35

Расширенный WPF

В предыдущей главе вы узнали о некоторой центральной функциональности WPF. В этой главе мы продолжим изучение программирования с использованием WPF. Вы узнаете о некоторых важнейших аспектах создания готовых приложений, таких как привязка данных и привязка команд, а также получите представление об анимации и программировании 3-D.

Главные темы настоящей главы:

- привязка данных;
- команды;
- анимация;
- 3-D;
- интеграция Windows Forms.

Привязка данных

В предыдущей главе вы видели несколько средств привязки данных, когда речь шла о стилизации `ListBox`. Но, конечно же, существует множество других. WPF делает в этом направлении еще один гигантский шаг вперед по сравнению с Windows Forms. В этом разделе мы дадим вам начальное представление о привязке данных с WPF и обсудим следующие темы:

- общий обзор;
- привязка с XAML;
- привязка простого объекта;
- объектный поставщик данных;
- привязка списка;
- привязка к XML.

Общий обзор

В привязке данных WPF целью может быть любое свойство зависимости элемента WPF, и каждое свойство объекта CLR может служить источником. Поскольку элемент WPF реализован как класс .NET, каждый элемент WPF также может служить источником данных. На рис. 35.1 показана связь между источником и целью. Объект `Binding` определяет соединение.

Рис. 35.1. Связь между источником и целью

Привязка поддерживает несколько моделей связи между целью и источником. Привязка может быть *однонаправленной* (one-way), когда исходная информация движется к цели, но если пользователь изменит информацию через пользовательский интерфейс, то источник не будет обновлен. Для обновления источника необходима двунаправленная привязка.

В табл. 35.1 перечислены режимы привязки и их требования.

Таблица 35.1. Режим привязки

Режим привязки	Описание
Однократная (`OneTime`)	Привязка направлена от источника к цели и происходит лишь однажды, при запуске приложения или изменении контекста данных. Здесь вы получаете "снимок" данных.
Однонаправленная (`OneWay`)	Привязка направлена от источника к цели. Это удобно для доступа к данным, позволяющим только чтение, поскольку нет необходимости изменять исходные данные источника через пользовательский интерфейс. Чтобы получить обновления в пользовательский интерфейс, источник должен реализовать интерфейс `INotifyPropertyChanged`.
Двунаправленная (`TwoWay`)	При двунаправленной привязке пользователь может вносить изменения в данные через пользовательский интерфейс. Привязка работает в обоих направлениях — от источника к цели и от цели к источнику. Источник должен реализовывать свойства, доступные по чтению/записи, чтобы изменения могли обновляться по направлению от пользовательского интерфейса к источнику.
Однонаправленная к источнику (`OneWayToSource`)	При однонаправленной привязке к источнику, если целевое свойство меняется, то обновляется исходный объект.

Привязка с XAML

Элемент WPF может не только служить целью для привязки данных, но с тем же успехом может служить и источником. Вы можете привязать некоторое свойство одного компонента WPF к цели из другого элемента WPF.

В следующем примере используется забавная рожица, которую мы создали ранее и которая состоит из фигур WPF; она привязывается к бегунку (slider), чтобы можно было двигать ее в пределах окна. `Slider` — элемент-источник по имени `slider`. Свойство `Value` дает действительное значение позиции бегунка. Целью привязки данных будет внутренний элемент `Canvas`. Внутренний элемент `Canvas` по имени `FunnyFace` содержит все фигуры, необходимые для рисования рожицы. Это полотно (canvas) содержится внутри внешнего элемента `Canvas`, так что ее можно позиционировать внутри внешнего элемента, изменяя присоединенные свойства. Присоединенное

свойство `Canvas.Left` устанавливает расширение маркировки `Binding`, `ElementName` устанавливается в `slider` для ссылки на WPF элемент-бегунок, а `Path` устанавливается в `Value`, чтобы получать значение из свойства `Value`.

```
<Window x:Class="DataBindingSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Data Binding" Height="345" Width="310">
 <StackPanel>
   <Canvas Height="210" Width="280">
     <Canvas Canvas.Top="0"
         Canvas.Left="{Binding Path=Value, ElementName=slider}"
         Name="FunnyFace" Height="210" Width="230">
     <Ellipse Canvas.Left="20" Canvas.Top="50" Width="100" Height="100"
         Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
     <Ellipse Canvas.Left="40" Canvas.Top="65" Width="25" Height="25"
         Stroke="Blue" StrokeThickness="3" Fill="White" />
     <Ellipse Canvas.Left="50" Canvas.Top="75" Width="5" Height="5"
         Fill="Black" />
     <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
         Data="M 32,125 Q 65,122 72,108" />
     <Line X1="94" X2="102" Y1="144" Y2="166" Stroke="Blue" StrokeThickness="4" />
     <Line X1="84" X2="103" Y1="169" Y2="166" Stroke="Blue" StrokeThickness="4" />
     <Line X1="62" X2="52" Y1="146" Y2="168" Stroke="Blue" StrokeThickness="4" />
     <Line X1="38" X2="53" Y1="160" Y2="168" Stroke="Blue" StrokeThickness="4" />
     </Canvas>
   </Canvas>
   <Slider Name="slider" Orientation="Horizontal" Value="10" Maximum="100" />
 </StackPanel>
</Window>
```

Запустив приложение, вы сможете перемещать бегунок, при этом рожица будет перемещаться, что можно видеть на рисунках 35.2 и 35.3.

Вместо определения привязки информации в коде XAML, как было сделано в примере с расширением метаданных `Binding`, вы можете сделать это в отделенном коде. Взглянем еще раз на XAML-версию привязки:

```
<Canvas Canvas.Top="0"
  Canvas.Left="{Binding Path=Value, ElementName=slider}"
  Name="FunnyFace" Height="210" Width="230">
```

При этом вам нужно будет создать новый объект `Binding` и установить свойства `Path` и `Source`. Свойство `Source` должно быть установлено на объект-источник. В данном случае это будет объект WPF `slider`. `Path` устанавливается в экземпляр `PropertyPath`, инициализируемый именем свойства объекта-источника — `Value`.

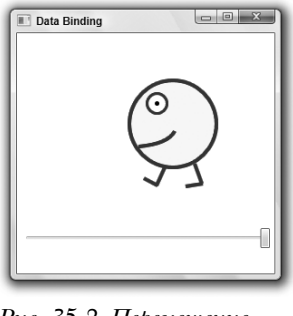

Рис. 35.2. Перемещение рожицы вправо

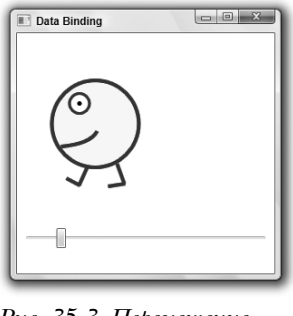

Рис. 35.3. Перемещение рожицы влево

С целевым объектом вы можете вызывать метод SetBinding() для определения привязки. Здесь целью служит объект Canvas по имени FunnyFace. Метод SetBinding() требует двух параметров: первый — это свойство зависимости, а второй — объект привязки. Свойство Canvas.Left должно быть привязано, поэтому свойство привязки типа DependencyProperty может быть доступно через поле Canvas.LeftProperty.

```
Binding binding = new Binding();
binding.Path = new PropertyPath("Value");
binding.Source = slider;
FunnyFace.SetBinding(Canvas.LeftProperty, binding);
```

Для класса Binding вы можете конфигурировать множество опций привязки, которые описаны в табл. 35.2.

Таблица 35.2. Свойства класса Binding

Свойство	Описание
Source	Свойством Source вы определяете объект-источник для привязки данных.
RelativeSource	Свойством RelativeSource вы можете специфицировать источник по отношению к объекту-цели. Это удобно для отображения сообщений об ошибках, когда источник ошибки поступает из того же элемента управления.
ElementName	Если источник — элемент WPF, вы можете специфицировать источник свойством ElementName.
Path	С помощью свойства Path вы можете специфицировать путь к объекту-источнику. Это может быть свойство объекта-источника, но индексаторы и свойства дочерних элементов также поддерживаются.
Xpath	С источником данных XML вы можете определить выражение запроса XPath, чтобы получать данные для привязки.
Mode	Mode задает направление для привязки. Свойство Mode имеет тип BindingMode — перечисление, состоящее из следующих значений: Default, OneTime, OneWay, TwoWay, OneWayToSource. Значение по умолчанию зависит от цели: для TextBox двунаправленная привязка принята по умолчанию; для элемента Label, который доступен только для чтения, стандартной является однонаправленная привязка. OneTime означает, что данные загружаются из источника только при инициализации, а OneWay также выполняет обновления из источника в цели. Привязка TwoWay обеспечивает запись изменений WPF-элемента обратно в источник. OneWayToSource означает, что данные никогда не читаются, а только записываются из цели в источник.
Converter	С помощью свойства Converter вы можете специфицировать класс конвертера, который преобразует данные из пользовательского интерфейса и обратно. Класс конвертера должен реализовывать интерфейс IValueConverter, который определяет методы Convert() и ConvertBack(). Вы можете передавать параметры методам конвертера со свойством ConverterParameter. Конвертер может быть зависимым от культуры; культура может быть установлена в свойстве ConverterCulture.
FallbackValue	Посредством свойства FallbackValue вы можете определять значение по умолчанию, используемое в случае, когда привязка не возвращает значения.
ValidationRules	С помощью свойства ValidationRules вы можете определить коллекцию объектов ValidationRule, проверяемых перед тем, как источник обновится по целевым элементам WPF. Класс ExceptionValidationRule наследуется от класса ValidationRule и проверяется на предмет исключений.

