоговые окна задач и система мастеров. Существуют тысячи элементов управления ActiveX, которые могли бы здорово обогатить ваши приложения. А некоторые технологии рассчитаны на сценарии, принципиально отличающиеся от тех, которые характерны для WPF, но все равно они могли бы занять достойное место в WPF-приложении. Например, к ним относятся высокопроизводительная визуализация на базе DirectX и платформонезависимая визуализация HTML.

Быть может, еще до появления WPF вы вложили немало усилий в разработку собственных интерфейсов и элементов управления и не хотите отправлять все это в корзину. Или на основе какой-то другой технологии написали приложение с очень сложным главным окном (скажем, САПР) и теперь просто

хотите немного «отшлифовать» его, добавив красивые меню, панели инструментов и другие элементы WPF. Или уже создано веб-приложение с богатейшим HTML-содержимым, которое желательно улучшить, но не переписывать полностью.

В предыдущих главах мы рассказывали об интероперабельности WPF и HTML. Учитывая, что HTML-содержимое может отображаться в WPF-элементе `Frame` или `WebBrowser`, а WPF-содержимое можно включить в HTML-страницу (в виде XBAP-приложения или автономного XAML-файла), открывается возможность использовать любое HTML-содержимое (и внедренные в него фрагменты, написанные с помощью Silverlight, Flash и других технологий) совместно с новым WPF-контентом. К счастью, поддержка интероперабельности в WPF этим отнюдь не ограничивается. В WPF-приложения и элементы управления можно относительно легко включать всевозможные сторонние API, в частности все, о чем шла речь в двух предыдущих абзацах. Иногда это делается с помощью механизмов, описанных в данной главе, в том числе благодаря встроенной в каркас .NET Framework интероперабельности между управляемым и неуправляемым кодом, а порой (в случае вызова из WPF различных API Windows Forms) используются управляемые классы, определенные в других технологиях, которые просто находятся в сборках, не относящихся к WPF.

На рис. 19.1 перечислены различные технологии построения пользовательских интерфейсов и показаны способы их сочетания. Платформа Win32 – это общий «колодец», из которого черпают все остальные технологии, работающие в Windows: MFC, WTL, OpenGL и прочие. Обратите внимание, что от WPF к любой другой технологии, кроме Silverlight и ActiveX, ведет прямой путь. А в этих двух случаях необходимо организовать промежуточный слой с использованием какой-то третьей технологии. В Silverlight имеется механизм для работы вне HTML, и он задействован в Visual Studio и Expression Blend. Идея в том, чтобы, воспользовавшись классом `HostingRenderTargetBitmap`, получить представление Silverlight-содержимого в виде растрового изображения, которое затем передать WPF-классу `InteropBitmap` или `WriteableBitmap`. Но это довольно примитивная поддержка, поэтому на рисунке она не показана.

В этой главе рассматриваются все связи между различными технологиями, представленные черными линиями. Связь между Win32 и Windows Forms обеспечивают стандартные для .NET Framework механизмы интероперабельности между управляемым и неуправляемым кодом (и тот факт, что Windows Forms базируется на Win32), а линии между Win32 и ActiveX/DirectX вообще условны, так как Win32 не отделяют от ActiveX и DirectX сколько-нибудь серьезные барьеры.

Предметом нашего внимания будет главным образом встраивание элементов одного типа в приложения другого типа. Сначала мы порознь рассмотрим оба направления интероперабельности между WPF и Win32, а затем оба направления интероперабельности между WPF и Windows Forms. Интероперабельность между WPF и DirectX рассматривается в одном разделе, поскольку в обоих направлениях она реализуется одинаково. Завершается глава рассмотрением различных способов достижения интероперабельности между WPF

и ActiveX. Хотя прежде всего нас будет интересовать встраивание элементов управления, мы поговорим и еще об одном важном сценарии, который не так прост, как может показаться на первый взгляд: запуск гетерогенных диалоговых окон.

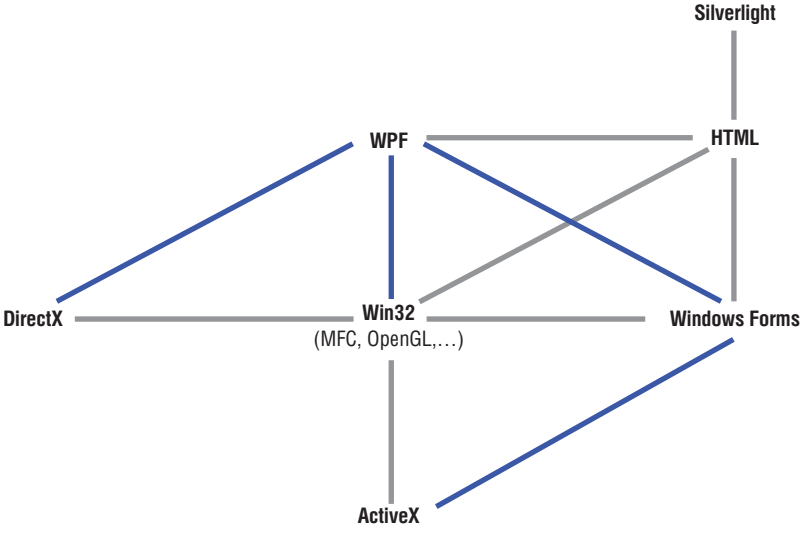


Рис. 19.1. *Отношения между различными технологиями построения пользовательских интерфейсов в Windows*

ПРЕДУПРЕЖДЕНИЕ

Невозможно расположить не-WPF-содержимое поверх WPF-содержимого (если только это не D3DImage)!

Как и в случае размещения HTML-содержимого в элементе `Frame` или `WebBrowser`, на любое не-WPF-содержимое, размещаемое в WPF-приложении, налагаются дополнительные ограничения, не применяемые к «родному» WPF-содержимому. Например, к не-WPF-содержимому невозможно применить геометрические преобразования `Transform`. Кроме того, содержимое, созданное по одной технологии, нельзя наложить на содержимое, созданное по другой технологии. Можно вкладывать Win32-элементы в WPF-компоненты, последние – в формы Windows Forms, а те, в свою очередь, в WPF-приложение, и глубина вложенности может быть произвольной, но при одном непременном условии: за рисование каждого пиксела отвечает одна и только одна технология. Единственным исключением из этого правила является DirectX – и то только в случае использования описанного ниже механизма `D3DImage`, – потому что на внутреннем уровне WPF прибегает для визуализации к услугам DirectX. Поэтому для рисования одного и того же множества пикселов можно комбинировать WPF и DirectX, так как в конечном итоге за их рисование отвечает одна и та же технология (DirectX).

Встраивание элементов управления Win32 в WPF-приложения

В Win32 все элементы управления считаются «окнами», и Win32 API взаимодействует с ними через описатели окон, которые называются также HWND. Все технологии построения пользовательских интерфейсов в Windows (например, DirectX и MFC) в конечном итоге так или иначе используют HWND, поэтому умение работать с HWND создает основу для применения прочих технологий.

Хотя различные подсистемы WPF (компоновка, анимация и т. д.) не умеют работать с HWND напрямую, в WPF имеется подкласс FrameworkElement, который может содержать произвольный HWND. Он называется System.Windows.Interop.HwndHost; с его помощью элементы управления на базе HWND выглядят и ведут себя, почти как настоящие элементы управления WPF.

Для демонстрации использования HwndHost в WPF-приложении рассмотрим встраивание элемента управления Win32, который наделит WPF-приложение способностью работать с веб-камерой. Поддержка видео в WPF не включает никаких средств для взаимодействия с локальными устройствами видеозахвата, в частности с простыми веб-камерами. Но технология Microsoft DirectShow эту возможность обеспечивает, так что благодаря интероперабельности с Win32 мы сможем воспользоваться ею и в WPF-приложении.

Элемент управления Win32 Webcam

В листинге 19.1 приведено определение элемента управления Win32 Webcam на неуправляемом C++. Это обертка вокруг нескольких COM-объектов DirectShow.

Листинг 19.1. Webcam.h – определение API элемента управления Webcam

```
#if !defined(WEBCAM_H)
#define WEBCAM_H

#include <wtypes.h>

class Webcam
{
public:
  static HRESULT Initialize(int width, int height);
  static HRESULT AttachToWindow(HWND hwnd);
  static HRESULT Start();
  static HRESULT Pause();
  static HRESULT Stop();
  static HRESULT Repaint();
  static HRESULT Terminate();
  static int GetWidth();
  static int GetHeight();
};
#endif // !defined(WEBCAM_H)
```

Класс Webcam предназначен для работы с устройством видеозахвата, сконфигурированным по умолчанию, поэтому он содержит только простые статические методы для управления этим устройством. При инициализации задаются ширина и высота элемента (позже их можно будет получить с помощью методов GetWidth и GetHeight). Затем, сообщив классу Webcam (посредством метода AttachToWindow), в каком HWND он должен себя визуализировать, мы можем управлять им с помощью методов Start, Pause и Stop.

В листинге 19.2 показана реализация класса Webcam. Для краткости код методов Webcam::Initialize и Webcam::Terminate опущен, но в коде, прилагаемом к книге (*http://informit.com/title/9780672331190*), реализация приведена полностью.

Листинг 19.2. Webcam.cpp – реализация класса Webcam

```
LRESULT WINAPI WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  switch (msg)
  {
    case WM_ERASEBKGND:
      DefWindowProc(hwnd, msg, wParam, lParam);
      Webcam::Repaint();
      break;

    default:
      return DefWindowProc(hwnd, msg, wParam, lParam);
  }
  return 0;
}

HRESULT Webcam::Initialize(int width, int height)
{
  _width = width;
  _height = height;

  // Создать и зарегистрировать оконный класс
  WNDCLASS wc;
  wc.style         = CS_VREDRAW | CS_HREDRAW;
  wc.lpfnWndProc   = WndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = GetModuleHandle(NULL);
  wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_SCROLLBAR+1);
  wc.lpszMenuName  = 0;
  wc.lpszClassName = L"WebcamClass";
  RegisterClass(&wc);

  HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
      IID_IGraphBuilder, (void **)&_graphBuilder);

  ...Создание нескольких COM-объектов и вызов их методов...
```

```
  return hr;
}

HRESULT Webcam::AttachToWindow(HWND hwnd)
{
  if (!_initialized || !_windowlessControl)
    return E_FAIL;

  _hwnd = hwnd;

  // Установить размер и положение видео
  RECT rcDest;
  rcDest.left = 0;
  rcDest.right = _width;
  rcDest.top = 0;
  rcDest.bottom = _height;
  _windowlessControl->SetVideoClippingWindow(hwnd);
  return _windowlessControl->SetVideoPosition(NULL, &rcDest);
}

HRESULT Webcam::Start()
{
  if (!_initialized || !_graphBuilder || !_mediaControl)
    return E_FAIL;

  _graphBuilder->Render(_pin);
  return _mediaControl->Run();
}

HRESULT Webcam::Pause()
{
  if (!_initialized || !_mediaControl)
    return E_FAIL;

  return _mediaControl->Pause();
}

HRESULT Webcam::Stop()
{
  if (!_initialized || !_mediaControl)
    return E_FAIL;

  return _mediaControl->Stop();
}

HRESULT Webcam::Repaint()
{
  if (!_initialized || !_windowlessControl)
    return E_FAIL;

  return _windowlessControl->RepaintVideo(_hwnd, GetDC(_hwnd));
}

HRESULT Webcam::Terminate()
```

```
{
  HRESULT hr = Webcam::Stop();

  ...Освободить несколько COM-объектов...
  return hr;
}

int Webcam::GetWidth()
{
  return _width;
}

int Webcam::GetHeight()
{
  return _height;
}
```

Реализация начинается с написания простой оконной процедуры Win32, которая гарантирует, что видео будет перерисовано при получении сообщения `WM_ERASEBKGND`. В методе `Initialize` определяется и регистрируется оконный класс `WebcamClass`, а затем создаются и инициализируются COM-объекты, относящиеся к DirectShow. (Все они освобождаются в методе `Terminate`.) Метод `AttachToWindow` не только сообщает DirectShow, в какое окно выводить видео, но также устанавливает размеры видео в соответствии со значениями, переданными методу `Initialize`. Все остальные методы – просто обертки вокруг методов DirectShow.

Использование элемента управления Webcam в WPF

Первым шагом при использовании элемента управления `Webcam` в WPF-приложении является создание проекта, который способен «увидеть» неуправляемый элемент из управляемого кода, относящегося к WPF. Есть много способов интегрировать управляемый код в неуправляемый. Если вы уверенно владеете C++, то обычно наилучшим решением является использование C++/CLI. В особенности это относится к классу `Webcam`, потому что он не экспортирует никакую функциональность за пределы DLL, в которую откомпилирован.

FAQ

Что такое C++/CLI?

C++/CLI – это вариант языка C++ с поддержкой управляемого кода. Игнорируя ныне не рекомендуемые расширения Managed C++, которые присутствовали в ранних версиях Visual C++, C++/CLI является *правильным* способом создания и использования компонентов .NET на языке C++. (Аббревиатура CLI означает Common Language Infrastructure – общеязыковая инфраструктура – и относится к стандартизированным Ecma частям общеязыковой среды выполнения .NET Framework.) Язык C++/CLI стандартизирован Ecma (равно как CLI и C#).

Вообще говоря, Visual C++ – это реализация стандарта C++/CLI корпорацией Microsoft, Visual C# – реализация языка C# корпорацией Microsoft, а общеязыковая среда выполнения (CLR) – реализация языка CLI корпорацией Microsoft. Для использования управляемых механизмов из кода на Visual C++ часто достаточно откомпилировать код с флагом `/clr`, изменить несовместимые флаги и освоить некоторые синтаксические новации, относящиеся к управляемым типам данных.

КОПНЕМ ГЛУБЖЕ

Сочетание управляемого и неуправляемого кода

C++/CLI – это языково-зависимый механизм сочетания управляемого и неуправляемого кода (равно как управляемых и неуправляемых данных) на уровне исходного кода. Но в .NET Framework есть также две языково-независимых технологии интеграции управляемого и неуправляемого кода (иначе говоря, они могут работать с любым .NET-совместимым языком):

- Вызов платформенно-зависимого кода (или PInvoke). Позволяет вызвать любую статическую точку входа в DLL из программы на любом управляемом языке при условии, что сигнатура неуправляемой функции переобъявлена в управляемом коде. Это аналог механизма `Declare` в языке Visual Basic 6.
- Интероперабельность с COM. Позволяет использовать COM-компоненты в любом управляемом языке точно так же, как управляемые компоненты, и наоборот.

Перечислим некоторые преимущества C++/CLI по сравнению с PInvoke и в части интероперабельности с COM:

- Управляемый и неуправляемый код может находиться в одной DLL.
- Обращаться к статическим точкам входа в DLL можно напрямую, без переопределения сигнатур неуправляемых функций.
- Если неуправляемый API изменится, то при компиляции компонентов, нуждающихся в обновлении, будут выданы сообщения об ошибках. При использовании PInvoke нужно не забыть изменить сигнатуры, иначе дело кончится ошибками на этапе выполнения.
- К COM-объектам можно обращаться напрямую, что позволяет избежать разнообразных ограничений механизма интероперабельности с COM. С другой стороны, прямое обращение к COM-объектам из управляемого кода чревато ошибками, хотя в состав Visual C++ входит несколько шаблонов (например, `com_handle`), облегчающих эту задачу.

В листинге 19.3 определен элемент WPF `Window` – тоже написанный на C++/CLI – и для интеграции с элементом управления Win32 `Webcam` используется тип `HwndHost`. Поскольку в этом коде используются и определяются управляемые типы данных, он должен компилироваться с флагом `/clr`.

ПРЕДУПРЕЖДЕНИЕ

Visual C++ не поддерживает откомпилированный XAML!

Именно поэтому в листинге 19.3 окно `Window` определено целиком в процедурном коде. Другие варианты – загрузить и разобрать XAML-код во время выполнения (как показано в главе 2 «Все тайны XAML») или определить `Window` на другом языке, который поддерживает откомпилированный XAML.

Листинг 19.3. Window1.h – элемент WPF Window с использованием класса, производного от HwndHost

```
#include "stdafx.h"
#include "Webcam.h"

#using <mscorlib.dll>
#using <PresentationFramework.dll>
#using <WindowsBase.dll>
#using <PresentationCore.dll>

using namespace System;
using namespace System::Windows;
using namespace System::Windows::Controls;
using namespace System::Windows::Interop;
using namespace System::Runtime::InteropServices;

ref class MyHwndHost : HwndHost
{
protected:
  virtual HandleRef BuildWindowCore(HandleRef hwndParent) override
  {
    HWND hwnd = CreateWindow(L"WebcamClass", // зарегистрированный класс
      NULL,                                  // заголовок
      WS_CHILD,                              // стиль
      CW_USEDEFAULT, 0,                      // позиция
      Webcam::GetWidth(),                    // ширина
      Webcam::GetHeight(),                   // высота
      (HWND)hwndParent.Handle.ToInt32(),     // родитель
      NULL,                                  // меню
      GetModuleHandle(NULL),                 // hInstance
      NULL);                                 // необязательный параметр

    if (hwnd == NULL)
      throw gcnew ApplicationException("CreateWindow failed!");

    Webcam::AttachToWindow(hwnd);

    return HandleRef(this, IntPtr(hwnd));
  }
```

```
  virtual void DestroyWindowCore(HandleRef hwnd) override
  {
    // Чистая формальность:
    ::DestroyWindow((HWND)hwnd.Handle.ToInt32());
  }
};

ref class Window1 : Window
{
public:
  Window1()
  {
    DockPanel^ panel = gcnew DockPanel();
    MyHwndHost^ host = gcnew MyHwndHost();
    Label^ label = gcnew Label();
    label->FontSize = 20;
    label->Content = "The Win32 control is docked to the left.";
    panel->Children->Add(host);
    panel->Children->Add(label);
    this->Content = panel;

    if (FAILED(Webcam::Initialize(640, 480)))
    {
      ::MessageBox(NULL, L"Failed to communicate with a video capture device.",
        L"Error", 0);
    }
    Webcam::Start();
  }

  ~Window1()
  {
    Webcam::Terminate();
  }
};
```

Первое, на что нужно обратить внимание в листинге 19.3, – определение подкласса `MyHwndHost` класса `HwndHost`. Это необходимо потому, что `HwndHost` – абстрактный класс. Он содержит два метода, которые требуется переопределить:

- `BuildWindowCore` – в этом методе определяется `HWND` окна элемента. Именно здесь обычно производится инициализация. В качестве параметра этому методу передается родительский `HWND`. Если метод не вернет дочерний `HWND`, родитель которого совпадает с переданным параметром, WPF возбудит исключение `InvalidOperationException`.
- `DestroyWindowCore` – в этом методе можно произвести необходимую очистку, когда необходимость в `HWND` отпадает.

В обоих методах `HWND` представлены типом `HandleRef`. Это тонкая обертка (класс определен в пространстве имен `System.Runtime.InteropServices`), которая привязывает время жизни `HWND` к управляемому объекту. Обычно при создании `HandleRef` конструктору передается ссылка `this` в качестве управляемого объекта.

В листинге 19.3 внутри метода BuildWindowCore вызывается функция Win32 API CreateWindow для создания окна класса WebcamClass, который был зарегистрирован в листинге 19.2; при этом в качестве описателя родительского окна передается полученный от вызывающей программы HWND. Описатель HWND, возвращенный методом CreateWindow (обернутый в HandleRef), не только возвращается в качестве значения BuildWindowCore, но и передается методу Webcam::AttachToWindow, чтобы видео правильно визуализировалось. В методе DestroyWindowCore вызывается функция Win32 API DestroyWindow, уничтожающая HWND.

СОВЕТ

В типичной реализации подкласса HwndHost функция CreateWindow вызывается в методе BuildWindowCore, а функция DestroyWindow — в методе DestroyWindowCore. Отметим, однако, что вызывать DestroyWindow необязательно, и связано это с тем, что Win32 автоматически удаляет дочерний HWND одновременно с уничтожением родительского. Поэтому в листинге 19.3 реализацию метода DestroyWindowCore можно было бы оставить пустой.

В конструкторе Window экземпляр MyHwndHost создается и добавляется на панель DockPanel, как любой другой элемент FrameworkElement. Затем инициализируется объект Webcam и запускается визуализация видеопотока.

СОВЕТ

В некоторых приложениях с инициализацией содержимого Win32 необходимо подождать до тех пор, пока не будет отрисовано WPF-содержимое. В таких случаях инициализацию следует производить в обработчике события ContentRendered объекта Window.

Последняя показанная в листинге 19.4 часть WPF-приложения webcam — это метод main, который создает окно Window и запускает приложение Application. Он также откомпилирован с флагом /clr. На рис. 19.2 изображено работающее приложение.

Листинг 19.4. HostingWin32.cpp — точка входа в приложение

```
#include "Window1.h"

using namespace System;
using namespace System::Windows;
using namespace System::Windows::Media;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
```

```
    Application^ application = gcnew Application();
    Window^ window = gcnew Window1();
    window->Title = "Hosting Win32 DirectShow Content in WPF";
    window->Background = Brushes::Orange;
    application->Run(window);
    return 0;
}
```



Рис. 19.2. *Видео, снимаемое веб-камерой, в окне WPF-приложения*

СОВЕТ

Флаг `/clr` компилятора Visual C++ можно применять для компиляции как всего проекта, так и отдельных файлов. Возникает искушение компилировать как управляемый код весь проект целиком, но обычно разумнее делать это для каждого файла в отдельности. В противном случае вы только создадите себе лишнюю работу, не получив никакой выгоды.

Флаг `/clr` работает прекрасно, но зачастую увеличивает время построения, а иногда требует внесения в код изменений. Например, при наличии флага `/clr` C-файлы необходимо компилировать как написанные на C++, но для компиляции иногда требуется изменять синтаксис. Кроме того, управляемый код не может работать, когда установлена блокировка загрузчика Windows, поэтому компиляция функции `DllMain` (или любого вызываемого из нее кода) в виде управляемого кода приводит к ошибке во время выполнения (к счастью, со вполне внятным сообщением).

Отметим, что при первой установке флага `/clr` нужно будет изменить несовместимые с ним флаги (например, `/Gm` и `/EHsc`). Впрочем, компилятор четко объясняет, что именно надлежит сделать.

Обратите внимание на серую область под окном видеопотока на рис. 19.2. Появляется она по очень простой причине. Элемент `MyHwndHost` пристыкован к левому краю панели `DockPanel`, однако размер элемента `Webcam` фиксирован при инициализации (640×480).

Если бы мы изменили реализацию метода `Webcam::AttachToWindow` в листинге 19.2 так, чтобы он определял размер `HWND`, то видео можно было бы растянуть на всю область. Такое изменение проделано в коде ниже, а результат показан на рис. 19.3.

```
HRESULT Webcam::AttachToWindow(HWND hwnd)
{
  if (!_initialized || !_windowlessControl)
    return E_FAIL;

  _hwnd = hwnd;

  // Установить размер и положение видео
  RECT rcDest;
  GetClientRect(hwnd, &rcDest);
  _windowlessControl->SetVideoClippingWindow(hwnd);
  return _windowlessControl->SetVideoPosition(NULL, &rcDest);
}
```



Рис. 19.3. *Теперь элемент управления Webcat занимает всю отведенную ему прямоугольную область*

Наверное, лучшим решением в этом приложении было бы назначить производному от `HwndHost` элементу фиксированный размер (или, по крайней мере, не растягивать его). Но важно понимать, что система компоновки WPF при-

меняется только к HwndHost. Внутри этого элемента для выставления желаемого размера следует «играть по правилам» Win32.

Поддержка навигации с помощью клавиатуры

Помимо двух абстрактных методов, которые должны быть реализованы обязательно, в классе HwndHost есть несколько виртуальных методов – их можно переопределить, если требуется естественно переходить между WPF-элементами и встроенными Win32-элементами с помощью клавиатуры. К нашему элементу Webcam это не относится, потому что он никогда не получает фокус клавиатуры. Но если элемент поддерживает ввод, то вы наверняка захотите, чтобы он обладал кое-какими привычными возможностями:

- Переход в Win32-элемент при нажатии клавиши Tab
- Выход из Win32-элемента при нажатии клавиши Tab
- Поддержка клавиш доступа

На рис. 19.4 представлен пример окна WPF Window с двумя элементами управления WPF, расположенными по обе стороны от элемента управления Win32 (обернутого классом HwndHost) с четырьмя дочерними элементами управления Win32. Далее мы обсудим все три вида элементов на этом рисунке. Числами показан ожидаемый порядок навигации. Для трех элементов управления WPF (1, 6 и HwndHost, в который вложены элементы 2–5) порядок навигации неявно определяется последовательностью добавления в коллекцию дочерних элементов родителя, хотя может быть задан и явно с помощью свойства TabIndex. Что же касается элементов управления Win32 (2–5), то их порядок навигации определяется логикой приложения.

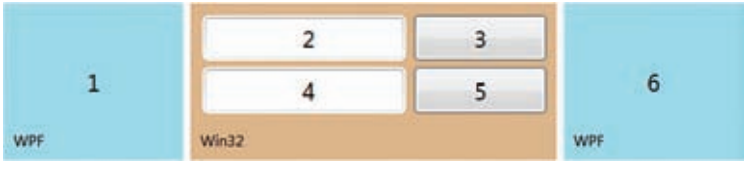


***Рис. 19.4.** Ситуация, когда клавиатурная навигация важна для правильного обхода встроенных элементов управления Win32*

Вход в элемент управления Win32

Под входом в элемент управления Win32 понимаются две вещи:

- Если фокус принадлежит *предшествующему* элементу WPF, то нажатие клавиши Tab передает фокус *первому* потомку родительского элемента Win32. На рис. 19.4 это означает передачу фокуса от 1 к 2.
- Если фокус принадлежит *последующему* элементу WPF, то нажатие сочетания клавиш Shift+Tab передает фокус *последнему* потомку родительского элемента Win32. На рис. 19.4 это означает передачу фокуса от 6 к 5.

Оба действия довольно легко поддержать путем переопределения в подклассе HwndHost метода TabInto, который вызывается, когда HwndHost получает фокус в результате нажатия клавиш Tab или Shift+Tab. На C++/CLI типичная реализация выглядит следующим образом:

```
virtual bool TabInto(TraversalRequest^ request) override
{
  if (request->FocusNavigationDirection == FocusNavigationDirection::Next)
    SetFocus(hwndForFirstWin32Control);
  else
    SetFocus(hwndForLastWin32Control);
  return true;
}
```

Параметр метода TabInto информирует о том, какие клавиши нажал пользователь: Tab (значение равно FocusNavigationDirection.Next) или Shift+Tab (значение равно FocusNavigationDirection.Previous). Получив эту информацию, код решает, какому потомку передать фокус: первому или последнему. Сам же фокус передается с помощью функции Win32 API SetFocus. Передав фокус нужному элементу, метод возвращает true, показывая, что запрос успешно обработан.

Выход из элемента управления Win32

Но, конечно, одной поддержки входа в элемент Win32 недостаточно. Если не поддержать также и *выход*, то клавиатурная навигация так и застрянет в элементе управления Win32. На рис. 19.4 для реализации выхода из элемента Win32 мы должны обеспечить переход от 5 к 6 по нажатию клавиши Tab и от 2 к 1 по нажатию сочетания клавиш Shift+Tab.

Поддержать переход в этом направлении несколько сложнее. Дело в том, что после того как фокус попадает элементу управления Win32, WPF уже не контролирует ход событий. Приложение по-прежнему получает от Windows сообщения, которые в конечном итоге дойдут до HwndHost, но механизм клавиатурной навигации WPF не «видит», что происходит с фокусом.

Поэтому и метода TabOutOf, который можно было бы переопределить, не существует. Зато есть метод TranslateAccelerator, который вызывается, когда приложение получает от Windows сообщение WM_KEYDOWN или WM_SYSKEYDOWN (как и одноименная функция Win32 API). В листинге 19.5 приведена типичная реализация метода TranslateAccelerator на C++/CLI; ее цель – поддержать выход из элемента управления Win32 (а также переходы между такими элементами).

Листинг 19.5. Типичная реализация TranslateAccelerator на C++/CLI

```
virtual bool TranslateAccelerator(MSG% msg, ModifierKeys modifiers) override
{
  if (msg.message == WM_KEYDOWN && msg.wParam == IntPtr(VK_TAB))
  {
    // Обработать Shift+Tab
    if (GetKeyState(VK_SHIFT))
```

```
      {
        if (GetFocus() == hwndOfFirstControl)
        {
          // Мы находимся в начале, поэтому передаем фокус
          // предшествующему элементу WPF
          return this->KeyboardInputSite->OnNoMoreTabStops(
            gcnew TraversalRequest(FocusNavigationDirection::Previous));
        }
        else
          return (SetFocus(hwndOfPreviousControl) != NULL);
      }
      // Обработать Shift без Tab
      else
      {
        if (GetFocus() == hwndOfLastControl)
        {
          // Мы находимся в конце, поэтому передаем фокус
          // следующему элементу WPF
          return this->KeyboardInputSite->OnNoMoreTabStops(
            gcnew TraversalRequest(FocusNavigationDirection::Next));
        }
        else
          return (SetFocus(hwndOfNextControl) != NULL);
      }
    }
  }
```

Методу `TranslateAccelerator` передается ссылка на «исходное» сообщение Windows (представленное в виде управляемой структуры `System.Windows.Interop.MSG`) и элемент перечисления `ModifierKeys`, показывающий, были ли нажаты какие-то из клавиш Shift, Alt, Ctrl или Windows. (Эту информацию можно получить также с помощью функции Win32 API `GetKeyState`.)

Код в листинге выше предпринимает действия только в том случае, когда получено сообщение `WM_KEYDOWN` и нажата клавиша Tab (сюда включается и случай нажатия Shift+Tab). Определив с помощью `GetKeyState`, что именно нажато – Tab или Shift+Tab, – программа должна понять, нужно ли передать фокус *за пределы* всего объемлющего элемента или остаться *внутри* этого элемента. Выход за пределы производится, когда фокус принадлежит первому дочернему элементу и пользователь нажал Shift+Tab или когда фокус принадлежит последнему дочернему элементу и пользователь нажал Tab. В этих случаях мы вызываем метод `OnNoMoreTabStops` объекта, возвращаемого свойством `KeyboardInputSite` элемента `HwndHost`. Таким способом мы сообщаем WPF, что она должна снова начать управлять фокусом. Методу `OnNoMoreTabStops` необходимо передать значение типа `FocusNavigationDirection`, чтобы он знал, какой элемент WPF должен получить фокус (1 или 6 на рис. 19.4). Реализация метода `TranslateAccelerator` должна возвращать `true`, если событие клавиатуры обработано. В противном случае событие всплывает или туннелируется другим элементам. В листинге 19.5 мы обошли молчанием следующее: логика вычис-

ления значений *hwndOfPreviousControl* и *hwndOfNextControl* зависит от конкретного приложение, которое должно на основе анализа HWND текущего элемента определить, какой элемент Win32 станет следующим или предыдущим.

При такой реализации методов TranslateAccelerator и TabInto пользователь представленного на рис. 19.4 приложения сможет с помощью клавиш Tab и Shift+Tab обойти все элементы в порядке от 1 к 6 и от 6 к 1 соответственно.

ПРЕДУПРЕЖДЕНИЕ

При компиляции проекта на C++/CLI могут возникнуть конфликты с определением TranslateAccelerator!

В стандартном заголовочном файле Windows *winuser.h* символ TranslateAccelerator определен как синоним функции Win32 TranslateAcceleratorW (при компиляции с определенным символом UNICODE) или TranslateAcceleratorA (при компиляции с неопределенным символом UNICODE). Поэтому при компиляции написанного на C++ проекта, в который включаются заголовки Win32, вполне вероятен конфликт с WPF-методом TranslateAccelerator. Чтобы предотвратить ошибки компиляции, следует отменить определение этого символа непосредственно перед методом TranslateAccelerator:

```
#undef TranslateAccelerator
```

Поддержка клавиш доступа

Последнее, что нужно от клавиатурной навигации, – это умение переходить к элементу управления при нажатии клавиши доступа (которая часто называется *мнемонической* клавишей). Например, с полями ввода на рис. 19.4, скорее всего, будут ассоциированы метки с клавишами доступа (обозначаемыми подчеркнутой буквой). Если такое поле встроено в WPF-приложение, все равно хотелось бы, чтобы при нажатии комбинации Alt и клавиши доступа фокус передавался соответствующему элементу.

Для поддержки клавиш доступа необходимо переопределить метод OnMnemonic класса HwndHost. Как и TranslateAccelerator, он получает исходное сообщение Windows и элемент перечисления ModifierKeys. Поэтому для поддержки двух клавиш доступа, a и b, можно было бы предложить такую реализацию:

```
virtual bool OnMnemonic(MSG% msg, ModifierKeys modifiers) override
{
  // Проверяем, что получили ожидаемое сообщение
  if (msg.message == WM_SYSCHAR && (modifiers | ModifierKeys.Alt))
  {
    // Преобразуем IntPtr в char
    char key = (char)msg.wParam.ToPointer();

    // Обрабатываем только символы 'a' и 'b'
    if (key == 'a')
```

```
        return (SetFocus(someHwnd) != NULL);
      else if (key == 'b')
        return (SetFocus(someOtherHwnd) != NULL);
    }
    return false;
  }
```

СОВЕТ

Поскольку компилятор C++/CLI вошел только в состав Visual C++ 2005, иногда необходимо модернизировать старый код под новый компилятор. Временами это бывает непросто – из-за приведения компилятора в более точное соответствие со стандартом ISO и различных изменений в заголовочных файлах и библиотеках Windows. Хотя процесс и не автоматический, у перехода на последнюю версию компилятора Visual C++ есть много плюсов даже с точки зрения неуправляемого кода.

FAQ

Как открыть модальный диалог Win32 из WPF-приложения?

Никто не мешает вам и дальше применять свою любимую технику показа диалоговых окон (например, вызывать функцию Win32 `DialogBox`). В C++/CLI для этого достаточно прямого вызова. А в языке типа C# для вызова нужной функции можно воспользоваться технологией PInvoke. Единственная хитрость в том, как получить `HWND` элемента WPF `Window`, который должен передаваться в качестве родителя диалогового окна.

К счастью, получить `HWND` для любого WPF-элемента `Window` можно с помощью класса `WindowInteropHelper` из пространства имен `System.Windows.Interop`. На C++/CLI это выглядит так:

```
WindowInteropHelper^ helper = gcnew WindowInteropHelper(wpfParentWindow);
HWND hwnd = (HWND)helper->Handle.ToPointer();
DialogBox(hinst, MAKEINTRESOURCE(MYDIALOG), hwnd, (DLGPROC)MyDialogProc);
```

Встраивание элементов управления WPF в Win32-приложения

В Win32-приложение можно встроить массу интересных возможностей WPF: трехмерную графику, поддержку форматированных документов, анимацию, изменение стилей и прочее. Даже если все это изобилие вам без надобности, все равно можно воспользоваться такими полезными средствами, как гибкая компоновка и независимость от разрешения устройства.

Интероперабельность на основе HWND – двусторонний механизм, поэтому элементы управления WPF можно встраивать в Win32-приложение точно так же, как элементы управления Win32 в WPF-приложение. В этом разделе мы покажем, как встроить элемент управления WPF DocumentViewer, средство просмотра XPS-документов, в простое Win32-окно с помощью класса HwndSource.

Введение в HwndSource

Класс HwndSource прямо противоположен HwndHost: он представляет элемент WPF Visual в виде описателя HWND. В листинге 19.6 демонстрируется использование класса HwndSource на примере исходного файла на C++ из проекта Win32, включенного в код, который прилагается к данной книге. Он откомпилирован с флагом /clr, так что является управляемым кодом, содержащим как управляемые, так и неуправляемые типы данных.