Привязка простого объекта

Для привязки объектов CLR вместе с классами .NET вы должны определять свойства, как показано в следующем примере класса `Book` и его свойств `Title`, `Publisher`, `Isbn` и `Authors`.

```
public class Book
{
  public Book(string title, string publisher, string isbn,
  params string[] authors)
  {
     this.title = title;
     this.publisher = publisher;
     this.isbn = isbn;
     foreach (string author in authors)
     {
        this.authors.Add(author);
     }
  }
  public Book()
     : this("unknown", "unknown", "unknown")
  {
  }
  public string Title { get; set; }
  public string Publisher { get; set; }
  public string Isbn { get; set; }
  public override string ToString()
  {
     return title;
  }
  private readonly List<string> authors = new List<string>();
  public string[] Authors
  {
     get { return authors.ToArray(); }
  }
}
```

В пользовательском интерфейсе определены несколько меток и элементов — текстовых полей для отображения информации о книге. Используя расширения разметки `Binding`, элементы управления `TextBox` привязываются к свойствам класса `Book`. В расширении разметки `Binding` не определяется ничего помимо свойства `Path`, служащего для привязки свойства класса `Book`. Определять источник необходимости нет, поскольку источник определяется присвоением `DataContext`, что видно в приведенном ниже отделенном коде. Режим привязки для элемента `TextBox` выбирается по умолчанию, т.е. является двунаправленным.

```
<Window x:Class="ObjectBindingSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Object Binding Sample" Height="300" Width="340"
>
<Grid Name="bookGrid" Margin="5" >
  <Grid.ColumnDefinitions>
     <ColumnDefinition Width="30*" />
     <ColumnDefinition Width="70*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
     <RowDefinition Height="50" />
     <RowDefinition Height="50" />
     <RowDefinition Height="50" />
```

```
      <RowDefinition Height="50" />
      <RowDefinition Height="50" />
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.Row="0">Title:</Label>
    <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="0"
         Text="{Binding Title}" />
    <Label Grid.Column="0" Grid.Row="1">Publisher:</Label>
    <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="1"
         Text="{Binding Publisher}" />
    <Label Grid.Column="0" Grid.Row="2">ISBN:</Label>
    <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="2"
         Text="{Binding Isbn}" />
    <Button Margin="5" Grid.Column="1" Grid.Row="4" Click="bookButton_Click"
         Name="bookButton">Open Dialog</Button>
  </Grid>
  </Window>
```

В отделенном коде создается новый объект `Book`, и книга присваивается свойству `DataContext` элемента управления `Grid`. Свойство `DataContext` — свойство зависимости, определенное в базовом классе `FrameworkElement`. Присваивание `DataContext` с элементом `Grid` означает, что каждый элемент `Grid` имеет привязку по умолчанию к одному и тому же контексту данных.

```
public partial class Window1 : System.Windows.Window
{
  private Book book1 = new Book();
  public Window1()
  {
     InitializeComponent();
     book1.Title = "Professional C# 2005 with .NET 3.0";
     book1.Publisher = "Wrox Press";
     book1.Isbn = "978-0470124727";
     bookGrid.DataContext = book1;
  }
}
```

После запуска приложения вы можете увидеть привязанные данные, как показано на рис. 35.4.

Для демонстрации двунаправленной привязки (изменения при вводе в элемент WPF отражаются внутри объекта CLR) реализован метод `OnOpenBookDialog()`. Этот метод назначен событию `Click` кнопки `bookButton`, что можно видеть в коде XAML. При щелчке на кнопке всплывает окно сообщения, отображающее текущий заголовок и номер ISBN объекта `book1`. На рис. 35.5 показан вывод окна сообщения после внесения изменений во время выполнения.

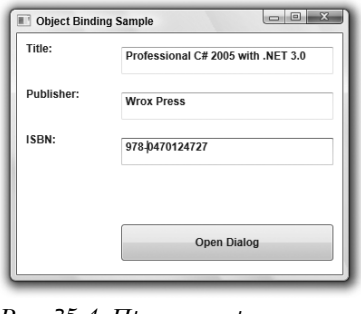

Рис. 35.4. Привязка простого объекта

Рис. 35.5. Вывод окна сообщения

```
void bookButton_Click(object sender, RoutedEventArgs e)
{
  string message = book1.Title;
  string caption = book1.Isbn;
  MessageBox.Show(message, caption);
}
```

Объектный поставщик данных

Вместо определения объекта в отделенном коде вы можете определить его в XAML. Чтобы обеспечить такую возможность, вам следует обратиться к пространству имен посредством объявления пространства имен в корневом элементе XML. XML-атрибут `xmlns:src="clr-namespace:Wrox.ProCsharp.WPF"` присваивает пространству имен .NET `Wrox.ProCSharp.WPF` XML-псевдоним `src`.

Один объект класса `Book` теперь определен в элементе `Book` внутри ресурсов `Window`. Присваивая значения XML-атрибутам `Title` и `Publisher`, вы устанавливаете значения свойств класса `Book`. Установка `x:Key="theBook"` определяет идентификатор ресурса, так что вы можете ссылаться на объект `book`. В элементе `TextBox` теперь определен `Source` с расширением привязки `Binding` для ссылки на ресурс `theBook`.

Расширения XAML разметки можно комбинировать. В следующем примере расширение разметки `StaticResource` используется для ссылки на ресурс — книгу и содержится внутри расширения разметки `Binding`.

```
<Window x:Class="Wrox.ProCSharp.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
  Title="Object Binding Sample" Height="300" Width="340">
<Window.Resources>
  <src:Book x:Key="theBook" Title="Professional C# 2008" Publisher="Wrox Press" />
</Window.Resources>
<!-- ... -->
  <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="0"
       Text="{Binding Source={StaticResource theBook}, Path=Title}" />
<!-- ... -->
```

Если пространство имен .NET, на которое нужно сослаться, находится в другой сборке, вы также должны добавить имя сборки в объявление XML:

```
xmlns:system="clr-namespace:System;assembly=mscorlib".
```

Вместо определения экземпляра объекта непосредственно внутри кода XAML, вы можете также определить поставщика объектных данных, ссылающегося на класс для вызова метода. Чтобы использовать `ObjectDataProvider`, лучше всего создать класс-фабрику, который возвращает объект для отображения, как показано на примере класса `BookFactory`:

```
public class BookFactory
{
 private List<Book> books = new List<Book>();
 public BookFactory()
 {
   books.Add(new Book("Professional C# with 2008", "Wrox Press", "978-0470191378"));
 }
 public Book GetTheBook()
 {
   return books[0];
 }
}
```

Элемент ObjectDataProvider может быть определен в разделе ресурсов. XML-атрибут ObjectType определяет имя класса; в MethodName вы специфицируете имя метода, который вызывается для получения объекта — книги:

```
<Window.Resources>
<ObjectDataProvider ObjectType="src:BookFactory" MethodName="GetTheBook"
     x:Key="theBook">
</ObjectDataProvider>
</Window.Resources>
```

Свойства, которые вы можете специфицировать в классе ObjectDataProvider, перечислены в табл. 35.3.

Таблица 35.3. Свойства класса ObjectDataProvider

Свойство	Описание
ObjectType	Свойство ObjectType определяет тип, экземпляр которого следует создать.
ConstructorParameters	Используя коллекцию ConstructorParameters, вы можете добавлять параметры в класс для создания экземпляра.
MethodName	Свойство MethodName определяет имя метода, вызываемого поставщиком объектных данных.
MethodParameters	С помощью свойства MethodParameters вы можете присвоить параметры метода, определенного свойством MethodName.
ObjectInstance	С помощью свойства ObjectInstance вы можете получать и устанавливать объект, используемый классом ObjectDataProvider. Например, вы можете программно присвоить существующий объект вместо определения ObjectType, так что экземпляр объекта будет создан посредством ObjectDataProvider.
Data	С помощью свойства Data вы можете обращаться к лежащему в основе объекту, используемому для привязки данных. Если определен MethodName, то через свойство Data вы можете обратиться к объекту, возвращенному этим методом.

Привязка списка

Привязка списка выполняется чаще, чем привязка простых объектов. Но привязка списка очень похожа на привязку простого объекта. Вы можете назначить полный список DataContext в отделенном коде или же использовать ObjectDataProvider, обращающийся к фабрике объектов и возвращающий список. С помощью элементов, поддерживающих привязку к списку (например, ListBox), привязывается сразу весь список. Посредством элементов, которые поддерживают привязку только одного объекта (например, TextBox), привязывается только один текущий элемент из списка.

Усовершенствуем класс BookFactory, чтобы он мог возвращать сразу список объектов — книг.

```
public class BookFactory
{
 private List<Book> books = new List<Book>();
 public BookFactory()
 {
  books.Add(new Book("Professional C# 2008",
       "Wrox Press", "978-0470191378", "Christian Nagel", "Bill Evjen",
       "Jay Glynn", "Karli Watson", "Morgan Skinner"));
```

```
  books.Add(new Book("Professional C# 2005 with .NET 3.0",
       "Wrox Press", "978-0-7645-7534-1", "Christian Nagel", "Bill Evjen",
       "Jay Glynn", "Karli Watson", "Morgan Skinner", "Allen Jones"));
  books.Add(new Book("Professional C# 2005",
       "Wrox Press", " 978-0-7645-7534-1", "Christian Nagel", "Bill Evjen",
       "Jay Glynn", "Karli Watson", "Morgan Skinner", "Allen Jones"));
  books.Add(new Book("Beginning Visual C#",
       "Wrox Press", "978-0-7645-4382-1", "Karli Watson", "David Espinosa",
       "Zach Greenvoss", "Jacob Hammer Pedersen", "Christian Nagel",
       "John D. Reid", "Matthew Reynolds", "Morgan Skinner", "Eric White"));
  books.Add(new Book("ASP.NET Professional Secrets",
       "Wiley", "978-0-7645-2628-2", "Bill Evjen", "Thiru Thangarathinam",
       "Bill Hatfield", "Doug Seven", "S. Srinivasa Sivakumar",
       "Dave Wanta", "Jason T. Roff"));
  books.Add(new Book("Design and Analysis of Distributed Algorithms",
       "Wiley", "978-0-471-71997-7", "Nicolo Santoro"));
 }
 public IEnumerable<Book> GetBooks()
 {
  return books;
 }
}
```

В конструкторе отделенного кода WPF для класса `Window1` создается экземпляр `BookFactory` и вызывается метод `GetBooks()` для присвоения массива `Book` с `DataContext` экземпляра `Winbdow1`:

```
public partial class Window1 : System.Windows.Window
{
 private BookFactory factory = new BookFactory();
 public Window1()
 {
  InitializeComponent();
  this.DataContext = factory.GetBooks();
 }
}
```

В XAML вам нужен просто элемент управления, поддерживающий списки, такой как `ListBox`, и привязка свойства `ItemsSource`, как показано ниже:

```
<Window x:Class="Wrox.ProCSharp.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="List Binding Sample" Height="300" Width="518"
  >
 <DockPanel>
  <Grid >
     <Grid.ColumnDefinitions>
       <ColumnDefinition />
     </Grid.ColumnDefinitions>
     <Grid.RowDefinitions>
       <RowDefinition />
       <RowDefinition />
       <RowDefinition />
       <RowDefinition />
     </Grid.RowDefinitions>
     <ListBox HorizontalAlignment="Left" Margin="5" Grid.RowSpan="4"
       Grid.Row="0" Grid.Column="0" Name="booksList"
       ItemsSource="{Binding}" />
  </Grid>
 </DockPanel>
</Window>
```

Поскольку Window имеет массив Book, назначенный DataContext, и ListBox помещен внутрь Window, этот ListBox показывает все книги в шаблоне по умолчанию, как показано на рис. 35.6.

Для более гибкой компоновки ListBox вам придется определить шаблон, как это было описано в предыдущей главе, когда речь шла о стилизации ListBox. Шаблон ItemTemplate, содержащийся в стиле listBoxStyle, определяет DataTemplate с элементом Label. Содержимое метки привязано к Title. Шаблон элемента повторяется для каждого элемента в списке.

Элемент ListBox имеет присвоенное свойство Style. Как и ранее, ItemsSource установлен в привязку по умолчанию. На рис. 35.7 показан вывод приложения с новым стилем ListBox.

```
<Window.Resources>
<Style x:Key="listBoxStyle" TargetType="{x:Type ListBox}" >
  <Setter Property="ItemTemplate">
     <Setter.Value>
       <DataTemplate>
           <Label Content="{Binding Title}" />
       </DataTemplate>
     </Setter.Value>
  </Setter>
</Style>
</Window.Resources>
<!-- ... -->
<ListBox HorizontalAlignment="Left" Margin="5"
     Style="{StaticResource listBoxStyle}" Grid.RowSpan="4"
     ItemsSource="{Binding}" />
```

Привязка типа "ведущая–детали"

Вместо простого отображения всех элементов внутри списка можно отобразить лишь детальную информацию об одном выбранном его элементе. Чтобы добиться этого, не понадобится много усилий. Вы просто определяете элементы для отображения текущего выбора. В нашем примере приложения определены три элемента Label с расширением разметки Binding, установленным в свойства Book — Title, Publisher и Isbn. В ListBox потребуется внести только одно важное изменение. По умолчанию метки просто привязываются к первому элементу в списке. За счет настройки свойства ListBox как IsSynchronizedWithCurrentItem="True" выбор окна списка устанавливается в текущий элемент. На рис. 35.8 вы можете видеть результат; выбранный элемент отображается в метках раздела деталей.

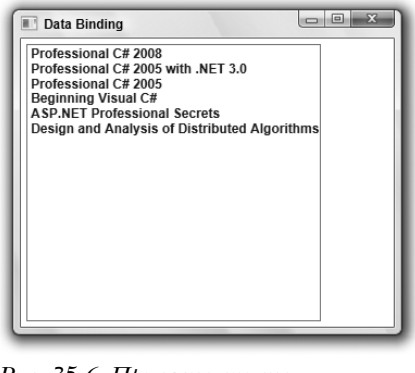

Рис. 35.6. Привязка списка

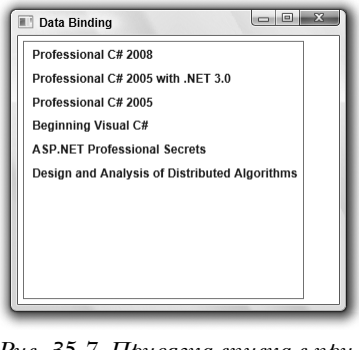

Рис. 35.7. Привязка списка с применением стиля

```
<Window x:Class="Wrox.ProCSharp.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="List Binding Sample" Height="300" Width="518"
  >
 <Window.Resources>
  <Style x:Key="listBoxStyle" TargetType="{x:Type ListBox}" >
     <Setter Property="ItemTemplate">
       <Setter.Value>
           <DataTemplate>
             <Label Content="{Binding Title}" />
           </DataTemplate>
       </Setter.Value>
     </Setter>
  </Style>
  <Style x:Key="labelStyle" TargetType="{x:Type Label}">
     <Setter Property="Width" Value="190" />
     <Setter Property="Height" Value="40" />
     <Setter Property="Margin" Value="5,5,5,5" />
  </Style>
 </Window.Resources>
 <DockPanel>
  <Grid >
  <Grid.ColumnDefinitions>
     <ColumnDefinition />
     <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
     <RowDefinition />
     <RowDefinition />
     <RowDefinition />
     <RowDefinition />
  </Grid.RowDefinitions>
  <ListBox IsSynchronizedWithCurrentItem="True" HorizontalAlignment="Left"
     Margin="5" Style="{StaticResource listBoxStyle}"
     Grid.RowSpan="4" ItemsSource="{Binding}" />
  <Label Style="{StaticResource labelStyle}" Content="{Binding Title}"
     Grid.Row="0" Grid.Column="1" />
  <Label Style="{StaticResource labelStyle}" Content="{Binding Publisher}"
     Grid.Row="1" Grid.Column="1" />
  <Label Style="{StaticResource labelStyle}" Content="{Binding Isbn}"
     Grid.Row="2" Grid.Column="1" />
  </Grid>
 </DockPanel>
</Window>
```

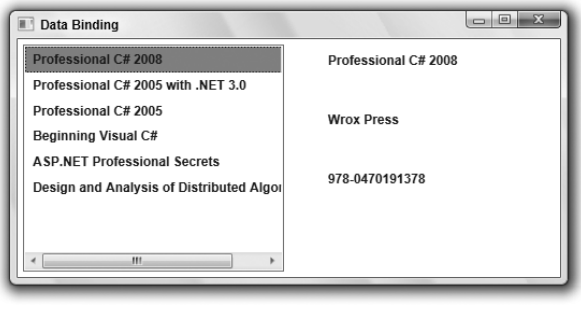

Рис. 35.8. Привязка типа "ведущая–детали"

Преобразование значений

Авторы книги по-прежнему отсутствуют в выводе. Если вы привяжете свойство `Authors` к элементу `Label`, будет вызван метод `ToString()` класса `Array`, который просто вернет имя типа. Одним из решений этой проблемы может быть привязка свойства `Authors` к `ListBox`. Для `ListBox` вы можете определить шаблон специфического представления. Другое же решение заключается в преобразовании строкового массива, возвращенного свойством `Authors`, в строку и использование строки для привязки.