Листинг 19.6. HostingWPF.cpp – встраивание элемента управления в диалоговое окно Win32

```
#include "stdafx.h"
#include "HostingWPF.h"
#include "commctrl.h"

#using <PresentationFramework.dll>
#using <PresentationCore.dll>
#using <WindowsBase.dll>

LRESULT CALLBACK DialogFunction(HWND hDlg, UINT message, WPARAM wParam,
  LPARAM lParam)
{
  switch (message)
  {
    case WM_INITDIALOG:
    {
      // Описываем HwndSource
      System::Windows::Interop::HwndSourceParameters p;
      p.Windowstyle = WS_VISIBLE | WS_CHILD;
      p.PositionX = 10;
      p.PositionY = 10;
      p.Width = 500;
      p.Height = 350;
      p.ParentWindow = System::IntPtr(hDlg);

      System::Windows::Interop::HwndSource^ source =
        gcnew System::Windows::Interop::HwndSource(p);

      // Присоединяем DocumentViewer к HwndSource
      source->RootVisual = gcnew System::Windows::Controls::DocumentViewer();
```

```
        return TRUE;
      }

      case WM_CLOSE:
        EndDialog(hDlg, LOWORD(wParam));
        return TRUE;
    }
    return FALSE;
}

[System::STAThread]
int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine, int nCmdShow)
{
  DialogBox(hInstance, (LPCTSTR)IDD_MYDIALOG, NULL, (DLGPROC)DialogFunction);
  return 0;
}
```

В этом проекте на языке описания ресурсов Win32 определено простое диалоговое окно (сценарий не показан). Точка входа в приложение (`_tWinMain`) показывает это диалоговое окно с помощью функции Win32 `DialogBox`, указывая `DialogFunction` в качестве оконной процедуры, которая получает сообщения Win32.

В самой функции `DialogFunction` обрабатываются только два сообщения: `WM_INITDIALOG`, которое создает и встраивает элемент управления WPF на этапе инициализации, и `WM_CLOSE`, закрывающее диалоговое окно. Во время обработки сообщения `WM_INITDIALOG` создается структура `HwndSourceParameters` и инициализируются некоторые ее поля, информирующие `HwndSource` о начальном размере, положении и стиле окна. Но самое главное – передается описатель `HWND` родительского окна (в данном случае самого диалогового окна). Для программистов на Win32 такая инициализация не представляет собой ничего нового. Это та самая информация, которая обычно передается функции Win32 `CreateWindow`.

После заполнения структуры `HwndSourceParameters` программе остается сделать всего два простых шага, чтобы появилось WPF-содержимое: создать объект `HwndSource`, передав его конструктору структуру `HwndSourceParameters`, и записать в его свойство `RootVisual` (типа `System.Windows.Media.Visual`) указатель на нужный объект, в данном случае экземпляр класса `DocumentViewer`. Результат показан на рис. 19.5.

Хотя в этом примере мы выбрали стандартный элемент управления WPF, ничто не мешает применить такой же подход к встраиванию собственного произвольно сложного WPF-содержимого. Нужно лишь взять элемент верхнего уровня (например, `Grid` или `Page`) и с помощью `HwndSource` сделать его доступным приложению Win32 в виде одного «большого» `HWND`.

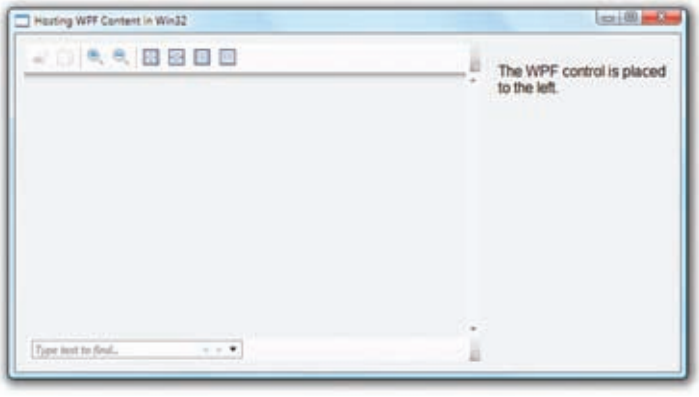


Рис. 19.5. *Элемент WPF DocumentViewer, встроенный в простое диалоговое окно Win32*

ПРЕДУПРЕЖДЕНИЕ

WPF должна работать в STA-потоке!

Как и в случае Windows Forms и еще более ранних технологий, главный поток приложения, в котором используется WPF, должен находиться в однопоточном подразделении. В листинге 19.6 точка входа должна быть снабжена атрибутом `STAThreadAttribute`, потому что весь файл компилируется как управляемый код, а по умолчанию управляемый код выполняется в MTA-подразделении.

Но в проектах, написанных на Visual C++, самый надежный способ принудительно поместить главный поток в STA – воспользоваться флагом компоновщика `/CLRTHREADATTRIBUTE:STA`. Этот способ работает вне зависимости от того, является ли точка входа управляемой или неуправляемой. Что же касается атрибута `STAThreadAttribute`, то его можно применить только к управляемой точке входа.

ПРЕДУПРЕЖДЕНИЕ

Не забудьте установить режим отладчика Visual C++ Mixed!

В больших Win32-приложениях часто имеет смысл включать WPF-части (да и вообще любой управляемый код) в DLL-библиотеки, загружаемые основным исполняемым файлом, но сам этот файл делать полностью неуправляемым. Однако при этом могут возникнуть кое-какие неприятности на этапе разработки.

По умолчанию отладчик Visual C++ работает в режиме `Auto`, то есть отлаживает только управляемый или только неуправляемый код в зависимости от типа исполняемого файла. Но если неуправляемый EXE-файл загружает DLL, содержащую управляемый код, то отлаживать его, когда включен режим отладки неуправляемого кода, невозможно. Решение простое – переключить отладчик в режим `Mixed` (Смешанный).

СОВЕТ

Если при создании объекта `HwndSource` не задавался родительский `HWND`, то родителем будет окно Win32 верхнего уровня, в строке заголовка которого отображается `HwndSourceParameters.Name`. Поэтому создание `HwndSource` без родителя и запись в его свойство `RootVisual` указателя на произвольное WPF-содержимое дает, по существу, тот же результат, что и создание окна WPF `Window` с последующей записью в его свойство `Content` ссылки на то же самое содержимое. Фактически `Window` – это просто обертка вокруг `HwndSource`, правда, функционально весьма насыщенная. Используя `HwndSource` напрямую для создания окна верхнего уровня, вы получаете больший контроль над различными битами стиля окна, но лишаетесь всех полезных членов, определенных в классе `Window` и связанных с ним (в том числе автоматического цикла обработки сообщений, организуемого методом `Application.Run`).

Обеспечение правильной компоновки

Поскольку описываемый сейчас вид интеграции относится к приложениям Win32, то никакой специальной поддержки для компоновки элемента WPF верхнего уровня нет. В листинге 19.6 элемент `DocumentViewer` первоначально располагается в точке `(10,10)` и имеет размер `(500,350)`. И ни размер, ни положение не изменятся, если вы явно не напишете соответствующий код. В листинге 19.7 сделано так, что элемент `DocumentViewer` всегда занимает всю область окна, даже если размер окна изменяется. Результат показан на рис. 19.6.

Листинг 19.7. HostingWPF.cpp – изменение размера элемента управления WPF

```
#include "stdafx.h"
#include "HostingWPF.h"
#include "commctrl.h"

#using <PresentationFramework.dll>
#using <PresentationCore.dll>
#using <WindowsBase.dll>

ref class Globals
{
public:
  static System::Windows::Interop::HwndSource^ source;
};

LRESULT CALLBACK DialogFunction(HWND hDlg, UINT message, WPARAM wParam,
  LPARAM lParam)
{
  switch (message)
  {
    case WM_INITDIALOG:
    {
        System::Windows::Interop::HwndSourceParameters p;
```

```
            p.Windowstyle = WS_VISIBLE | WS_CHILD;
            // Начальный размер и положение не играют роли, потому
            // что обрабатывается сообщение WM_SIZE:
            p.PositionX = 0; p.PositionY = 0;
            p.Width = 100; p.Height = 100;
            p.ParentWindow = System::IntPtr(hDlg);

            Globals::source = gcnew System::Windows::Interop::HwndSource(p);
            Globals::source->RootVisual =
              gcnew System::Windows::Controls::DocumentViewer();
            return TRUE;
        }

        case WM_SIZE:
          RECT r;
          GetClientRect(hDlg, &r);
          SetWindowPos((HWND)Globals::source->Handle.ToPointer(), NULL,
            r.left, r.top, r.right - r.left, r.bottom - r.top, 0);
          return TRUE;

        case WM_CLOSE:
          EndDialog(hDlg, LOWORD(wParam));
          return TRUE;
      }
      return FALSE;
    }

    [System::STAThreadAttribute]
    int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
      LPTSTR lpCmdLine, int nCmdShow)
    {
      DialogBox(hInstance, (LPCTSTR)IDD_MYDIALOG, NULL, (DLGPROC)DialogFunction);
      return 0;
    }
```

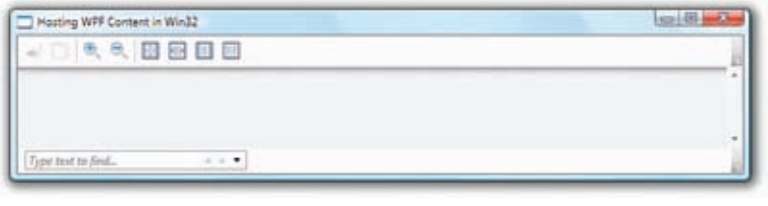


***Рис. 19.6.** Элемент WPF DocumentViewer размещен в простом диалоговом окне Win32 и изменяет размеры вместе с ним*

Наиболее важная часть кода в листинге 19.7 – обработка сообщения `WM_SIZE`. В нем используется функция Win32 API `GetClientRect` для получения текущего размера окна, а затем этот размер применяется к объекту `HwndSource` с помощью

функции Win32 API SetWindowPos. В новой реализации есть два интересных момента:

- Переменная HwndSource теперь «глобальна», то есть может использоваться в любой части программы. Но C++/CLI не допускает по-настоящему глобальных управляемых переменных, поэтому мы применяем стандартную технику – создание статической переменной в управляемом классе.
- Чтобы применять к объекту HwndSource функции Win32 API, в частности SetWindowPos, необходимо иметь его HWND. Его возвращает свойство Handle типа IntPtr. В C++/CLI можно затем вызвать метод ToPointer (который возвращает значение типа void*) и привести результат к типу HWND.

СОВЕТ

Глобальный доступ к HwndSource не нужен, если у вас есть соответствующий HWND. В классе HwndSource определен статический метод FromHwnd, который возвращает экземпляр HwndSource, соответствующий произвольному HWND (если, конечно, этот HWND действительно принадлежит объекту HwndSource). Это очень удобно при модернизации имеющегося кода Win32 с целью включения WPF-содержимого, поскольку описатели HWND часто передаются в качестве параметров. Такая техника позволяет обойтись без определения управляемого класса Globals.

СОВЕТ

Класс HwndSource можно использовать и в чистом WPF-приложении для обработки редко встречающихся сообщений Windows. В чистом WPF-приложении нет необходимости определять оконную процедуру и реагировать на сообщения Windows. Но это вовсе не значит, что сообщений Windows больше нет; окно верхнего уровня по-прежнему имеет описатель HWND и действует по правилам Win32. В предыдущем совете было сказано, что объект WPF Window на самом деле использует HwndSource для размещения произвольного содержимого в HWND верхнего уровня. А внутри WPF имеется оконная процедура, которая представляет различные сообщения по-своему. Например, WPF обрабатывает сообщения WM_SIZE и генерирует событие SizeChanged. Но имеются и такие сообщения Windows, которые WPF не передает программе. Однако для любого объекта WPF Window можно использовать HwndSource, чтобы получить доступ ко всем сообщениям. Ключом к этому является класс System.Windows.Interop.WindowInteropHelper, который возвращает HWND любого окна WPF Window. Получив описатель, вы можете найти соответствующий ему объект HwndSource (с помощью HwndSource.FromHwnd) и присоединить оконную процедуру, вызвав метод HwndSource AddHook.

В главе 8 «Особенности Windows 7» мы так и поступали, чтобы получить доступ к сообщениям WM_DWMCOMPOSITIONCHANGED. А показанное ниже окно Window перехватывает редкое сообщение WM_TCARD, которое подсистема Windows Help посылает при выборе некоторых команд в файле оперативной справочной системы приложения:

```
public partial class AdvancedWindow : Window
{
  ...
  void AdvancedWindow_Loaded(object sender, RoutedEventArgs e)
  {
    // Получить HWND текущего окна Window
    IntPtr hwnd = new WindowInteropHelper(this).Handle;

    // Получить HwndSource, соответствующий HWND
    HwndSource source = HwndSource.FromHwnd(hwnd);

    // Добавить оконную процедуру в HwndSource
    source.AddHook(new HwndSourceHook(WndProc));
  }

  private static IntPtr WndProc(
    IntPtr hwnd, int msg, IntPtr wParam, IntPtr lParam, ref bool handled)
  {
    // Обработать сообщение Win32
    if (msg == WM_TCARD)
    {
      ...
      handled = true;
    }
    return IntPtr.Zero;
  }

  // Определить номера сообщений Win32
  private const int WM_TCARD = 0x0052;
}
```

FAQ

Как открыть модальное диалоговое окно WPF из Win32-приложения?

Чтобы открыть окно WPF `Window` из Win32- или WPF-программы, следует создать соответствующий объект и вызвать его метод `ShowDialog`. Хитрость заключается в том, чтобы назначить окну WPF `Window` правильного родителя. Установить родителя модального диалогового окна необходимо для обеспечения ожидаемого поведения – диалоговое окно должно всегда оставаться поверх родительского, сворачиваться вместе с ним и т. д.

Проблема в том, что свойство `Owner` класса `Window` имеет тип `Window`, и не существует никакого другого свойства или метода, которые позволили бы записать в родителя произвольный `HWND`. Более того, нельзя даже сфабриковать объект `Window` с заданным `HWND`.

Решение в том, чтобы воспользоваться классом `WindowInteropHelper` в пространстве имен `System.Windows.Interop`. Он позволяет не только узнать `HWND` любого окна WPF `Window`, но и установить в качестве его владельца окно с произвольным `HWND`. На C++/CLI это выглядит так:

```
Nullable<bool> LaunchWpfDialogFromWin32Window(Window^ dialog, HWND parent)
{
  WindowInteropHelper^ helper = gcnew WindowInteropHelper(dialog);
  helper->Owner = parent;
  return dialog->ShowDialog();
}
```

Встраивание элементов управления Windows Forms в WPF-приложения

Как мы видели, WPF-приложение может включить элемент управления Win32, обернув произвольный `HWND` объектом `HwndHost`. А любой элемент управления Windows Forms легко представить в виде элемента Win32. (В отличие от элементов управления WPF, все они основаны на использовании `HWND`, поэтому в классе `System.Windows.Forms.Control` имеется свойство `Handle`, прямо возвращающее `HWND`.) Таким образом, для встраивания элементов управления Windows Forms в WPF-приложение можно применять рассмотренную выше технику.

Однако имеется возможность куда теснее интегрировать Windows Forms и WPF, не прибегая к хитроумной работе с `HWND`. Конечно, в этих технологиях и методы визуализации, и элементы управления разные. Но и тут, и там применяются развитые объектные модели .NET со схожими свойствами и событиями и в обоих случаях имеются службы (например, компоновка и привязка к данным), выходящие далеко за рамки общей основы – платформы Win32.

И WPF, пользуясь этими особенностями, располагает встроенной функциональностью для прямой интероперабельности с Windows Forms. Поддержка по-прежнему основана на механизме работы с описателями `HWND`, обсуждаемыми в двух предшествующих разделах, но сверх того имеется ряд средств, существенно упрощающих интеграцию. Вся трудная работа за вас уже сделана, поэтому совместное использование обеих технологий обычно даже не требует написания неуправляемого кода.

Как и в случае интероперабельности с Win32, в WPF определены два класса, по одному для каждого направления взаимодействия. Аналог `HwndHost` называется `WindowsFormsHost` и находится в пространстве имен `System.Windows.Forms.Integration` (в сборке *WindowsFormsIntegration.dll*).

Встраивание PropertyGrid с помощью процедурного кода

В начале этой главы отмечалось, что в Windows Forms есть ряд интересных элементов управления, отсутствующих в WPF. Один из них – весьма развитый элемент PropertyGrid – помогает пролить свет на глубокую интеграцию между Windows Forms и WPF, поэтому давайте включим его в окно WPF Window. (Разумеется, вы точно так же можете включать и элементы Windows Forms собственной разработки.)

Первым делом добавим в WPF-проект ссылки на сборки *System.Windows.Forms.dll* и *WindowsFormsIntegration.dll*. Самым подходящим местом для создания и присоединения элементов Windows Forms является обработчик события Loaded объекта Window. Например, рассмотрим следующее простое окно Window, содержащее элемент Grid с именем grid:

```
<Window x:Class="HostingWindowsFormsControl.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Hosting a Windows Forms Property Grid in WPF"
  Loaded="Window_Loaded">
    <Grid Name="grid"/>
</Window>
```

В показанном ниже обработчике события Loaded мы добавляем в сетку Grid элемент PropertyGrid, используя WindowsFormsHost как промежуточный элемент:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  // Создаем владельца и элемент управления PropertyGrid
  System.Windows.Forms.Integration.WindowsFormsHost host =
    new System.Windows.Forms.Integration.WindowsFormsHost();
  System.Windows.Forms.PropertyGrid propertyGrid =
    new System.Windows.Forms.PropertyGrid();

  // Добавляем PropertyGrid к владельцу, а владельца вставляем в сетку Grid
  host.Child = propertyGrid;
  grid.Children.Add(host);

  // Устанавливаем свойство элемента PropertyGrid
  propertyGrid.SelectedObject = this;
}
```

В плане интеграции нужно всего лишь создать объект WindowsFormsHost и записать в его свойство Child ссылку на интересующий нас объект. Это свойство может содержать ссылку на объект любого класса, производного от System.Windows.Forms.Control.

В последней строке мы записываем в свойство SelectedObject элемента PropertyGrid ссылку на само текущее окно WPF Window, в результате чего получается довольно любопытная ситуация. В сетке свойств PropertyGrid можно отобра-

жать, а иногда и редактировать свойства произвольного объекта .NET. Для этой цели используется механизм отражения. А поскольку объекты WPF – в то же время и объекты .NET, то `PropertyGrid` открывает способ «на лету» изменять свойства текущего окна без написания дополнительного кода. На рис. 19.7 показано определенное выше окно `Window` в действии. Запустив это приложение, вы увидите, как модифицируются значения ширины и высоты при изменении размеров окна; вы сможете изменить размер окна, введя новое значение ширины или высоты, а также сменить цвет фона, стиль рамки и т. д.

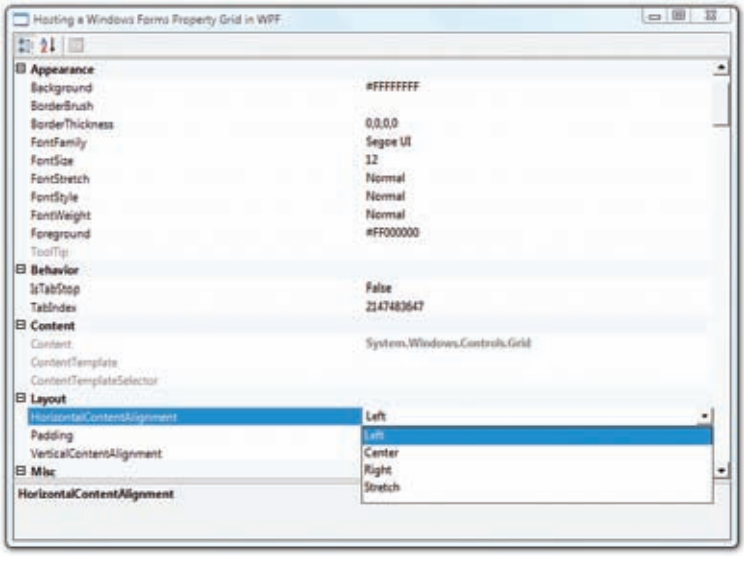


Рис. 19.7. *Встроенный элемент Windows Forms PropertyGrid позволяет динамически изменять свойства окна WPF Window*

Обратите внимание, что в таких свойствах, как `HorizontalContentAlignment`, все допустимые значения автоматически включены в раскрывающийся список благодаря стандартным средствам работы с перечислениями в .NET. Но рис. 19.7 демонстрирует и другие общие черты Windows Forms и WPF помимо того, что в основе обеих технологий лежит .NET. Заметьте, что свойства элемента `Window` сгруппированы по категориям: Behavior (Поведение), Content (Содержимое) и Layout (Компоновка). Эти названия взяты из атрибута `CategoryAttribute`, которым помечаются свойства как в Windows Forms, так и в WPF. Конвертеры типов, применяемые в WPF, также совместимы с Windows Forms, поэтому можно, к примеру, указать цвет «red» (красный), и он автоматически преобразуется в шестнадцатеричный формат ARGB (#FFFF0000). Еще одна приятная особенность `PropertyGrid` заключается в том, что показываются и присоединенные свойства, которые можно применить к объекту, причем в ожидаемом синтаксисе.

СОВЕТ

Класс WindowsFormsHost на самом деле является подклассом HwndHost, поэтому поддерживает все возможности интероперабельности с HWND, описанные выше. Это я говорю на случай, если вы захотите углубиться в низкоуровневые механизмы, например переопределить метод WndProc.

Встраивание элемента PropertyGrid с помощью XAML

Нет никаких причин создавать экземпляр WindowsFormsHost исключительно в процедурном коде; это можно сделать и в XAML-файле. Более того, ничто кроме ограничений на выразительные способности XAML не препятствует использованию в XAML элементов управления. (Элемент управления должен иметь конструктор по умолчанию, какие-то полезные свойства и т. д., если только вы не можете себе позволить использовать XAML2009.)

Не все элементы управления Windows Forms хорошо сочетаются с XAML, но с PropertyGrid в этом отношении все нормально. Скажем, приведенный выше XAML-код можно было бы заменить таким:

```
<Window x:Class="HostingWindowsFormsControl.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:swf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
  Title="Hosting a Windows Forms Property Grid in WPF"
  Loaded="Window_Loaded" x:Name="rootWindow">
  <Grid>
    <WindowsFormsHost>
      <swf:PropertyGrid x:Name="propertyGrid"
                        SelectedObject="{x:Reference rootWindow}"/>
    </WindowsFormsHost>
  </Grid>
</Window>
```

Пространство имен System.Windows.Forms.Integration уже включено в состав стандартных пространств имен, поэтому для использования WindowsFormsHost ничего дополнительно делать не нужно при условии, конечно, что в проекте есть ссылка на сборку *WindowsFormsIntegration.dll*. А если назначить пространству имен .NET System.Windows.Forms префикс swf, то объект PropertyGrid можно будет создать прямо в XAML-разметке. Отметим, что PropertyGrid можно добавить в качестве дочернего элемента WindowsFormsHost, поскольку свойство Child последнего является свойством содержимого. Свойства PropertyGrid также можно устанавливать в XAML, а не в C#. Благодаря расширению разметки x:Reference, в свойство SelectedObject можно записать ссылку на текущий экземпляр Window (которое теперь называется rootWindow), и, следовательно, весь пример реализуется без единой строчки процедурного кода!

СОВЕТ

Часто ошибочно считают, что расширение разметки относится к особенностям XAML2009, которые можно использовать только в автономном XAML-файле. Но, хотя оно появилось только в WPF 4, его можно с тем же успехом употреблять и в XAML2006 при условии, что проект ориентирован на .NET Framework версии 4 или более поздней. Мелкая неприятность заключается в том, что конструктор XAML в Visual Studio 2010 некорректно обрабатывает ключевое слово `x:Reference`, поэтому возникает ошибка конструирования, которую можно спокойно игнорировать:

```
Service provider is missing the INameResolver service
```

СОВЕТ

Установленный по умолчанию внешний вид элементов управления Windows Forms, встроенных в WPF-приложение, может показаться старомодным. Дело в том, что если явно не подключить визуальные стили эпохи Windows XP, то будет использоваться «классическая» библиотека Win32 Common Controls. Это можно сделать, внедрив в приложение специальный файл манифеста, но проще вызвать где-то в приложении метод `System.Windows.Forms.Application.EnableVisualStyles` еще до создания первого элемента Windows Forms. Шаблон Visual Studio для проектов типа Windows Forms автоматически вставляет вызов этого метода, но для проектов типа WPF это не так.

FAQ

Как открыть модальное диалоговое окно Windows Forms из WPF-приложения?

Ответ на такой вопрос должен бы быть простым; создать экземпляр класса, производного от `Form`, и вызвать его метод `ShowDialog`. Но, чтобы при этом получить корректное модальное диалоговое окно, необходимо вызывать перегруженный вариант `ShowDialog`, который принимает параметр `owner`. Однако же владелец должен иметь тип `IWin32Window`, который несовместим с элементом WPF `Window`.

Как объяснялось в предыдущем разделе, получить описатель `HWND` для WPF `Window` можно с помощью класса `WindowInteropHelper` из пространства имен `System.Windows.Interop`, но вот как получить `IWin32Window`? Придется написать специальный класс, который реализует этот интерфейс. К счастью, это довольно просто, потому что в интерфейсе `IWin32Window` определено всего одно свойство `Handle`. В коде ниже определен класс `OwnerWindow`, которым можно воспользоваться в подобной ситуации:

```
class OwnerWindow : IWin32Window
{
  private IntPtr handle;
  public IntPtr Handle
```

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

Имея такой класс, можно написать код, открывающий модальное диалоговое окно Windows Forms, родителем которого является окно WPF `Window`:

```
DialogResult LaunchWindowsFormsDialogFromWpfWindow(Form dialog, Window parent)
{
  WindowInteropHelper helper = new WindowInteropHelper(parent);
  OwnerWindow owner = new OwnerWindow();
  owner.Handle = helper.Handle;
  return dialog.ShowDialog(owner);
}
```

Встраивание элементов управления WPF в приложения Windows Forms

Элементы управления WPF можно встраивать в приложение Windows Forms благодаря наличию класса `ElementHost`, дополняющего `WindowsFormsHost`. Класс `ElementHost` аналогичен `HwndSource`, но адаптирован специально для размещения элементов WPF внутри формы Windows Forms `Form`, а не в окне с произвольным `HWND`. `ElementHost` – это элемент управления Windows Forms (наследует классу `System.Windows.Forms.Control`), который изначально знает, как отображать WPF-содержимое.

Для демонстрации использования `ElementHost` мы создадим простое приложение Windows Forms, которое будет содержать элемент управления WPF `Expander`. Сначала создадим в Visual Studio стандартный проект типа Windows Forms, а затем добавим `ElementHost` на панель элементов, выбрав из меню Tools (Сервис) команду Choose Toolbox Items (Выбор элементов для окна элементов управления). В результате откроется диалоговое окно, показанное на рис. 19.8.

После того как `ElementHost` окажется на панели элементов, его можно будет перетащить в форму, как любой другой элемент Windows Forms. При этом автоматически добавляются ссылки на необходимые сборки WPF (*PresentationFramework.dll*, *PresentationCore.dll* и т. д.). В листинге 19.8 приведен главный исходный файл проекта Windows Forms, в котором форма `Form` содержит элемент `ElementHost` с именем `elementHost`, пристыкованный к левому краю, и метку `Label` справа.

Листинг 19.8. Form1.cs – встраивание элемента WPF Expander в форму Windows Forms Form

```
using System.Windows.Forms;
using System.Windows.Controls;
```

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

      // Создаем элемент WPF Expander
      Expander expander = new Expander();
      expander.Header = "WPF Expander";
      expander.Content = "Content";

      // Добавляем его в ElementHost
      elementHost.Child = expander;
    }
  }
}
```

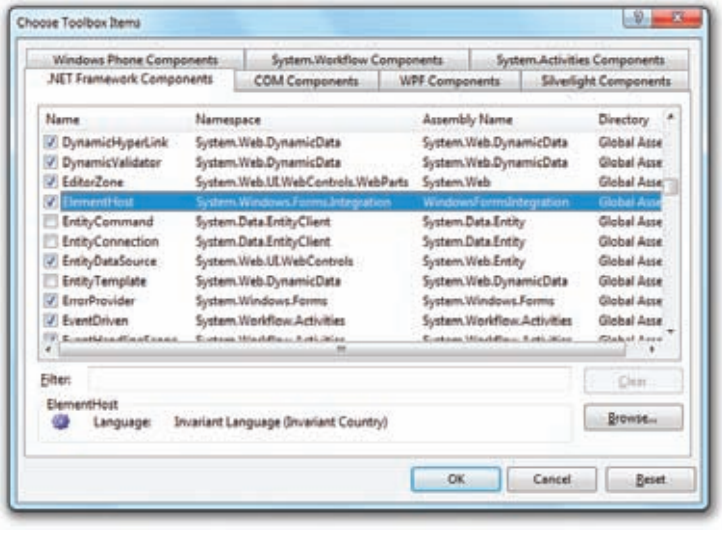


***Рис. 19.8.** Добавление ElementHost на панель элементов в проекте Windows Forms*

В этом коде пространство имен `System.Windows.Controls` добавлено исключительно для `Expander`. Сам элемент создается и инициализируется в конструкторе формы. Как и `WindowsFormsHost`, класс `ElementHost` имеет свойство `Child`, в которое можно записать ссылку на произвольный элемент типа `UIElement`. Значение этому свойству нужно присваивать в исходном коде, а не в конструкторе Windows Forms, и здесь мы записали в него ссылку на экземпляр `Expander`. Результат представлен на рис. 19.9. Отметим, что по умолчанию `Expander` занимает все место, отведенное под `ElementHost`.

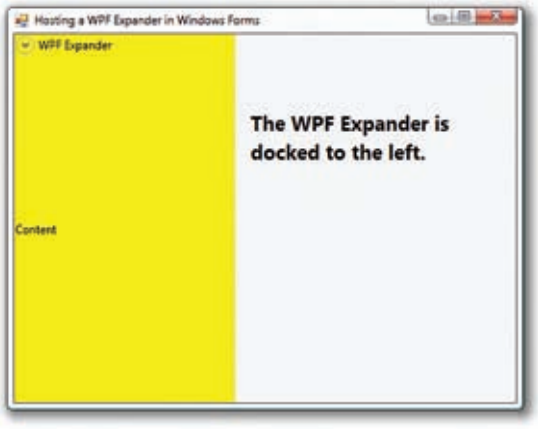


Рис. 19.9. *Приложение Windows Forms, содержащее элемент управления WPF Expander*

Если пойти чуть дальше, то можно использовать комбинацию `ElementHost` и `WindowsFormsHost`, чтобы получить элемент Windows Forms, встроенный в элемент управления WPF, который сам встроен в приложение Windows Forms! Все, что для этого нужно, – записать в свойство `Content` элемента WPF `Expander` ссылку на объект `WindowsFormsHost`, который может содержать произвольный элемент управления Windows Forms. В листинге 19.9 мы реализовали эту идею, поместив элемент управления Windows Forms `MonthCalendar` в элемент WPF `Expander`, все в одной форме Windows Forms. Результат показан на рис. 19.10.

Листинг 19.9. Form1.cs – использование интеграции Windows Forms и WPF в обоих направлениях

```
using System.Windows.Forms;
using System.Windows.Controls;
using System.Windows.Forms.Integration;

namespace WindowsFormsHostingWPF
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();

      // Создаем элемент WPF Expander
      Expander expander = new Expander();
      expander.Header = "WPF Expander";

      // Создаем элемент MonthCalendar и обертываем его в WindowsFormsHost
      WindowsFormsHost host = new WindowsFormsHost();
      host.Child = new MonthCalendar();
```

```
            // Помещаем WindowsFormsHost в Expander
            expander.Content = host;

            // Добавляем Expander в ElementHost
            elementHost.Child = expander;
        }
    }
}
```

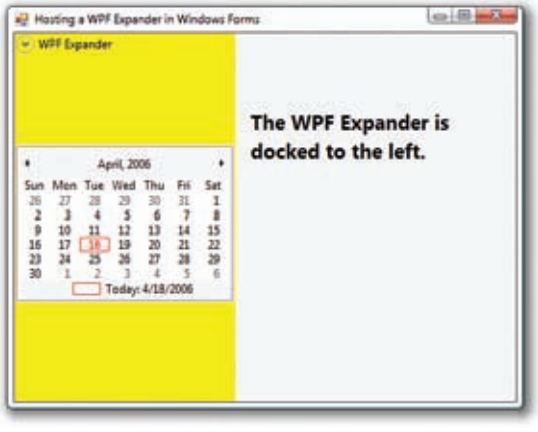


Рис. 19.10. *Элемент Windows Forms MonthCalendar находится внутри элемента WPF Expander, который сам помещен в форму Windows Forms*

КОПНЕМ ГЛУБЖЕ

Преобразование одного представления в другое

Одна из проблем, с которыми приходится сталкиваться при разработке гибридного приложения Windows Forms/WPF, – как быть с разными управляемыми типами данных, предназначенными для одного и того же. Например, в WPF есть типы `Color`, `Cursor`, `Size`, `Rect` и `Point`, отличающиеся от типов `Color`, `Cursor`, `Size`, `Rectangle` и `Point`, определенных в Windows Forms. Однако в большинстве случаев преобразовать один тип в другой достаточно просто. Например:

- В обоих типах `Color` имеется статический метод `FromArgb`, поэтому для создания одного объекта `Color` из другого достаточно передать этому методу значения свойств `A`, `R`, `G`, `B` исходного объекта `Color`.
- Чтобы получить кегль шрифта Windows Forms по кеглю шрифта WPF, надо умножить последний на 0,75. Чтобы получить кегль шрифта WPF по кеглю шрифта Windows Forms, надо разделить последний на 0,75.

В других случаях преобразование может оказаться сложнее. Так, для преобразования `System.Drawing.Bitmap` в `System.Windows.Media.Imaging.BitmapSource` необходимо работать на уровне представления, которое понимают обе технологии, – это тип Win32 `HBITMAP`.

Объект Windows Forms `Bitmap` основан на использовании `HBITMAP`, поэтому у него есть простой метод `GetHbitmap`, который возвращает описатель (в виде `IntPtr`). Правда, объект `BitmapSource` в WPF не имеет никакого отношения к `HBITMAP`, но, к счастью, имеется класс `System.Windows.Interop.Imaging`, в котором есть три статических вспомогательных метода для создания объектов `BitmapSource` из различных источников – области памяти, `HICON` и `HBITMAP`. Последнему методу `CreateBitmapSourceFromHBitmap` можно передать описатель и размеры, полученные от объекта Windows Forms `Bitmap`, и он вернет в ответ искомый объект WPF.

FAQ

Как открыть модальное диалоговое окно WPF из приложения Windows Forms?

Это делается почти так же, как в случае открытия модального диалогового окна WPF из Win32-приложения. Можно создать объект класса, производного от `Window`, и вызвать его метод `ShowDialog`. Но, чтобы все было корректно, нужно еще установить свойство `Owner`. Значением `Owner` должен быть объект `Window`, тогда как в приложении Windows Forms владельцем, без сомнения, является форма, то есть объект типа `System.Windows.Forms.Form`.

И снова воспользуемся классом `WindowInteropHelper`, чтобы установить в качестве владельца окно с произвольным описателем `HWND`. Иными словами, в его свойство `Owner` можно будет записать значение, полученное от свойства `Handle` формы `Form`. Вот как это делается:

```
bool? LaunchWpfDialogFromWindowsForm(Window dialog, Form parent)
{
  WindowInteropHelper helper = new WindowInteropHelper(dialog);
  helper.Owner = parent.Handle;
  return dialog.ShowDialog();
}
```

Сочетание содержимого DirectX с содержимым WPF

Как и содержимое Windows Forms, содержимое DirectX можно поместить в WPF-приложение с помощью `HwndHost`, а WPF-контент – в приложение DirectX с помощью `HwndSource`. В первой версии WPF интероперабельность на основе `HWND` была единственным способом объединить WPF и DirectX. Но учитывая, что WPF построена на базе DirectX, должна быть возможность гораздо более развитой интеграции между обеими технологиями без привлечения ортогонального в общем-то механизма `HWND`.

Начиная с версии WPF 3.5 SP1 (и WPF 3.0 SP2) стало возможно комбинировать WPF и DirectX напрямую в обоих направлениях. Новый механизм – подкласс `ImageSource` под названием `D3DImage` – не делает интероперабельность существенно проще, зато устраняет ограничения на перекрытие изображе-

ний, присущее всем другим способам организации интероперабельности. Это означает, что содержимое обоих типов можно смешивать, накладывать одно на другое и преобразовывать так, будто мы имеем дело с двумя элементами WPF. Возможности `D3DImage` не реализованы поверх `HWND`; это совершенно другой и более мощный механизм.

Объект `D3DImage` – это контейнер, который может содержать произвольную поверхность DirectX. (Несмотря на название, поверхность может включать как двумерное, так и трехмерное содержимое.) Поскольку `D3DImage` наследует `ImageSource`, то его можно использовать в самых разных местах – в `Image`, `ImageBrush` или `ImageDrawing`.

Для демонстрации `D3DImage` мы применим иной, нежели в предыдущих примерах, подход. В этом разделе мы возьмем простое неуправляемое приложение, написанное на C++, из комплекта DirectX SDK. (Детали кода не столь существенны, но его полную версию можно найти в приложении к этой книге по адресу *http://informit.com/title/9780672331190*.) Заимствованный из DirectX SDK пример так и останется неуправляемым, но вместо EXE-файла мы преобразуем его в DLL. Затем написанное на C# WPF-приложение сможет обратиться к реализованной на DirectX функциональности, вызвав с помощью PInvoke три неуправляемых функции, которые составляют открытый API.

Результатом является гипотетическая форма для заказа тигров, фоном которой служит вращающийся трехмерный тигр, реализованный с помощью DirectX, а поверх него расположены стандартные элементы управления WPF. Это показано на рис. 19.11.

В листинге 19.10 приведен XAML-код этого окна WPF `Window`. В качестве фона в нем указан элемент `D3DImage`, оформленный в виде кисти `ImageBrush`. Поверх него размещено несколько элементов управления WPF с 70-процентной непрозрачностью, чтобы продемонстрировать, как безукоризненно они сочетаются с фоном, созданным с помощью DirectX.



Рис. 19.11. *Окно WPF Window, содержащее трехмерное изображение вращающегося тигра, полученное с помощью DirectX, а поверх него элементы управления WPF*

Листинг 19.10. MainWindow.xaml – элемент управления WPF с DirectX-содержимым в качестве фона

```
<Window x:Class="WpfDirectX.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:interop="clr-namespace:System.Windows.Interop;assembly=PresentationCore"
  Title="Mixing DirectX with WPF" Height="350" Width="400">
  <Window.Background>
    <ImageBrush>
    <ImageBrush.ImageSource>
      <interop:D3DImage x:Name="d3dImage"
        IsFrontBufferAvailableChanged="d3dImage_IsFrontBufferAvailableChanged"/>
    </ImageBrush.ImageSource>
    </ImageBrush>
  </Window.Background>
  <Grid Margin="20" Opacity=".7" TextBlock.Foreground="White">
    ...
  </Grid>
</Window>
```

Очень важно обработать событие `IsFrontBufferAvailableChanged` объекта `D3DImage`. На протяжении времени жизни приложения поверхность DirectX иногда становится недоступной. (Это может случиться в разных ситуациях, например когда пользователь нажимает комбинацию клавиш Ctrl+Alt+Delete для вызова окна входа в систему или при смене видеодрайвера.) Следовательно, это событие может стать причиной инициализации (или повторной инициализации) содержимого DirectX, а также его очистки – в зависимости от значения свойства `IsFrontBufferAvailable` объекта `D3DImage`.

Задача связывания пустого объекта `D3DImage` с фактическим содержимым DirectX решается в застраничном файле, полный текст которого приведен в листинге 19.11.

Листинг 19.11. MainWindow.xaml.cs – работа D3DImage с содержимым DirectX, которое поставляется неуправляемой DLL-библиотекой, написанной на C++

```
using System;
using System.Runtime.InteropServices;
using System.Windows;
using System.Windows.Interop;
using System.Windows.Media;

namespace WpfDirectX
{
  // Три сигнатуры PInvoke для взаимодействия с неуправляемым кодом
  // на C++, находящимся в файле DirectXSample.dll
  class Sample
  {
    [DllImport("DirectXSample.dll")]
    internal static extern IntPtr Initialize(IntPtr hwnd, int width, int height);
```

```
  [DllImport("DirectXSample.dll")]
  internal static extern void Render();

  [DllImport("DirectXSample.dll")]
  internal static extern void Cleanup();
}

public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  protected override void OnSourceInitialized(EventArgs e)
  {
    base.OnSourceInitialized(e);
    // Теперь, когда мы можем получить HWND окна Window, выполним
    // инициализацию, которая производится также в момент,
    // когда становится доступен основной буфер
    d3dImage_IsFrontBufferAvailableChanged(this,
      new DependencyPropertyChangedEventArgs());
  }

  private void d3dImage_IsFrontBufferAvailableChanged(object sender,
    DependencyPropertyChangedEventArgs e)
  {
    if (d3dImage.IsFrontBufferAvailable)
    {
      // (Повторная) инициализация:
      IntPtr surface = Sample.Initialize(new WindowInteropHelper(this).Handle,
        (int)this.Width, (int)this.Height);

      if (surface != IntPtr.Zero)
      {
        d3dImage.Lock();
        d3dImage.SetBackBuffer(D3DResourceType.IDirect3DSurface9, surface);
        d3dImage.Unlock();

        CompositionTarget.Rendering += CompositionTarget_Rendering;
      }
    }
    else
    {
      // Очистка:
      CompositionTarget.Rendering -= CompositionTarget_Rendering;
      Sample.Cleanup();
    }
  }

  // Визуализируем схему DirectX, когда сама WPF готова к визуализации
```

```
    private void CompositionTarget_Rendering(object sender, EventArgs e)
    {
      if (d3dImage.IsFrontBufferAvailable)
      {
        d3dImage.Lock();
        Sample.Render();
        // Делаем всю область недействительной:
        d3dImage.AddDirtyRect(new Int32Rect(0, 0,
          d3dImage.PixelWidth, d3dImage.PixelHeight));
        d3dImage.Unlock();
      }
    }
  }
}
```

Код начинается с определения трех простых сигнатур PInvoke неуправляемых функций, экспортируемых из *DirectXSample.dll*. Хотя исходный код *DirectXSample.dll* здесь не показан, он имеется в прилагаемых к книге примерах. (Чтобы самостоятельно собрать эту библиотеку, необходимо сначала скачать и установить DirectX SDK с сайта *http://microsoft.com*.) Что именно делает код DirectX, не так важно; последовательность `Initialize`, `Render` и `Cleanup` вполне универсальна. Для `Initialize` необходим описатель `HWND`, поскольку его требует используемый DirectX API – создание устройства Direct3D.

Поскольку функция `Initialize` нуждается в `HWND`, мы не можем вызвать ее из конструктора `MainWindow` (если только не передать ему `HWND` для другого окна). Поэтому метод `OnSourceInitialized` переопределен таким образом, чтобы выполнять инициализацию в нем. В этот момент `WindowInteropHelper` уже может вернуть корректный `HWND` окна WPF `Window`. А вместо того чтобы дублировать код из обработчика события `d3dImage_IsFrontBufferAvailableChanged`, мы просто вызываем этот обработчик из `OnSourceInitialized`.

СОВЕТ

Если требуется получить `HWND` окна WPF `Window` до того, как оно показано, то можно воспользоваться методом `EnsureHandle` класса `WindowInteropHelper`. Этот метод создает окно Win32 (и генерирует событие `SourceInitialized`), если оно еще не было создано ранее, и возвращает его описатель `HWND`. А после вызова этого метода созданное окно можно даже не показывать! Кстати, Visual Studio 2010 именно так и поступает, когда выполняет построение проекта из командной строки.

В обработчике события `d3dImage_IsFrontBufferAvailableChanged`, в той его ветви, где выполняется инициализация, мы вызываем функцию `Initialize`, передавая ей `HWND` окна, ширину и высоту, а в ответ получаем указатель на интерфейс `IDirect3DSurface9`, замаскированный под `IntPtr` (типичный трюк, позволяющий не создавать управляемое определение интерфейса). Затем этот `IntPtr` можно передать методу `D3DImage.SetBackBuffer` (пока `D3DImage` заблокирован), чтобы ассоциировать содержимое. Неуправляемый метод `Render` требуется вы-

зывать по одному разу на каждый кадр, для чего идеально подходит обработчик статического события `Rendering` из класса `CompositionTarget`. В той ветви `d3dImage_IsFrontBufferAvailableChanged`, которая занимается очисткой (когда флаг `IsFrontBufferAvailable` равен `false`), мы просто отсоединяем обработчик события `Rendering` и вызываем неуправляемый метод `Cleanup`, чтобы написанный на C++ код имел возможность освободить захваченные ресурсы.

ПРЕДУПРЕЖДЕНИЕ

Не забывайте, что WPF хранит ссылку на поверхность Direct3D!

В гибридных приложениях, где имеется как управляемый, так и неуправляемый код, к распределению памяти следует подходить внимательно. Когда работаешь в основном с управляемым кодом, легко забыть о подсчете ссылок, поэтому имейте в виду, что WPF хранит ссылку на поверхность, передаваемую методу `SetBackBuffer`, до тех пор, пока флаг `IsFrontBufferAvailable` не станет равным `false` или `SetBackBuffer` не будет вызван снова. Чтобы разорвать эту ссылку, необходимо вызвать метод `SetBackBuffer`, передав в качество второго параметра `IntPtr.Zero`.

ПРЕДУПРЕЖДЕНИЕ

Объект D3DImage необходимо блокировать перед внесением любых модификаций в фоновый буфер!

Блокировка необходима для того, чтобы WPF не показывала неполные кадры. (Если вы еще не закончили рисование в буфере к моменту, когда WPF готова его вывести, то изображение будет показано неправильно.) К операциям, требующим блокировки, относятся вызовы методов объекта `D3DImage` — `SetBackBuffer` и `AddDirtyRect`, а также все операции визуализации DirectX с использованием указателя на интерфейс `IDirect3DSurface9`. Чтобы поставить блокировку, следует вызвать либо метод `D3DImage.Lock`, не возвращающий управление, пока WPF не закончит читать фоновый буфер, либо метод `D3DImage.TryLock`, который будет ждать только до истечения заданного пользователем тайм-аута. В любом случае не забудьте вызвать метод `D3DImage.Unlock`, закончив модификацию фонового буфера!

ПРЕДУПРЕЖДЕНИЕ

Дайте WPF возможность вывести содержимое фонового буфера!

Если вы адаптируете существующий код DirectX под работу совместно с WPF (как в этом примере), то не забудьте убрать все обращения к функции `Present` для устройства `Direct3D`. Дело в том, что WPF предоставляет свой собственный фоновый буфер, основываясь на внутреннем содержимом и содержимом поверхности, переданной методу `SetBackBuffer`. Если вы будете самостоятельно выводить содержимое фонового буфера, то помешаете нормальной работе системы визуализации.

Наконец, обработчик события CompositionTarget_Rendering вызывает неуправляемый метод Render (когда объект D3DImage заблокирован), а также делает недействительной всю область D3DImage, вызывая метод AddDirtyRect, которому передаются размеры D3DImage. WPF объединяет все измененные участки D3DImage с собственной внутренней поверхностью, а затем визуализирует результат. В некоторых приложениях этот процесс можно оптимизировать, более точно указывая недействительные области. Кроме того, иногда не требуется, чтобы вызуализация DirectX и объявление недействительными областей D3DImage происходили в *каждом* кадре.

КОПНЕМ ГЛУБЖЕ

Проверка того, что использование DirectX совместимо с D3DImage

Есть ряд мелких деталей, о которых нужно знать, когда пишешь программу, где предполагается обращение к DirectX API напрямую (в данном случае к неуправляемому коду на C++ в библиотеке *DirectXSample.dll*). Если их не учесть, то программа не будет работать вовсе или будет работать неоптимально.

Самое главное – поддерживается только DirectX версии 9 и более поздних, поскольку в методе D3DImage.SetBackBuffer используется только одно значение из перечисления D3DResourceType – IDirect3DSurface9! (Можно использовать более поздние версии Direct3D и работать с промежуточным устройством IDirect3DDevice9Ex.)

При работе в ОС Windows XP необходимо использовать функцию Direct3DCreate9, а затем можно создать устройство IDirect3DDevice9. Для этой поверхности следует задавать параметры D3DPOOL_DEFAULT, D3DUSAGE_RENDERTARGET и D3DFMT_X8R8G8B8 (RGB) или D3DFMT_A8R8G8B8 (ARGB). Но в Windows Vista и последующих версиях применение функции Direct3DCreate9Ex (и устройства IDirect3DDevice9Ex) обеспечивает более высокую производительность при условии, что используется драйвер экрана Windows Display Driver Model (WDDM), а видеокарта поддерживает необходимые возможности.

В Windows XP можно увеличить производительность (за счет аппаратного ускорения), если поверхность Direct3D создана как допускающая блокировку, но такие поверхности обычно работают медленнее в Windows Vista и последующих версиях. Эти и другие подобные тонкости позволяют в полной мере оценить, насколько программирование на уровне WPF проще по сравнению с DirectX!

ПРЕДУПРЕЖДЕНИЕ

D3DImage не работает в режиме программной визуализации!

Если визуализация производится потоком визуализации WPF (из-за недостаточно мощного оборудования, на удаленном рабочем столе и в других подобных ситуациях), то содержимое, находящееся внутри D3DImage, просто не показывается. Однако D3DImage работает при печати или в случае использования RenderTargetBitmap. Хотя в этих механизмах также применяется программная визуализация, они работают в потоке ГИП и потому указанное ограничение на них не распространяется.

Встраивание элементов управления ActiveX в WPF-приложения

В мире существуют тысячи элементов управления ActiveX, и все они легко встраиваются в WPF-приложения. Однако команде разработчиков WPF не пришлось для этого особо трудиться. Еще в Windows Forms 1.0 были встроены механизмы, обеспечивающие интероперабельность с элементами управления ActiveX. Вместо того чтобы дублировать их в WPF, было решено воспользоваться тем, что уже есть в Windows Forms. WPF получает эту функциональность задаром просто потому, что хорошо умеет взаимодействовать с Windows Forms.

Использование Windows Forms в качестве промежуточного слоя между ActiveX и WPF – быть может, не совсем оптимальное решение, зато вести разработку легко и приятно. Чтобы продемонстрировать, как элемент управления ActiveX встраивается в WPF-приложение, мы рассмотрим элемент Microsoft Terminal Services, поставляемый в комплекте с Windows. Он содержит практически всю функциональность удаленного рабочего стола, но для управления им достаточно всего нескольких простых API.

Собираясь воспользоваться элементом управления ActiveX, мы прежде всего должны получить управляемые, совместимые с Windows Forms определения нужных типов. Сделать это можно двумя способами:

- Использовать для ActiveX DLL утилиту ActiveX Importer (*AXIMP.EXE*). Она включена в компонент Windows SDK, относящийся к .NET Framework.
- Открыв любой проект типа Windows Forms в Visual Studio, добавить компонент на панель элементов. Его можно отыскать на вкладке COM Components (Компоненты COM) в диалоговом окне, которое появляется при выборе команд меню Tools→Choose Toolbox Items (Сервис→Выбор элементов для окна элементов управления). Затем этот элемент можно перетащить с панели элементов в любую форму. Тогда Visual Studio сама вызовет утилиту ActiveX Importer.

В любом случае генерируются два DLL-файла. Ссылки на них необходимо добавить в проект типа WPF (наряду со ссылками на сборки *System.Windows.Forms.dll* и *WindowsFormsIntegration.dll*). Первый файл – это сборка интероперабельности, содержащая «исходные» управляемые определения для неуправляемых интерфейсов, классов, перечислений и структур, которые имеются в библиотеке типов, находящейся внутри ActiveX DLL. Второй файл содержит элемент управления Windows Forms, соответствующий классу ActiveX. Первый файл называется по имени исходной библиотеки типов, второй – так же, но с префиксом *Ax*.

Для элемента управления Microsoft Terminal Services исходная библиотека ActiveX DLL называется *mstscax.dll* и находится в каталоге *system32* в папке *Windows*. (В диалоговом окне Choose Toolbox Items он отображается под названием Microsoft Terminal Services Client Control.) Утилита ActiveX Importer генерирует на его основе файлы *MSTSCLib.dll* и *AxMSTSCLib.dll*.

После того как все четыре сборки (*MSTSCLib.dll*, *AxMSTSCLib.dll*, *System.Windows.Forms.dll* и *WindowsFormsIntegration.dll*) будут включены в проект, можно добавить файлы, показанные в листингах 19.12 и 19.13; это код на XAML и C#, необходимый для встраивания элемента управления. Получившееся приложение показано на рис. 19.12.

Листинг 19.12. Window1.xaml – XAML-код WPF-приложения, включающего элемент Terminal Services

```
<Window x:Class="HostingActiveX.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Hosting the Terminal Services ActiveX Control">
  <DockPanel Name="panel" Margin="10">
    <StackPanel Margin="0,0,0,10" DockPanel.Dock="Top" Orientation="Horizontal">
      <TextBox x:Name="serverBox" Width="180" Margin="0,0,10,0"/>
      <Button x:Name="connectButton" Click="connectButton_Click">Connect</Button>
    </StackPanel>
  </DockPanel>
</Window>
```

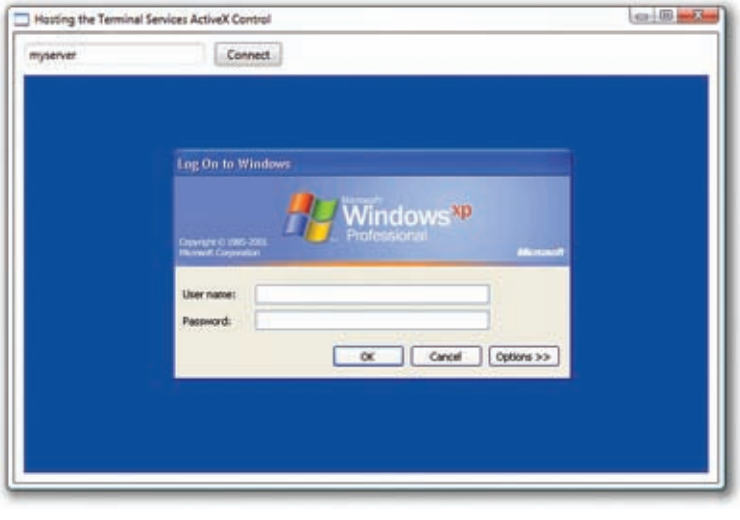


***Рис. 19.12.** Встраивание элемента ActiveX Terminal Services ActiveX в окно WPF-приложения*

Листинг 19.13. Window1.xaml.cs – код на C# для встраивания элемента ActiveX Terminal Services

```
using System;
using System.Windows;
using System.Windows.Forms.Integration;
```

```
namespace HostingActiveX
{
  public partial class Window1 : Window
  {
    AxMSTSCLib.AxMsTscAxNotSafeForScripting termServ;

    public Window1()
    {
      InitializeComponent();

      // Создаем владельца и элемент управления ActiveX
      WindowsFormsHost host = new WindowsFormsHost();
      termServ = new AxMSTSCLib.AxMsTscAxNotSafeForScripting();

      // Передаем элемент ActiveX владельцу, а владельца помещаем
      // на панель WPF
      host.Child = termServ;
      panel.Children.Add(host);
    }

    void connectButton_Click(object sender, RoutedEventArgs e)
    {
      termServ.Server = serverBox.Text;
      termServ.Connect();
    }
  }
}
```

Ничего особенного в XAML-коде в листинге 19.12 нет; в нем просто описана панель `DockPanel`, содержащая поле `TextBox` и кнопку `Button` для выбора сервера и подключения к нему. В листинге 19.13 на панель `DockPanel` помещается элемент `WindowsFormsHost`, а объект, представляющий элемент ActiveX в Windows Forms, делается потомком этого `WindowsFormsHost`. Этот элемент называется `AxMsTscAxNotSafeForScripting`. (До Windows Vista он назывался короче – `AxMsTscAx`.) Взаимодействие с элементом, носящим столь сложное имя, на самом деле устроено чрезвычайно просто. В его свойство `Server` следует записать строку, содержащую имя сервера, после чего к серверу можно подключиться, вызвав метод `Connect`.

Разумеется, создать элементы `WindowsFormsHost` *и* `AxMsTscAxNotSafeForScripting` можно непосредственно в XAML, тогда выделенный полужирным шрифтом код в листинге 19.13 можно будет убрать. Новый вариант XAML-кода приведен в листинге 19.14. Можно пойти еще дальше и, воспользовавшись привязкой к данным, убрать первую строку в методе `connectButton_Click`, но обработчик события все равно необходим для вызова метода `Connect`.

Листинг 19.14. Window1.xaml – модифицированный XAML-код WPF-приложения, включающего элемент Terminal Services

```
<Window x:Class="HostingActiveX.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:ax="clr-namespace:AxMSTSCLib;assembly=AxMSTSCLib"
    Title="Hosting the Terminal Services ActiveX Control">
    <DockPanel Name="panel" Margin="10">
      <StackPanel Margin="0,0,0,10" DockPanel.Dock="Top" Orientation="Horizontal">
        <TextBox x:Name="serverBox" Margin="0,0,10,0"/>
        <Button x:Name="connectButton" Click="connectButton_Click">Connect</Button>
      </StackPanel>
      <WindowsFormsHost>
        <ax:AxMsTscAxNotSafeForScripting x:Name="termServ"/>
      </WindowsFormsHost>
    </DockPanel>
  </Window>
```

СОВЕТ

Имеется возможность встроить элемент управления ActiveX в XBAP-приложение с частичным доверием или в автономную XAML-страницу, но сделать это с помощью механизма интероперабельности с Windows Forms не получится (потому что для этого нужен более высокий уровень доверия). Придется воспользоваться элементом `Frame` или `WebBrowser`, содержащим веб-страницу, в которую помещен элемент ActiveX. Например:

```
<Frame Source="pack://siteoforigin:,,,/webpage.html"/>
```

где файл *webpage.html* содержит следующий код:

```
<html>
  <body>
    <object Width="100%" Height="100%" ClassId="clsid:..."/>
  </body>
</html>
```

С точки зрения безопасности все будет выглядеть так, будто вы перешли на страницу *webpage.html* непосредственно в Internet Explorer. Возможно, появятся сообщения, относящиеся к безопасности, – в зависимости от настроек пользователя и текущей зоны. Но в некоторых случаях появления этих сообщений можно избежать, если элемент ActiveX подписан и безопасен.

FAQ

А можно ли представить элементы управления WPF как элементы управления ActiveX?

Никакой встроенной поддержки сверх интероперабельности на уровне `HWND` не предусмотрено, так что наилучшим решением будет создать обычный элемент управления (можете воспользоваться своей любимой технологией, например ActiveTemplate Library – ATL) и внедрить в него WPF-содержимое.

Резюме

Большинство разработчиков понимают, что с помощью WPF можно создавать действительно великолепные приложения. Но рассмотренные в этой главе механизмы интероперабельности с HWND, Windows Forms, DirectX и ActiveX вообще снимают все ограничения. Потому что теперь в вашем распоряжении плоды многолетних усилий, потраченные на разработку, тестирование и развертывание самых разных элементов управления и функций. Для организаций, затративших большие средства на создание существующего кода, это критически важно.

В этой главе обсуждалось пять различных ситуаций. Их названия не очень согласованны и довольно туманны, поэтому в табл. 19.1 приведена сводка, к которой можно обращаться, если забудете, что есть что.

Таблица 19.1. Пять основных классов интероперабельности

Класс	Применение
HwndHost	Встраивание HWND в WPF
WindowsFormsHost	Встраивание Windows Forms в WPF
D3DImage	Встраивание DirectX в WPF без HWND
HwndSource	Встраивание WPF в HWND
ElementHost	Встраивание WPF в Windows Forms

Однако преимущества интероперабельности не исчерпываются тем, что изложено в этой главе. Весь интерфейс приложения можно переделать на основе WPF, но при этом оставить неизменной стоящую за ним логику – даже если она реализована в неуправляемом коде. Для этого существуют самые разные способы, в том числе с использованием C++/CLI, PInvoke или интероперабельности с COM.

Несмотря на простоту и богатство возможностей, описанных в этой главе, у построения интерфейсов на основе одной лишь технологии WPF есть плюсы по сравнению с гибридными технологиями. Например, в чистом WPF-интерфейсе для всех элементов масштабирование и применение стилей происходят единообразно. Их можно накладывать друг на друга. Клавиатурная навигация и передача фокуса достигаются без особых усилий. Кроме того, не нужно думать о сложностях, связанных с одновременным использованием элементов, зависящих и не зависящих от разрешения устройства. Чистые WPF-интерфейсы открывают путь к работе в окружении с частичным доверием (в зависимости от того, как распределена основная логика работы приложения) – они даже могут быть адаптированы к Silverlight.

Даже сложные приложения, в разработку интерфейса которых были вложены значительные средства, могут получить выигрыш от использования WPF, если они хорошо структурированы. Например, мне как-то встретилась написанная с применением MFC программа, показывающая карты городов США.

В ней применялись различные примитивы MFC (естественно, основанные на GDI) для рисования линий и фигур. Воспользовавшись поверхностями WPF и переписав логику рисования с применением средств, обсуждавшихся в главе 15 «Двумерная графика», удалось получить WPF-версию ценой сравнительно небольших изменений в коде. Зато после перехода на WPF приложение смогло воспользоваться функциональностью, которую трудно было бы реализовать иначе: плавное изменение масштаба, поворот карты в трехмерной перспективе и т. д.

Таким образом, приложение, написанное до появления WPF, можно совершенствовать, улучшая внешний вид или наращивая функциональность, если, воспользовавшись интероперабельностью, постепенно вводить средства, предоставляемые WPF. Если же вы ранее разработали *элементы управления*, то есть и еще одно применение интероперабельности, которое необязательно влечет за собой изменение функциональности с точки зрения конечного пользователя: надстроить над элементом объектную модель WPF так, что клиенты смогут работать с вашим продуктом как с полноценным элементом управления WPF, не задумываясь о проблемах интероперабельности. Создание нестандартных элементов управления (реализованных только средствами WPF или гибридных) – тема следующей главы.

- Создание пользовательского элемента управления
- Создание нестандартного элемента управления

20

Пользовательские и нестандартные элементы управления

В главе 9 «Однодетные элементы управления» утверждалось, что ни один современный презентационный каркас нельзя считать полным, если в нем нет набора стандартных элементов управления, из которых можно быстро собрать традиционный пользовательский интерфейс. Полагаю, не будет преувеличением добавить, что современный презентационный каркас нельзя назвать полным, если он не предоставляет средств для создания собственных повторно используемых элементов управления. Такое желание может возникнуть потому, что приложению нужно что-то особенное или что на продаже уникальных элементов управления другим разработчикам можно заработать деньги. В этой главе мы расскажем о двух разновидностях таких элементов: *пользовательских* (более простых) и *нестандартных* (сложных, зато и более гибких).

Роль пользовательских и нестандартных элементов управления в WPF совершенно иная, чем в других технологиях. Раньше нестандартные элементы зачастую создавались просто ради эффектного внешнего вида. Но в WPF есть много способов придать существующему элементу нестандартный вид, не создавая нового. В главе 14 «Стили, шаблоны, обложки и темы» было показано, как с помощью стилей и шаблонов полностью изменить внешний облик встроенных элементов. А иногда можно просто вложить составное содержимое в имеющийся элемент и таким образом получить желаемое. В других технологиях для получения кнопки, украшенной изображением, или дерева, содержащего комбинированные списки, надо было писать нестандартный элемент, но в WPF это не так! (Я вовсе не хочу сказать, что теперь стало меньше возможностей продавать повторно используемые компоненты. Просто вариантов реализации стало больше.)

Решение о том, создавать или нет новый элемент управления, должно основываться не на внешнем виде, а на том API, который требуется предоставить.

Если ни один из существующих элементов не обладает *программным* интерфейсом, который позволил бы естественно выразить вашу идею, то вперед – создавайте пользовательский или нестандартный элемент. Но самая большая ошибка – создавать новый элемент, когда имеющегося вполне хватило бы!

FAQ



Я пришел к выводу, что нужен собственный элемент управления. Но какой – пользовательский или нестандартный?

Пользовательский элемент имеет смысл создавать, если возможности его повторного использования будут ограничены, а поддержка применения разнообразных стилей и тем вас не интересует. Нестандартный элемент создается, когда нужен полноценный элемент управления (не уступающий встроенным в WPF). Пользовательский элемент обычно содержит логическое дерево, определяющее его внешний вид, а логика его работы тяготеет к прямому взаимодействию с дочерними элементами. С другой стороны, нестандартный элемент, как правило, получает свой внешний вид от визуального дерева, определенного в отдельном шаблоне, а логика работы не меняется, даже если клиент полностью изменит визуальное дерево (применяя методы, описанные в главе 14).

Однако это различие в значительной степени определяется поведением Visual Studio. Среда Visual Studio подталкивает вас в том или ином направлении в зависимости от того, какого типа проект вы выбрали для создания элемента управления. Если создается пользовательский элемент, то вы получаете XAML-файл и соответствующий ему застраничный файл, так что элемент можно конструировать, как `Window` или `Page`. А для нестандартного элемента вы получаете обычный файл с расширением *.cs* (или *.vb*) плюс стиль темы с простым шаблоном элемента управления, включенным в типовой словарь проекта (*themes\generic.xaml*).

Но, чтобы ответить на этот вопрос компетентно, давайте сформулируем точные различия между пользовательскими и нестандартными элементами управления. Нестандартный элемент наследует классу `Control` или любому его подклассу. С другой стороны, пользовательский элемент – это по определению класс, производный от `UserControl`, который наследует классу `ContentControl`, а тот уже является подклассом `Control`. Таким образом, технически пользовательский элемент – частный случай нестандартного, однако в этой главе мы будем под *нестандартным элементом* понимать любой производный от `Control` класс, не являющийся пользовательским элементом.

Если элемент, который вы намереваетесь создать, может воспользоваться функциональностью, уже реализованной в каком-то классе, не наследующем `ContentControl` (например, `RangeBase` или `Selector`) или, наоборот, наследующем `ContentControl` (например, `HeaderedContentControl` или `Button`), то было бы логично унаследовать этому классу. Если же вашему элементу не нужна дополнительная функциональность класса `ContentControl` или ему подобного сверх той, что имеется в `Control`, то имеет смысл наследовать непосредственно `Control`. В обоих случаях вы будете писать нестандартный, а не пользовательский элемент.

Но если не требуется ни того ни другого, то выбор между наследованием ContentControl (нестандартный элемент) и UserControl (пользовательский элемент) не играет существенной роли, если на время забыть о порядке разработки. Дело в том, что класс UserControl мало чем отличается от своего базового класса ContentControl; по умолчанию у него другой шаблон, подразумевается режим выравнивания Stretch в обоих направлениях (а не Left и Top), свойства IsTabStop и Focusable равны false, а источники всех событий, генерируемых внутренними элементами, подменяются на сам UserControl. Вот, собственно, и все. На этапе выполнения WPF не рассматривает класс UserControl как особый случай. Поэтому ваше решение должно основываться на том, хотите вы создать «безвидный» элемент (тогда он будет нестандартным) или «наделенный видом» (тогда это пользовательский элемент).

Создание пользовательского элемента управления

Лучший способ понять, как создается пользовательский элемент управления, – создать его самому. Поэтому в этом разделе мы создадим элемент FileInputBox, который будет сочетать поле ввода TextBox с кнопкой Browse (Обзор).

Идея в том, что пользователь сможет либо ввести имя файла напрямую в поле TextBox, либо нажать кнопку, которая откроет стандартное диалоговое окно OpenFileDialog. Если затем пользователь выберет файл в этом диалоговом окне, то его полное имя будет автоматически скопировано в TextBox. Иными словами, элемент работает точно так же, как тег <INPUT TYPE="FILE"/> в HTML.

Создание пользовательского интерфейса элемента управления

В листинге 20.1 приведен XAML-файл, содержащий описание пользовательского интерфейса нашего элемента управления, а на рис. 20.1 показан результат его визуализации.

Листинг 20.1. FileInputBox.xaml – пользовательский интерфейс элемента FileInputBox

```
<UserControl x:Class="Chapter20.FileInputBox"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <DockPanel>
    <Button x:Name="theButton" DockPanel.Dock="Right" Click="theButton_Click">
      Browse...</Button>
    <TextBox x:Name="theTextBox"
      MinWidth="{Binding ActualWidth, ElementName=theButton}" Margin="0,0,2,0"/>
  </DockPanel>
</UserControl>
```

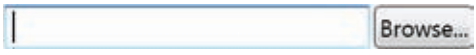


Рис. 20.1. *Пользовательский элемент управления FileInputBox сочетает поле ввода TextBox с кнопкой Button*

Кнопка `Button` пристыкована к правому краю, и для нее имеется обработчик события `Click` (который мы рассмотрим в следующем разделе). Поле ввода `TextBox` занимает все оставшееся место, кроме правого поля шириной две единицы, отделяющего его от кнопки. XAML-разметка очень проста, но легко справляется со всеми возникающими при компоновке ситуациями. Задавать минимальную ширину `MinWidth` элемента `TextBox` необязательно, но полезно, чтобы поле не оказалось слишком узким. А привязав минимальную ширину поля ввода к ширине кнопки (которая благодаря пристыковке к правому краю всегда достаточна для размещения содержимого), мы избежали «зашивания» значения в код.

На рис. 20.2 показано, что происходит, когда приложение следующим образом задает различные свойства элемента `FileInputBox`, унаследованные от классов `ContentControl` и `Control`:

```
<local:FileInputBox BorderBrush="Orange" BorderThickness="4" Background="Blue"
  HorizontalContentAlignment="Right"/>
```



Рис. 20.2. *Элемент FileInputBox автоматически учитывает визуальные свойства, унаследованные от своих базовых классов*

То, что эти свойства работают правильно, кажется само собой разумеющимся, но в действительности все происходит не настолько автоматически, как вы думаете. Внешний вид элемента `FileInputBox` определяется его шаблоном, унаследованным от `UserControl`. Так уж вышло, что в подразумеваемом по умолчанию шаблоне `UserControl` свойства, установленные на рис. 20.2, действительно учитываются:

```
<ControlTemplate TargetType="{x:Type UserControl}">
  <Border Background="{TemplateBinding Background}"
    BorderBrush="{TemplateBinding BorderBrush}"
    BorderThickness="{TemplateBinding BorderThickness}"
    Padding="{TemplateBinding Padding}">
    <ContentPresenter
      HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
      VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
  </Border>
</ControlTemplate>
```

Однако если бы класс FileInputBox наследовал непосредственно ContentControl (базовый класс UserControl), то эти свойства *не* учитывались бы, если только мы не снабдили бы FileInputBox специальным шаблоном. В теперешнем виде клиент FileInputBox может изменять не только его стиль, но и стили составляющих его элементов (TextBox, Button и DockPanel) – если создаст для них типизированные стили.