Класс `StringArrayConverter` преобразует строковый массив в строку. Классы конвертеров WPF должны реализовывать интерфейс `IValueConverter` из пространства имен `System.Windows.Data`. Этот интерфейс определяет методы `Convert()` и `ConvertBack()`. У `StringArrayConverter` метод `Convert()` преобразует строковый массив из переменной `value` в строку с помощью метода `String.Join()`. Параметр `separator` в `Join()` берется из переменной `parameter`, полученной методом `Convert()`.

Подробнее о методах класса `String` читайте в главе 8.

```
public class StringArrayConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
  System.Globalization.CultureInfo culture)
  {
     string[] stringCollection = (string[])value;
     string separator = (string)parameter;
     return String.Join(separator, stringCollection);
  }
  public object ConvertBack(object value, Type targetType, object parameter,
     System.Globalization.CultureInfo culture)
  {
     string s = (string)value;
     char separator = (char)parameter;
     return s.Split(separator);
  }
}
```

В коде XAML класс `StringArrayConverter` может быть объявлен как ресурс для обращения к нему из расширения разметки `Binding`:

```
<Window x:Class="Wrox.ProCSharp.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
  Title="List Binding Sample" Height="300" Width="518"
  >
   <Window.Resources>
  <src:StringArrayConverter x:Key="stringArrayConverter" />
  <!-- ... -->
```

Для многострочного вывода элемент `TextBlock` объявлен со свойством `TextWrapping`, установленным в `Wrap`, чтобы обеспечить возможность отображения множества авторов. В расширении разметки `Binding` значение `Path` устанавливается в `Authors`, которое определено как свойство, возвращающее строковый массив. Строковый массив преобразуется из ресурса `stringArrayConverter`, что задано свойством `Converter`. Метод `Convert` реализации конвертера принимает значение `ConverterParameter` , равное `', '`, в качестве разделитель авторов.

```
<TextBlock Width="190" Height="50" Margin="5" TextWrapping="Wrap"
  Text="{Binding Path=Authors,
  Converter={StaticResource stringArrayConverter},
  ConverterParameter=', ' }"
  Grid.Row="3" Grid.Column="1" />
```

На рис. 35.9 можно видеть подробную информацию о книге, включая список авторов.

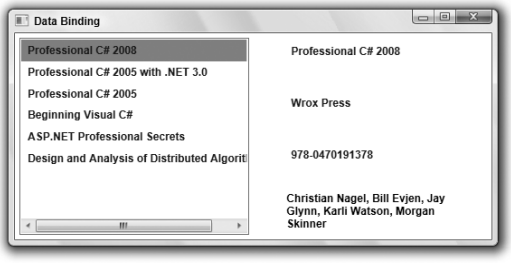

Рис. 35.9. Просмотр подробной информации о книге

Динамическое добавление элементов в список

Что, если элементы в список будут добавляться динамически? Элемент WPF должен быть извещен о добавлении в список новых элементов.

В коде XAML приложения WPF элемент `Button` добавляется внутри `StackPanel`. Событию `Click` назначается обработчик — метод `OnAddBook()`:

```
<!-- ... -->
<DockPanel>
<StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom" Height="60">
<Button Click="addBookButton_Click" Name="addBookButton" Margin="5"
     Width="80" Height="40">Add Book</Button>
</StackPanel>
<Grid >
  <!-- ... -->
```

В методе `addBookButton_Click()`, который реализует код обработчика события для `addBookButton`, новый объект `Book` добавляется к списку. Если вы протестируете приложение с `BookFactory`, в том виде, как оно у нас реализовано сейчас, то никакого уведомления о добавлении нового объекта в список элементы WPF не получат.

```
void addBookButton_Click(object sender, RoutedEventArgs e)
{
factory.AddBook(new Book(".NET 2.0 Wrox Box", "Wrox Press",
     "978-0-470-04840-5"));
}
```

Объект, присвоенный `DataContext`, должен реализовывать интерфейс `INotifyCollectionChanged`. Этот интерфейс определяет событие `CollectionChanged`, используемое приложением WPF. Вместо реализации этого интерфейса в собственном классе коллекции вы можете определить обобщенный класс коллекции `ObservableCollection<T>` из пространства имен `System.Collections.ObjectModel` в сборке `WindowsBase`. После этого при добавлении нового элемента в коллекцию этот новый элемент будет немедленно отображен в `ListBox`.

```
public class BookFactory
{
  private ObservableCollection<Book> books = new ObservableCollection<Book>();
  // ...
```

```
    public void AddBook(Book b)
    {
         books.Add(b);
    }
    public IEnumerable<Book> GetBooks()
    {
         return books;
    }
}
```

Шаблоны данных

В этой главе вы уже видели, как элементы управления могут быть настроены посредством шаблонов. Также вы уже определяли шаблон для типа данных, например, для класса `Book`. Независимо от того, где этот класс используется, шаблон определит для него вид по умолчанию.

В нашем примере `DataTemplate` определен внутри ресурсов `Window`. Свойство `DataType` ссылается на класс `Book` из пространства имен `Wrox.ProCSharp.WPF`. Шаблон определяет рамку с двумя элементами-метками, находящимися в панели `StackPanel`. С помощью элемента `ListBox` вы можете видеть, что нет ссылок ни на какой шаблон. Единственное свойство, определенное `ListBox` — это `ItemsSource` со значением для стандартного расширения разметки `Binding`. Поскольку `DataTemplate` не определяет ключа, он используется всеми списками, содержащими объекты `Book`. На рис. 35.10 показан вывод приложения с шаблоном данных.

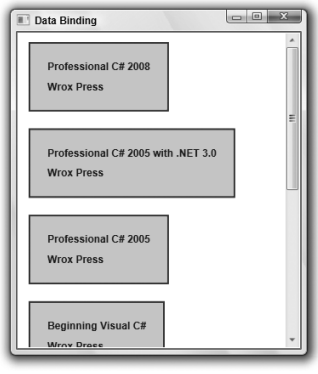

Рис. 35.10. Пример использования шаблона данных

```
<Window x:Class="Wrox.ProCSharp.WPF.DataTemplateDemo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
  Title="Data Binding" Height="300" Width="300"
>
<Window.Resources>
  <DataTemplate DataType="{x:Type src:Book}">
     <Border BorderBrush="Blue" BorderThickness="2" Background="LightBlue"
           Margin="10" Padding="15">
       <StackPanel>
           <Label Content="{Binding Path=Title}" />
           <Label Content="{Binding Path=Publisher}" />
       </StackPanel>
     </Border>
  </DataTemplate>
</Window.Resources>
  <Grid>
     <ListBox ItemsSource="{Binding}" />
  </Grid>
</Window>
```

В случае если вы хотите использовать другой шаблон данных с тем же типом данных, то можете создать селектор шаблона данных. Селектор шаблона данных реализован как класс-наследник `DataTemplateSelector`.

Приведем селектор шаблона данных, реализованный выбором шаблона в зависимости от издателя. Шаблон определен внутри ресурсов `Window`. Один шаблон мо-

жет быть доступен по ключевому имени WroxBookTemplate, а другой — по имени WileyBookTemplate:

```
<DataTemplate x:Key="WroxBookTemplate" DataType="{x:Type src:Book}">
  <Border BorderBrush="Blue" BorderThickness="2" Background="LightBlue"
     Margin="10" Padding="15">
    <StackPanel>
      <Label Content="{Binding Path=Title}"/>
      <Label Content="{Binding Path=Publisher}"/>
    </StackPanel>
  </Border>
</DataTemplate>
<DataTemplate x:Key="WileyBookTemplate" DataType="{x:Type src:Book}">
  <Border BorderBrush="Yellow" BorderThickness="2"
     Background="LightGreen" Margin="10" Padding="15">
    <StackPanel>
      <Label Content="{Binding Path=Title}"/>
      <Label Content="{Binding Path=Publisher}"/>
    </StackPanel>
  </Border>
</DataTemplate>
```

Для выбора шаблона в классе BookDataTemplateSelector переопределяется метод SelectTemplate базового класса DataTemplateSelector. Реализация выбирает шаблон в зависимости от свойства Publisher класса Book.

```
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.WPF
{
  public class BookDataTemplateSelector : DataTemplateSelector
  {
    public override DataTemplate SelectTemplate(object item,
         DependencyObject container)
    {
      if (item != null & & item is Book)
      {
        Window window = Application.Current.MainWindow;
        Book book = item as Book;
        switch (book.Publisher)
        {
          case "Wrox Press":
            return window.FindResource("WroxBookTemplate")
                 as DataTemplate;
          case "Wiley":
            return window.FindResource("WileyBookTemplate")
                 as DataTemplate;
          default:
            return window.FindResource("BookTemplate") as DataTemplate;
        }
      }
      return null;
    }
  }
}
```

Для доступа к классу BookDataTemplateSelector из кода XAML класс определен внутри ресурсов Window:

```
<src:BookDataTemplateSelector x:Key="bookTemplateSelector" />
```

Теперь класс селектора может быть присвоен свойству `ItemTemplateSelector` класса `ListBox`:

```
<ListBox ItemsSource="{Binding}"
 ItemTemplateSelector=
       "{StaticResource bookTemplateSelector}" />
```

При запуске этого приложения вы можете видеть разные шаблоны данных, в зависимости от издателя, как показано на рис. 35.11.

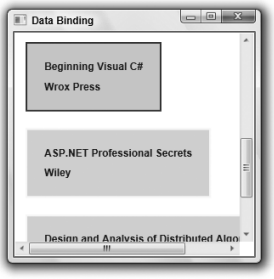

Рис. 35.11. Шаблоны данных, зависящие от издателя

Привязка к XML

Механизм привязки данных WPF имеет специальную поддержку для привязки данных XML. Вы можете применять `XmlDataProvider` в качестве источника данных и привязывать элементы, используя выражения `XPath`. Для иерархического отображения вы можете применить элемент управления `TreeView` и создавать представление элементов с применением `HierarchicalDataTemplate`.

Приведенный ниже XML-файл, содержащий элементы `Book`, будет использован в качестве источника данных в наших следующих примерах.

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
<Book isbn="978-0-470-12472-7">
  <Title>Professional C# 2008</Title>
  <Publisher>Wrox Press</Publisher>
  <Author>Christian Nagel</Author>
  <Author>Bill Evjen</Author>
  <Author>Jay Glynn</Author>
  <Author>Karli Watson</Author>
  <Author>Morgan Skinner</Author>
</Book>
<Book isbn="978-0-7645-4382-1">
  <Title>Beginning Visual C# 2008</Title>
  <Publisher>Wrox Press</Publisher>
  <Author>Karli Watson</Author>
  <Author>David Espinosa</Author>
  <Author>Zach Greenvoss</Author>
  <Author>Jacob Hammer Pedersen</Author>
  <Author>Christian Nagel</Author>
  <Author>John D. Reid</Author>
  <Author>Matthew Reynolds</Author>
  <Author>Morgan Skinner</Author>
  <Author>Eric White</Author>
</Book>
</Books>
```

Подобно определению поставщика объектных данных, вы можете определить поставщика данных XML. И `ObjectDataProvider`, и `XmlDataProvider` унаследованы от одного и того же базового класса `DataSourceProvider`. В примере с `XmlDataProvider` свойство `Source` устанавливается в ссылку на XML-файл `books.xml`. Свойство `XPath` определяет XPath-выражение для обращения к корневому элементу XML `Books`. Элемент `Grid` ссылается на источник данных XML с помощью свойства `DataContext`. В контексте данных для `Grid` все элементы `Book` требуются для привязки списка, поэтому выражение XPath установлено в `Book`. Внутри `Grid` вы можете найти элемент `ListBox`, который привязан к контексту данных по умолчанию и использует `DataTemplate` для включения заголовка в элементы `TextBlock` как составные части

ListBox. Внутри Grid вы также можете видеть три элемента Label с привязкой данных, установленной выражениями XPath, для отображения заголовка, издателя и номеров ISBN.

```
<Window x:Class="XmlBindingSample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="XML Binding" Height="348" Width="498"
  >
<Window.Resources>
  <XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />
  <DataTemplate x:Key="listTemplate">
     <TextBlock Text="{Binding XPath=Title}" />
  </DataTemplate>
  <Style x:Key="labelStyle" TargetType="{x:Type Label}">
     <Setter Property="Width" Value="190" />
     <Setter Property="Height" Value="40" />
     <Setter Property="Margin" Value="5" />
  </Style>
</Window.Resources>
  <Grid DataContext="{Binding Source={StaticResource books}, XPath=Book}">
     <Grid.ColumnDefinitions>
       <ColumnDefinition />
       <ColumnDefinition />
     </Grid.ColumnDefinitions>
     <Grid.RowDefinitions>
       <RowDefinition />
       <RowDefinition />
       <RowDefinition />
       <RowDefinition />
     </Grid.RowDefinitions>
  <ListBox IsSynchronizedWithCurrentItem="True" Margin="5,5,5,5"
     Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
     ItemTemplate="{StaticResource listTemplate}" />
  <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Title}"
       Grid.Row="0" Grid.Column="1" />
  <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Publisher}"
       Grid.Row="1" Grid.Column="1" />
  <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=@isbn}"
       Grid.Row="2" Grid.Column="1" />
</Grid>
</Window>
```

На рис. 35.12 показан результат привязки к XML.

Если данные XML должны быть отображены иерархически, то для этого вы можете использовать элемент управления TreeView.

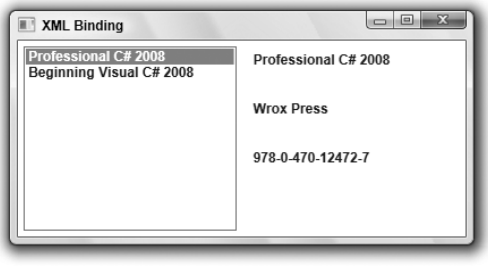

Рис. 35.12. Пример привязки к XML

Проверка привязки

Доступны несколько опций для проверки достоверности данных, поступивших от пользователя, прежде чем они будут использованы объектами .NET:

- обработка исключений;
- информация об ошибках данных;
- специальные правила проверки достоверности.

Обработка исключений

Одной из опций, которые продемонстрированы здесь, является то, что класс .NET генерирует исключение при установке неправильных данных, как показано в классе `SomeData`. Свойство `Value1` принимает значение, большее или равное 5 и меньшее 12.

```
public class SomeData
{
  private int value1;
  public int Value1 {
    get
    {
      return value1;
    }
    set
    {
      if (value < 5 || value > 12)
        throw new ArgumentException(
              "value must not be less than 5 or greater than 12");
      value1 = value;
    }
  }
}
```

В конструкторе класса `Window1` инициализируется новый объект класса `SomeData` и передается `DataContext` для привязки данных:

```
public partial class Window1 : Window
{
  SomeData p1 = new SomeData() { Value1 = 11 };
  public Window1()
  {
    InitializeComponent();
    this.DataContext = p1;
  }
```

Метод-обработчик событий `buttonSubmit_Click` отображает окно сообщения для показа действительного значения экземпляра `SomeData`:

```
  private void buttonSubmit_Click(object sender, RoutedEventArgs e)
  {
    MessageBox.Show(p1.Value1.ToString());
  }
}
```

При простой привязке данных здесь свойство `Text` объекта `TextBox` привязывается к свойству `Value1`. Если теперь вы запустите приложение и попытаетесь изменить значение на неправильное, вы можете убедиться, что значение не будет изменено, щелкнув на кнопке Submit (Отправить). WPF перехватывает и игнорирует исключения, сгенерированные методом доступа `set` свойства `Value1`.