СОВЕТ

Если требуется, чтобы определенные в приложении типизированные стили не влияли на элементы, находящиеся внутри вашего элемента управления, то проще всего назначить им явный стиль Style (который может быть равен null, если нужно оставить внешний вид по умолчанию).

С точки зрения визуализации такое использование FileInputBox:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Chapter20">
  <StackPanel Margin="20">
    <local:FileInputBox/>
  </StackPanel>
</Window>
```

это не более чем короткий способ включить все логическое дерево элементов, описанное в файле *FileInputBox.xaml*, в свой пользовательский интерфейс:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Chapter20">
  <StackPanel Margin="20">
    <UserControl>
      <DockPanel>
        <Button DockPanel.Dock="Right">Browse...</Button>
        <TextBox MinWidth="{Binding ActualWidth, ElementName=theButton}"
          Margin="0,0,2,0"/>
      </DockPanel>
    </UserControl>
  </StackPanel>
</Window>
```

Это и само по себе полезно, хотя может быть достигнуто путем назначения произвольному существующему элементу управления явного шаблона, содержащего элементы DockPanel, Button и TextBox (если не обращать внимания на тонкие различия между визуальным и логическим деревьями). Однако пользовательские элементы управления, как правило, еще и добавляют некое поведение.

Наделение пользовательского элемента управления поведением

В листинге 20.2 приведен полный текст застраничного файла для разметки из листинга 20.1. Он наделяет элемент `FileInputBox` ожидаемым поведением при нажатии кнопки, позволяет получить текст, находящийся в поле `TextBox` в виде свойства, допускающего чтение и запись, и раскрывает событие `FileNameChanged`, соответствующее событию `TextChanged` элемента `TextBox`. Обработчик `TextChanged` помечает это событие как обработанное (чтобы остановить дальнейшее всплытие) и генерирует вместо него событие `FileNameChanged`.

Листинг 20.2. FileInputBox.xaml.cs – логика работы элемента FileInputBox

```
using System;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Win32;

namespace Chapter20
{
  public partial class FileInputBox : UserControl
  {
    public FileInputBox()
    {
      InitializeComponent();
      theTextBox.TextChanged += new TextChangedEventHandler(OnTextChanged);
    }

    private void theButton_Click(object sender, RoutedEventArgs e)
    {
      OpenFileDialog d = new OpenFileDialog();
      if (d.ShowDialog() == true) // Результат может быть равен true, false или null
        this.FileName = d.FileName;
    }

    public string FileName
    {
      get { return theTextBox.Text; }
      set { theTextBox.Text = value; }
    }

    void OnTextChanged(object sender, TextChangedEventArgs e)
    {
      e.Handled = true;
      if (FileNameChanged != null)
        FileNameChanged(this, EventArgs.Empty);
    }

    public event EventHandler<EventArgs> FileNameChanged;
  }
}
```

Вот и все! Если вы не собираетесь широко распространять свой пользовательский элемент или максимально тесно интегрировать его с различными подсистемами WPF, то зачастую достаточно предоставить обычные методы, события и свойства .NET, чтобы элемент был «в меру хорош». На рис. 20.3 этот элемент представлен в действии.

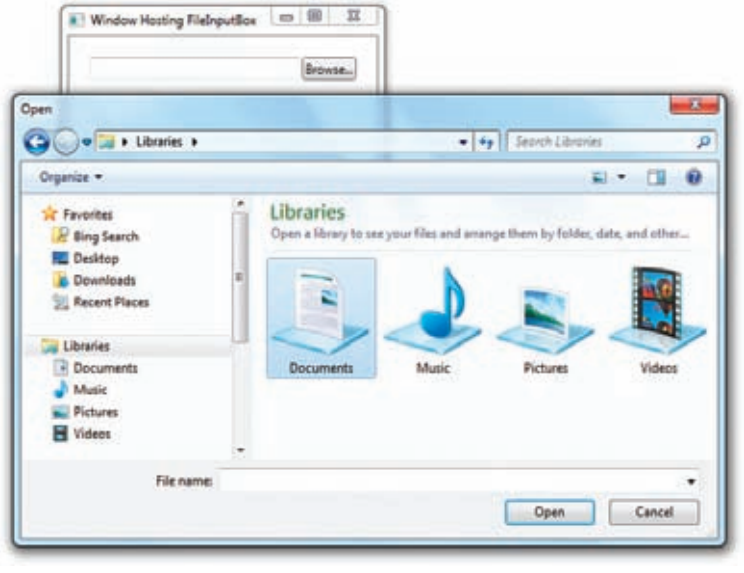


Рис. 20.3. *Элемент FileInputBox открывает стандартное диалоговое окно OpenFileDialog при нажатии кнопки*

Использовать такой элемент в программе тоже просто. Если нужно вставить его в элемент `Window` или `Page`, находящийся в той же сборке, то достаточно сослаться на соответствующее пространство имен, в данном случае `Chapter20`:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Chapter20">
  <StackPanel Margin="20">
    <local:FileInputBox/>
  </StackPanel>
</Window>
```

Если же вы хотите поместить элемент в отдельную сборку, то в директиву `clr-namespace` нужно включить информацию не только о пространстве имен, но и о сборке:

```
xmlns:local="clr-namespace:Chapter20;assembly=Chapter20Controls"
```

КОПНЕМ ГЛУБЖЕ

Защита пользовательских элементов управления от непреднамеренного использования

Ниже показан корректный способ инициализации элемента FileInputBox, когда в его внутреннее поле TextBox записывается значение свойства FileName, равное c:\Lindsay.htm:

```
<local:FileInputBox FileName="c:\Lindsay.htm"/>
```

Но поскольку FileInputBox наследует классу ContentControl, то пользователь может попытаться инициализировать его еще двумя способами:



```
<local:FileInputBox Content="c:\Lindsay.htm"/>
```

или



```
<local:FileInputBox>c:\Lindsay.htm</local:FileInputBox>
```

Как вы думаете, что при этом произойдет? Подразумеваемое по умолчанию значение свойства Content (панель DockPanel, содержащая кнопку Button и поле ввода TextBox) будет полностью заменено этой строкой! Разумеется, это совсем не то, чего хотел пользователь, иначе он попросту воспользовался бы элементом TextBlock!

К счастью, вы можете воспрепятствовать таким ошибкам. В случае FileInputBox можно сделать свойством содержимого не Content, а FileName:

```
[ContentProperty("FileName")]
public partial class FileInputBox : UserControl
{
  ...
}
```

Тогда строка

```
<local:FileInputBox>c:\Lindsay.htm</local:FileInputBox>
```

станет эквивалентна такой:

```
<local:FileInputBox FileName="c:\Lindsay.htm"/>
```

Но как избежать разрушительных последствий явной установки свойства Content? Один из способов – добавить в класс FileInputBox такой метод:

```
protected override void OnContentChanged(object oldContent, object newContent)
{
  if (oldContent != null)
    throw new InvalidOperationException("You can't change Content!");
}
```

Другое решение – поместить пользовательский интерфейс своего элемента управления в шаблон (а не в свойство Content) и привязать TextBox.Text к свойству Content. Но тогда уж лучше с самого начала писать нестандартный, а не пользовательский элемент!

Включение в пользовательский элемент управления свойств зависимости

Элемент FileInputBox можно улучшить, превратив FileName из обычного свойства .NET в свойство зависимости. Тогда клиенты этого элемента смогут использовать его в качестве приемника привязки к данным, упростится его включение в нестандартный шаблон элемента управления и т. д.

Чтобы превратить FileName в свойство зависимости, необходимо добавить в класс поле типа DependencyProperty, инициализировать его и изменить реализацию FileName, воспользовавшись механизмом свойств зависимости:

```
public static readonly DependencyProperty FileNameProperty =
  DependencyProperty.Register("FileName", typeof(string), typeof(FileInputBox));

public string FileName
{
  get { return (string)GetValue(FileNameProperty); }
  set { SetValue(FileNameProperty, value); }
}
```

По соглашению имена полей во встроенных объектах WPF имеют вид *PropertyName*Property. Рекомендуется придерживаться этого соглашения и в своих элементах во избежание ненужной путаницы.

Однако приведенная выше реализация FileName в виде свойства зависимости некорректна. Она больше не связана со свойством Text внутреннего элемента TextBox! Чтобы значение FileName изменялось при модификации свойства Text, можно добавить такую строку в метод OnTextChanged:

```
void OnTextChanged(object sender, TextChangedEventArgs e)
{
  this.FileName = theTextBox.Text;
  e.Handled = true;
  if (FileNameChanged != null)
    FileNameChanged(this, EventArgs.Empty);
}
```

А чтобы Text изменялось при модификации FileName, очень соблазнительно добавить в аксессор записи FileName такую строку:



```
set { theTextBox.Text = value; SetValue(FileNameProperty, value); }
```

Но это никуда не годная идея, потому что, как было объяснено в главе 3 «Основные принципы WPF», аксессор записи не вызывается, если, конечно, кто-то не захочет установить свойство .NET в процедурном коде. Когда свойство устанавливается в XAML-коде, в ходе привязки к данным и т. п., WPF вызывает метод SetValue напрямую.

Чтобы корректно реагировать на любое изменение свойства зависимости FileName, можно было бы зарегистрировать уведомление, рассылаемое системой свойств зависимости. Но самый простой способ синхронизировать Text и File-

Name – воспользоваться привязкой к данным. В листинге 20.3 приведена реализация класса FileInputBox, модифицированная так, чтобы FileName стало свойством зависимости. Предполагается, что в XAML-коде элемента FileInputBox теперь используется привязка к данным:

```
<UserControl x:Class="Chapter20.FileInputBox"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Name="root">
  <DockPanel>
    <Button x:Name="theButton" DockPanel.Dock="Right" Click="theButton_Click">
      Browse...</Button>
    <TextBox x:Name="theTextBox"
      MinWidth="{Binding ActualWidth, ElementName=theButton}"
      Text="{Binding FileName, ElementName=root}" Margin="0,0,2,0"/>
  </DockPanel>
</UserControl>
```

Листинг 20.3. FileInputBox.xaml.cs – модифицированный вариант листинга 20.2, в котором FileName – свойство зависимости

```
using System;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Win32;

namespace Chapter20
{
  public partial class FileInputBox : UserControl
  {
    public FileInputBox()
    {
      InitializeComponent();
      theTextBox.TextChanged += new TextChangedEventHandler(OnTextChanged);
    }

    private void theButton_Click(object sender, RoutedEventArgs e)
    {
      OpenFileDialog d = new OpenFileDialog();
      if (d.ShowDialog() == true) // Результат может быть равен true, false или null
        this.FileName = d.FileName;
    }

    public string FileName
    {
      get { return (string)GetValue(FileNameProperty); }
      set { SetValue(FileNameProperty, value); }
    }

    private void OnTextChanged(object sender, TextChangedEventArgs e)
    {
```

```
            e.Handled = true;
            if (FileNameChanged != null)
              FileNameChanged(this, EventArgs.Empty);
          }

          public static readonly DependencyProperty FileNameProperty =
          DependencyProperty.Register("FileName", typeof(string), typeof(FileInputBox));

          public event EventHandler<EventArgs> FileNameChanged;
        }
      }
```

После того как для `TextBox.Text` настроена привязка к данным (по умолчанию двусторонняя), стандартная реализация свойств зависимости обеспечивает правильную работу без написания дополнительного кода, несмотря на то, что значение `FileName` хранится отдельно от `TextBox`.

ПРЕДУПРЕЖДЕНИЕ

Не следует реализовывать в обертках свойств зависимости никакой логики, кроме вызова методов GetValue и SetValue!

Отклоняясь от этой стандартной реализации, вы вносите семантику, которая применяется только при установке свойства в процедурном коде. Чтобы реагировать на любые обращения к `SetValue`, необходимо зарегистрировать уведомление об изменении свойства зависимости и поместить логику в метод обратного вызова. Или придумать какой-то другой механизм реагирования на изменение значения, основанный на привязке к данным, как, например, в листинге 20.3.

СОВЕТ

Объект типа `FrameworkPropertyMetadata`, передаваемый методу `DependencyProperty.Register`, содержит несколько свойств для настройки поведения свойства зависимости. Помимо присоединения обработчика изменений, можно еще задать значение по умолчанию, указать, наследуется ли свойство дочерними элементами, настроить подразумеваемый по умолчанию поток данных в случае привязки к данным, указать, нужно ли при изменении значения перекомпоновывать или перерисовывать элемент, и т. д.

Включение в пользовательский элемент управления маршрутизируемых событий

Если уж вы пошли на то, чтобы снабдить пользовательский элемент управления свойствами зависимости, то, наверное, стоит потратить немного времени на преобразование некоторых событий в маршрутизируемые. Клиент может написать триггер, срабатывающий при возникновении маршрутизируемого

события, но для обычных событий .NET это невозможно. В случае элемента FileInputBox имеет смысл сделать событие FileNameChanged маршрутизируемым и всплывающим, тем более что обертываемое событие TextChanged само является таковым!

В главе 6 «События ввода: клавиатура, мышь, стилус и мультисенсорные устройства» было сказано, что маршрутизируемое событие определяется почти так же, как свойство зависимости. Требуется создать поле типа RoutedEvent (по соглашению имена таких полей заканчиваются суффиксом Event), зарегистрировать его и при желании предоставить обычное событие .NET, обертывающее вызовы методов AddHandler и RemoveHandler. В листинге 20.4 показано, как выглядит превращение события FileNameChanged в маршрутизируемое и всплывающее. Помимо реализации поддержки маршрутизируемости, закрытый метод OnTextChanged также изменен – теперь он генерирует маршрутизируемое событие методом RaiseEvent, унаследованным от класса UIElement.

Листинг 20.4. FileInputBox.xaml.cs – модифицированный вариант листинга 20.3, в котором FileNameChanged – маршрутизируемое событие

```
using System;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Win32;

namespace Chapter20
{
  public partial class FileInputBox : UserControl
  {
    public FileInputBox()
    {
      InitializeComponent();
      theTextBox.TextChanged += new TextChangedEventHandler(OnTextChanged);
    }

    private void theButton_Click(object sender, RoutedEventArgs e)
    {
      OpenFileDialog d = new OpenFileDialog();
      if (d.ShowDialog() == true) // Результат может быть равен true, false или null
        this.FileName = d.FileName;
    }

    public string FileName
    {
      get { return (string)GetValue(FileNameProperty); }
      set { SetValue(FileNameProperty, value); }
    }

    private void OnTextChanged(object sender, TextChangedEventArgs e)
    {
      e.Handled = true;
      RoutedEventArgs args = new RoutedEventArgs(FileNameChangedEvent);
```

```
        RaiseEvent(args);
      }

      public event RoutedEventHandler FileNameChanged
      {
        add { AddHandler(FileNameChangedEvent, value); }
        remove { RemoveHandler(FileNameChangedEvent, value); }
      }

      public static readonly DependencyProperty FileNameProperty =
      DependencyProperty.Register("FileName", typeof(string), typeof(FileInputBox));

      public static readonly RoutedEvent FileNameChangedEvent =
        EventManager.RegisterRoutedEvent("FileNameChanged",
        RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(FileInputBox));
    }
  }
```

Создание нестандартного элемента управления

В предыдущем разделе мы проиллюстрировали создание пользовательского элемента управления на примере элемента `FileInputBox`, а теперь напишем нестандартный элемент `PlayingCard` (игральная карта). Если при разработке пользовательского элемента мы обычно начинаем с интерфейса, а потом добавляем поведение, то при проектировании нестандартного элемента имеет смысл поставить во главу угла поведение – поскольку его пользовательский интерфейс можно полностью подменить.

Программирование поведения нестандартного элемента

Для элемента `PlayingCard` определено понятие *лица*, которое может принимать 52 значения. По карте можно щелкнуть. Для карты должно быть определено понятие *«выбрана»*, причем щелчок меняет состояние выбрана–не выбрана.

Прежде чем приступать к реализации, полезно поискать сходство между новым элементом управления и встроенными в WPF. Если существует похожий элемент, то можно будет в качестве базового класса взять что-то более специфическое, чем `Control`, и получить в свое распоряжение готовую функциональность.

В случае `PlayingCard` понятие лица – разновидность свойства `Foreground`, имеющегося у всех элементов управления. Но `Foreground` – это кисть `Brush`, а я хотел бы задавать в качестве значения лица простые строки типа `"H2"` (двойка червей) или `"SQ"` (дама пик). Можно было бы «похитить» какое-то имеющееся свойство типа `string` (например, `TextBlock.Text`), как описано в главе 14, но такой трюк вызовет недоумение у клиентов нашего элемента. Поэтому лучше бы реализовать отдельное свойство `Face`.

Понятие «быть нажатой» – это то, что применимо к кнопке, поэтому вроде бы естественно взять в качество базового класса `Button`. Но как быть с понятием *«выбрана»*? Так ведь класс `ToggleButton` уже поддерживает его в виде свойства `IsChecked`, равно как и идею «нажимаемости»! Поэтому `ToggleButton` представляется идеальным кандидатом на роль базового класса.

Первая попытка

В листинге 20.5 приведена реализация элемента управления `PlayingCard`, производного от `ToggleButton`.

Листинг 20.5. PlayingCard.cs – логика работы нестандартного элемента PlayingCard

```
using System.Windows.Media;
using System.Windows.Controls.Primitives;

namespace Chapter20
{
  public class PlayingCard : ToggleButton
  {
    public string Face
    {
      get { return face; }
      set { face = value; Foreground = (Brush)TryFindResource(face); }
    }

    private string face;
  }
}
```

Поскольку события `Click`, `Checked` и `Unchecked`, а также свойство `IsChecked` унаследованы от класса `ToggleButton`, то в классе `PlayingCard` осталось реализовать лишь свойство `Face`. В листинге 20.5 поданная на вход строка служит ключом ресурса, используемого в качестве свойства `Foreground` элемента. Так как мы вызываем метод `TryFindResource`, то попытка передать недопустимую строку приведет к сбросу `Foreground` в `null` – вполне разумное поведение. Но это означает также, что мы должны где-то хранить ресурсы с ключами `"HA"`, `"H2"`, `"H3"` и т. д. Не проблема – их можно было бы поместить в коллекцию `Resources` объекта `PlayingCard`, тогда метод `TryFindResource` сможет их найти.

В качестве визуальных образов игральных карт я нарисовал 52 изображения в Adobe Illustrator – по одному для каждой карты, а затем экспортировал их в XAML с помощью программы, которая находится по адресу *http://mikeswanson.com/xamlexport*. Каждый из 52 ресурсов – это кисть `DrawingBrush` с различными объектами `GeometryDrawing`. Именно эти ресурсы помещаются в коллекцию `Resources` объекта `PlayingCard`. Было бы смешно даже пытаться перевести такой объем XAML-кода на C#, поэтому применим другой подход – разобьем определение `PlayingCard` на две части: XAML и C#, а код в листинге 20.5 сделаем застраничным. В листингах 20.6 и 20.7 показано, что при этом получается.

Листинг 20.6. PlayingCard.xaml.cs – код из листинга 20.5 теперь стал застраничным файлом

```
using System.Windows.Media;
using System.Windows.Controls.Primitives;

namespace Chapter20
{
  public partial class PlayingCard : ToggleButton
  {
    public PlayingCard()
    {
      InitializeComponent();
    }

    public string Face
    {
      get { return face; }
      set { face = value; Foreground = (Brush)TryFindResource(face); }
    }

    private string face;
  }
}
```

Листинг 20.7. PlayingCard.xaml – ресурсы для нестандартного элемента управления PlayingCard

```
<ToggleButton x:Class="Chapter20.PlayingCard"
              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
              xmlns:local="clr-namespace:Chapter20">
<ToggleButton.Resources>
  <DrawingBrush x:Key="HA" Stretch="Uniform">
  <DrawingBrush.Drawing>
    ...
  </DrawingBrush.Drawing>
  </DrawingBrush>
  <DrawingBrush x:Key="H2" Stretch="Uniform">
  <DrawingBrush.Drawing>
    ...
  </DrawingBrush.Drawing>
  </DrawingBrush>
  ...
  <Style TargetType="{x:Type local:PlayingCard}">
    ...
    <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type local:PlayingCard}">
        ...
        <Rectangle Fill="{TemplateBinding Foreground}"/>
        ...
```

```
        </ControlTemplate>
      </Setter.Value>
      </Setter>
    </Style>
  </ToggleButton.Resources>
  </ToggleButton>
```

Изменения в коде на C# – это тот минимум, который необходим для поддержки компиляции кода класса, находящегося в двух разных файлах. В листинге 20.7 в коллекцию `Resources` помещаются все 52 объекта `DrawingBrush` и там же находится типизированный стиль с шаблоном, исправляющим внешний вид (чтобы карта не выглядела, как кнопка). В стиле находятся триггеры, запускающие анимацию по событиям `Checked`, `Unchecked`, `MouseEnter` и `MouseLeave` (их код опущен). Можно было бы вместо этого воспользоваться менеджером визуальных состояний, потому что в классе `ToggleButton` состояния `Checked` и `Unchecked` прописаны в группе `CheckStates` и к тому же учитываются состояния `Normal` и `MouseOver` из группы `CommonStates` класса `ButtonBase`.

Смысл шаблона в том, что прямоугольник `Rectangle` закрашивается кистью `Foreground`, которая в момент присваивания значения свойству `Face` ассоциируется с одним из ресурсов `DrawingBrush`. Полный текст листинга 20.7 занимает *больше 100 страниц* (я не шучу!) из-за размера и количества объектов `DrawingBrush`. Поэтому в книгу он, конечно, не включен, но на сайте *http://informit.com/title/9780672331190* приведен в полном объеме.

На рис. 20.4 показаны примеры использования элемента `PlayingCard`, для чего визуализировано следующее окно `Window`, в котором каждому экземпляру присвоено уникальное значение свойства `Face`, а затем они с помощью поворотов расположены в виде веера.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Chapter20"
        Title="Window Hosting PlayingCards">
<Window.Background>
  ...
</Window.Background>
  <Viewbox>
    <Canvas Width="220" Height="400">
      <local:PlayingCard Face="C3" Width="100" Height="140"
                         Canvas.Left="0" Canvas.Top="100">
        <local:PlayingCard.RenderTransform>
          <RotateTransform CenterX="50" CenterY="140" Angle="300"/>
        </local:PlayingCard.RenderTransform>
      </local:PlayingCard>
      <local:PlayingCard Face="CQ" Width="100" Height="140"
                         Canvas.Left="10" Canvas.Top="100">
        <local:PlayingCard.RenderTransform>
          <RotateTransform CenterX="50" CenterY="140" Angle="310"/>
        </local:PlayingCard.RenderTransform>
      </local:PlayingCard>
```

```
        ...
      </Canvas>
    </Viewbox>
  </Window>
```





Карта выдвигается, когда над ней находится указатель мыши



При щелчке по карте выделенная карта «выпрыгивает», при повторном щелчке становится на место

Рис. 20.4. *Сдача карт, каждая из которых реагирует на наведение указателя мыши и выбор*

Такой подход к реализации класса `PlayingCard` работает, и на бумаге результат смотрится вполне прилично. Но если вы запустите это приложение, то увидите, что производительность оставляет желать лучшего. И памяти оно потребляет слишком много. Причем при добавлении в окно каждой новой карты ситуация ухудшается. Проблема в том, что все 52 ресурса `DrawingBrush` хранятся внутри элемента управления, поэтому у каждого экземпляра оказывается своя копия всей коллекции! (100 печатных страниц × 13 экземпляров = уйма памяти!)

Кроме того, поведение этого элемента таит неожиданности для клиентов. Например, если в показанном выше элементе `Window` попробовать установить

свойство Resources для конкретного объекта PlayingCard, то возникнет исключение с сообщением о том, что словарь ResourceDictionary нельзя повторно инициализировать.

Сам заголовок раздела «Первая попытка» наводит на мысль, что мы пошли в неверном направлении. Логика, представленная в листингах 20.5 и 20.6, не сфокусирована на поведении элемента управления PlayingCard, а диктует детали визуальной реализации, требуя наличия ресурсов с определенными ключами, которые можно было бы присвоить в качестве значения Foreground.

Чтобы решить проблему быстро, можно переместить содержимое словаря PlayingCard.Resources в словарь Application.Resources, принадлежащий клиенту. Так мы решим проблемы с быстродействием и памятью, но нарушим инкапсуляцию элемента. Если автор приложения забудет включить эти ресурсы, то увидит изображение, показанное на рис. 20.5.



Рис. 20.5. *Если нужные ресурсы отсутствуют, то карты ничем не отличаются от ToggleButton*

Итак, разрабатывая эту версию класса PlayingCard, мы продолжали мыслить в терминах модели пользовательского элемента управления, который «владеет» своим интерфейсом. Необходимо порвать с таким подходом и реорганизовать код.

Рекомендуемый подход

Возвращаясь к листингу 20.5, мы должны убрать доступ к ресурсам и установку свойства Foreground, оставив эту деталь на усмотрение стиля, применяемого к PlayingCard:

```
public string Face
{
  get { return face; }
  set { face = value; Foreground = (Brush)TryFindResource(face); }
}
```

Разумным местом для стиля PlayingCard будет типовой словарь сборки (*themes\generic.xaml*, см. главу 14). Таким образом, для применения стиля к элементу PlayingCard (чтобы изображение не выглядело, как на рис. 20.5) нужно включить в статический конструктор класса PlayingCard такую строку:

```
DefaultStyleKeyProperty.OverrideMetadata(typeof(PlayingCard),
  new FrameworkPropertyMetadata(typeof(PlayingCard)));
```

А чтобы упростить применение свойства Face совместно с различными подсистемами WPF, следует преобразовать его в свойство зависимости. В листинге 20.8 все эти изменения осуществлены, и мы получаем окончательную версию класса PlayingCard.

Листинг 20.8. PlayingCard.cs – окончательный вариант логики работы нестандартного элемента PlayingCard

```
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls.Primitives;

namespace Chapter20
{
  public class PlayingCard : ToggleButton
  {
    static PlayingCard()
    {
      // Переопределяем стиль
      DefaultStyleKeyProperty.OverrideMetadata(typeof(PlayingCard),
        new FrameworkPropertyMetadata(typeof(PlayingCard)));

      // Регистрируем свойство зависимости Face
      FaceProperty = DependencyProperty.Register("Face",
        typeof(string), typeof(PlayingCard));
    }

    public string Face
    {
      get { return (string)GetValue(FaceProperty); }
      set { SetValue(FaceProperty, value); }
    }

    public static DependencyProperty FaceProperty;
  }
}
```

Кажется как-то чересчур просто, однако же вся необходимая логика присутствует. В этом коде отражена самая сущность игральной карты: от объекта

ToggleButton ее отличает только наличие строкового свойства Face. Все остальное – чисто визуальные различия.

СОВЕТ

При создании проекта типа WPF Custom Control Library в Visual Studio и при выборе из меню Add (Добавить) команды New Item (Новый элемент) для добавления нестандартного элемента управления в существующий проект Visual Studio автоматически создает файл, в котором уже есть нужный вызов метода DefaultStyleKeyProperty.OverrideMetadata и заготовка стиля в типовом словаре (если словаря еще нет, он создается). Однако XAML-файл с частичным определением класса *не* создается. Поэтому при использовании этих механизмов вы вряд ли попадете в ловушку, продемонстрированную в ходе первой попытки реализовать класс PlayingCard.

Создание пользовательского интерфейса нестандартного элемента управления

Чтобы придать нашей карте соответствующий ее назначению интерфейс, необходимо поместить в типовой словарь сборки подходящий стиль и ресурсы. (Кроме того, если вы собираетесь изменять внешний вид визуальных объектов в зависимости от темы, то надо заполнить еще и тематические словари.) Чтобы получить такой же результат, как на рис. 20.4, следует переместить все ресурсы, которые раньше находились *внутри* PlayingCard (см. листинг 20.7), в типовой словарь.

Также необходимо модифицировать следующую строку в шаблоне элемента управления в листинге 20.7:

```
<Rectangle Fill="{TemplateBinding Foreground}"/>
```

Закрашивать главный прямоугольник Rectangle в соответствии со значением Foreground теперь неуместно, потому что объект PlayingCard больше не устанавливает это значение, а заставлять клиентов настраивать эту кисть было бы невежливо.

Вместо этого мы записываем в свойство Fill тот ресурс DrawingBrush из типового словаря, ключ которого совпадает с текущим значением Face. Для этой цели следует использовать статический ресурс, потому что механизм динамических ресурсов не производит поиск ни в типовом, ни в тематическом словаре. Поскольку Face – свойство зависимости, то первое, что приходит на ум, – изменить значение Fill следующим образом:



```
<Rectangle>
<Rectangle.Fill>
  <StaticResource ResourceKey="{TemplateBinding Face}"/>
</Rectangle.Fill>
</Rectangle>
```

К сожалению, во время выполнения возникает такое малопонятное сообщение:

```
Cannot convert the value in attribute 'ResourceKey' to object of type ''.
```

Если заменить TemplateBinding эквивалентной привязкой Binding:



```
<Rectangle>
<Rectangle.Fill>
  <StaticResource ResourceKey=
    "{Binding Face, RelativeSource={RelativeSource TemplatedParent}}"/>
</Rectangle.Fill>
</Rectangle>
```

то исключение все равно возникает, но сообщение хотя бы имеет смысл:

```
'Binding' cannot be set on the 'ResourceKey' property of type
'StaticResourceExtension'. A 'Binding' can only be set on a DependencyProperty
of a DependencyObject.
```

ResourceKey не является свойством зависимости (и не может быть таковым, потому что класс StaticResourceExtension даже не наследует DependencyObject), поэтому использовать его в качестве приемника привязки к данным невозможно.

Если мы определим в качестве ключа ресурса DrawingBrush объект типа ComponentResourceKey (в котором TypeInTargetAssembly содержит тип PlayingCard, а ResourceId – название карты), а не просто строку, то сможем восстановить код, который программно устанавливает Foreground с помощью метода TryFindResource, но при этом оставить привязку TemplateBinding к Foreground. (Использовать класс ComponentResourceKey важно потому, что иначе методы FindResource и TryFindResource не смогут найти ресурсы в типовом или тематическом словаре.) Есть, впрочем, и другой вариант, позволяющий оставить код, приведенный в листинге 20.8, не отказываясь от строковых ключей: определить 52 триггера свойств (по одному на каждое допустимое значение Face) и записать в Fill ресурс, заданный на этапе компиляции. Длинно, зато просто. В листинге 20.9 показано 13 из 52 триггеров.