```
<Label Margin="5" Grid.Row="0" Grid.Column="0">Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1" Text="{Binding Path=Value1}" />
```

Чтобы отобразить ошибку, как только контекст поля ввода изменится, вы можете установить свойство `ValidatesOnException` расширения разметки `Binding` в `True`. При неверном значении (как только генерируется исключение, когда должно быть установлено значение) `TextBox` окружается красной рамкой, как показано на рис. 35.13.

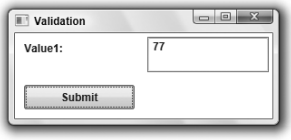

Рис. 35.13. Вывод рамки вокруг `TextBox` при генерации исключения

```
<Label Margin="5" Grid.Row="0"
 Grid.Column="0">Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
     Text="{Binding Path=Value1, ValidatesOnExceptions=True}" />
```

Чтобы иным способом вернуть информацию об ошибке пользователю, вы можете присвоить присоединенное свойство `ErrorTemplate`, определенное классом `Validation`, к шаблону, определяющему пользовательский интерфейс для ошибок. Новый шаблон для обозначения ошибок показан здесь с ключом `validationTemplate`. Шаблон `ControlTemplate` помещает красный восклицательный знак перед существующим содержимым элемента управления.

```
<ControlTemplate x:Key="validationTemplate">
  <DockPanel>
    <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
    <AdornedElementPlaceholder/>
  </DockPanel>
</ControlTemplate>
```

Установка `validationTemplate` в присоединенное свойство `Validation.ErrorTemplate` активизирует шаблон с `TextBox`:

```
<Label Margin="5" Grid.Row="0" Grid.Column="0">Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
     Text="{Binding Path=Value1, ValidatesOnExceptions=True}"
     Validation.ErrorTemplate="{StaticResource validationTemplate}" />
```

Новый вид приложения показан на рис. 35.14.

Другой вариант специального сообщения об ошибке заключается в регистрации события `Error` класса `Validation`. В этом случае свойство `NotifyOnValidationError` должно быть установлено в `True`.

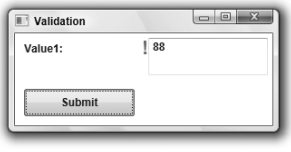

Рис. 35.14. Изменение внешнего вида приложения при генерации исключения

Информация об ошибке сама по себе может быть доступна через коллекцию Errors класса `Validation`. Чтобы отобразить информацию об ошибке в `ToolTip` элемента управления `TextBox`, вы можете создать триггер свойства, как показано ниже. Триггер активизируется, как только свойство `HasError` класса `Validation` устанавливается в `True`. Триггер устанавливает свойство `ToolTip` элемента `TextBox`.

```
<Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="ToolTip"
           Value="{Binding RelativeSource=
                {x:Static RelativeSource.Self},
           Path=(Validation.Errors)[0].ErrorContent}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Информация об ошибках данных

Другой способ обращения с ошибками состоит в том, что объект .NET реализует интерфейс `IDataErrorInfo`.

Класс `SomeData` теперь изменен для реализации интерфейса `IDataErrorInfo`. Этот интерфейс определяет свойство `Error` и индексатор со строковым аргументом. При проверке достоверности WPF во время привязки данных индексатор вызывается, и имя проверяемого свойства передается в качестве аргумента `columnName`. С такой реализацией значение проверяется на предмет допустимости, в противном случае передается строка ошибки.

В данном случае проверка достоверности выполняется на свойстве `Value2`, которое реализовано с использованием простой нотации свойства C# 3.0:

```
public class SomeData : IDataErrorInfo
{
  private int value1;
  public int Value1 {
    get
    {
      return value1;
    }
    set
    {
      if (value < 5 || value > 12)
        throw new ArgumentException(
             "value must not be less than 5 or greater than 12");
      value1 = value;
    }
  }
  public int Value2 { get; set; }
  string IDataErrorInfo.Error
  {
    get
    {
      return null;
    }
  }

  string IDataErrorInfo.this[string columnName]
  {
    get
    {
      if (columnName == "Value2")
      {
        if (this.Value2 < 0 || this.Value2 > 80)
          return "age must not be less than 0 or greater than 80";
      }
      return null;
    }
  }
}
```

Для сущностного класса .NET не ясно? что должен возвращать индексатор; например, чего ожидать от объекта типа `Person`, вызывающего индексатор? Вот почему лучше ясно реализовать интерфейс `ISataErrorInfo`. Таким образом, индексатор может быть доступен только с использованием этого интерфейса, и класс .NET может предусматривать другую реализацию для других целей.

Если вы устанавливаете свойство ValidatesOnDataErrors класса Binding в true, то во время привязки используется интерфейс IDataErrorInfo. Здесь при изменении TextBox механизм привязки вызывает индексатор интерфейса и передает Value2 переменной columnName:

```
<Label Margin="5" Grid.Row="1" Grid.Column="0">Value2:</Label>
<TextBox Margin="5" Grid.Row="1" Grid.Column="1"
     Text="{Binding Path=Value2, ValidatesOnDataErrors=True}" />
```

Специальные правила проверки достоверности

Чтобы получить больше контроля над процессом проверки достоверности, вы можете реализовать специальное правило проверки достоверности. Класс, реализующий такое правило, должен наследоваться от базового класса ValidationRule.

В предыдущих двух примерах также были использованы правила проверки достоверности. Два класса-наследника абстрактного базового класса ValidationRule — это DataErrorValidationRule и ExceptionValidationRule. Класс DataErrorValidationRule активизируется установкой свойства ValidatesOnDataErrors и использует интерфейс IDataErrorInfo; ExceptionValidationRule имеет дело с исключениями и активизируется установкой свойства ValidatesOnException.

Здесь правило проверки достоверности реализовано для верификации регулярного выражения. Класс RegularExpressionValidationRule наследуется от базового класса ValidationRule и переопределяет абстрактный метод Validate(), определенный в базовом классе. В этой реализации класс RegEx из пространства имен System.Text.RegularExpressions используется для проверки достоверности выражения, определенного свойством Expression.

```
public class RegularExpressionValidationRule : ValidationRule
{
  public string Expression { get; set; }
  public string ErrorMessage { get; set; }
  public override ValidationResult Validate(object value,
       CultureInfo cultureInfo)
  {
    ValidationResult result = null;
    if (value != null)
    {
      Regex regEx = new Regex(Expression);
      bool isMatch = regEx.IsMatch(value.ToString());
      result = new ValidationResult(isMatch, isMatch ?
           null : ErrorMessage);
    }
    return result;
  }
}
```

Вместо использования расширения разметки Binding теперь привязка осуществляется как дочерний элемент элемента TextBox.Text. Объект привязки теперь определяет свойство Email, реализованное в простом синтаксисе свойства. Свойство UpdateSourceTrigger определяет, когда источник должен быть обновлен. Возможные опции для обновления источника таковы:

- когда значение свойства меняется, что происходит при вводе каждого символа пользователем;
- когда элемент утрачивает фокус;
- явно.

ValidationRules — это свойство класса Binding, содержащее элементы ValidationRule. Ниже приведено правило проверки достоверности, используемое специальным классом RegularExpressionValidationRule, где свойство Expression установлено в регулярное выражение, проверяющее ввод на соответствие формату адреса электронной почты, а свойство ErrorMessage выдает сообщение об ошибке в случае, если введенные в TextBox данные неверны.

```
<Label Margin="5" Grid.Row="2" Grid.Column="0"> Email: </Label>
<TextBox Margin="5" Grid.Row="2" Grid.Column="1">
  <TextBox.Text>
    <Binding Path="Email" UpdateSourceTrigger="LostFocus">
      <Binding.ValidationRules>
        <src:RegularExpressionValidationRule
            Expression="^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)
|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$"
            ErrorMessage="Email is not valid" />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

Привязка команд

WPF включает элементы управления Menu и ToolBar, служащие тем же целям, что и элементы, известные вам по Windows Forms: запускать команды. К этим элементам управления вы можете добавлять обработчики событий, чтобы обеспечить функциональность команд. Однако вы можете запускать команды, выбирая пункты меню, щелкая на кнопках в панели инструментов либо нажимая определенные клавиши на клавиатуре. Чтобы обработать все эти разнообразные действия, WPF предлагает другое средство — команды.

Некоторые из элементов управления WPF предоставляют реализацию предопределенных команд, которые позволяют очень легко получить некоторую функциональность.

В WPF предусмотрены некоторые предопределенные команды в классах команд ApplicationCommands, EditingCommands, ComponentCommands и NavigationCommands. Все эти классы команд являются статическими, со статическими свойствами, возвращающими объекты RoutedUICommands. Например, некоторые из свойств ApplicationCommands таковы: New, Open, Save, SaveAs, Print и Close, т.е., это команды, известные вам по многим приложениям.

Чтобы приступить к работе с командами, создайте простой проект WPF и добавьте элемент управления Menu с пунктами для операций отмены, повтора, вырезки, копирования и вклейки. TextBox по имени textContent занимает остальное место окна и обеспечивает возможность многострочного ввода. Внутри окна создается панель DockPanel, определяющая компоновку. К верхней части окна пристыкован элемент управления Menu с элементами MenuItem. Установлен заголовок, определяющий текст меню. Знак _ определяет букву, к которой вы можете получить быстрый доступ с клавиатуры, без использования мыши. Когда вы нажимаете клавишу <Alt>, под этой буквой из текста заголовка появляется подчеркивание. Свойство Command определяет команду, ассоциированную с пунктом меню.