Листинг 20.9. Generic.xaml – типовой словарь, содержащий стиль и шаблон элемента PlayingCard

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:Chapter20">
  ...
  <Style TargetType="{x:Type local:PlayingCard}">
    ...
    <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type local:PlayingCard}">
        ...
      <Rectangle Name="faceRect"/>
        ...
```

```
<ControlTemplate.Triggers>
  <Trigger Property="Face" Value="HA">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource HA}"/>
  </Trigger>
  <Trigger Property="Face" Value="H2">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H2}"/>
  </Trigger>
  <Trigger Property="Face" Value="H3">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H3}"/>
  </Trigger>
  <Trigger Property="Face" Value="H4">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H4}"/>
  </Trigger>
  <Trigger Property="Face" Value="H5">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H5}"/>
  </Trigger>
  <Trigger Property="Face" Value="H6">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H6}"/>
  </Trigger>
  <Trigger Property="Face" Value="H7">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H7}"/>
  </Trigger>
  <Trigger Property="Face" Value="H8">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H8}"/>
  </Trigger>
  <Trigger Property="Face" Value="H9">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H9}"/>
  </Trigger>
  <Trigger Property="Face" Value="H10">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource H10}"/>
  </Trigger>
  <Trigger Property="Face" Value="HJ">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource HJ}"/>
  </Trigger>
  <Trigger Property="Face" Value="HQ">
    <Setter TargetName="faceRect" Property="Fill"
            Value="{StaticResource HQ}"/>
  </Trigger>
  <Trigger Property="Face" Value="HK">
```

```
            <Setter TargetName="faceRect" Property="Fill"
                    Value="{StaticResource HK}"/>
          </Trigger>
          ...
        </ControlTemplate.Triggers>
      </Setter.Value>
      </Setter>
    </Style>
  </ResourceDictionary>
```

Конечно, раз уж мы вручную отображаем значения `Face` на ключи ресурсов, то могли бы принять соглашение о том, что `Face` принимает значения от 0 до 51; это было бы удобнее для алгоритмов работы с картами. А потом можно было бы добавить свойства `Suit` (масть) и `Rank` (достоинство), чтобы с информацией стало проще работать.

При таком подходе устраняются проблемы с производительностью, характерные для первой попытки, поскольку типовые ресурсы разделяются всеми экземплярами `PlayingCard`. (А если вы не хотите, чтобы какой-то ресурс находился в общем пользовании, то можете пометить его ключевым словом `x:Shared="False"`.) Но это еще не все – полное отделение пользовательского интерфейса от логики позволяет максимально гибко изменять стиль элемента `PlayingCard`. В отличие от первой версии, нам не требуется кисть для каждой карты, поэтому можно даже подставить шаблон, который будет представлять карту в виде простого текстового блока `TextBlock`. Если вы пожелаете сделать ресурсы такого элемента, как `PlayingCard`, настраиваемыми и всячески облегчить их переопределение клиентом, то можете объявить 52 статических свойства, которые возвращают ключ `ComponentResourceKey`, соответствующий каждому ресурсу.

КОПНЕМ ГЛУБЖЕ

Другие подходы к проектированию элемента PlayingCard

Вместо того чтобы встраивать возможность «быть выбранным» в сам класс `PlayingCard`, можно поместить объекты `PlayingCard` в список `ListBox` и воспользоваться реализованным в нем поведением выборки. Затем с помощью свойства `SelectionMode` можно разрешить одиночную или множественную выборку.

Но если помещать элементы в `ListBox`, то такой симпатичный веер, как на рис. 20.4 и 20.5, уже просто так не получится. Однако можно написать нестандартную «веерную» панель и встроить ее в элемент `ListBox` в качестве шаблона `ItemsPanel`. В следующей главе мы создадим такую панель, которую назовем `FanCanvas`.

Можно также реализовать `PlayingCard` в виде простого класса, а не нестандартного элемента управления, и с помощью шаблона данных связать с ним подходящие визуальные объекты. Можно даже использовать обычные строки, если имеется шаблон данных, который сможет преобразовать их в лица карт!

СОВЕТ

В разделе «Программирование поведения нестандартного элемента» мы говорили о выборе подходящего базового класса, у которого можно было бы заимствовать как можно больше уже готового поведения. Но и с точки зрения пользовательского интерфейса в WPF имеется много встроенных элементов, которые было бы разумно использовать в своем шаблоне элемента управления.

Для такого нетрадиционного интерфейса, как у элемента `PlayingCard`, разумно начать с нуля. Но если будете писать другие элементы, то обратите внимание на многочисленные малознакомые повторно используемые компоненты в пространстве имен `System.Windows.Controls.Primitives`, например `BulletDecorator`, `ResizeGrip`, `ScrollBar`, `Thumb`, `Track` и прочие.

Некоторые соображения о более сложных элементах управления

Элемент `PlayingCard` обладает минимальной интерактивностью, которую мы сумели обработать в шаблоне с помощью простых триггеров или визуальных состояний. Но для более интерактивных элементов нужна иная техника. Представьте, например, что требуется превратить пользовательский элемент `FileInputBox`, рассмотренный в начале главы, в нестандартный. Тогда его интерфейс (для удобства повторяем XAML-разметку ниже) пришлось бы перенести в шаблон:

```
<DockPanel>
  <Button x:Name="theButton" DockPanel.Dock="Right" Click="theButton_Click">
    Browse...</Button>
  <TextBox x:Name="theTextBox"
    MinWidth="{Binding ActualWidth, ElementName=theButton}"
    Text="{Binding FileName, ElementName=root}" Margin="0,0,2,0"/>
</DockPanel>
```

Но как присоединить щелчок по кнопке к обработчику события `theButton_Click` элемента `FileInputBox`? Установить обработчик события `Click` в шаблоне элемента управления не получится. (Можно, правда, переопределить обработчик `theButton_Click` в застраничном файле для типового словаря. Но это, по сути дела, означает переработку всей логики элемента, а кроме того, всякий, кто захотел бы подменить шаблон по умолчанию своим собственным, должен был бы проделать то же самое!)

Для реализации такого рода интерактивности есть два разумных подхода, и оба применяются во встроенных в WPF элементах в зависимости от ситуации:

- Части элементов управления
- Команды

В этом разделе мы на примере `PlayingCard` рассмотрим также определение и использование новых состояний элемента управления.

Части элемента управления

В главе 14 отмечалось, что *часть элемента управления* – это некий «контракт» между элементом управления и его шаблоном. Элемент управления может поискать в своем шаблоне элемент с известным именем, а затем сделать с ним все, что заблагорассудится.

Решив, какие элементы сделать частями элементов управления, вы должны будете выбрать для каждого имя. Общепринято называть их `PART_XXX`, где *XXX* – имя элемента управления. Затем необходимо документировать существование частей, пометив свой класс атрибутом `TemplatePartAttribute` (для каждой части должен быть указан отдельный атрибут). Для той версии элемента `FileInputBox`, которая ожидает, что в шаблоне будет кнопка Browse (Обзор), это может выглядеть следующим образом:

```
[TemplatePart(Name="PART_Browse", Type=typeof(Button))]
public class FileInputBox : Control
{
  ...
}
```

WPF безразлично наличие атрибута `TemplatePartAttribute`, это просто документация, которая может быть полезна инструментам конструирования.

Чтобы обработать специально помеченные части элементов управления, необходимо переопределить метод `OnApplyTemplate`, унаследованный от класса `FrameworkElement`. Он вызывается при каждом применении шаблона, так что вам предоставляется возможность адекватно отреагировать на динамическое изменение шаблона. Для получения экземпляров элементов внутри шаблона следует вызвать метод `GetTemplateChild`, также унаследованный от `FrameworkElement`. В показанной ниже реализации мы находим кнопку Browse с известным именем и присоединяем необходимый обработчик к ее событию `Click`:

```
public override void OnApplyTemplate()
{
  base.OnApplyTemplate();

  // Найти кнопку Button в текущем шаблоне
  Button browseButton = base.GetTemplateChild("PART_Browse") as Button;

  // Присоединить обработчик события
  if (browseButton != null)
    browseButton.Click += new RoutedEventHandler(theButton_Click);
}
```

Отметим, что эта реализация справляется и с шаблонами, в которых нет части `PART_Browse`, и, значит, переменная `browseButton` равна `null`. Именно так и рекомендуется поступать, чтобы элемент мог работать с любым шаблоном, варьируя предоставляемую функциональность. В конце концов, кто-то вполне может изменить стиль элемента `FileInputBox`, так что кнопки Browse в шаблоне не будет. Если вы не согласны с этой рекомендацией и хотите производить более строгую проверку, то всегда можно возбудить исключение в методе `OnApply-`

Template в случае, когда шаблон не содержит ожидаемой части. Но такой элемент управления вряд ли уживется с графическим конструктором, например с Expression Blend.

Команды

Есть и более гибкий способ присоединения логики к частям шаблона – команды. В случае элемента FileInputBox, представляющего идею обзора файловой системы, шаблон мог бы следующим образом ассоциировать команду с подэлементом:

```
<Button Command="{x:Static local:FileInputBox.BrowseCommand}">Browse...</Button>
```

Это не только позволяет обойтись без магических имен, но еще и элемент, вызывающий срабатывание команды, больше не обязан быть кнопкой Button!

Для реализации такой команды в классе FileInputBox должно быть определено статическое свойство .NET типа RoutedCommand или RoutedUICommand (причем за ним обязательно должно стоять статическое поле, которое может быть закрытым):

```
private static RoutedUICommand browseCommand = new
  RoutedUICommand("Browse...", "BrowseCommand", typeof(FileInputBox));

public static RoutedUICommand BrowseCommand
{
  get { return browseCommand; }
}
```

Элемент управления должен связать эту команду с требуемой логикой (в данном случае с событием theButton_Click) в своем статическом конструкторе:

```
static FileInputBox()
{
  // Описать жест, приводящий к срабатыванию команды
  CommandManager.RegisterClassInputBinding(typeof(FileInputBox),
    new MouseGesture(MouseAction.LeftClick));

  // Связать команду с логикой обработки:
  CommandManager.RegisterClassCommandBinding(typeof(FileInputBox),
    new CommandBinding(browseCommand, theButton_Click));
}
```

Состояния элементов управления

В главе 14 говорилось о том, что в WPF 4 появилась возможность определять *состояния элементов управления*, чтобы помочь таким инструментам конструирования, как Expression Blend. И пользовательские, и нестандартные элементы поддерживают состояния. Любой класс, производный от Control, уже поддерживает три состояния из группы ValidationStates: Valid, InvalidFocused и InvalidUnfocused. Элемент PlayingCard автоматически поддерживает группу CheckStates (в которую включены состояния Checked, Unchecked и Indeterminate) –

благодаря своему базовому классу ToggleButton – и группу CommonStates (с состояниями Normal, MouseOver, Pressed и Disabled) – посредством базового класса ButtonBase.

Такое богатство, уже имеющееся в базовых классах PlayingCard, делает излишним определение дополнительных состояний. Впрочем, может оказаться полезным понятие перевертывания карты рубашкой вверх. Графический дизайнер вполне мог бы придумать и подключить красивый рисунок рубашки, не заботясь о том, какие события или свойства приводят к перевертыванию карты.

На такой случай имеет смысл завести два состояния – Front (лицом вверх) и Back (рубашкой вверх) – и включить их в новую группу состояний FlipStates. (В каждой группе должно быть состояние, подразумеваемое по умолчанию.) Само существование состояний необходимо документировать, пометив класс PlayingCard двумя атрибутами TemplateVisualState:

```
[TemplateVisualState(Name="Front", GroupName="FlipStates")]
[TemplateVisualState(Name="Back", GroupName="FlipStates")]
public class PlayingCard : ToggleButton
{
  ...
}
```

Определившись с состояниями и документировав их, остается только реализовать переходы в определенные состояния в подходящие моменты времени, для чего служит метод GoToState из класса VisualStateManager. Обычно это делается в каком-нибудь вспомогательном методе, например:

```
internal void ChangeState(bool useTransitions)
{
  // Предполагаем, что IsShowingFace – свойство, которое определяет состояние:
  if (this.IsShowingFace)
    VisualStateManager.GoToState(this, "Front", useTransitions);
  else
    VisualStateManager.GoToState(this, "Back", useTransitions);
}
```

ПРЕДУПРЕЖДЕНИЕ

Элементы управления не должны добавлять состояния в группы, определенные в базовом классе!

Новые состояния следует добавлять только в новые группы. Поскольку все группы состояний работают независимо друг от друга, переходы между состояниями в новой группе не могут нарушить логику работы базового класса. Однако если добавлять новые состояния в существующую группу, то нет никакой гарантии, что реализованная в базовом классе логика перехода между состояниями будет по-прежнему работать правильно.

ПРЕДУПРЕЖДЕНИЕ

Имена состояний должны быть уникальны, даже если они находятся в разных группах!

Несмотря на то что состояния распределены по группам, у элемента управления не должно быть двух состояний с одинаковым именем. Это ограничение может показаться удивительным, пока вы не задумаетесь о том, как реализуются переходы состояний, и не осознаете, что метод `GoToState` класса `VisualStateManager` ничего не знает о группах состояний. Группы – это всего лишь средство документирования, позволяющее лучше понять поведение отдельных состояний элемента управления и возможные переходы.

Именно из-за этого ограничения имена состояний часто бывают весьма специфичны. Например, среди состояний элемента `CalendarDayButton` есть `Normal` (из группы `CommonStates`), `NormalDay` (из группы `BlackoutDayStates`), `RegularDay` (из группы `DayStates`), `Unfocused` (из группы `FocusStates`), `CalendarButtonUnfocused` (из группы `CalendarButtonFocusStates`) и прочие. Их нельзя назвать просто `Default` или `Normal`.

Элементы управления вызывают такой метод в следующих обстоятельствах:

- Из метода `OnApplyTemplate` (с параметром `useTransitions=false`)
- При первой загрузке элемента управления (с параметром `useTransitions=false`)
- Из обработчиков событий (например, его необходимо вызывать из обработчика события `PropertyChanged` при изменении свойства `IsShowingFace`)

Нет ничего страшного в том, чтобы вызвать метод `GoToState`, когда конечное состояние совпадает с текущим (в этом случае метод ничего не делает). Поэтому методы типа показанного выше `ChangeState`, как правило, устанавливают текущее состояние для *каждой* группы, не заботясь о том, какое именно свойство изменилось.

ПРЕДУПРЕЖДЕНИЕ

Сразу после загрузки элемент управления должен явно осуществить переходы в подразумеваемые по умолчанию состояния из каждой группы!

Элемент управления, который не переходит явно в состояние (или состояния) по умолчанию, создает тонкую ошибку, проявляющуюся в его клиентах. Перед начальным переходом в состояние из каждой группы элемент еще не находится *ни в одном* из этих состояний. Это означает, что первый переход в состояние, отличное от подразумеваемого по умолчанию, не вызовет перехода из состояния по умолчанию, который мог бы определить клиент.

Совершая начальный переход, следует передать `false` в качестве значения параметра `useTransitions` метода `GoToState`, чтобы переход был выполнен мгновенно.

В классе Control тоже есть похожий вспомогательный метод, который, по существу, реализован следующим образом:

```
internal virtual void ChangeVisualState(bool useTransitions)
{
  // Обработать состояния в группе ValidationStates:
  if (Validation.GetHasError(this))
  {
    if (this.IsKeyboardFocused)
      VisualStateManager.GoToState(this, "InvalidFocused", useTransitions);
    else
      VisualStateManager.GoToState(this, "InvalidUnfocused", useTransitions);
  }
  else
  {
    VisualStateManager.GoToState(this, "Valid", useTransitions);
  }
}
```

ChangeVisualState – виртуальный метод, а во всех остальных элементах управления WPF он переопределен. Так, в классе ButtonBase мы встречаем следующую реализацию:

```
internal override void ChangeVisualState(bool useTransitions)
{
  // Обработать состояния в группе ValidationStates:
  base.ChangeVisualState(useTransitions);

  // Независимо обработать состояния в группе CommonStates:
  if (!this.IsEnabled)
    VisualStateManager.GoToState(this, "Disabled", useTransitions);
  else if (this.IsPressed)
    VisualStateManager.GoToState(this, "Pressed", useTransitions);
  else if (this.IsMouseOver)
    VisualStateManager.GoToState(this, "MouseOver", useTransitions);
  else
    VisualStateManager.GoToState(this, "Normal", useTransitions);

  // Независимо обработать состояния в группе FocusStates:
  if (this.IsKeyboardFocused)
    VisualStateManager.GoToState(this, "Focused", useTransitions);
  else
    VisualStateManager.GoToState(this, "Unfocused", useTransitions);
}
```

А в классе ToggleButton эта реализация переопределена так:

```
internal override void ChangeVisualState(bool useTransitions)
{
  // Обработать состояния в группах ValidationStates, CommonStates и FocusStates:
  base.ChangeVisualState(useTransitions);
```

```
    // Независимо обработать состояния в группе CheckStates:
    if (this.IsChecked == true)
      VisualStateManager.GoToState(this, "Checked", useTransitions);
    else if (this.IsChecked == false)
      VisualStateManager.GoToState(this, "Unchecked", useTransitions);
    else // this.isChecked == null
    {
      // Пытаемся перейти в состояние Indeterminate.
      // Если оно не определено, переходим в состояние Unchecked
      if (!VisualStateManager.GoToState(this, "Indeterminate", useTransitions))
        VisualStateManager.GoToState(this, "Unchecked", useTransitions);
    }
  }
```

Метод `GoToState` возвращает `false`, если не может выполнить переход в указанное состояние. Это бывает, когда применен шаблон, в котором просто нет соответствующего определения `VisualState`. Элементы управления не должны непредсказуемо вести себя в этой ситуации, и обычно они просто игнорируют значение, возвращаемое методом `GoToState`. Однако `ToggleButton` пытается перейти в состояние `Unchecked`, если состояния `Indeterminate` не существует. (Отметим, что на значении свойства `IsChecked` это не отражается; элемент `ToggleButton` логически по-прежнему находится в неопределенном состоянии, хотя выглядит так же, как неотмеченный.)

Хотя класс `PlayingCard` не может переопределить метод `ChangeVisualState` класса `ToggleButton` (потому что он объявлен внутренним для сборки WPF), он тем не менее наследует все его поведение, будучи производным от `ToggleButton`. Код в методе `ChangeState` класса `PlayingCard` прекрасно работает в счастливом неведении о существовании метода `ChangeVisualState`, так что получившийся элемент поддерживает все ожидаемые состояния из всех пяти групп.

КОПНЕМ ГЛУБЖЕ

Поддержка автоматизации ГИП

Чтобы нестандартный элемент управления считался полноценным классом, он должен поддерживать автоматизацию ГИП. Для этого обычно создается класс-спутник, производный от `FrameworkElementAutomationPeer`, которому по соглашению присваивается имя *`ControlName`*`AutomationPeer` и который описывает элемент управления для системы автоматизации. Затем в своем элементе вы должны переопределить метод `OnCreateAutomationPeer` (унаследованный от `UIElement`) так, чтобы он возвращал экземпляр класса-спутника:

```
protected override AutomationPeer OnCreateAutomationPeer()
{
  return new FileInputBoxAutomationPeer(this);
}
```

Когда возникает событие, о котором нужно сообщить системе автоматизации, вы можете получить экземпляр класса-спутника и сгенерировать событие автоматизации, например:

```
FileInputBoxAutomationPeer peer =
  UIElementAutomationPeer.FromElement(myControl) as FileInputBoxAutomationPeer;
if (peer != null)
  peer.RaiseAutomationEvent(AutomationEvents.StructureChanged);
```

СОВЕТ

Хорошо проработанному элементу управления иногда желательно знать, работает ли он в *режиме конструирования* (например, отображается в конструкторе Visual Studio или Expression Blend). В статическом классе `System.ComponentModel.DesignerProperties` имеется присоединенное свойство `IsInDesignMode`, которое возвращает эту информацию. Инструменты конструирования в подходящий момент изменяют значение по умолчанию, поэтому нестандартный элемент может узнать свой режим работы, вызвав статический метод `GetIsInDesignMode` и передав ему ссылку на себя самого.

Резюме

Если вы читаете книгу по порядку, то уже знаете о WPF достаточно, чтобы процесс создания нестандартного элемента управления не казался вам чем-то непостижимым. Но для тех, кто только начинает изучать WPF, это занятие – даже при всех подсказках Visual Studio – не из легких. А если такому пользователю нет особого дела до применения стилей и поддержки тем, а требуется лишь написать простое приложение и элементы управления, подобные имеющимся в Windows Forms, то все эти дополнительные сложности ему совершенно ни к чему! Именно поэтому в WPF и различаются пользовательские и нестандартные элементы.

Конечно, этими двумя подходами возможности создания повторно используемых компонентов в WPF не исчерпываются. Например, можно создать нестандартный низкоуровневый элемент управления, наследующий непосредственно классу `FrameworkElement`. А также класс, производный от `Panel`, для реализации нестандартных схем компоновки. Именно эта тема и будет рассмотрена в следующей (и последней) главе книги.

- Взаимодействие между родителями и потомками
- Создание панели SimpleCanvas
- Создание панели SimpleStackPanel
- Создание панели OverlapPanel
- Создание панели FanCanvas

21

Компоновка с помощью нестандартных панелей

В главе 5 «Компоновка с помощью панелей» мы рассматривали разнообразные панели, включенные в состав WPF. Если ни одна из встроенных панелей вам не подходит, можете написать собственную. Разумеется, учитывая гибкость готовых панелей, свойства компоновки, задаваемые для дочерних элементов (см. главу 4 «Задание размера, положения и преобразований элементов»), и возможность вкладывать одни панели в другие для создания сколь угодно сложной компоновки, маловероятно, что вам когда-нибудь потребуется писать свою панель. Собственно, теоретически в нестандартной панели вообще *не должно* возникать необходимости; с помощью процедурного кода (более или менее сложного) можно добиться любой компоновки с помощью одной лишь панели `Canvas`. Вопрос лишь в том, на какую легкость и автоматизм вы рассчитываете при повторной компоновке определенного вида макетов.

Например, может возникнуть потребность написать вариант `WrapPanel`, который размещает потомков в стопку (с возможным оборачиванием) в направлении, отличном от двух стандартных. Или вариант `StackPanel`, который выстраивает стопку снизу вверх, хотя это можно довольно легко сделать и с помощью панели `DockPanel`, если для каждого элемента задавать значение `Bottom` свойства `Dock`. Убедительным стимулом для создания нестандартной панели может стать виртуализация пользовательского интерфейса, например `VirtualizingWrapPanel` наподобие существующей `VirtualizingStackPanel`. Можно также создать панель, реализующую автоматическое перетаскивание и бросание, как `ToolBarTray`.

Хотя написания нестандартных панелей можно избежать за счет комбинирования более простых панелей, все же специальная панель может пригодиться, если приходится многократно решать задачу размещения элементов каким-то уникальным способом. Инкапсуляция логики компоновки в панель может сделать разработку макета пользовательского интерфейса менее под-

верженной ошибкам и обеспечить лучшую согласованность. Кроме того, панели, разработанные для специфических ситуаций, иногда могут показывать лучшую производительность, чем сверхгибкие панели WPF, особенно если удается заменить глубоко вложенные стандартные панели одной, пусть даже с ограниченной функциональностью.

Чтобы понять, из каких шагов состоит разработка нестандартной панели, мы сначала создадим две панели, повторяющие уже имеющуюся в WPF функциональность. А уже потом приступим к созданию двух уникальных панелей. Хочу вас порадовать – никаких специальных механизмов для создания нестандартных панелей не существует; все делается так же, как при создании панелей встроенных. Однако нам придется ближе познакомиться с тем, как панели взаимодействуют со своими потомками; этот вопрос мы в главах 4 и 5 обошли молчанием.

Взаимодействие между родителями и потомками

В главах 4 и 5 было сказано, что родители и потомки совместно работают над определением окончательного положения и размера. Чтобы достичь разумного компромисса между потребностями родителя и его потомков, применяется двухпроходная компоновка. Первый проход называется *измерением*, второй – *размещением*.

Этап измерения

На этапе измерения родитель спрашивает потомков, сколько места они хотели бы получить с учетом того, что имеется в наличии. Панели (и их дочерние элементы в тех случаях, когда это имеет смысл) отвечают на этот вопрос, переопределяя метод `MeasureOverride`, унаследованный от класса `FrameworkElement`. Например:

```
protected override Size MeasureOverride(Size availableSize)
{
  ...
  // Выяснить у каждого дочернего элемента желательный размер
  // с учетом имеющегося места
  foreach (UIElement child in this.Children)
  {
    child.Measure(new Size(...));
    // Ответ дочернего элемента находится в child.DesiredSize
   ...
  }
  ...
  // Сообщаю своему родителю, сколько места я хотел бы получить,
  // учитывая переданный им параметр availableSize
  return new Size(...);
}
```

Все дочерние элементы панели представлены в коллекции `Children` (типа `UIElementCollection`), а для того чтобы запросить у потомка предпочтительный для него размер, достаточно вызвать его метод `Measure` (унаследованный от `UIElement`). Этот метод не возвращает никакого значения, но после вызова полученный от потомка ответ находится в свойстве `DesiredSize` этого потомка. Родительская панель решает, как подстроить свое поведение с учетом предпочтительных для дочерних элементов размеров.

ПРЕДУПРЕЖДЕНИЕ

В методе MeasureOverride панель обязана вызвать метод Measure каждого потомка!

Возможно, вы захотите реализовать панель, которая не находит никакого применения возвращенным потомками значениям `DesiredSize` просто потому, что размер потомков ей безразличен. И тем не менее панель *обязана* в любом случае опросить своих потомков (вызывая метод `Measure`), потому что некоторые элементы ведут себя неправильно, если их метод `Measure` не вызывался. Это как задать супруге вопрос: «Как прошел день?» – ответ вас не очень-то интересует, просто вы хотите избежать упреков. (По крайней мере, мне так говорили. Лично меня ответ всегда интересует!)

В показанном выше коде на C#, как и во всех реализациях метода `MeasureOverride`, используются две важных переменных типа `Size`, которые мы обсудим в следующих разделах.

Size, передаваемый методу Measure каждого потомка

Это значение должно описывать размер области, которую вы готовы выделить потомку. Это может быть вся область, которую вы получили от родителя (ее размеры находятся в переданном параметре `availableSize`), какая-то ее часть или некое абсолютное значение – все зависит от того, что вы собираетесь делать.

Можно также в качестве одного или обоих размеров в переменной `Size` задавать значение `Double.PositiveInfinity`. Это позволит узнать, сколько места хотел бы получить потомок в идеальной ситуации. Иными словами, следующая строка означает: «Сколько ты запросил бы, если бы располагал всем местом в мире?»

```
child.Measure(new Size(Double.PositiveInfinity, Double.PositiveInfinity));
```

Система компоновки автоматически учитывает такие свойства потомков, как `Margin` (см. главу 4), поэтому метод `MeasureOverride` потомка получает тот размер, который вы передали методу `Measure`, за вычетом полей. Это также означает, что в параметре `availableSize`, который родительская панель передала вашей собственной реализации метода `MeasureOverride`, ваши поля тоже вычтены.

Size, возвращаемый методом MeasureOverride

Возвращаемый вами объект `Size` описывает предпочтительный для вас размер (то есть вы отвечаете на вопрос родителя точно так же, как ваши потомки ответили вам). Можно, конечно, вернуть абсолютный размер, но это означало бы игнорирование пожеланий потомков. Более вероятно, что вы вернете значение, которое позволит адаптировать размер под содержимое, то есть получить столько места, сколько нужно для размещения всех потомков в идеальных для них условиях, но не больше.

ПРЕДУПРЕЖДЕНИЕ

Нельзя просто взять и вернуть availableSize, переданный методу MeasureOverride!

То ли из-за простоты реализации, то ли по жадности, но может возникнуть искушение просто вернуть то значение параметра `availableSize`, которое было передано методу `MeasureOverride`. По существу, это означает: «Дай мне все место, что у тебя есть».

Однако если в объекте `availableSize` можно задавать бесконечные значения размеров (`Double.PositiveInfinity`), то в `DesiredSize` это запрещено. Даже если вам предлагается неограниченное место, вы должны выбрать для себя конкретный размер. Если потомок вернет бесконечное значение какого-нибудь размера, то реализация метода `Measure` в классе `UIElement` возбудит исключение `InvalidOperationException` с таким пояснительным сообщением: `"Layout measurement override of element 'XXX' should not return PositiveInfinity as its DesiredSize, even if Infinity is passed in as available size"` (Переопределения измерений макета элемента 'XXX' не должны возвращать значения PositiveInfinity для параметра DesiredSize, даже если в качестве доступного размера передано значение Infinity).

Если дочерний элемент только один, то этап измерения сводится к возврату полученного от него `DesiredSize` в качестве своего собственного размера. Когда дочерних элементов несколько, то необходимо произвести те или иные вычисления с полученными от них значениями ширины и высоты в зависимости от того, как вы планируете эти элементы размещать.

Этап размещения

После того как этап измерения был рекурсивно выполнен для всего дерева элементов, наступает время их физического размещения. На этом этапе родитель *извещает* потомков о том, в какой точке они размещены и сколько им выделено места (значение может отличаться от того значения `Size`, которое раньше передавалось методу `MeasureOverride`). Для этой цели панели (и их дочерние элементы в тех случаях, когда это имеет смысл) переопределяют метод `ArrangeOverride`, унаследованный от класса `FrameworkElement`. Например:

```
protected override Size ArrangeOverride(Size finalSize)
{
  ...
  // Сообщить каждому дочернему элементу, сколько ему выделено места
  foreach (UIElement child in this.Children)
  {
    child.Arrange(new Rect(...));
    // Размеры дочернего элемента теперь находятся в child.ActualHeight
    // и child.ActualWidth
    ...
  }
  ...
  // Устанавливаю собственные размеры (ActualHeight и ActualWidth)
  return new Size(...);
}
```

Вы сообщаете каждому дочернему элементу его позицию и размер, передавая его методу `Arrange` (унаследованному от `UIElement`) параметры типа `Rect` и `Size`. Например, чтобы выделить потомку область выбранного им предпочтительного размера, достаточно просто передать значение свойства `DesiredSize` этого потомка. Этот размер гарантированно подходит, потому что все измерения завершились еще до начала размещения.

Но, в отличие от `Measure`, метод `Arrange` не принимает бесконечных размеров (и переданный вам размер `finalSize` тоже никогда не будет бесконечным). Дочерний элемент может занять не всю область, какую вы задали, а, скажем, ее часть. В таком случае родитель решает, что делать (и нужно ли что-то делать вообще). Фактический размер, выбранный дочерним элементом, можно получить из его свойств `ActualHeight` и `ActualWidth` после возврата из метода `Arrange`.

Размер, который вы возвращаете из метода `ArrangeOverride`, становится значением свойств `RenderSize` и `ActualHeight/ActualWidth` вашей панели. Этот размер не должен быть бесконечным, но, в отличие от метода `MeasureOverride`, разрешается возвращать то значение, которое было передано в параметре `finalSize`, если вы хотите занять все предоставленное место, поскольку ни один из указанных в нем размеров не может быть бесконечным.

На этапе размещения, как и на этапе измерения, свойства, подобные `Margin`, обрабатываются автоматически, поэтому из размеров, передаваемых потомкам (и из переданных вам в параметре `finalSize`), поля уже вычтены. Кроме того, на этапе размещения автоматически учитывается выравнивание. Если дочерний элемент забирает все предложенное ему место (например, передавая `DesiredSize` своему методу `Arrange`), то выравнивание никак не проявляется, потому что не остается никакого лишнего места, в котором можно было бы что-то выровнять. Если же вы предлагаете потомку больше места, чем ему нужно, то результат учета свойств `HorizontalAlignment` и `VerticalAlignment` виден.

ПРЕДУПРЕЖДЕНИЕ

Не делайте в методах MeasureOverride и ArrangeOverride ничего такого, что может изменить компоновку!

В методах `MeasureOverride` и `ArrangeOverride` можно делать разные экзотические вещи, например применять к дочерним элементам дополнительные преобразования (в режиме `LayoutTransform` или `RenderTransform`). Но следите за тем, чтобы код каким-нибудь образом не повлиял на компоновку, иначе программа может зациклиться!

Метод или свойство делает компоновку недействительной, если вызывает метод `UIElement.InvalidateMeasure` либо `UIElement.InvalidateArrange`. Но эти методы открыты, поэтому узнать, откуда они вызываются, не так-то просто. В тех свойствах зависимости в WPF, где упомянутые методы используются, этот факт документирован с помощью одного или нескольких флагов метаданных из перечисления `FrameworkPropertyMetadataOptions`: `AffectsMeasure`, `AffectsArrange`, `AffectsParentArrange`, `AffectsParentMeasure`.

Если вы полагаете, что не можете обойтись без кода, изменяющего компоновку, и спланировали, как не попасть в бесконечный цикл, то можете вынести эту логику в отдельный метод, а затем с помощью `Dispatcher.BeginInvoke` запланировать его выполнение после завершения текущего прохода компоновки. Но для этого не забудьте задать значение `DispatcherPriority`, большее чем `Loaded`.

Создание панели SimpleCanvas

Прежде чем браться за создание уникальных панелей, попробуем продублировать поведение уже существующих. Для начала реализуем упрощенный вариант класса `Canvas`, который назовем `SimpleCanvas`. Панель `SimpleCanvas` ведет себя так же, как `Canvas`, но из свойств `Left`, `Top`, `Right` и `Bottom`, присоединенных к ее дочерним элементам, учитывает только `Left` и `Top`. Так сделано для того, чтобы сократить объем повторяющегося кода, потому что поддержка `Right` и `Bottom` мало чем отличается от поддержки `Left` и `Top`. (В результате этап размещения в `SimpleCanvas` выполняется чуточку быстрее, чем в `Canvas`, но только для тех дочерних элементов, к которым не присоединены свойства `Left` и `Top`.)

Реализация `SimpleCanvas` (как и любой другой панели) состоит из четырех шагов:

1. Создать класс, производный от `Panel`.
2. Определить свойства, потенциально полезные для настройки компоновки, в том числе присоединенные свойства для потомков.
3. Переопределить метод `MeasureOverride` и измерить в нем все дочерние элементы.
4. Переопределить метод `ArrangeOverride` и разместить в нем все дочерние элементы.

В листинге 21.1 приведен полный код класса `SimpleCanvas`.

Листинг 21.1. SimpleCanvas.cs – реализация класса SimpleCanvas

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace CustomPanels
{
  public class SimpleCanvas : Panel
  {
    public static readonly DependencyProperty LeftProperty =
      DependencyProperty.RegisterAttached("Left", typeof(double),
      typeof(SimpleCanvas), new FrameworkPropertyMetadata(Double.NaN,
      FrameworkPropertyMetadataOptions.AffectsParentArrange));

    public static readonly DependencyProperty TopProperty =
      DependencyProperty.RegisterAttached("Top", typeof(double),
      typeof(SimpleCanvas), new FrameworkPropertyMetadata(Double.NaN,
      FrameworkPropertyMetadataOptions.AffectsParentArrange));

    [TypeConverter(typeof(LengthConverter)),AttachedPropertyBrowsableForChildren]
    public static double GetLeft(UIElement element)
    {
      if (element == null) { throw new ArgumentNullException("element"); }
      return (double)element.GetValue(LeftProperty);
    }

    [TypeConverter(typeof(LengthConverter)),AttachedPropertyBrowsableForChildren]
    public static void SetLeft(UIElement element, double length)
    {
      if (element == null) { throw new ArgumentNullException("element"); }
      element.SetValue(LeftProperty, length);
    }

    [TypeConverter(typeof(LengthConverter)),AttachedPropertyBrowsableForChildren]
    public static double GetTop(UIElement element)
    {
      if (element == null) { throw new ArgumentNullException("element"); }
      return (double)element.GetValue(TopProperty);
    }

    [TypeConverter(typeof(LengthConverter)),AttachedPropertyBrowsableForChildren]
    public static void SetTop(UIElement element, double length)
    {
      if (element == null) { throw new ArgumentNullException("element"); }
      element.SetValue(TopProperty, length);
    }

    protected override Size MeasureOverride(Size availableSize)
    {
```

```
        foreach (UIElement child in this.Children)
        {
          // Даем каждому дочернем элементу столько места, сколько он хочет
          if (child != null)
            child.Measure(new Size(Double.PositiveInfinity,
                                   Double.PositiveInfinity));
        }

        // Для самой панели SimpleCanvas никакого места не нужно
        return new Size(0, 0);
      }

      protected override Size ArrangeOverride(Size finalSize)
      {
        foreach (UIElement child in this.Children)
        {
          if (child != null)
          {
            double x = 0;
            double y = 0;

            // Если присоединенные свойства Left или Top заданы, учтем их,
            // иначе поместим дочерний элемент в точку (0,0)
            double left = GetLeft(child);
            double top = GetTop(child);

            if (!Double.IsNaN(left)) x = left;
            if (!Double.IsNaN(top)) y = top;

            // Поместим в выбранную точку (x,y) с размером DesiredSize
            child.Arrange(new Rect(new Point(x, y), child.DesiredSize));
          }
        }

        // Меня устроит любой выделенный мне размер
        return finalSize;
      }
    }
  }
```

Сначала мы определяем присоединенные свойства `Left` **и** `Top`**, каждое из которых состоит из поля типа** `DependencyProperty` **и двух статических методов** `Get/Set`**. Как и для панели** `Canvas`**, по умолчанию свойства** `Left` **и** `Top` **принимают значение** `Double.NaN`**, интерпретируемое как «не установлено». При регистрации конструктору класса** `FrameworkPropertyMetadata` **передается значение** `FrameworkPropertyMetadataOptions.AffectsParentArrange`**, чтобы WPF знала, что при изменении этих свойств для дочерних элементов родительская панель** `SimpleCanvas` **должна повторить этап размещения, позволяя тем самым поместить элемент в новую позицию.**

Статические методы Get/Set — это стандартная реализация присоединенных свойств. Отметим ассоциацию с конвертером типа LengthConverter, которая позволяет задавать в XAML строковые значения этих свойств в различных форматах, например "Auto" (соответствует Double.NaN) или число с явно указанной единицей измерения ("px", "in", "cm" либо "pt"). Атрибут AttachedPropertyBrowsableForChildren сообщает конструктору, что оба свойства следует включить в список тех свойств, которые можно задавать для дочерних элементов.

Реализация метода MeasureOverride очень проста, что и неудивительно, принимая во внимание требуемое поведение панели SimpleCanvas. Он просто говорит дочерним элементам, что они могут получить столько места, сколько пожелают, а затем сообщает родителю, что лично для себя ему ничего не нужно (потому что его потомки не отсекаются по границам, если только свойство ClipToBounds не равно true; это поведение унаследовано от класса FrameworkElement).

Вся интересная работа производится в методе ArrangeOverride. Каждый потомок помещается в точку (0,0) с размером DesiredSize, если для него не установлены присоединенные свойства Left или Top. Чтобы проверить это, ArrangeOverride просто вызывает методы GetLeft и GetTop и смотрит, вернули они Double.NaN или что-то другое.

Как видите, панели нет нужды заботиться о каких-либо свойствах дочерних элементов, относящихся к компоновке (Height, MinHeight, MaxHeight, Width, MinWidth, MaxWidth, Margin, Padding, Visibility, HorizontalAlignment, VerticalAlignment, LayoutTransform и прочих). Переход от одного дочернего элемента к другому с помощью клавиши табуляции также обрабатывается автоматически. Последовательность перехода определяется порядком добавления дочерних элементов на родительскую панель.

В проекте, включенном в исходный код, который прилагается к этой книге, панель SimpleCanvas используется следующим образом:

```
<Window x:Class="CustomPanels.SimpleCanvasWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:CustomPanels"
        Title="Four Buttons in a SimpleCanvas">
  <local:SimpleCanvas>
    <Button Content="1" Background="Red"/>
    <Button local:SimpleCanvas.Left="40" local:SimpleCanvas.Top="40"
            Content="2" Background="Orange"/>
    <Button local:SimpleCanvas.Left="80" local:SimpleCanvas.Top="80"
            Content="3" Background="Yellow"/>
    <Button local:SimpleCanvas.Left="120" local:SimpleCanvas.Top="120"
            Content="4" Background="Lime"/>
  </local:SimpleCanvas>
</Window>
```

В XAML-разметке элемента Window пространству имен .NET CustomPanels сопоставляется префикс local, поэтому в ссылках на саму панель и ее присоединенные свойства следует употреблять префикс local:. Поскольку откомпили-

рованный файл *SimpleCanvas.cs* находится в той же сборке, то задавать в директиве `clr-namespace` атрибут `Assembly` необязательно.

Отметим, что в классе `SimpleCanvas` можно было бы повторно использовать уже реализованные в классе `Canvas` присоединенные свойства `Left` и `Top`, отказавшись от собственной реализации и заменив две строки в методе `ArrangeOverride`:

```
double left = Canvas.GetLeft(child);
double top = Canvas.GetTop(child);
```

Тогда эту панель надо было бы использовать так:

```
<Window x:Class="CustomPanels.SimpleCanvasWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:CustomPanels"
        Title="Four Buttons in a SimpleCanvas">
  <local:SimpleCanvas>
    <Button Content="1" Background="Red"/>
    <Button Canvas.Left="40" Canvas.Top="40"
            Content="2" Background="Orange"/>
    <Button Canvas.Left="80" Canvas.Top="80"
            Content="3" Background="Yellow"/>
    <Button Canvas.Left="120" Canvas.Top="120"
            Content="4" Background="Lime"/>
  </local:SimpleCanvas>
</Window>
```

Однако такой подход, когда одна панель заимствует присоединенные свойства другой, стандартным не назовешь.

Создание панели SimpleStackPanel

Теперь повторим реализацию еще одной существующей панели, но на этот раз для измерения и размещения придется немного потрудиться. Мы создадим панель `SimpleStackPanel`, очень похожую на `StackPanel`. Существенное различие состоит только в том, что в нашей версии не будет некоторых оптимизаций производительности. В листинге 21.2 приведен полный код.

Листинг 21.2. SimpleStackPanel.cs – реализация SimpleStackPanel

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace CustomPanels
{
  public class SimpleStackPanel : Panel
  {
    // Направление роста стопки
    public static readonly DependencyProperty OrientationProperty =
```

```
  DependencyProperty.Register("Orientation", typeof(Orientation),
  typeof(SimpleStackPanel), new FrameworkPropertyMetadata(
  Orientation.Vertical, FrameworkPropertyMetadataOptions.AffectsMeasure));

public Orientation Orientation
{
  get { return (Orientation)GetValue(OrientationProperty); }
  set { SetValue(OrientationProperty, value); }
}

protected override Size MeasureOverride(Size availableSize)
{
  Size desiredSize = new Size();

  // Разрешаем дочерним элементам неограниченно расти в направлении стопки,
  // для чего подменяем переданное значение.
  if (Orientation == Orientation.Vertical)
    availableSize.Height = Double.PositiveInfinity;
  else
    availableSize.Width = Double.PositiveInfinity;

  foreach (UIElement child in this.Children)
  {
    if (child != null)
    {
      // Запрашиваем у первого дочернего элемента предпочтительный размер,
      // предоставляя ему неограниченное место в направлении роста стопки,
      // но в другом направлении лишь столько места, сколько выделили нам.
      child.Measure(availableSize);

      // Наш предпочтительный размер – это сумма размеров дочерних элементов
      // в направлении роста стопки и размер наибольшего элемента –
      // в другом направлении.
      if (Orientation == Orientation.Vertical)
      {
        desiredSize.Width = Math.Max(desiredSize.Width,
                                     child.DesiredSize.Width);
        desiredSize.Height += child.DesiredSize.Height;
      }
      else
      {
        desiredSize.Height = Math.Max(desiredSize.Height,
                                      child.DesiredSize.Height);
        desiredSize.Width += child.DesiredSize.Width;
      }
    }
  }
  return desiredSize;
}

protected override Size ArrangeOverride(Size finalSize)
{
```

```
        double offset = 0;

        foreach (UIElement child in this.Children)
        {
          if (child != null)
          {
            if (Orientation == Orientation.Vertical)
            {
              // Переменная offset определяет сдвиг дочернего элемента
              // относительно начала стопки. Выделяем ему столько ширины,
              // сколько нам предоставлено, и столько высоты, сколько он захочет.
              child.Arrange(new Rect(0, offset, finalSize.Width,
                                     child.DesiredSize.Height));

              // Обновляем offset для следующего дочернего элемента
              offset += child.DesiredSize.Height;
            }
            else
            {
              // Переменная offset определяет сдвиг дочернего элемента
              // относительно начала стопки. Выделяем ему столько высоты,
              // сколько нам предоставлено, и столько ширины, сколько он захочет.
              child.Arrange(new Rect(offset, 0, child.DesiredSize.Width,
                                     finalSize.Height));

              // Обновляем offset для следующего дочернего элемента
              offset += child.DesiredSize.Width;
            }
          }
        }

        // Забираем себе все предоставленное нам место
        return finalSize;
      }
    }
  }
```

Как и в листинге 21.1, начинаем с определения свойства зависимости — `Orientation`. По умолчанию оно равно `Vertical`, а флаги `FrameworkPropertyMetadataOptions` показывают, что при изменении значения необходимо повторно выполнить этап измерения (после которого, естественно, повторяется этап размещения).

В методе `MeasureOverride` в направлении, отличном от направления роста стопки, каждому дочернему элементу предоставляется столько места, сколько выделено родителю (возможно, бесконечно много); зато в направлении роста стопки потомку выделяется неограниченное место. Узнав предпочтительный размер потомка, `SimpleStackPanel` обновляет свой собственный предпочтительный размер. В направлении роста стопки этот размер равен сумме размеров всех потомков. А в другом направлении предпочтительный размер совпадает с размером самого широкого (или самого высокого) потомка.

В методе ArrangeOverride переменная offset («указатель стопки», если угодно) указывает, куда нужно поместить следующий дочерний элемент. Каждому потомку выделяется весь размер панели в направлении роста стопки и указанный им предпочтительный размер в направлении, отличном от направления роста стопки. И в конце SimpleStackPanel забирает все выделенное ей место, возвращая в качестве результата полученный от родителя размер finalSize. Тем самым мы обеспечили такое же поведение, как у встроенной панели StackPanel.

Создание панели OverlapPanel

OverlapPanel — настоящая нестандартная панель в том смысле, что она не имеет встроенного аналога. Ее код основан на работе, проделанной при создании SimpleStackPanel, но несколько дополнительных штрихов делают эту панель уникальной. Как и SimpleStackPanel, она последовательно упаковывает дочерние элементы в стопку исходя из значения свойства Orientation. Только вместо того чтобы позволить потомкам размещаться вне выделенной ей области, эта панель начинает накладывать их друг на друга, когда место заканчивается. Таким образом, дочерним элементам выделяется столько же места, сколько в SimpleStackPanel, но их позиции равномерно «сжимаются» так, чтобы была заполнена вся ширина или высота (в зависимости от Orientation) панели. Если панели OverlapPanel выделено больше места, чем требуется для упаковки дочерних элементов, то она раздвигает их, опять-таки ставя целью заполнить всю область стопки. На рис. 21.1 визуализировано следующее окно Window, включающее панель OverlapPanel:

```
<Window x:Class="CustomPanels.OverlapPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:CustomPanels"
        Title="Four Buttons in an OverlapPanel">
  <local:OverlapPanel>
    <Button FontSize="40" Content="1" Background="Red"/>
    <Button FontSize="40" Content="2" Background="Orange"/>
    <Button FontSize="40" Content="3" Background="Yellow"/>
    <Button FontSize="40" Content="4" Background="Lime"/>
  </local:OverlapPanel>
</Window>
```

При таком равномерном наложении и растягивании панель OverlapPanel чем-то напоминает сетку Grid с одним столбцом (или с одной строкой), в которой у каждого дочернего элемента есть своя ячейка пропорционального размера *. Основное различие состоит в том, что каждому потомку разрешено рисовать вне своей эффективной «ячейки», чего в настоящей сетке не происходит, если только не поместить каждый дочерний элемент на свою собственную панель Canvas. Однако если обернуть элемент панелью Canvas, то потеряется возможность растягивания. Глядя на рис. 21.1, нельзя сказать, то ли кнопки действительно перекрывают друг друга, то ли они обрезаны по краю, но если бы

элементы были непрямоугольными или полупрозрачными, как на рис. 21.2, то разница стала бы очевидна.



Рис. 21.1. *Панель OverlapPanel с четырьмя кнопками в окнах разного размера*



Рис. 21.2. *Если для кнопок на рис. 21.1 задать Opacity=0.5, то становится видно, что они перекрываются, а не обрезаны*

В листинге 21.3 приведен полный код класса `OverlapPanel` и полужирным шрифтом выделены места, где он отличается от класса `SimpleStackPanel`, показанного в листинге 21.2.

Листинг 21.3. OverlapPanel.cs – модифицированная панель SimpleStackPanel, которая накладывает дочерние элементы друг на друга или растягивает их

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace CustomPanels
{
  public class OverlapPanel : Panel
  {
    double _totalChildrenSize = 0;

    // Направление роста стопки
    public static readonly DependencyProperty OrientationProperty =
      DependencyProperty.Register("Orientation", typeof(Orientation),
      typeof(OverlapPanel), new FrameworkPropertyMetadata(Orientation.Vertical,
      FrameworkPropertyMetadataOptions.AffectsMeasure));

    public Orientation Orientation
    {
      get { return (Orientation)GetValue(OrientationProperty); }
      set { SetValue(OrientationProperty, value); }
    }

    protected override Size MeasureOverride(Size availableSize)
    {
      Size desiredSize = new Size();

      foreach (UIElement child in this.Children)
      {
        if (child != null)
        {
          // Смотрим, какой размер предпочтет потомок,
          // если ему выделить все имеющееся у нас место
          child.Measure(availableSize);

          // Наш предпочтительный размер – это сумма размеров дочерних элементов
          // в направлении роста стопки и размер наибольшего элемента
          // в другом направлении.
          if (Orientation == Orientation.Vertical)
          {
            desiredSize.Width = Math.Max(desiredSize.Width,
                                         child.DesiredSize.Width);
            desiredSize.Height += child.DesiredSize.Height;
          }
          else
          {
            desiredSize.Height = Math.Max(desiredSize.Height,
                                          child.DesiredSize.Height);
            desiredSize.Width += child.DesiredSize.Width;
```

```
      }
    }
  }

  _totalChildrenSize = (Orientation == Orientation.Vertical ?
                        desiredSize.Height : desiredSize.Width);

  return desiredSize;
}

protected override Size ArrangeOverride(Size finalSize)
{
  double offset = 0;
  double overlap = 0;

  // Вычисляем величину перекрытия путем деления недостающего
  // места поровну на всех потомков
  if (Orientation == Orientation.Vertical)
  {
    if (finalSize.Height > _totalChildrenSize)
      // Если нам выделили места больше, чем _totalChildrenSize,
      // то отрицательное перекрытие означает растяжение
      overlap = (_totalChildrenSize - finalSize.Height) / this.Children.Count;
    else
      // В этом случае this.DesiredSize дает фактический меньший размер
      overlap = (_totalChildrenSize - this.DesiredSize.Height) /
                this.Children.Count;
  }
  else
  {
    if (finalSize.Width > _totalChildrenSize)
      // Если нам выделили места больше, чем _totalChildrenSize,
      // то отрицательное перекрытие означает растяжение
      overlap = (_totalChildrenSize - finalSize.Width) / this.Children.Count;
    else
      // В этом случае this.DesiredSize дает фактический меньший размер
      overlap = (_totalChildrenSize - this.DesiredSize.Width) /
                this.Children.Count;
  }

  foreach (UIElement child in this.Children)
  {
    if (child != null)
    {
      if (Orientation == Orientation.Vertical)
      {
        // Переменная offset определяет сдвиг дочернего элемента
        // относительно начала стопки. Выделяем ему столько ширины,
        // сколько нам предоставлено, и столько высоты, сколько он захочет
        // или больше в случае, когда перекрытие отрицательно.
        child.Arrange(new Rect(0, offset, finalSize.Width,
```

```
                child.DesiredSize.Height + (overlap > 0 ? 0 : -overlap)));

              // Обновляем offset для следующего дочернего элемента
              offset += (child.DesiredSize.Height - overlap);
            }
            else
            {
              // Переменная offset определяет сдвиг дочернего элемента
              // относительно начала стопки. Выделяем ему столько высоты,
              // сколько нам предоставлено, и столько ширины, сколько он захочет
              // или больше в случае, когда перекрытие отрицательно.
              child.Arrange(new Rect(offset, 0,
                child.DesiredSize.Width + (overlap > 0 ? 0 : -overlap),
                finalSize.Height));

              // Обновляем offset для следующего дочернего элемента
              offset += (child.DesiredSize.Width - overlap);
            }
          }
        }

        // Забираем себе все предоставленное нам место
        return finalSize;
      }
    }
  }
```

Единственная разница между методами `MeasureOverride` в классах `OverlapPanel` и `SimpleStackPanel` заключается в том, что в первом случае потомку не выделяется неограниченное место в направлении роста стопки; он получает размер `availableSize` в обоих направлениях. Связано это с тем, что панель пытается уплотнить потомков в отведенном пространстве, даже если его недостаточно. Кроме того, запоминается суммарный размер потомков в направлении роста стопки (эта величина одновременно является собственным предпочтительным размером панели в данном направлении). Результат хранится в поле `_totalChildrenSize`, которое используется в методе `ArrangeOverride`.

В методе `ArrangeOverride` вычисляется разность между располагаемым и желаемым размерами, а на ее основе – величина перекрытия `overlap`, которую нужно будет вычитать из переменной `offset` при размещении каждого дочернего элемента. Если значение `overlap` положительно, то оно представляет количество логических пикселов в области перекрытия соседних потомков, а если отрицательно, то количество логических пикселов, добавляемых к размеру каждого потомка.

Обратите внимание на странного вида выражение, прибавляемое к размеру потомка в направлении роста стопки при каждом вызове `child.Arrange`:

```
(overlap > 0 ? 0 : -overlap)
```

Это означает, что мы увеличиваем размер потомка на абсолютную величину `overlap`, но только если `overlap` отрицательно. Это необходимо для того, чтобы

сами потомки растягивались, если между ними остается пустое пространство (см. рис. 21.1). Не будь этой поправки, растянутые кнопки выглядели бы, как показано на рис. 21.3.

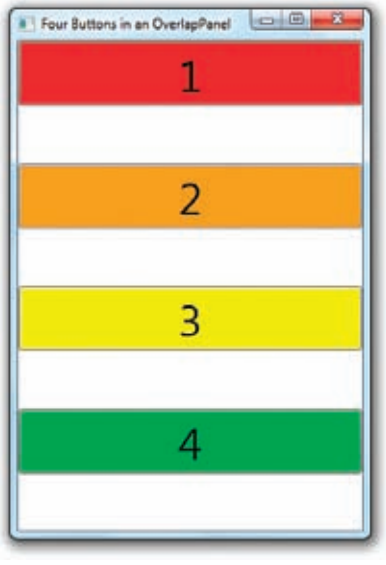


Рис. 21.3. *Как повела бы себя панель OverflowPanel, если бы не увеличивала размер дочерних элементов в направлении роста стопки*

Отметим, что кнопки на рис. 21.1 растягиваются только потому, что в классе `Button` свойство `VerticalAlignment` по умолчанию равно `Stretch`. Если бы для каждой кнопки было задано выравнивание `VerticalAlignment`, равное `Top`, то даже при корректной реализации `OverlapPanel` все равно получился бы результат, показанный на рис. 21.3. Но это нормально; задача панели – сообщить дочернему элементу, сколько места ему выделено, а уж элемент сам решает, как им распорядиться: растянуть себя на всю предоставленную область или выровняться по каким-то краям.

Создание панели FanCanvas

Наша последняя панель `FanCanvas` необычна и служит особой цели – расположить свои дочерние элементы в виде веера. Эффектным приложением такой панели была бы раскладка игральных карт – как в предыдущей главе. Да и другие применения можно придумать. Впервые панель `FanCanvas` была упомянута в главе 10 «Многодетные элементы управления» в качестве значения свойства `ItemsPanel` списка `ListBox`, в котором отображаются фотографии. В листинге 21.4 приведен полный код класса `FanCanvas`.

Листинг 21.4. FanCanvas.cs – реализация класса FanCanvas

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace CustomPanels
{
  public class FanCanvas : Panel
  {
    public static readonly DependencyProperty OrientationProperty =
      DependencyProperty.Register("Orientation", typeof(Orientation),
      typeof(FanCanvas), new FrameworkPropertyMetadata(Orientation.Horizontal,
      FrameworkPropertyMetadataOptions.AffectsArrange));

    public static readonly DependencyProperty SpacingProperty =
      DependencyProperty.Register("Spacing", typeof(double),
      typeof(FanCanvas), new FrameworkPropertyMetadata(10d,
      FrameworkPropertyMetadataOptions.AffectsArrange));

    public static readonly DependencyProperty AngleIncrementProperty =
      DependencyProperty.Register("AngleIncrement", typeof(double),
      typeof(FanCanvas), new FrameworkPropertyMetadata(10d,
      FrameworkPropertyMetadataOptions.AffectsArrange));

    public Orientation Orientation
    {
      get { return (Orientation)GetValue(OrientationProperty); }
      set { SetValue(OrientationProperty, value); }
    }

    public double Spacing
    {
      get { return (double)GetValue(SpacingProperty); }
      set { SetValue(SpacingProperty, value); }
    }

    public double AngleIncrement
    {
      get { return (double)GetValue(AngleIncrementProperty); }
      set { SetValue(AngleIncrementProperty, value); }
    }

    protected override Size MeasureOverride(Size availableSize)
    {
      foreach (UIElement child in this.Children)
      {
        // Даем каждому дочернему элементу столько места, сколь он хочет
        if (child != null)
          child.Measure(new Size(Double.PositiveInfinity,
                                 Double.PositiveInfinity));
```

```
    }

    // Для самой FanCanvas места не нужно, как и для SimpleCanvas
    return new Size(0, 0);
  }

  protected override Size ArrangeOverride(Size finalSize)
  {
    // Центрируем дочерние элементы
    Point location = new Point(0,0);
    double angle = GetStartingAngle();

    foreach (UIElement child in this.Children)
    {
      if (child != null)
      {
        // Выделяем потомку область предпочтительного размера
        child.Arrange(new Rect(location, child.DesiredSize));

        // ПРЕДУПРЕЖДЕНИЕ: подменяем заданное RenderTransform преобразованием,
        //                 которое располагает потомков в виде веера
        child.RenderTransform = new RotateTransform(angle,
        child.RenderSize.Width / 2, child.RenderSize.Height);

        // Подготавливаем смещение и угол для следующего потомка
        if (Orientation == Orientation.Vertical)
          location.Y += Spacing;
        else
          location.X += Spacing;

        angle += AngleIncrement;
      }
    }

    // Забираем себе все предоставленное нам место
    return finalSize;
  }

  double GetStartingAngle()
  {
    double angle;

    if (this.Children.Count % 2 != 0)
      // Нечетное, значит, угол среднего потомка равен 0
      angle = -AngleIncrement * (this.Children.Count / 2);
    else
      // Четное, два средних потомка отстоят на половину
      // AngleIncrement по разные стороны от 0
      angle = -AngleIncrement * (this.Children.Count / 2) + AngleIncrement / 2;

    // Поворачиваем на 90 градусов, если ориентация вертикальная
```

```
        if (Orientation == Orientation.Vertical)
          angle += 90;