```
<Window x:Class="Wrox.ProCSharp.WPF.WPFEditorWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="WPF Editor" Height="300" Width="300"
  >
```

```
  <DockPanel>
  <Menu DockPanel.Dock="Top">
     <MenuItem Header="_Edit">
       <MenuItem Name="editUndoMenu" Header="_Undo"
           Command="ApplicationCommands.Undo"/>
       <MenuItem Name="editRedoMenu" Header="_Redo"
           Command="ApplicationCommands.Redo"/>
       <Separator/>
       <MenuItem Name="editCutMenu" Header="Cu_t"
           Command="ApplicationCommands.Cut"/>
       <MenuItem Name="editCopyMenu" Header="_Copy"
           Command="ApplicationCommands.Copy"/>
       <MenuItem Name="editPasteMenu" Header="_Paste"
           Command="ApplicationCommands.Paste"/>
    </MenuItem>
  </Menu>
    <TextBox Name="textContent" TextWrapping="Wrap" AcceptsReturn="True"
         AcceptsTab="True"/>
  </DockPanel>
</Window>
```

Это все, что вам нужно, чтобы реализовать функциональность буфера обмена. Класс `TextBox` уже включает функциональность для этих предопределенных привязок команд. Запустив приложение и начав вводить текст в текстовое поле, вы увидите, что соответствующие пункты меню станут активными. Выбор текста в поле сделает активными пункты меню, управляющие вырезкой и копированием. На рис. 35.15 показано работающее приложение.

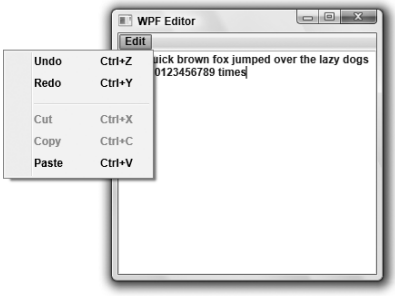

Рис. 35.15. Приложение простого редактора во время выполнения

Теперь нужно изменить приложение, добавив привязки команд, которые ранее не были определены элементами управления. Речь идет о командах открытия и сохранения файла, которые мы добавим к редактору.

Чтобы сделать их доступными, понадобятся добавить дополнительные элементы `MenuItem` в элемент `Menu`, как показано ниже:

```
<MenuItem Header="_File">
 <MenuItem Name="fileNewMenu" Header="_New" Command="ApplicationCommands.New" />
 <MenuItem Name="fileOpenMenu" Header="_Open" Command="ApplicationCommands.Open" />
 <Separator />
 <MenuItem Name="fileSave" Header="_Save" Command="ApplicationCommands.Save" />
 <MenuItem Name="fileSaveAs" Header=Save _As" />
</MenuItem>
```

Команды также могут быть доступны из панели инструментов. В элементе `ToolBar` определены те же команды, что и в меню. Для размещения панели инструментов элемент `ToolBar` вставлен внутрь `ToolBarTray`:

```
<ToolBarTray DockPanel.Dock="Top">
   <ToolBar>
     <Button Command="ApplicationCommands.New">
          <Image Source="toolbargraphics/New.bmp" />
     </Button>
     <Button Command="ApplicationCommands.Open">
          <Image Source="toolbargraphics/Open.bmp" />
     </Button>
     <Button Command="ApplicationCommands.Save">
          <Image Source="toolbargraphics/Save.bmp" />
     </Button>
   </ToolBar>
</ToolBarTray>
```

Теперь привязки команд должны быть определены для ассоциации команд с обработчиками событий. Привязки команд могут быть присвоены любому классу WPF, производному от базового класса `UIElement`, находящемуся очень высоко в иерархии. Привязки команд добавляются к свойству `CommandBindings` в виде элементов `CommandBinding`. Класс `CommandBinding` имеет свойство `Command`, в котором вы специфицируете объект, реализующий интерфейс `ICommand`, а также события `CanExecute` и `Executed` для указания обработчиков событий. Ниже приведены привязки команд, назначенные классу `Window`. Событие `Executed` установлено в методы-обработчики событий, реализующие функциональность, выполняемую командами. Если команда не должна быть доступна постоянно, вы можете также установить обработчик `CanExecute`, который будет решать, когда данная команда должна быть доступна.

```
<Window.CommandBindings>
 <CommandBinding Command="ApplicationCommands.New" Executed="NewFileExecuted" />
 <CommandBinding Command="ApplicationCommands.Open" Executed="OpenFileExecuted" />
 <CommandBinding Command="ApplicationCommands.Save" Executed="SaveFileExecuted"
 CanExecute="SaveFileCanExecute" />
 <CommandBinding Command="ApplicationCommands.SaveAs" Executed="SaveAsFileExecuted"
    CanExecute="SaveAsFileCanExecute" />
</Window.CommandBindings>
```

В коде метода-обработчика `NewFileExecuted()` происходит очистка текстового поля и запись имени файла `untitled.txt` в свойство `Title` класса `Window`.

В `OpenFileExecuted()` создается `Microsoft.Win32.OpenFileDialog` и отображается в виде диалогового окна. При успешном выходе из диалога выбранный файл открывается и записывается в элемент управления `TextBox`.

> Диалог открытия файла в WPF не предопределен. Вы можете либо создать собственный диалог для выбора файлов и папок, либо воспользоваться классом **`OpenFileDialog`** из пространства имен **`Microsoft.Win32`**, который служит оболочкой диалога Win32.

```
public partial class Window1 : System.Windows.Window
{
private string filename;
private readonly string defaultFilename;
private const string appName = "WPF Editor";
private bool isChanged = false;
public Window1()
{
  defaultFilename =
     Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
     @"\untitled.txt";
```

```
  InitializeComponent();
  NewFile();
}
private void OnFileNew(object sender, ExecutedRoutedEventArgs e)
{
  NewFile();
}
private void NewFile()
{
  textContent.Clear();
  filename = defaultFilename;
  SetTitle();
  isChanged = false;
}
private void SetTitle()
{
  Title = String.Format("{0} {1}", System.IO.Path.GetFileName(filename), appName);
}
private void OnFileOpen(object sender, ExecutedRoutedEventArgs e)
{
try
{
  OpenFileDialog dlg = new OpenFileDialog();
  bool? dialogResult = dlg.ShowDialog();
  if (dialogResult)
  {
    filename = dlg.FileName;
    SetTitle();
    textContent.Text = File.ReadAllText(filename);
  }
}
catch (IOException ex)
{
  MessageBox.Show(ex.Message, "Error WPF Editor", MessageBoxButton.OK,
     MessageBoxImage.Error);
}
}
```

Обработчик `SaveFileCanExecute()` возвращает решение о том, должна ли быть доступна команда сохранения файла, в зависимости от того, было ли изменено содержимое:

```
private void SaveFileCanExecute(object
sender, CanExecuteRoutedEventArgs e)
{
  if (isChanged)
  {
    e.CanExecute = true;
  }
  else
  {
    e.CanExecute = false;
  }
}
```

Затем приложение откроет файл примера `sample.txt`, как показано на рис. 35.16.

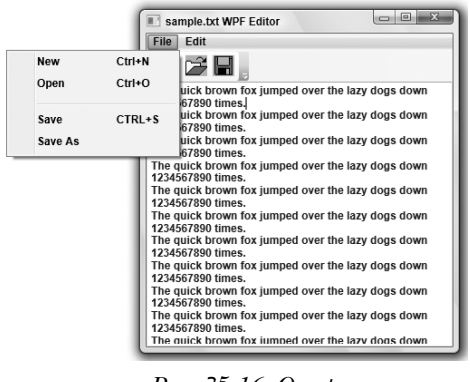

Рис. 35.16. Открытие файла `sample.txt`

Анимация

С помощью анимации вы можете выполнять плавные перемещения, используя движение элементов, изменение цветов, трансформации и тому подобное. WPF существенно облегчает задачу создания анимации. Вы можете анимировать значение и свойство зависимости. Для этого предусмотрены разнообразные классы, позволяющие анимировать значения различных свойств, в зависимости от их типа.

Основные элементы анимации описаны ниже.

- **Временная шкала** (Timeline). Временная шкала определяет, как значение изменяется во времени. Доступны различные виды временных шкал для изменения различных типов значений. Базовый класс для всех временных шкал — это `Timeline`. Чтобы анимировать `double`, можно использовать класс `DoubleAnimation`. Класс `Int32Animation` — класс анимации для целочисленных значений.
- **Раскадровка** (Storyboard). Раскадровка используется для комбинирования анимаций. Сам по себе класс `Storyboard` наследуется от базового класса `TimeLineGroup`, который, в свою очередь, унаследован от `Timeline`. С помощью `DoubleAnumation` вы можете анимировать значения `double`, а `StoryBoard` позволяет комбинировать все анимации, которые связаны между собой.
- **Триггеры**. С помощью триггеров вы может запускать и останавливать анимации. Вы уже знакомы с триггерами свойств. Триггеры свойств срабатывают при изменении значения свойств. Вы можете создать триггер события. Такой триггер срабатывает при наступлении события.

Пространство имен для классов анимации — `System.Windows.Media.Animation`.

Временная шкала

`TimeLine` определяет то, как значение меняется во времени. Наш первый пример анимирует размер эллипса. Здесь используется `DoubleAnimation`, представляющая временную шкалу изменения значения `double`. Свойство `Triggers` класса `Ellipse` установлено в `EventTrigger`. Триггер событий инициируется при загрузке эллипса, как определено в свойстве `RoutedEvent` объекта `EventTrigger`. `BeginStoryboard` — действие триггера, которое запускает раскадровку `StoryBoard`. С этой раскадровкой используется элемент `DoubleAnimation` для анимирования свойства `Width` класса `Ellipse`. Анимация изменяет ширину эллипса от 100 до 300 за 3 секунды, и обращает эту же анимацию через 3 секунды.