        return angle;
      }
    }
  }
```

У FanCanvas есть кое-какие общие черты с тремя предыдущими панелями. На SimpleStackPanel и OverflowPanel она похожа тем, что дочерние элементы, по существу, располагаются в одном направлении. Как и у остальных панелей, у FanCanvas имеется свойство зависимости Orientation, хотя по умолчанию оно равно Horizontal, а в метаданных поднят флаг AffectsArrange вместо AffectsMeasure. Изменение ориентации не влияет на этап измерения, потому что реализация метода MeasureOverride не зависит от значения Orientation.

В классе FanCanvas определены еще два свойства зависимости, управляющих параметрами веера. Свойство Spacing управляет промежутком между соседними потомками в терминах логических пикселов. Оно аналогично переменной overlap в классе OverlapPanel, только в данном случае речь идет о степени *неперекрытия*. Свойство AngleIncrement управляет углом поворота следующего потомка относительно предыдущего. Угол выражен в градусах. По умолчанию оба свойства, Spacing и AngleIncrement, равны 10 и, как и Orientation, влияют только на этап размещения. Тот факт, что это свойства зависимости, открывает путь к разным любопытным анимациям панели.

В имени класса FanCanvas присутствует слово «Canvas», поскольку реализация метода MeasureOverride идентична той, что имеется в классе Canvas (и в классе SimpleCanvas выше в этой главе). Это означает, что каждому дочернему элементу выделяется столько места, сколько он хочет, а родителю сообщается о том, что для самой панели места вообще не нужно (опять-таки потому, что потомки не обрезаются по границам панели, если свойство ClipToBounds не равно true). Именно поэтому измерение не зависит от ориентации; алгоритму безразлично, в каком направлении растет стопка. Слово «Canvas» также оправдывает относительную простоту политики компоновки! Более совершенная реализация учитывала бы точные углы и промежутки между потомками, чтобы вычислить размер ограничивающего прямоугольника для самой панели. А сейчас клиент FanCanvas должен явно задавать размеры и поля FanCanvas, чтобы получить желаемый результат.

Логика метода ArrangeOverride довольно близка к SimpleStackPanel, если не учитывать тот факт, что каждый потомок поворачивается с помощью преобразования RenderTransform на увеличивающийся с каждым разом угол. Начальный угол возвращает метод GetStartingAngle, который гарантирует, что либо средний потомок не повернут вовсе, либо – если число потомков четное – два средних потомка повернуты на равные углы относительно среднего положения (0° при горизонтальной или 90° при вертикальной ориентации).

Обычно панель не должна изменять свойства своих дочерних элементов (в частности, RenderTransform). Это может привести к недоразумениям, когда свойст-

ва потомков уже установлены клиентом, и нарушить сделанные клиентом предположения о работе программы. Можно было бы поступить иначе и определить однодетный элемент управления `FanCanvasItem`, который неявно содержит каждого потомка; тогда преобразования можно было бы применять к нему. Но так обычно поступают при разработке многодетных элементов управления, а не панелей. Панель `FanCanvas` прекрасно работает в тех ситуациях, где с ее ограничениями можно смириться.

На рис. 21.4 показана панель `FanCanvas` в сочетании с экземплярами нестандартного элемента управления `PlayingCard`, разработанного в предыдущей главе. По-разному задавая свойства `Spacing` и `AngleIncrement`, можно получить всякие интересные раскладки!



`Spacing`=10, `AngleIncrement`=10 (по умолчанию)



`Spacing`=30, `AngleIncrement`=10



`Spacing`=10, `AngleIncrement`=30



`Spacing`=0, `AngleIncrement`=30



`Spacing`=110, `AngleIncrement`=50

Рис. 21.4. *Панель FanCanvas в сочетании с элементом управления PlayingCard из предыдущей главы*

Резюме

В этой главе мы рассмотрели механизмы взаимодействия родительских панелей со своими дочерними элементами – как они находят компромисс, дающий прекрасные результаты в самых разных ситуациях. Разработка собственных панелей считается «дополнительным вопросом» только потому, что делать это приходится редко. Как вы могли убедиться, написать класс панели довольно просто. Благодаря протоколу измерения/размещения и всей той работе, которую автоматически проделывает WPF, существующие элементы управления можно помещать на совершенно новые панели, и они будут вести себя вполне корректно.

Как и при создании нестандартных элементов управления, нужно сначала потратить некоторое время на выбор подходящего базового класса. Впрочем, для панелей возможностей не так уж много. Как правило, имеет смысл наследовать прямо классу `Panel` – так мы и поступали в примерах из этой главы. Если вы планируете поддержать виртуализацию пользовательского интерфейса, то в качестве базового класса стоит взять `VirtualizingPanel`; этому абстрактному классу наследует, в частности, `VirtualizingStackPanel`. Иногда бывает удобно унаследовать какому-нибудь другому подклассу `Panel` (например, `Canvas` или `DockPanel`), особенно если вы собираетесь поддержать те же присоединенные свойства, что уже определены в этом классе.

Алфавитный указатель

Символы

\\ (обратная косая черта), 56
{} (фигурные скобки), 56, 429

A

About, диалоговое окно
  Help, команда, 227
  маршрутизируемые события, 197
  первый вариант кода, 100
  перенос свойств шрифта во внутреннюю панель StackPanel, 117
  присоединенные события, 200
  установка свойств шрифта в корневом элементе, 111
Action, свойство (класса QueryContinueDragEventArgs), 208
ActiveEditingMode, свойство (класса InkCanvas), 366
ActiveX элементы управления, 788
ActualHeight, свойство (класса FrameworkElement), 128
ActualWidth, свойство (класса FrameworkElement), 128
AddBackEntry, метод, 256
AddHandler, метод, 194
Aero Glass, 292
AlternationCount, свойство (класса ItemsControl), 321
AlternationIndex, свойство (класса ItemsControl), 321
AmbientLight, 632, 637
AmbientMaterial, класс, 644
AnchoredBlock, класс, 375
AngleX, свойство (класса SkewTransform), 141
AngleY, свойство (класса SkewTransform), 141
Angle, свойство (класса RotateTransform), 137
AnnotationService, класс, 380
ApplicationCommands, класс, 225
ApplicationPath, свойство (класса JumpTask), 279
Application, класс
  Properties, коллекция, 239
  Windows, коллекция, 239
  Run, метод, 236
  объяснение, 236
  приложения в одном экземпляре, 240
  события, 238
  создание приложения без, 241
Apply, метод, 286
ArcSegment, класс, 538
Arguments, свойство (класса JumpTask), 279
ArrangeOverride, метод, переопределение, 828
AutoReverse, свойство (классы анимаций), 686
AxisAngleRotation3D, класс, 627
AxMsTscAxNotSafeForScripting, элемент управления, 790

B

BAML (Binary Application Markup Language), язык, 69, 70
  декомпиляция в XAML, 72
BAML2006Reader, класс, 78
BaseValueSource, перечисление, 115
BeginTime, свойство (классы анимации), 685
BezierSegment, класс, 538
Binding, расширение разметки, 66
BindingMode, перечисление, 459
BitmapCache, класс, 600
BitmapCacheBrush, класс, 601
BitmapEffect, класс, 595

BitmapScalingMode, свойство (класса RenderOptions), 353
BlackoutDates, свойство (элемента управления Calendar), 387
BlockUIContainer, блок, 369
BlurEffect, 594
BooleanToVisibilityConverter, 437
BrushConverter, конвертер типа, 54
BuildWindowCore, метод, 756
Button, класс, 107, 306
ButtonAutomationPeer, класс, 307
ButtonBase, класс, 305
By, свойство (классы анимации), 684

C

C++/CLI, 753
Calendar, элемент управления, 386
CAML (Compiled Application Markup Language), язык, 70
Cancel, метод, 221
CanExecute, метод, 225
CanExecuteChanged, метод, 225
CanUserAddRows, свойство (класса DataGrid), 344
CanUserDeleteRows, свойство (класса DataGrid), 344
Canvas, панель, 147
  моделирование с помощью Grid, 169
Center, свойство (класса RadialGradientBrush), 582
CenterX, свойство
  класса RotateTransform, 137
  класса SkewTransform, 141
CenterY, свойство
  класса RotateTransform, 137
  класса SkewTransform, 141
CheckBox, класс, 308
Class, ключевое слово, 67
ClearAllBindings, метод, 416
ClearBinding, метод, 416
ClearHighlightsCommand, 380
ClearValue, метод, 115
CLI (Common Language Infrastructure), 753
Click, событие, 306
ClickCount, свойство (класса MouseButtonEventArgs), 206
ClickMode, свойство (класса ButtonBase), 305
ClickOnce, 247
  и неуправляемый код, 249
  кэширование, 265
ClipboardCopyMode, свойство (DataGrid), 342
ClipToBounds, свойство (панели), 173
/clr, флаг компилятора, 758
clr-namespace, директива, 62
Code, ключевое слово, 71
CoerceValueCallback, делегат, 116
Collapsed, значение (перечисления Visibility), 131
CollectionViewSource, класс, 449
Color, структура, 576
CombinedGeometry, класс, 545
ComboBox, элемент управления, 326
  ComboBoxItem, объекты, 331
  IsEditable, свойство, 329
  IsReadOnly, свойство, 329
  SelectionChanged, событие, 331
  события, 326
ComboBoxItem, класс, 331
ComCtl32.dll, 298
Complete, метод, 221
ComponentCommands, класс, 225
CompositeCollection, класс, 466
CompositionTarget_Rendering, обработчик события, 787
ConstantAttenuation, свойство (класса PointLight), 634
ContainerUIElement3D, класс, 660
Content, действие при построении, 394
Content, свойство, 58
  ContentControl, класс, 493
  Frame, класс, 316
ContentControl, класс, 304, 493
ContentElement, класс, 100
ContextMenu, элемент управления, 347
ContextMenuService, класс, 348
Control, класс, 100
Convert, метод, 436
ConvertXmlStringToObjectGraph, метод, 90
CountToBackgroundConverter, класс, 435
CreateBitmapSourceFromHBitmap, метод, 781
CreateHighlightCommand, 380
CreateInkStickyNoteCommand, 380
CreateTextStickyNoteCommand, 380
CreateWindow, функция, 757
CurrentItem, свойство (интерфейса ICollectionView), 447

CustomCategory, свойство (класса JumpItem), 280

D

D3DImage, класс, 781
DashStyle, свойство (класса Pen), 549
Data, свойство (класса DragEventArgs), 207
DataContext, свойство, 426
DataGrid, элемент управления
  CanUserAddRows, свойство, 344
  CanUserDeleteRows, свойство, 344
  ClipboardCopyMode, свойство, 342
  EnableColumnVirtualization, свойство, 342
  EnableRowVirtualization, свойство, 342
  FrozenColumnCount, свойство, 343
  RowDetailsVisibilityMode, свойство, 342
  SelectionMode, свойство, 341
  SelectionUnit, свойство, 341
  взаимодействие с буфером обмена, 341
  виртуализация, 342
  выбор строк и столбцов, 341
  дополнительная информация для строк, 342
  замораживание столбцов, 343
  пример, 337
  редактирование данных, 343
  типы столбцов, 339
DataGridCheckBoxColumn, 339
DataGridComboBoxColumn, 339
DataGridHyperlinkColumn, 339
DataGridTemplateColumn, 339
DataGridTextColumn, 339
DataTrigger, класс, 485
DatePicker, элемент управления, 388
DateValidationError, событие, 389
DeadCharProcessedKey, свойство (класса KeyEventArgs), 202
DeleteStickyNotesCommand, 380
DependencyObject, класс, 99, 108
DependencyPropertyHelper, класс, 115
DesiredSize, свойство (класса FrameworkElement), 127
DestroyWindowCore, метод, 756
DialogFunction, функция, 766
DiffuseMaterial, 640
Direct3D, 32
Direction, свойство
  класса DirectionalLight, 633
  класса SpotLight, 637
DirectionalLight, 632
DirectX
  интероперабельность с WPF, 36, 781
  когда использовать, 34
  разработка, 30
  сравнение с WPF, 34
DispatcherObject, класс, 98
DispatcherPriority, перечисление, 242
DispatcherTimer, класс, 676
DisplayDateEnd, свойство (элемента управления Calendar), 387
DisplayDateStart, свойство (элемента управления Calendar), 387
DisplayMemberPath, свойство, 321, 423
Dock, свойство (класса DockPanel), 154
DockPanel, панель
  взаимодействие со свойствами компоновки дочерних элементов, 157
  моделирование с помощью Grid, 170
  примеры, 154
  свойства, 154
DoNothing, значение (Binding), 439
DoubleAnimation, класс, 680
DownloadFileGroupAsync, метод, 271
DragEventArgs, класс, 207
Drawing, класс, 534
DrawingBrush, класс, 584
DrawingContext, класс
  методы, 554
  пример изображения, 554
DrawingImage, класс, 536
DrawingVisual, класс, 535
DropDownClosed, событие, 326
DropDownOpened, событие, 326
DropShadowEffect, 594
Duration, свойство (классы анимации), 682
DwmExtendFrameIntoClientArea, функция, 292
DynamicResource, расширение разметки, 407

E

Ease, метод, 710
EaseIn, режим, 712
EaseInOut, режим, 712
EasingFunction, свойство, 689, 706

EasingFunctionBase, класс, 711
EasingMode, свойство, 706
EditingCommands, класс, 226
EditingMode, свойство (класса InkCanvas), 366
EditingModeInverted, свойство (класса InkCanvas), 366
ElementHost, класс, 777
ElementName, свойство (класса Binding), 417
EllipseGeometry, класс, 537
EmbeddedResource, действие при построении, 395
EmissiveMaterial, 645
EnableClearType, свойство (класса BitmapCacheBrush), 601
EnableColumnVirtualization, свойство (элемента DataGrid), 342
EnableRowVirtualization, свойство (элемента DataGrid), 342
EnableVisualStyles, метод, 776
EndLineCap, свойство (класса Pen), 549
EndMember, значение (свойства NodeType), 82
EndObject, значение (свойства NodeType), 82
EndPoint, свойство (класса LinearGradientBrush), 578
EscapePressed, свойство (класса QueryContinueDragEventArgs), 208
EvenOdd, значение (свойства FillRule), 541
ExceptionValidationRule, объект, 464
Execute, метод, 225
Expander, класс, 318
Expansion, свойство (класса ManipulationDelta, 217
Expression Blend, 36, 38
ExtendGlassFrame, метод, 294

F

FanCanvas, 842
FileInputBox, элемент управления
  защита от непреднамеренного использования, 801
  маршрутизируемые события, 804
  поведение, 799
  пользовательский интерфейс, 796
  свойства зависимости, 802
FillBehavior, свойство (классы анимации), 690
FillRule, свойство (класса PathGeometry), 541
Filter, свойство (интерфейса ICollectionView), 446
FindResource, метод, 410
FirstDayOfWeek, свойство (элемента Calendar), 388
FlowDirection, свойство (класса FrameworkElement), 134
FlowDocument, класс, 367
FlowDocumentPageViewer, элемент управления, 378
FlowDocumentReader, элемент управления, 378
FlowDocumentScrollViewer, элемент управления, 378
FontSizeConverter, конвертер типа, 54
FormatConvertedBitmap, класс, 357
Frame, класс, 250, 314
FrameworkContentElement, класс, 100, 106, 367
FrameworkElement, класс, 100, 106
  ActualHeight, свойство, 128
  ActualWidth, свойство, 128
  DesiredSize, свойство, 127
  FlowDirection, свойство, 134
  Height, свойство, 126
  HorizontalAlignment, свойство, 132
  HorizontalContentAlignment, свойство, 133
  LayoutTransform, свойство, 135
  Margin, свойство, 128
  Padding, свойство, 128
  RenderSize, свойство, 128
  RenderTransform, свойство, 135
  Triggers, свойство, 111
  VerticalAlignment, свойство, 132
  Visibility, свойство, 131
  Width, свойство, 126
FrameworkPropertyMetadata, 804
Freezable, класс, 99
From, свойство (классы анимации), 683
FromArgb, метод, 780
FrozenColumnCount, свойство (элемента DataGrid), 343

G

GDI (Graphics Device Interface), 30
  GDI+, 30
  и аппаратное ускорение, 34
Geometry3D, класс, 647

GeometryCombineMode, перечисление, 545
GeometryDrawing, класс, 534
GeometryModel3D, 632, 639
  Geometry3D, класс, 647
  MeshGeometry3D, класс, 648
    Normals, свойство, 651
    Positions, свойство, 648
    TextureCoordinates, свойство, 653
    TriangleIndices, свойство, 650
  материалы, 639
    AmbientMaterial, 644
    DiffuseMaterial, 640
    EmissiveMaterial, 645
    комбинирование, 647
GetCommandLineArgs, метод, 238
GetExceptionForHR, метод, 76
GetGeometry, метод, 538
GetHbitmap, метод, 781
GetInstalledVoices, метод, 735
GetIntermediateTouchPoints, метод, 212
GetObject, значение (свойства NodeType), 82
GetPosition, метод, 206
GetTouchPoint, метод, 212
GetValueSource, метод, 115
GetVisualChild, метод, 556
GlyphRunDrawing, класс, 535
GradientBrush, класс, 61
GradientOrigin, свойство (класса RadialGradientBrush), 582
GradientStop, объекты, 577
GrammarBuilder, класс, 742
Grid, панель, 158
  ShowGridLines, свойство, 162
  взаимодействие со свойствами компоновки дочерних элементов, 170
  задание общего размера для строк и столбцов, 166
  задание размеров строк и столбцов
    абсолютное, 162
    автоматическое, 162
    в процентах, 164
    пропорциональное, 163
    структуры GridLength, 164
  интерактивное задание размера с помощью GridSplitter, 165
  моделирование Canvas, 169
  моделирование DockPanel, 170
  моделирование StackPanel, 170
  свойства ячеек, 161
  сравнение с другими панелями, 169
GridLength, структуры, 164
GridLengthConverter, 164
GridSplitter, класс, 165
GridView, элемент управления, 335
GridViewColumn, объект, 335
GroupBox, элемент управлени, 316
GroupDescriptions, свойство (интерфейса ICollectionView), 443
GroupName, свойство (класса RadioButton), 310

H

Handled, свойство (класса RoutedEventArgs), 196
HandleRef, 756
HasContent, свойство (класса ContentControl), 304
HasItems, свойство (класса ItemsControl), 320
Header, свойство (класса ToolBar), 354
HeaderedItemsControl, класс, 345
Height, свойство (класса FrameworkElement), 126
Help, команда, 227
Hidden, значение (перечисления Visibility), 131
HierarchicalDataTemplate, класс, 434, 455
HitTest, метод, 560
HitTestCore, метод, 565
HitTestFilterCallback, делегат, 565
HitTestResultCallback, делегат, 564
HorizontalAlignment, свойство (класса FrameworkElement), 132
HorizontalContentAlignment, свойство (класса FrameworkElement), 133
HostingWin32.cpp, файл, 757
HostingWPF.cpp, файл, 765
HwndHost, класс, 757
HwndSource, класс, 765
HwndSource, переменная, 770

I

ICC (International Color Consortium), 577
ICommand, интерфейс, 225
Icon, свойство (класса MenuItem), 346

IconResourceIndex, свойство (класса JumpTask), 279
IconResourcePath, свойство (класса JumpTask), 279
ICustomTypeDescriptor, интерфейс, 420
IEasingFunction, интерфейс, 710
IList, интерфейс, 59
Image, элемент управления, 356
ImageBrush, класс, 588
ImageDrawing, класс, 534, 535, 536
ImageSource, класс, 357
ImageSourceConverter, конвертер типа, 356
ImeProcessedKey, свойство (класса KeyEventArgs), 202
InAir, свойство (класса StylusDevice), 210
InitializeComponent, метод, 70, 234
InitialShowDelay, свойство (класса ToolTip), 314
InkCanvas, класс, 365
Inline, элементы
  AnchoredBlock, 375
  InlineUIContainer, 378
  LineBreak, 376
  Span, 374
  определение, 373
Inlines, свойство (класса TextBlock), 362
InlineUIContainer, класс, 378
InnerConeAngle, свойство класса PointLights, 637
InputGestureText, свойство (класса MenuItem), 346
InputHitTest, метод, 574
IntelliSense, 97
Inverted, свойство (класса StylusDevice), 210
IsAdditive, свойство (классы анимации), 689
IsAsync, свойство (класса Binding), 457
IsCheckable, свойство (класса MenuItem), 346
IsChecked, свойство (класса ToggleButton), 308
IsCumulative, свойство (классы анимации), 690
IsDefault, свойство (класса Button), 107, 307
IsDefaulted, свойство (класса Button), 307
IsDirective, свойство, 83
IsDown, свойство (класса KeyEventArgs), 202
IsEditable, свойство (класса ComboBox), 329
IsFrontBufferAvailableChanged, обработчик события, 785
IsGrouping, свойство (класса ItemsControl), 320
IsIndeterminate, свойство (класса ProgressBar), 384
IsKeyboardFocused, свойство (класса UIElement), 204
IsKeyDown, метод, 204
IsMouseDirectlyOver, свойство (класса UIElement), 205
IsNetworkDeployed, свойство, 271
IsolatedStorage, пространство имен, 247
IsolatedStorageFile, класс, 247
IsolatedStorageFileStream, класс, 247
IsPressed, свойство (класса ButtonBase), 305
IsReadOnly, свойство (класса ComboBox), 329
IsRepeat, свойство (класса KeyEventArgs), 203
IsSelected, свойство (класса Selector), 326
IsSelectionActive, свойство (класса Selector), 326
IsSynchronizedWithCurrentItem, свойство (класса Selector), 425
IsTextSearchCaseSensitive, свойство (класса ItemsControl), 331
IsTextSearchEnabled, свойство (класса ItemsControl), 331
IsThreeState, свойство (класса ToggleButton), 308
IsToggled, свойство (класса KeyEventArgs), 202
IsUp, свойство (класса KeyEventArgs), 202
ItemHeight, свойство (класса WrapPanel), 152
Items, свойство (класса ItemsControl), 319, 425
ItemsCollection, класс, 334
ItemsControl, класс
  AlternationCount, свойство, 321
  AlternationIndex, свойство, 321
  DisplayMemberPath, свойство, 321
  HasItems, свойство, 320

IsGrouping, свойство, 320
IsTextSearchCaseSensitive, свойство, 331
IsTextSearchEnabled, свойство, 331
Items, свойство, 319
ItemsPanel, свойство, 321
ItemsSource, свойство, 320
управление поведением прокрутки, 325
ItemsPanel, свойство (класса ItemsControl), 321
ItemsSource, коллекция, 343
ItemsSource, свойство (класса ItemsControl), 320, 425
ItemWidth, свойство (класса WrapPanel), 152
IValueConverter, интерфейс, 435
IXamlLineInfo, интерфейс, 83, 90
IXamlLineInfoConsumer, интерфейс, 90

J

JavaScript, 32
JournalOwnership, свойство (класса Frame), 254
JumpItemRejectionReason, перечисление, 286
JumpItemsRejected, событие, 285
JumpItemsRemovedByUser, событие, 286

K

Kaxaml, 45
Key, перечисление, 202
Key, свойство (класса KeyEventArgs), 202
KeyboardDevice, свойство (класса KeyEventArgs), 203
KeyboardNavigation, класс, 353
KeyDown, событие, 202
KeyEventArgs, класс, 202
KeyStates, свойство
класса KeyEventArgs, 203
класса QueryContinueDragEventArgs, 207
KeyUp, событие, 202

L

Label, класс, 311
LastChildFill, свойство (класса DockPanel), 155
LayoutTransform, свойство (класса FrameworkElement), 135
LengthConverter, конвертер типа, 130
Light, объекты
AmbientLight, 632, 637
DirectionalLight, 632, 633
PointLight, 632, 634
SpotLight, 632, 635
определение, 632
яркость, 634
Line, класс, 570
LinearAttenuation, свойство (класса PointLight), 634
LinearGradientBrush, класс, 577
LineBreak, класс, 376
LineGeometry, класс, 538
LineJoin, свойство (класса Pen), 549
LineNumber, свойство, 83
LinePosition, свойство, 83
LineSegment, класс, 538
LINQ (Language Integrated Query), 452
ListBox, элемент управления
SelectionMode, свойство, 332
идентификаторы автоматизации, 335
поддержка множественного выбора, 333
помещение объектов PlayingCard в, 816
пример, 332
прокрутка, 334
расположение объектов по горизонтали, 324
сортировка объектов, 334
ListView, элемент управления, 335
Load, метод, 63, 89
LoadAsync, метод, 65
LoadComponent, метод, 72
LocBaml, создание сателлитной сборки, 402
LogicalChildren, свойство, 106
LogicalTreeHelper, класс, 104
LookDirection, свойство (класса Camera), 611

M

mage.exe, утилита, 248
mageUI.exe, графическая утилита, 248
Main, метод, 235
MainWindow, класс, 233

ManipulationBoundaryFeedback, событие, 221
ManipulationCompleted, событие, 216
ManipulationDelta, событие, 216
ManipulationDeltaEventArgs, объект, 217
ManipulationStarted, событие, 216
ManipulationStarting, событие, 216, 223
Margin, свойство (класса FrameworkElement), 128
MarkupExtension, класс, 55
MatrixCamera, класс, 620
MatrixTransform3D, класс, 631
MatrixTransform, класс, 142
MeasureOverride, метод, переопределение, 826
MediaCommands, класс, 226
MediaElement, класс
  воспроизведение аудио, 726
  воспроизведение видео, 728
MediaPlayer, класс,
MediaTimeline, класс
  воспроизведение аудио, 726
  воспроизведение видео, 728
Menu, элемент управления, 345
MenuItem, класс, 346
MergedDictionaries, свойство (класса ResourceDictionary), 408
MeshGeometry3D, класс, 648
  Normals, свойство, 651
  Positions, свойство, 648
  TextureCoordinates, свойство, 653
  TriangleIndices, свойство, 650
Microsoft Anna, 734
Microsoft Surface, 38
Model3D, класс, 631
  см. также GeometryModel3D
  Light, объекты
    AmbientLight, 632, 637
    DirectionalLight, 632, 633
    PointLight, 632, 634
    SpotLight, 632, 635
    определение, 632
    яркость, 634
Model3DGroup, класс, 632, 654
ModelUIElement3D, класс, 658
ModelVisual3D, класс, 656
Modifiers, свойство (класса KeyboardDevice), 203
Mouse, класс, 208
MouseButtonEventArgs, класс, 206
MouseButtonState, перечисление, 206
MouseEventArgs, класс, 206
MouseWheelEventArgs, класс, 206
mscorlib, сборка, 75
MultiBinding, класс, 467
MultiPoint Mouse SDK, 212
MyHwndHost, класс, 756

N

Name, свойство, 65
NamespaceDeclaration, значение (свойства NodeType), 82
Navigate, метод, 252
NavigationCommands, класс, 226
NavigationProgress, событие, 257
NavigationStopped, событие, 257
NavigationWindow, класс, 250
NodeType, свойство, 82
None, значение (свойства NodeType), 83
NonZero, значение (свойства FillRule), 541
NoRegisteredHandler, значение (перечисления JumpItemRejectionReason), 286
Normals, свойство (класса MeshGeometry3D), 651
null-области, 205

O

Object, класс, 98
ObjectDataProvider, класс, 456
OneTime, режим привязки, 459
OneWay, режим привязки, 459
OneWayToSource, режим привязки, 459
OnMnemonic, метод, 763
OnNoMoreTabStops, метод, 762
Opacity, свойство (кисти), 592
OpacityMask, свойство (кисти), 592
OpenGL, 30
Orientation, свойство
  класса ProgressBar, 384
  класса StackPanel, 150
  класса WrapPanel, 152
OriginalSource, свойство (класса RoutedEventArgs), 196
OrthographicCamera, класс
  LookDirection, свойство, 611
  Position, свойство, 608
  UpDirection, свойство, 614
  коллизии в Z-буфере, 611

мертвая зона, 611
сравнение с PerspectiveCamera, 617
OuterConeAngle, свойство
класса PointLights, 637
OverlapPanel, 837
Overlay, свойство (класса TaskbarItemInfo), 288

P

pack, схема URI, 399
Padding, свойство (класса FrameworkElement), 128
Page, элементы, 249
PageFunction, класс, 260, 261
Parse, метод, 90
partial, ключевое слово, 68
PasswordBox, элемент управления, 364
Path, класс, 572
PathGeometry, класс, 538
Pen, класс, 548
PerspectiveCamera, класс, *см.* OrthographicCamera, класс
PInvoke, 293
PixelFormats, перечисление, 358
Play, метод, 723
PlayingCard, элемент управления
поведение
застраничный файл, 807
окончательная реализация, 812
первая попытка реализации, 807
ресурсы, 807
типовые ресурсы, 814
пользовательский интерфейс, 813
помещение в ListBox, 816
PointLight, класс, 634
PolyBezierSegment, класс, 538
Polygon, класс, 572
Polyline, класс, 571
PolyLineSegment, класс, 538
PolyQuadraticBezierSegment, класс, 538
Position, свойство (камеры), 608
Positions, свойство (класса MeshGeometry3D), 648
PreferredXamlNamespace, свойство, 83
PressureFactor, свойство (класса StylusPoint), 210
PreviewKeyDown, событие, 202
PreviewKeyUp, событие, 202
PrintLogicalTree, метод, 105
PrintVisualTree, метод, 105
PriorityBinding, класс, 468
ProgressBar, элемент управления, 384
помещение на панель задач, 287
шаблон в виде секторной диаграммы, 500, 511
ProgressState, свойство (класса TaskbarItemInfo), 288
ProgressValue, свойство (класса TaskbarItemInfo), 288
PromptBuilder, класс, 735
PropertyGrid
встраивание с помощью XAML, 775
встраивание с помощью процедурного кода, 773
PropertyGroupDescription, класс, 444

Q

QuadraticAttenuation, свойство (класса PointLight), 634
QuadraticBezierSegment, класс, 538
QuaternionRotation3D, класс, 627
QueryContinueDragEventArgs, класс, 207

R

RadialGradientBrush, класс, 581
RadioButton, класс, 309
RadiusX, свойство
класса RadialGradientBrush, 582
класса Rectangle, 568
RadiusY, свойство
класса RadialGradientBrush, 582
класса Rectangle, 568
Range, свойство (класса PointLight), 636
RectangleGeometry, класс, 537
Reference, расширение разметки, 66
Refresh, метод, 255
Register, метод, 108
RelativeSource, свойство (объекта Binding), 418
RemovedByUser, значение (перечисления JumpItemRejectionReason), 286
RemoveHandler, метод, 194
RenderAtScale, свойство (класса BitmapCache), 599
Rendering, событие, 677
RenderSize, свойство (класса FrameworkElement), 127
RenderTargetBitmap, класс, 598

RenderTransform, свойство (класса FrameworkElement), 135
RenderTransformOrigin, свойство (класса FrameworkElement), 136
RepeatBehavior, свойство (классы анимации), 687
RepeatButton, класс, 307
ResizeBehavior, свойство (класса GridSplitter), 166
ResizeDirection, свойство (класса GridSplitter), 166
Resource, действие при построении, 394
ResourceDictionary, класс, 408
ResourceDictionaryLocation, класс, 526
RichTextBox, элемент управления, 364
RotateTransform, 137
RotateTransform3D, класс, 627
Rotation, свойство (класса ManipulationDelta, 217
RoutedEvent, свойство (класса RoutedEventArgs), 196
RoutedUICommand, класс, 226
RoutingStrategy, перечисление, 195
RowDetailsVisibilityMode, свойство (класса DataGrid), 342
Run, метод, 236

S

Save, метод, 90
ScaleTransform, 139, 178
ScaleTransform3D, класс, 623
Scalr, свойство (класса ManipulationDelta, 216
scRGB, цветовое пространство, 575
ScrollBar, элемент управления, 175
ScrollViewer, элемент управления, 175
SelectedDateChanged, событие, 389
SelectedIndex, свойство (класса Selector), 326
SelectedItem, свойство (класса Selector), 326
SelectedValue, свойство (класса Selector), 326
SelectionChanged, событие, 326, 331
SelectionMode, свойство
  Calendar, элемент управления, 386
  DataGrid, элемент управления, 341
  ListBox, элемент управления, 332
SelectionUnit, свойство (элемента управления DataGrid), 341
SelectiveScrollingGrid, панель, 172
Selector, класс, 325
SelectVoice, метод, 735
SelectVoiceByHints, метод, 735
Separator, элемент управления, 346
SetBinding, метод, 416
SetCurrentValue, метод, 116
SetOutputToDefaultAudioDevice, метод, 735
SetOutputToWaveFile, метод, 735
SetResourceReference, метод, 410
Settings, класс, 247
ShaderEffect, 596
ShowDialog, метод, 245
ShowDuration, свойство (класса ToolTip), 314
ShowFrequentCategory, свойство (класса JumpList), 284
ShowGridLines, свойство (класса Grid), 162
ShowOnDisabled, свойство
  класса ContextMenuService, 348
  класса ToolTipService, 314
ShowRecentCategory, свойство (класса JumpList), 284
ShutdownMode, перечисление, 239
Silverlight, 38, 40, 215
SimpleCanvas, панель, 830
SimpleQuadraticEase, класс, 711
SimpleStackPanel, панель, 834
SkewTransform, класс, 141
Skip, метод, 88
Slider, элемент управления, 385
SnapsToDevicePixels, свойство (класса BitmapCache), 39, 599, 601
Snoop, 36
SolidColorBrush, класс, 61, 575
SortDescription, класс, 450
SortDescriptions, коллекция, 441
SortDescriptions, свойство
  интерфейса ICollectionView, 440
  класса ItemsCollection, 334
SoundPlayer, класс, 723
SoundPlayerAction, класс, 724
Source, свойство
  класса MediaElement, 726
  класса RoutedEventArgs, 196
SourceName, свойство (класса Trigger), 492
SpeakAsync, метод, 734
SpeakAsyncCancelAll, метод, 735
SpeechRecognitionEngine, класс, 738

SpeechSynthesizer, класс, 735
SpeedRatio, свойство (классы анимации), 686
SpotLight, 632
SpreadMethod, свойство (класса LinearGradientBrush), 578
sRGB, цветовое пространство, 575
SRGS (Speech Recognition Grammar Specification), 741
SSML (Speech Synthesis Markup Language), 735
StackPanel, панель, 150
  взаимодействие со свойствами компоновки дочерних элементов, 151
  в сочетании с Menu, 347
  задание свойств шрифтов, 117
  моделирование с помощью Grid, 170
StartMember, значение (свойства NodeType), 82
StartObject, значение (свойства NodeType), 82
StartPoint, свойство (класса LinearGradientBrush), 578
StartupUri, свойство (класса Application), 237
STAThreadAttribute, 767
StaticResource, расширение разметки, 407
StatusBar, элемент управления, 354
StopLoading, метод, 255
Storyboard, класс (раскадровка)
  TargetName, свойство, 694
  TargetProperty, свойство, 691
  внутри триггеров событий, 690
  как временная шкала, 698
StreamGeometry, класс, 542
Stretch, вид выравнивания, 132
Stretch, перечисление, 178
Stretch, свойство
  класса DrawingBrush, 585
  класса MediaElement, 728
StringFormat, свойство (объекта Binding), 428
Stroke, класс, 365
StylusButtonEventArgs, класс, 211
StylusButtons, свойство (класса StylusDevice), 210
StylusDevice, класс, 210
StylusDownEventArgs, класс, 211
StylusEventArgs, класс, 211
StylusPoint, объекты, 210
StylusSystemGestureEventArgs, класс, 211
Subclass, ключевое слово, 68
Surface Toolkit for Windows Touch, 224
SystemKey, свойство (класса KeyEventArgs), 202
System.Object, тип XAML, 53
SystemSounds, класс, 723
System.String, тип XAML, 53

T

TabControl, элемент управления, 336
TabInto, метод, 761
TabletDevice, свойство (класса StylusDevice), 210
TargetName, свойство, 694
TargetNullValue, свойство (объекта Binding), 418
TargetProperty, свойство, 694
TargetType, свойство
  класса ControlTemplate, 492
  класса Style, 478
TaskDialog, 296
TemplateBindingExtension, класс, 493
TextBlock, элемент управления, 360
TextBox, элемент управления, 362
TextElement, класс, 368
  Inline, элементы
    AnchoredBlock, 375
    InlineUIContainer, 378
    LineBreak, 376
    Span, 374
    определение, 373
  блоки
    AnchoredBlock, класс, 375
    BlockUIContainer, 369
    List, 369
    Paragraph, 369
    Section, 369
    Table, 369
    пример кода, 370
TextFormattingMode, свойство (класса TextOptions), 359
TextHintingMode, свойство (класса TextOptions), 360
TextOptions, класс, 359
TextRenderingMode, свойство (класса TextOptions), 359
TextureCoordinates, свойство (класса MeshGeometry3D), 653

ThemeDictionaryExtension, 527
ThemeInfoAttribute, 526
Thickness, класс, 128
ThicknessConverter, конвертер типа, 130
ThumbButtonInfos, свойство (класса TaskbarItemInfo), 290
ThumbnailClipMargin, свойство (класса TaskbarItemInfo), 289
TileMode, перечисление, 587
TileMode, свойство (класса DrawingBrush), 586
To, свойство (классы анимации), 683
ToggleButton, класс, 308
ToolBar, элемент управления, 351
ToolBarOverflowPanel, панель, 171
ToolBarPanel, панель, 171
ToolBarTray, панель, 171
ToolPanel, панель, 171
ToolTip, класс, 312
ToolTipService, класс, 314
TouchDevice, свойство (класса TouchEventArgs), 212
TouchDown, событие, 213
TouchEventArgs, класс, 212
TouchMove, событие, 213
TouchUp, событие, 213
TraceSource, объект, 438
Transform, метод, 90
Transform, свойство (камеры), 615
Transform3D, 620
  MatrixTransform3D, 621, 631
  RotateTransform3D, 621, 627
  ScaleTransform3D, 621, 623
  Transform3DGroup, 621
  TranslateTransform3D, 620, 623
  комбинирование, 630
TransformConverter, конвертер типа, 143
TransformToAncestor, метод, 666
TransformToDescendant, метод, 670
Transitions, свойство (класса VisualStateGroup), 514
TranslateAccelerator, метод, 761
TranslateTransform, 142
TranslateTransform3D, класс, 623
Translation, свойство (класса ManipulationDelta, 216
TreeView, элемент управления, 349
TreeViewItem, класс, 349
TriangleIndices, свойство (класса MeshGeometry3D), 650
TriggerBase, класс, 111
Triggers, свойство (класса FrameworkElement), 111
TryFindResource, метод, 410
TwoWay, режим привязки, 459

U

UICulture, элемент, 401
Uid, директива, 401
UIElement, класс, 99
  IsKeyboardFocused, свойство, 204
  IsMouseDirectlyOver, свойство, 205
  RenderTransformOrigin, свойство, 136
  привязка к, 422
UIElement3D, класс, 36, 99
  ContainerUIElement3D, 660
  ModelUIElement3D, 658
UniformGrid, панель, 172
UpdateLayout, метод, 128
UpdateSourceExceptionFilter, свойство (объекта Binding), 465
UpdateSourceTrigger, перечисление, 461
UpDirection, свойство (камеры), 614
URI
  для доступа к двоичным ресурсам, 397
  со схемой pack, 399
UseLayoutRounding, свойство, 39
USER, подсистема, 30

V

ValidateValueCallback, делегат, 116
ValidationRules, свойство (объекта Binding), 462
Value, значение (свойства NodeType), 82
ValueSource, структура, 115
VerticalAlignment, свойство (класса FrameworkElement), 132
VideoDrawing, класс, 534
Viewbox, класс, 178
Viewport2DVisual3D, класс, 36, 660
Viewport3D, класс, 663
Viewport3DVisual, класс, 666
VirtualizingPanel, класс, 152
VirtualizingStackPanel, панель, 152
Visibility, свойство (класса FrameworkElement), 131
Visible, значение (перечисления Visibility), 131

Visual, класс, 99, 106
TransformToAncestor, метод, 666
Visual3D, класс, 99, 656
ModelVisual3D, 656
TransformToAncestor, метод, 670
TransformToDescendant, метод, 670
UIElement3D, класс, 658
VisualBrush, класс, 589
Visual C++, 766, 767
VisualChildrenCount, свойство, 556
VisualStateGroup, метод, 514
Visual State Manager (менеджер визуальных состояний), 38
и анимация
переходы, 716
шаблон кнопки с визуальными состояниями, 712
учет визуальных состояний
с помощью состояний элемента, 508
с помощью частей элемента, 505
Visual Studio 2010, 35
Visual Studio отладчик, 277
VisualTreeHelper, класс, 104
vshost32.exe, 277

W

Width, свойство (класса FrameworkElement), 126
Window, класс, 232
WindowInteropHelper, класс, 781
Windows, коллекция, 239
Windows 7, средства пользовательского интерфейса
Aero Glass, 292
TaskDialog, 296
настройка элементов на панели задач, 287
индикатор выполнения, 287
кнопки управления, 290
наложения, 288
содержимое эскиза, 289
поддержка в WPF 4, 38
списки переходов, 273
JumpPath, 282
JumpTask, 275
и отладчик Visual Studio, 277
связывание с приложением, 274
WindowsFormsHost, класс, 775
Windows Media Player, 727
Windows XP, особенности поведения WPF, 40
WorkingDirectory, свойство (класса JumpTask), 279
WPF
история, 30
основные возможности, 32
WPF Toolkit, 35
WPF XAML Vocabulary Specification 2006 (MS-WPFXV), 46
WrapPanel, панель, 152
примеры, 153
примеры взаимодействие со свойствами компоновки дочерних элементов, 154
размещение справа налево, 154
свойства, 152
WriteableBitmap, класс, 37

X

x, префикс, 49
x:Arguments, ключевое слово, 75, 93
x:Array, ключевое слово, 95
x:AsyncRecords, ключевое слово, 93
x:Boolean, ключевое слово, 93
x:Byte, ключевое слово, 93
x:Char, ключевое слово, 93
x:ClassAttributes, ключевое слово, 93
x:ClassModifier, ключевое слово, 93
x:Class, ключевое слово, 68, 93
x:Code, ключевое слово, 93
x:ConnectionId, ключевое слово, 93
x:Decimal, ключевое слово, 93
x:Double, ключевое слово, 93
x:FactoryMethod, ключевое слово, 76, 94
x:FieldModifier, ключевое слово, 94
x:Int16, ключевое слово, 94
x:Int32, ключевое слово, 94
x:Int64, ключевое слово, 94
x:Key, ключевое слово, 94
x:Members, ключевое слово, 77, 94
x:Name, ключевое слово, 66, 94, 492
x:Null, ключевое слово, 95
x:Object, ключевое слово, 94
x:Property, ключевое слово, 77, 94
x:Reference, ключевое слово, 96, 776
x:Shared, ключевое слово, 94, 409
x:Single, ключевое слово, 94
x:Static, ключевое слово, 96
x:String, ключевое слово, 94
x:Subclass, ключевое слово, 94

x:SynchronousMode, ключевое слово, 95
x:TimeSpan, ключевое слово, 95
x:TypeArguments, ключевое слово, 95
x:Type, ключевое слово, 96
x:Uid, ключевое слово, 95
x:Uri, ключевое слово, 95
x:XData, ключевое слово, 95
XAML (eXtensible Application Markup Language), 33
BAML (Binary Application Markup Language)
декомпиляция, 72
определение, 69
CAML (Compiled Application Markup Language), 70
XAML2009, *см.* XAML2009
XBAP-приложения
см. XAML Browser Application (XBAPs)
автономные страницы, 271
анимация с помощью триггеров событий и раскадровок, 690
TargetName, свойство, 694
TargetProperty, свойство, 691
запуск из триггера свойства, 697
раскадровка как временная шкала, 698
встраивание элемента PropertyGrid, 775
встроенный процедурный код, 71
вынесение, 408
генерируемый исходный код, 70
достоинства, 44
доступ к двоичным ресурсам, 395
загрузка и разбор во время выполнения, 63
записыватели (XAML), 77, 78
XamlServices, класс, 89
запись в объекты, 86
запись в формате XML, 88
циклы обработки узлов, 81
запуск примеров, 44
клиент Twitter, 469
ключевые слова, 92
компиляция, 67
конвертеры типов, 52
BrushConverter, 54
FontSizeConverter, 54
GridLengthConverter, 164
ImageSourceConverter, 356
LengthConverter, 130
ThicknessConverter, 130
TransformConverter, 143
в процедурном коде, 164
отключение преобразования типов, 74
поиск, 54
объект Binding, 417
объектные элементы, 47
атрибуты, 47
именование, 65
обработка дочерних элементов, 63
объявление, 47
позиционирование, 132
преобразование типов, 61
преобразования, *см.* преобразования
свойство содержимого, 58
словари, 60
списки, 59
управление размером, 126
объяснение, 33, 43
определение, 45
сновные свойства, 43
порядок обработки свойств и событий, 48
пространства имен XAML, 50
расширения разметки, 55
в процедурном коде, 57
параметры, 55
расширяемость, 61
системные типы данных, 75
спецификации, 46
считыватели (XAML), 77
NodeType, свойство, 82
XamlServices, класс, 89
поток узлов XAML, 84
пример XAML-разметки, 83
совместимость разметки, 86
циклы обработки узлов, 81
типичные возражения, 97
{} (фигурные скобки), 429
элементы свойств, 51
XAML Browser Application (XBAPs), 263
безопасность, 269
загрузка по требованию, 269
интегрированная навигация, 268
кэширование ClickOnce, 265
ограничения, 265
развертывание, 268
с полным доверием, 267

XAML Cruncher, 45
XAML Object Mapping Specification 2006 (MS-XAML), 46
XAML2009
  введение, 72
  встроенные типы данных, 75
  гибкость присоединения обработчиков событий, 76
  определение новых свойств, 77
  поддержка универсальных классов, 73
  словарные ключи, 74
  создание объектов с помощью
    конструктора с аргументами, 75
    фабричных методов, 76
  универсальные классы, 74
XamlBackgroundReader, класс, 78
XamlMember, класс, 83
XamlObjectReader, класс, 78
XamlObjectWriter, класс, 78
XamlPad, 45
XAMLPAD2009, 44
XamlPadX, 45, 102
XamlReader, класс, 78
  Load, метод, 63
  LoadAsync, метод, 65
XamlServices, класс, 89
XamlType, класс, 83
XamlWriter, класс, 72, 78
XamlXmlReader, класс, 80
  поток узлов XAML, 84
  пример XAML-содержимого, 83
  совместимость разметки, 86
XamlXmlWriter, класс, 78
XML Paper Specification (XPS), 367
XML Path Language (XPath), 453
xml:lang, атрибут, 92
xml:space, атрибут, 92
XmlDataProvider, класс, 452
XmlnsDefinitionAttribute, атрибут, 48
XNA Framework, 31

Z

Z-буфер, коллизии, 611
Z-порядок, 149

A

абсолютный размер, 162
автоматизация
  идентификаторы, 335
  пользовательского интерфейса, поддержка в нестандартных элементах управления, 823
автоматический выбор размера, 160
автономные XAML-страницы, 271
анимация, 116, 675
  вдоль пути, 706
  и менеджер визуальных состояний
    переходы, 716
    шаблон Button с визуальными состояниями, 712
  и привязка к данным, 701
  классы
    AutoReverse, свойство, 686
    BeginTime, свойство, 685
    By, свойство, 684
    DoubleAnimation, класс, 680
    Duration, свойство, 682
    EasingFunction, свойство, 689
    FillBehavior, свойство, 690
    From, свойство, 683
    IsAdditive, свойство, 689
    IsCumulative, свойство, 690
    RepeatBehavior, свойство, 687
    SpeedRatio, свойство, 686
    To, свойство, 683
    линейная интерполяция, 680
    универсальные типы, 679
    управление продолжительностью, 682
  переходные функции, 38
    BackEase, 708
    BounceEase, 708
    CircleEase, 708
    EasingMode, свойство, 706
    ElasticEase, 709
    ExponentialEase, 709
    SineEase, 709
    написание, 710
    степенные, 707
  повторное использование, 681
  покадровая, 676
  по таймеру, 676
  с опорными кадрами, 699
    дискретные опорные кадры, 703
    линейные опорные кадры, 700
    переходные опорные кадры, 706
    сплайновые опорные кадры, 702
  триггеры событий и раскадровки в XAML, 690
    TargetName, свойство, 694

TargetProperty, свойство, 691
запуск из триггера свойства, 697
раскадровка как временная шкала, 698
аппаратное ускорение, 32
аргументы командной строки, получение, 238
ассемблер, 34
атрибут
XML, 52
свойств XAML, 47
событий XAML, 47
аудио, поддержка
MediaElement, 725
MediaPlayer,
MediaTimeline, 726
SoundPlayer, 723
SoundPlayerAction, класс, 724
SystemSounds, класс, 723
внедренные ресурсы, 733
объяснение, 722
распознавание речи
описание грамматики на языке SRGS, 741
описание грамматики с помощью класса GrammarBuilder, 742
преобразование произнесенных слов в текст, 738
синтез речи, 734
GetInstalledVoices, метод, 735
PromptBuilder, класс, 735
SelectVoice, метод, 735
SelectVoiceByHints, метод, 735
SetOutputToWaveFile, метод, 735
SpeakAsync, метод, 734
Speech Synthesis Markup Language (SSML), 735
SpeechSynthesizer, класс, 735

Б

безопасность XBAP-приложений, 269
Безье кривые, 538
блокировка D3DImage, 786
буфер обмена, взаимодействие (DataGrid), 341

В

визуализация, 559
повышение производительности
BitmapCache, класс, 600
BitmapCacheBrush, класс, 601
RenderTargetBitmap, класс, 598
текста, 39
TextOptions, класс, 359
усовершенствования в WPF 4, 359
управление, 428
конвертеры значений, 434, 435, 438, 439
форматирование строк, 428
шаблоны данных, 431
визуальные объекты (Visual), 552
DrawingContext, методы, 554
DrawingVisual, класс, 553
отображение на экране, 556
проверка попадания в Visual, 559
виртуализация, 334, 342
внедренные ресурсы, 733
внутренняя панель, 322
временные шкалы, 698
время загрузки, 37
время холодного запуска, 242
всплытие, 195
встроенные команды, 225
выбор строк и ячеек, 341
выражения, 116

Г

генерируемый исходный код, 70
геометрические объекты
CombinedGeometry, класс, 545
EllipseGeometry, класс, 537
Geometry3D, класс, 647
GeometryGroup, класс, 542
LineGeometry, класс, 538
MeshGeometry3D, класс, 648
PathGeometry, класс, 538
RectangleGeometry, класс, 537
StreamGeometry, класс, 542
кривые Безье, 538
определение, 537
представление в виде строк, 546
пример домика, 603
составные геометрические объекты, 542
гиперссылки, 253
градиенты
LinearGradientBrush, класс, 577
RadialGradientBrush, класс, 581
объекты GradientStop, 577

перечисление GradientSpreadMethod, 578
прозрачные цвета, 584
грамматики
GrammarBuilder, класс, 742
Speech Recognition Grammar Specification (SRGS), 741
графическая система
с запоминанием, 35, 533
с непосредственной визуализацией, 35, 533
группировка, 443
группы загрузки, 269

Д

двоичные ресурсы
доступ
в первоисточнике, 398
из XAML, 395
из процедурного кода, 400
к ресурсам, внедренным в другую сборку, 398
локализация, 400
подготовка проекта для нескольких культур, 401
пометка пользовательского интерфейса идентификаторами локализации, 401
создание сателлитной сборки с помощью LocBaml, 402
определение, 393
двумерная графика
визуальные объекты, 552
DrawingContext, методы, 554
DrawingVisual, класс, 553
отображение на экране, 556
проверка попадания в Visual, 559
геометрические объекты
CombinedGeometry, класс, 545
EllipseGeometry, класс, 537
Geometry3D, класс, 647
GeometryGroup, класс, 542
LineGeometry, класс, 538
MeshGeometry3D, класс, 648
PathGeometry, класс, 538
RectangleGeometry, класс, 537
StreamGeometry, класс, 542
кривые Безье, 538
определение, 537
представление в виде строк, 546
пример домика, 603
составные геометрические объекты, 542
кисти
BitmapCacheBrush, класс, 601
DrawingBrush, класс, 584
ImageBrush, класс, 588
LinearGradientBrush, класс, 577
RadialGradientBrush, класс, 581
SolidColorBrush, класс, 575
VisualBrush, класс, 589
как маски непрозрачности, 592
консолидация с помощью логических ресурсов, 404
мозаичные, 584
объяснение, 575
применение без использования логических ресурсов, 402
преобразования
MatrixTransform, 142
RotateTransform, 137
ScaleTransform, 139
SkewTransform, 141
Transform3D, 620
TranslateTransform, 142
и отсечение, 175
комбинирование, 143
поддержка, 145
применение, 135
преобразования систем координат, 660, 665
Visual.TransformToAncestor, метод, 666
Visual3D.TransformToAncestor, метод, 670
Visual3D.TransformToDescendant, метод, 670
объяснение, 666
рисунки
Drawing, класс, 534
DrawingContext, методы, 554
DrawingImage, класс, 536
DrawingVisual, класс, 535
GeometryDrawing, класс, 534
GlyphRunDrawing, класс, 535
ImageDrawing, класс, 534
Pen, класс, 548
VideoDrawing, класс, 534
пример изображения, 550
усовершенствования в WPF 3.5, 36
фигуры, 566

Ellipse, класс, 569
Line, класс, 570
Path, класс, 572
Polygon, класс, 572
Polyline, класс, 571
Rectangle, класс, 568
изображение, составленное из объектов Shape, 573
как работают, 569
чрезмерное увлечение, 567
эффекты, 594
декларативное программирование, 33
декораторы, 178
делегаты
CoerceValueCallback, 116
ValidateValueCallback, 116
контравариантность, 202
деревья
визуальные, 102
логические, 100
диалоговые окна
About
первый вариант кода, 100
перенос свойств шрифта во внутреннюю панель StackPanel, 117
установка свойств шрифта в корневом элементе, 111
TaskDialog, 296
модальные, открытие из приложения
Win32, 771
WPF, 764, 776
Windows Forms, 781
немодальные, 233
нестандартные, 244
результат диалогового окна, 244
стандартные, 243
диапазонные элементы управления, 383
ProgressBar, 384
Slider, 385
документы потоковые
блоки
AnchoredBlock, класс, 375
BlockUIContainer, 369
List, 369
Paragraph, 369
Section, 369
Table, 369
определение, 367
отображение, 378
пример кода, 370
добавление комментариев, 380
объекты Inline
AnchoredBlock, класс, 375
InlineUIContainer, 378
LineBreak, 376
Span, 374
определение, 373
создание, 367

Ж

жесты ввода
выполнение команд с помощью, 228
журнал, 254

З

загрузка XAML во время выполнения, 63
записыватели (XAML), 77, 78
XamlServices, класс, 89
запись в объекты, 86
запись в формате XML, 88
циклы обработки узлов, 81
заставки, 242
застраничные файлы, 68, 807
затухание, 634
захват событий мыши, 208

И

идентификаторы локализации, пометка пользовательского интерфейса, 401
иерархия классов, 98
изображение, пример
из объектов Shape, 573
из рисунков, 550
средствами DrawingContext, 554
файл WindowHostingVisual.cs, 556
изолированное хранилище, 246
именованные элементы, 492
Ингебретсен Робби, 45
инерция, добавление, 219
интеграция WPF, 32
интерактивность, 36
интероперабельность, 747
C++/CLI, 753
перекрывающееся содержимое, 749
с DirectX, 36, 781
с элементами ActiveX, 788

с элементами управления Win32, 750
HwndSource, класс, 765
Webcam, элемент управления, 750
клавиатурная навигация, 760
компоновка, 768
открытие модальных диалоговых окон, 764, 771
с элементами управления Windows Forms, 772
ElementHost, класс, 777
PropertyGrid, 773
открытие модальных диалоговых окон, 776, 781
преобразование представлений, 780

K

календарные элементы управления
Calendar, 386
DatePicker, 388
камеры
LookDirection, свойство, 611
MatrixCamera, класс, 620
Position, свойство, 608
Transform, свойство, 615
UpDirection, свойство, 614
коллизии в Z-буфере, 611
мертвая зона, 611
ортографическая и перспективная, 617
системы координат, 607
кисти
BitmapCacheBrush, класс, 601
DrawingBrush, класс, 584
ImageBrush, класс, 588
LinearGradientBrush, класс, 577
RadialGradientBrush, класс, 581
SolidColorBrush, класс, 575
VisualBrush, класс, 589
как маски непрозрачности, 592
консолидация с помощью логических ресурсов, 404
мозаичные, 584
объяснение, 575
применение без использования логических ресурсов, 402
клавиатурная навигация
настройка, 353
поддержка в элементах Win32, 760
клиентская конфигурация .NET, 37
кнопки
Button, класс, 107, 306
ButtonBase, класс, 305
CheckBox, класс, 308
RadioButton, класс, 309
RepeatButton, класс, 307
ToggleButton, класс, 308
определение, 305
отмены, 306
по умолчанию, 306
стилизация со встроенной анимацией, 695
шаблон с визуальными состояниями, 712
коллекции
ItemsSource, 343
Properties, 239
SortDescriptions, 441
Triggers, 111
Windows, 239
представления, 440
группировка, 443
навигация, 447
создание нового, 449
сортировка, 440
фильтрация, 446
привязка к, 422
словари, 60
списки, 59
команды, 224
встроенные, 225
выполнение с помощью жестов ввода, 228
реализация в нестандартных элементах управления, 819
элементы управления со встроенными привязками к, 229
комбинирование
материалов, 647
преобразований, 143
преобразований Transform3D, 630
комментарии, добавление в потоковые документы, 380
композиция в XAML, 51
компоновка
нестандартные панели, 125, 825
FanCanvas, 842
OverlapPanel, 837
SimpleCanvas, 830
SimpleStackPanel, 834

взаимодействие между родителями и потомками, 826
обработка переполнения содержимого, 173
масштабирование, 177
отсечение, 173
прокрутка, 175
панели, *см.* панели
позиционирование элементов, 132
выравнивание по горизонтали и по вертикали, 132
выравнивание растяжением, 133
выравнивание содержимого, 133
направление потока, 134
преобразования
MatrixTransform, 142
RotateTransform, 137
ScaleTransform, 139
SkewTransform, 141
Transform3D, 620
TranslateTransform, 142
и отсечение, 175
комбинирование, 143
поддержка, 145
применение, 135
создание панели, как в Visual Studio, 182
управление размером элементов, 126
видимость, 131
высота и ширина, 126
нежелательность явного задания, 127
поля и отступы, 128
конвертеры значений, 434
Binding.DoNothing, 439
ValueMinMaxToIsLargeArcConverter, 504
временная отмена привязки к данным, 439
настройка отображения данных, 438
согласование несовместимых типов данных, 435
конвертеры типов, 52
BrushConverter, 54
FontSizeConverter, 54
GridLengthConverter, 164
ImageSourceConverter, 356
LengthConverter, 130
ThicknessConverter, 130
TransformConverter, 143
в процедурном коде, 164
отключение преобразования типов, 74
поиск, 54
контейнеры
Expander, класс, 318
Frame, класс, 314
GroupBox, класс, 316
Label, класс, 311
ToolTip, 312
навигационные, 249
контекст использования, 427
контекст объявления, 427
кэширование композиции
BitmapCache, класс, 600
BitmapCacheBrush, класс, 601
поддержка со стороны Viewport2DVisual3D, 662

Л

линейная интерполяция, 680
Лобо Лестер, 45
логические деревья, 100
логические ресурсы, 402
взаимодействие с системными ресурсами, 411
консолидация кистей, 404
необобществляемые, 409
определение и применение в процедурном коде, 410
поиск ресурса, 406
прямой доступ, 411
статические и динамические, 406
локализация двоичных ресурсов, 400
подготовка проекта для нескольких культур, 401
пометка пользовательского интерфейса идентификаторами локализации, 401
создание сателлитной сборки с помощью LocBaml, 402
локальные значения, очистка, 115

М

манипулирования события, 216
добавление инерции, 219
маршрутизируемые события, 193
RoutedEventArgs, класс, 196
включение в пользовательский элемент управления, 804
консолидация обработчиков, 201

на примере диалогового окна About, 196
прерывание, 199
присоединенные события, 200
реализация, 194
стратегии маршрутизации, 195
масштабирование, 177
ScaleTransform3D, класс, 623
вдоль неглавной оси, 627
растрового изображения по ближайшей соседней точке, 357
материалы, 639
AmbientMaterial, 644
DiffuseMaterial, 640
EmissiveMaterial, 645
комбинирование, 647
меню
ContextMenu, элемент управления, 347
Menu, элемент управления, 345
MenuItem, класс, 346
мертвая зона (камеры), 611
мнемонические клавиши, 763
многодетные элементы управления
ComboBox, элемент управления, 326
ComboBoxItem, объекты, 331
IsEditable, свойство, 329
IsReadOnly, свойство, 329
SelectionChanged, событие, 331
события, 326
ContextMenu, 347
DataGrid, элемент управления
CanUserAddRows, свойство, 344
CanUserDeleteRows, свойство, 344
ClipboardCopyMode, свойство, 342
EnableColumnVirtualization, свойство, 342
EnableRowVirtualization, свойство, 342
FrozenColumnCount, свойство, 343
RowDetailsVisibilityMode, свойство, 342
SelectionMode, свойство, 341
SelectionUnit, свойство, 341
взаимодействие с буфером обмена, 341
виртуализация, 342
выбор строк и столбцов, 341
дополнительная информация для строк, 342
замораживание столбцов, 343
пример, 337
редактирование данных, 343
типы столбцов, 339
GridView, элемент управления, 335
ItemsControl, класс
AlternationCount, свойство, 321
AlternationIndex, свойство, 321
DisplayMemberPath, свойство, 321
HasItems, свойство, 320
IsGrouping, свойство, 320
IsTextSearchCaseSensitive, свойство, 331
IsTextSearchEnabled, свойство, 331
Items, свойство, 319
ItemsPanel, свойство, 321
ItemsSource, свойство, 320
управление поведением прокрутки, 325
ListBox, элемент управления
SelectionMode, свойство, 332
идентификаторы автоматизации, 335
поддержка множественного выбора, 333
помещение объектов PlayingCard в, 816
пример, 332
прокрутка, 334
расположение объектов по горизонтали, 324
сортировка объектов, 334
ListView, 335
Menu, 345
Selector, класс, 325
StatusBar, 354
TabControl, 336
ToolBar, 351
TreeView, 349
управление поведением прокрутки, 325
многодокументный интерфейс (MDI), 240
многопоточные приложения, 241
мультисенсорный ввод, поддержка, 38

Н

навигация
в XBAP-приложениях, 268
в представлениях, 447
клавиатурная, поддержка в элементах управления Win32, 760
наложения, для элементов на панели задач, 288
наследование
значений свойств, 111
иерархия классов, 98
стилей, 475
независимость от разрешающей способности, 32
независимые от устройства пикселы, 130
немодальные диалоговые окна, 233
неявные пространства имен .NET, 49
неявные стили, создание, 479
нормали к поверхности, 652

О

обертывающие события, 194
обложки, 472, 517
вредоносные, предотвращение, 523
отсутствие стилей, отладка, 520
примеры, 518, 522
процедурный код, 521
обработка ошибок, 464
обработчики событий, 76
объектные элементы, 47
атрибуты, 47
именование, 65
обработка дочерних элементов, 63
объявление, 47
позиционирование, 132
выравнивание по горизонтали и по вертикали, 132
выравнивание растяжением, 133
выравнивание содержимого, 133
направление потока, 134
преобразование типов, 61
преобразования
MatrixTransform, 142
RotateTransform, 137
ScaleTransform, 139
SkewTransform, 141
Transform3D, 620
TranslateTransform, 142
и отсечение, 175
комбинирование, 143
поддержка, 145
применение, 135
свойство содержимого, 58
словари, 60
списки, 59
управление размером, 126
видимость, 131
высота и ширина, 126
нежелательность явного задания, 127
поля и отступы, 128
объекты
Object, класс, 98
визуальные деревья, 102
запись в, 86
логические деревья, 100
привязка к, 420
создание с помощью
конструктора с аргументами, 75
фабричных методов, 76
однодетные элементы управления
ContentControl, класс, 304
и произвольные объекты, 305
кнопки
Button, класс, 107, 306
ButtonBase, класс, 305
CheckBox, класс, 308
RadioButton, класс, 309
RepeatButton, класс, 307
ToggleButton, класс, 308
определение, 305
отмены, 306
по умолчанию, 306
стилизация со встроенной анимацией, 695
шаблон с визуальными состояниями, 712
контейнеры
Expander, класс, 318
Frame, класс, 314
GroupBox, класс, 316
Label, класс, 311
ToolTip, 312
навигационные, 249
определение, 304
однопоточное подразделение (STA), 235
отладчик (Visual C++), 767
отрезки, 374
отсечение, 173, 175

очистка
локальных значений, 115
привязок, 416

П

панели, 146
Canvas, 169, 172
DockPanel, панель
взаимодействие со свойствами компоновки дочерних элементов, 157
моделирование с помощью Grid, 170
примеры, 154
свойства, 154
Grid, панель, 158
ShowGridLines, свойство, 162
взаимодействие со свойствами компоновки дочерних элементов, 170
задание общего размера для строк и столбцов, 166
задание размеров строк и столбцов, 162–164
интерактивное задание размера с помощью GridSplitter, 165
моделирование Canvas, 169
моделирование DockPanel, 170
моделирование StackPanel, 170
свойства ячеек, 161
сравнение с другими панелями, 169
SelectiveScrollingGrid, 172
StackPanel, панель, 150
взаимодействие со свойствами компоновки дочерних элементов, 151
в сочетании с Menu, 347
задание свойств шрифтов, 117
моделирование с помощью Grid, 170
TabPanel, 171
ToolBarOverflowPanel, 171
ToolBarPanel, 171
ToolBarTray, 171
UniformGrid, 172
WrapPanel, панель, 152
примеры, 153
примеры взаимодействие со свойствами компоновки дочерних элементов, 154
размещение справа налево, 154
свойства, 152
нестандартные панели, 125, 825
FanCanvas, 842
OverlapPanel, 837
SimpleCanvas, 830
SimpleStackPanel, 834
взаимодействие между родителями и потомками, 826
переполнение содержимого, *см.* переполнение содержимого, обработка
создание панели, как в Visual Studio, 182
панель задач, настройка, 287
добавление кнопок управления, 290
индикатор выполнения для элемента, 287
наложения для элементов, 288
содержимое эскиза, 289
перекрывающееся содержимое, 749
переполнение содержимого, обработка, 173
масштабирование, 177
отсечение, 173
прокрутка, 175
переходные функции, 38
BackEase, 708
BounceEase, 708
CircleEase, 708
EasingMode, свойство, 706
ElasticEase, 709
ExponentialEase, 709
SineEase, 709
написание, 710
степенные, 707
переходы (анимация), 716
перечисления
BaseValueSource, 115
BindingMode, 459
DayOfWeek, 388
DispatcherPriority, 242
GeometryCombineMode, 545
GradientSpreadMethod, 578
JumpItemRejectionReason, 286
Key, 202
MouseButtonState, 206
PixelFormats, 358
RoutingStrategy, 195
ShutdownMode, 239
Stretch, 178

StretchDirection, 179
TileMode, 587
UpdateSourceTrigger, 461
Visibility, 131
Петцольд Чарльз, 45
пикселы
границы, 39
независимые от устройства, 130
построители текстур, 596
поворот
RotateTransform3D, класс, 627
с инерцией, 219
с помощью жестов ввода, 217
позиционирование элементов, 132
выравнивание по горизонтали и по вертикали, 132
выравнивание растяжением, 133
выравнивание содержимого, 133
направление потока, 134
пользовательские интерфейсы
пользовательских элементов управления, 796
пометка идентификатором локализации, 401
элемента PlayingCard, 813
пользовательские элементы управления
защита от непреднамеренного использования, 801
маршрутизируемые события, 804
поведение, 794, 799
пользовательский интерфейс, 796
свойства зависимости, 802
сравнение с нестандартными, 795
порядок обхода, 649
поставщики данных, 451
ObjectDataProvider, класс, 456
XmlDataProvider, класс, 452
поток данных, настройка, 459
похищение свойства зависимости, 499
правила проверки, 461
правило правой руки, 649, 654
правописания проверка, 363
правосторонняя система координат, 609
представления
группировка, 443
навигация, 447
настройка, 440
создание, 448, 449
сортировка, 440
фильтрация, 446
преобразование типов, отключение, 74
преобразования
MatrixTransform, 142
RotateTransform, 137
ScaleTransform, 139
SkewTransform, 141
Transform3D, 620
TranslateTransform, 142
и отсечение, 175
комбинирование, 143
поддержка, 145
применение, 135
преобразования систем координат, 660, 665
Visual.TransformToAncestor, метод, 666
Visual3D.TransformToAncestor, метод, 670
Visual3D.TransformToDescendant, метод, 670
объяснение, 666
прерывание
загрузки страницы, 255
маршрутизации события, 199
привязка к данным, 36
Binding, объект, 414
ElementName, свойство, 417
IsAsync, свойство, 457
RelativeSource, свойство, 418
StringFormat, свойство, 428
TargetNullValue, свойство, 418
UpdateSourceExceptionFilter, свойство, 465
UpdateSourceTrigger, свойство, 460
ValidationRule свойство, 462
в XAML, 417
в процедурном коде, 414
обобществление источника с помощью DataContext, 426
правила проверки, 461
привязка к UIElement, 422
привязка к коллекциям, 422
привязка ко всему объекту, 420
привязка к свойствам .NET, 419
удаление, 416
CompositeCollection, класс, 466
Language Integrated Query (LINQ), 452
MultiBinding, класс, 467
PriorityBinding, класс, 468
анимация и, 701

асинхронная, 457
временная отмена, 439
клиент Twitter на чистом XAML, 469
к методам, 458
настройка потока данных, 459
настройка представлений, 440
группировка, 443
навигация, 447
создание, 449
сортировка, 440
фильтрация, 446
определение, 414
отладка, 438
поставщики данных, 451
ObjectDataProvider, класс, 456
XmlDataProvider, класс, 452
управление визуализацией, 428
конвертеры значений, 440
форматирование строк, 428
шаблоны данных, 431
приложения
Windows-приложения, стандартные, 231
Application, класс, 236
ClickOnce, 247
Window, класс, 232
в одном экземпляре, 240
заставки, 242
многопоточные, 241
нестандартные диалоговые окна, 244
получение аргументов командной строки, 238
состояние приложения, 246
стандартные диалоговые окна, 243
установщик Windows, 247
XBAP-приложения, *см.* XAML Browser Application (XBAPs)
автономные XAML-страницы, 271
в стиле гаджетов, 261
встраивание элементов управления Win32 в WPF-приложения, 750
клавиатурная навигация, 760
элемент управления Webcam, 750
встраивание элементов управления Windows Forms в WPF-приложения, 772
PropertyGrid, 773
встраивание элементов управления WPF в Win32-приложения
HwndSource, класс, 765
компоновка, 768
встраивание элементов управления WPF в приложения Windows Forms
ElementHost, класс, 777
открытие модального диалогового окна, 781
преобразование одного представления в другое, 780
связывание списка переходов, 274
с многодокументным интерфейсом (MDI), 240
с навигацией, 249
Navigate, метод, 252
возврат данных от страницы, 259
гиперссылки, 253
журнал, 254
навигационные контейнеры, 249
передача данных странице, 258
события навигации, 256
элементы Page, 249
с частичным доверием, 37
присоединенные свойства
как механизм расширяемости, 120
на примере диалогового окна About, 117
определение, 117
поставщики, 119
присоединенные события, 200
проверка попадания
в Visual, 559
в случае перекрывающихся объектов, 564
методы обратного вызова, 566
при наличии нескольких объектов, 560
простая, 560
в трехмерном пространстве, 662
при вводе, 559
InputHitTest, метод, 574
прозрачные области и события мыши, 205
прозрачные цвета, 584
производительность
визуализации
BitmapCache, класс, 600

BitmapCacheBrush, класс, 601
RenderTargetBitmap, класс, 598
кэширование композиции
BitmapCache, класс, 600
BitmapCacheBrush, класс, 601
поддержка со стороны Viewport2DVisual3D, 662
прокрутка
управление в многодетных элементах управления, 175
элемента ListBox, 334
пространства имен, 48
.NET, 48
неявные, 49
XAML, 50
XML, 48, 51
отображение, 48
процедурный код
анимация с помощью таймера, 676
внутри XAML, 71
встраивание PropertyGrid, 773
доступ к двоичным ресурсам из, 400
использование объекта Binding, 414
классы анимации
AutoReverse, свойство, 686
BeginTime, свойство, 685
By, свойство, 684
DoubleAnimation, класс, 680
Duration, свойство, 682
EasingFunction, свойство, 689
FillBehavior, свойство, 690
From, свойство, 683
IsAdditive, свойство, 689
IsCumulative, свойство, 690
RepeatBehavior, свойство, 687
SpeedRatio, свойство, 686
To, свойство, 683
линейная интерполяция, 680
универсальные типы, 679
управление продолжительностью, 682
конвертеры типов, 52, 164
BrushConverter, 54
FontSizeConverter, 54
GridLengthConverter, 164
ImageSourceConverter, 356
LengthConverter, 130
ThicknessConverter, 130
TransformConverter, 143
обложки, 521
определение и применение ресурсов, 410
покадровая анимация, 676
расширения разметки, 57
сравнение с XAML, 46

P

равномерное масштабирование, 624
разбор XAML во время выполнения, 63
развертывание
ClickOnce, 247
XBAP-приложения, 269
усовершенствования в WPF 3.5, 37
усовершенствования в WPF 4, 39
установщик Windows, 248
распознавание речи
описание грамматики на языке SRGS, 741
описание грамматики с помощью класса GrammarBuilder, 742
преобразование произнесенных слов в текст, 738
расширения разметки, 55
в процедурном коде, 57
параметры, 55
расширяемость XAML, 61
ресурсы, 393
без ключей, 479
двоичные, *см.* двоичные ресурсы
доступ, 395–400
локализация, 400–402
определение, 393
логические ресурсы, 402
взаимодействие с системными ресурсами, 411
консолидация кистей, 404
необобществляемые, 409
определение и применение в процедурном коде, 410
поиск ресурса, 406
прямой доступ, 411
статические и динамические, 406
рисунки
Drawing, класс, 534
DrawingBrush, класс, 535
DrawingContext, класс
методы, 554
DrawingImage, класс, 535
DrawingVisual, класс, 535
GeometryDrawing, класс, 534

GlyphRunDrawing, класс, 535
ImageDrawing, класс, 534
Pen, класс, 548
VideoDrawing, класс, 534
пример изображения, 550
усовершенствования в WPF 3.5, 36, 37

C

сателлитные сборки, создание с помощью LocBaml, 402
свойства
.NET, привязка к, 419
Properties, коллекция, 239
атрибуты, 47
зависимости, *см.* свойства зависимости
обертывающие, 108
порядок обработки, 48
путь к, 322
триггеры, 109, 481, 697
элементы, 51
свойства зависимости, 106, 477
включение в пользовательский элемент управления, 802
наследование значений свойств, 111
поддержка нескольких поставщиков, 113
вычисление, 116
определение базового значения, 114
приведение, 116
применение анимации, 116
проверка, 116
похищение, 499
присоединенные свойства
как механизм расширяемости, 120
на примере диалогового окна About, 117
определение, 117
поставщики, 119
реализация, 107
триггеры свойств, 109
уведомление об изменении, 109
свойство содержимого, 58, 63
синтез речи, 734
GetInstalledVoices, метод, 735
PromptBuilder, класс, 735
SelectVoice, метод, 735
SelectVoiceByHints, метод, 735
SetOutputToWaveFile, метод, 735
SpeakAsync, метод, 734
Speech Synthesis Markup Language (SSML), 735
SpeechSynthesizer, класс, 735
системы координат, 607
словари, 60
события
Click, 306
DateValidationError, 389
DropDownOpened, 326
JumpItemsRejected, 285
JumpItemsRemovedByUser, 286
Rendering, 676
SelectedDateChanged, 389
SelectionChanged, 326, 331
атрибуты, 47
клавиатуры, 202
маршрутизируемые
RoutedEventArgs, класс, 196
включение в пользовательский элемент управления, 804
на примере диалогового окна About, 197
определение, 193
прерывание, 199
присоединенные, 200
реализация, 194
стратегии маршрутизации, 195
мультисенсорные
манипулирование, 216
простые касания, 211, 212
мыши, 205
MouseButtonEventArgs, 206
MouseEventArgs, 206
MouseWheelEventArgs, 206
захват, 208
перетаскивание, 207
прозрачные и null-области, 205
навигации, 256
обертывающие, 194
порядок обработки, 48
стилуса, 209
создание объектов
с помощью конструктора с аргументами, 75
с помощью фабричных методов, 76
сортировка, 334, 440
состояния
визуальные
учет с помощью VSM, 505

учет с помощью триггеров, 500
сохранение и восстановление, 246
элементов управления, 508, 819
спецэффекты, 36
списки, 59
Listbox, элемент управления
SelectionMode, свойство, 332
идентификаторы автоматизации, 335
поддержка множественного выбора, 333
помещение объектов PlayingCard в, 816
пример, 332
прокрутка, 334
расположение объектов по горизонтали, 324
сортировка объектов, 334
ListView, элемент управления, 335
переходов, *см.* списки переходов
списки переходов, 38, 273
JumpPath, 282
добавление, 283
недавние и часто посещаемые, 284
реакция на отказ от добавления или на удаление, 285
JumpTask, 275
настройка поведения, 277
пример, 275
и отладчик Visual Studio, 277
связывание с приложением, 274
стандартные диалоговые окна, 243
стили, 472, 473
без ключей, 479
именованные, 479
комбинирование с шаблонами, 514
консолидация свойств в одном месте, 474
наследование, 475
неявные, создание, 479
обобществление, 475
ограничения, 477
отсутствующие, отладка, 520
поведение Setter, 477
по умолчанию, 115
темы, 115, 525
типизированные, 479
триггеры, 481
выражение логики с помощью, 486
данных, 485
конфликтующие, 487
свойств, 481
учет визуальных состояний с помощью, 500
столбцы (Grid)
автоматически генерируемые, 339
задание общего размера, 166
задание размеров
абсолютное, 162
автоматический выбор, 162
в процентах, 164
интерактивное с помощью GridSplitter, 165
пропорциональное, 163
структуры GridLength, 164
замораживание, 343
типы, 339
страницы
XAML автономные, 271
возврат данных от, 259
обновление, 255
передача данных, 258
прерывание загрузки, 255
элементы Page, 249
строки
представление геометрических объектов в виде, 546
форматирование, 428
считыватели (XAML), 77
NodeType, свойство, 82
XamlServices, класс, 89
поток узлов XAML, 84
пример XAML-разметки, 83
совместимость разметки, 86
циклы обработки узлов, 81

Т

текст
InkCanvas, класс, 365
PasswordBox, элемент управления, 364
RichTextBox, элемент управления, 364
TextBlock, элемент управления, 360
TextBox, элемент управления, 362
TextOptions, класс, 359
визуализация, 39, 358
преобразование в слышимую речь, *см.* синтез речи

преобразование произнесенных слов в, 738
текстур координаты, 654
темы, 472, 524
словари, 526
стили, 115, 525
типовые словари, 526, 814
типовые словари, 526, 814
типы данных
в XAML2009, 75
согласование несовместимых, 435
трехмерная графика
Model3D, класс, 631
GeometryModel3D, 639
Light, класс, 631
Model3DGroup, класс, 654
ModelVisual3D, класс, 632, 656
Transform3D, 620
MatrixTransform3D, 621, 631
RotateTransform3D, 621, 627
ScaleTransform3D, 621, 623
Transform3DGroup, 621
TranslateTransform3D, 620, 623
комбинирование, 630
UIElement3D, класс, 658
Viewport2DVisual3D, класс, 660
Viewport3D, класс, 663
Visual3D, 656
аппаратное ускорение, 32
и GDI, 34
границы пикселов, 39
источники света, 607
камеры
LookDirection, свойство, 611
MatrixCamera, класс, 620
Position, свойство, 608
Transform, свойство, 615
UpDirection, свойство, 614
коллизии в Z-буфере, 611
мертвая зона, 611
ортографическая и перспективная, 617
системы координат, 607
координаты текстур, 654
материалы, 639
AmbientMaterial, 644
DiffuseMaterial, 640
EmissiveMaterial, 645
комбинирование, 647
независимость от разрешающей способности, 32
объяснение, 602
преобразования систем координат, 660, 665
Visual.TransformToAncestor, метод, 666
Visual3D.TransformToAncestor, метод, 670
Visual3D.TransformToDescendant, метод, 670
объяснение, 666
пример домика, 603
проверка попадания в трехмерном пространстве, 662
системы координат, 608
усовершенствования в WPF 3.5, 36
триггеры, 481
в шаблонах элементов управления, 490
выражение логики с помощью, 486
логического И, 487
логического ИЛИ, 486
данных, 111, 485
конфликтующие, 487
свойств, 109, 481
событий, 111
учет визуальных состояний с помощью, 500
туннелирование, 195

У

универсальные классы, поддержка в XAML2009, 73
управляемый код, сочетание с неуправляемым, 754
установщик Windows, 248

Ф

фигуры (Shape), 566
Ellipse, класс, 569
Line, класс, 570
Path, класс, 572
Polygon, класс, 572
Polyline, класс, 571
Rectangle, класс, 568
изображение из объектов Shape, 573
как работают, 569
чрезмерное увлечение, 567
фильтрация, 446
форматирование строк, 428

Ц

цветовые профили, 577

Ч

части элемента управления, 505, 818

Ш

шаблоны
  HierarchicalDataTemplate, 455
  данных, 431
  для тем Windows, 530
  определение, 472, 488
  селекторы, 434
  темы, 525
  элементов управления, *см.* шаблоны элементов управления
шаблоны данных
  HierarchicalDataTemplate, 455
  селекторы шаблонов, 434
шаблоны элементов управления, 488
  TargetType, свойство, 492
  визуальные состояния
    учет с помощью VSM, 505
    учет с помощью триггеров, 500
  именованные элементы, 492
  комбинирование со стилями, 514
  ограничение типа целевого элемента, 492
  повторное использование, 496
  простой пример, 489
  редактирование, 516
  с триггерами, 490
  учет свойств шаблона-родителя
    Content, свойство, 493
    похищение существующих свойств, 499
    прочие свойства, 496

Э

Эйлера углы, 628
элемент XML, 52
элементы управления
  ActiveX, 788
  Calendar, 386
  ComboBox, 326
    ComboBoxItem, объекты, 331
    IsEditable, свойство, 329
    IsReadOnly, свойство, 329
    SelectionChanged, событие, 331
    события, 326
  ContextMenu, 347
  DataGrid
    CanUserAddRows, свойство, 344
    CanUserDeleteRows, свойство, 344
    ClipboardCopyMode, свойство, 342
    EnableColumnVirtualization, свойство, 342
    EnableRowVirtualization, свойство, 342
    FrozenColumnCount, свойство, 343
    RowDetailsVisibilityMode, свойство, 342
    SelectionMode, свойство, 341
    SelectionUnit, свойство, 341
    взаимодействие с буфером обмена, 341
    виртуализация, 342
    выбор строк и столбцов, 341
    дополнительная информация для строк, 342
    замораживание столбцов, 343
    пример, 337
    редактирование данных, 343
    типы столбцов, 339
  DatePicker, 388
  GridView, 335
  InkCanvas, 365
  ItemsControl, класс
    AlternationCount, свойство, 321
    AlternationIndex, свойство, 321
    DisplayMemberPath, свойство, 321
    HasItems, свойство, 320
    IsGrouping, свойство, 320
    IsTextSearchCaseSensitive, свойство, 331
    IsTextSearchEnabled, свойство, 331
    Items, свойство, 319
    ItemsPanel, свойство, 321
    ItemsSource, свойство, 320
    управление поведением прокрутки, 325
  ListBox
    SelectionMode, свойство, 332
    идентификаторы автоматизации, 335

поддержка множественного выбора, 333
помещение объектов PlayingCard в, 816
пример, 332
прокрутка, 334
расположение объектов по горизонтали, 324
сортировка объектов, 334
ListView, 335
Menu, 345
PasswordBox, 364
ProgressBar, 384
RichTextBox, 364
ScrollViewer, 175
Selector, класс, 325
Slider, 385
StatusBar, 354
TabControl, 336
TextBlock, 360
TextBox, 362
ToolBar, 351
TreeView, 349
кнопки
Button, класс, 107, 306
ButtonBase, класс, 305
CheckBox, класс, 308
RadioButton, класс, 309
RepeatButton, класс, 307
ToggleButton, класс, 308
определение, 305
отмены, 306
по умолчанию, 306
стилизация со встроенной анимацией, 695
шаблон с визуальными состояниями, 712
контейнеры
Expander, класс, 318
Frame, класс, 314
GroupBox, класс, 316
Label, класс, 311
ToolTip, 312
навигационные, 249
нестандартные
автоматизация ГИП, 823
застраничный файл, 807
команды, 819
объяснение, 794
пользовательский интерфейс, 813
ресурсы, 808
создание, 806
состояния, 819
сравнение с пользовательскими, 795
типовые ресурсы, 814
части, 818
объяснение, 303
пользовательские элементы управления
защита от непреднамеренного использования, 801
маршрутизируемые события, 804
поведение, 794, 799
пользовательский интерфейс, 796
свойства зависимости, 802
создание, 796
сравнение с нестандартными, 795
со встроенными привязками к командам, 229
состояния, 508, 819
части элемента управления, 505, 818
эффекты, 594

Я

яркость источника света, 634

Марк САММЕРФИЛД

Qt. Профессиональное программирование

Разработка кроссплатформенных приложений на C++

560 стр., книга в продаже



Книга Марка Саммерфилда открывает путь к овладению разнообразными паттернами и приемами создания приложений с использованием библиотеки разработки кроссплатформенных приложений Qt.

Основной акцент сделан на создании моделей, графических представлений и гибридных приложений «рабочий стол + Интернет», на многопоточной обработке данных и приложениях, содержащих мультимедийные объекты и форматированный текст. Представлено подробное введение в подсистемы анимации и конечных автоматов, включенные в версию Qt 4.6.

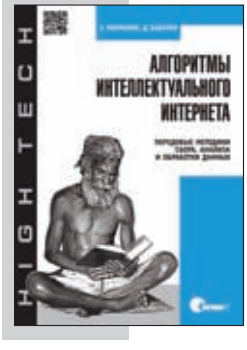
В книге приведены примеры кода, протестированные на платформах Windows, Mac OS X и Linux с использованием Qt 4.6 (а многие работают также с версией Qt 4.5) и написанные с ориентацией на будущие версии Qt.

Харалампос МАРМАНИС и Дмитрий БАБЕНКО

Алгоритмы интеллектуального Интернета

Передовые методики сбора, анализа и обработки данных

480 стр., книга в продаже



Если вас интересуют вопросы искусственного интеллекта, если у вас есть собственный блог, новостной портал, wiki-справочник или сайт с онлайн-игрой, если вы планируете создать веб-приложение, которое должно учитывать введенные каждым пользователем данные, его поведение в системе на протяжении некоторого периода времени, а также другую потенциально полезную информацию – эта книга будет вам безусловно необходима.

В издании рассматриваются проблемы, с которыми сталкиваются все веб-приложения: поиск, кластеризация, релевантность и т. д. Примеры в этой книге даны на языке Java и разработаны таким образом, чтобы с их помощью можно было проиллюстрировать универсальную методику – алгоритм, – которая находит применение в широком диапазоне сценариев.

Издание в первую очередь адресовано программистам и веб-разработчикам, однако множество примеров и новых идей будут полезны и руководителям разного уровня, желающим лучше разобраться в технологиях и их возможностях с точки зрения бизнеса.

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:
главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

Москва:
московский филиал издательства –
ул. Беговая, д. 13 (м. Динамо),
тел. (495) 638-5305

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству

www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!