```
<Window x:Class="EllipseAnimation.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ellipse Animation" Height="300" Width="300">
  <Grid>
    <Ellipse Height="50" Width="100" Fill="SteelBlue">
      <Ellipse.Triggers>
        <EventTrigger RoutedEvent="Ellipse.Loaded">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard Duration="00:00:06" RepeatBehavior="Forever">
                <DoubleAnimation
                     Storyboard.TargetProperty="(Ellipse.Width)"
                     Duration="0:0:3" AutoReverse="True"
                     FillBehavior="Stop" RepeatBehavior="Forever"
                     AccelerationRatio="0.9" DecelerationRatio="0.1"
                     From="100" To="300" />
```

```
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Ellipse.Triggers>
  </Ellipse>
 </Grid>
</Window>
```

На рис. 35.17 и 35.18 показаны два состояния анимированного эллипса.

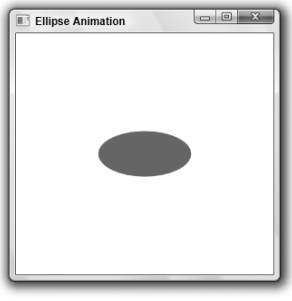

Рис. 35.17. Эллипс с шириной 100

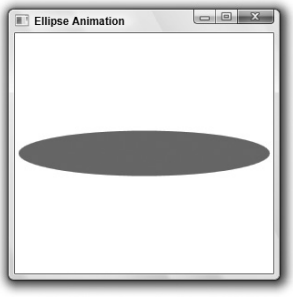

Рис. 35.18. Эллипс с шириной 300

Речь идет об анимации, представляющей собой нечто большее, чем обычная оконная анимация, которая появляется на экране постоянно и немедленно. Вы можете добавить анимацию к бизнес-приложениям, что сделает пользовательский интерфейс более отзывчивым.

Следующий пример демонстрирует приличную анимацию, и также показывает, как анимация может быть определена в стиле. Внутри ресурсов `Window` вы можете видеть стиль `AnimatedButtonStyle` для кнопок. В шаблоне определяется прямоугольник по имени `outline`. Шаблон имеет тонкий контур шириной `0.4`.

Шаблон определяет триггер свойств для свойства `IsMouseOver`. Свойство `EnterActions` этого триггера применяется, как только курсор мыши перемещается над кнопкой. Действие запуска — `BeginStoryboard`. Действие триггера `BeginStoryboard` может включать, а потому и запускать элементы `Storyboard`. Элемент `Storyboard` определяет `DoubleAnimation` для анимации значения `double`. Значение свойства, изменяемой в этой анимации — это `Rectangle.StrokeThickness` элемента `Rectangle` по имени `outline`. Значение плавно изменяется на `1.2`, как задает свойство `By`, в течение периода `0.3` секунды, что указано в свойстве `Duration`. В конце анимации толщина штриха сбрасывается в свое исходное значение, поскольку установлено `AutoReverse="True"`. В итоге имеем следующее: как только курсор мыши перемещается над кнопкой, толщина контура увеличивается на `1.2` в течение `0.3` секунды. На рис. 35.19 показана кнопка без анимации, а на рис. 35.20 — кнопка в момент через `0.3` секунды после начала движения мыши. К сожалению, показать анимацию со всеми промежуточными видами в печатном документе невозможно.

```
<Window x:Class="AnimationSample.ButtonAnimation"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Animation Sample" Height="300" Width="300">
<Window.Resources>
  <Style x:Key="AnimatedButtonStyle" TargetType="{x:Type Button}">
     <Setter Property="Template">
```

```
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Button}">
              <Grid>
                  <Rectangle Name="outline" RadiusX="9" RadiusY="9" Stroke="Black"
                     Fill="{TemplateBinding Background}" StrokeThickness="0.4">
                  </Rectangle>
                  <ContentPresenter VerticalAlignment="Center"
                     HorizontalAlignment="Center"
                       />
              </Grid>
              <ControlTemplate.Triggers>
                  <Trigger Property="IsMouseOver" Value="True">
                     <Trigger.EnterActions>
                       <BeginStoryboard>
                            <Storyboard>
                            <DoubleAnimation Duration="0:0:0.3"
                            AutoReverse="True"
                                 Storyboard.TargetProperty=
                                   "(Rectangle.StrokeThickness)"
                                 Storyboard.TargetName="outline" By="1.2" />
                            </Storyboard>
                       </BeginStoryboard>
                     </Trigger.EnterActions>
                  </Trigger>
              </ControlTemplate.Triggers>
            </ControlTemplate>
        </Setter.Value>
      </Setter>
  </Style>
</Window.Resources>
<Grid>
  <Button Style="{StaticResource MyButtonStyle}" Width="200" Height="100">
     Click Me!
  </Button>
</Grid>
</Window>
```

Свойства класса `Timeline` перечислены в табл. 35.4.

Рис. 35.19. Кнопка без анимации

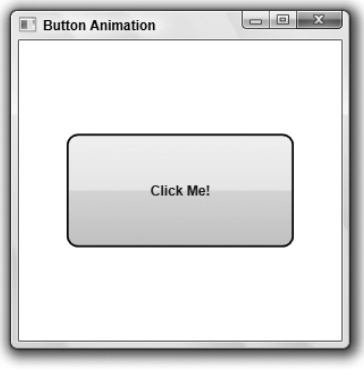

Рис. 35.20. Кнопка в момент через `0.3` секунды после начала движения мыши

Таблица 35.4. Свойства класса `Timeline`

Свойство	Описание
`AutoReverse`	Свойством `AutoReverse` вы можете специфицировать, должно ли значение, подверженное анимации, возвращаться в свое исходное значение по ее окончанию.
`SpeedRatio`	`SpeedRatio` позволяет изменять скорость продвижения анимации. Этим свойством вы можете определить отношение к родителю. По умолчанию оно равно 1; установка в меньшее значение замедляет анимацию, установка в значение больше 1 заставляет ее продвигаться быстрее.
`BeginTime`	Посредством `BeginTime` можно указать задержку между стартом события триггера и моментом запуска анимации. Здесь можно задавать дни, часы, минуты, секунды и доли секунды. Значение может не отражать реальное время, в зависимости от временного масштаба — `SpeedRatio`. Например, если временной масштаб установлен равным 2, а время запуска — 6 секунд, то анимация запустится через 3 секунды.
`AccelerationRatio` `DecelerationRatio`	При анимации значения не обязаны изменяться линейным образом. Вы можете специфицировать `AccelerationRatio` и `DecelerationRatio`, чтобы задать ускорение и замедление. Сумма обоих значений не должна превышать 1.
`Duration`	Свойство `Duration` позволяет указать длительность времени одной итерации анимации.
`RepeatBehavior`	Присвоение структуры `RepeatBehavior` свойству `RepeatBehavior` позволяет определить, сколько раз либо насколько долго должна повторяться анимация.
`FillBehavior`	Свойство `FillBehavior` важно в том случае, если родительская временная шкала имеет другую длительность. Например, если временная шкала родителя короче, чем шкала данной анимации, установка `FillBehavior` в `Stop` означает, что данную анимацию следует остановить. Если же временная шкала родителя длиннее, чем продолжительность данной анимации, то `HoldEnd` оставляет текущую анимацию активной перед тем, как сбросить ее в исходное значение (если установлено `AutoReverse`).

В зависимости от типа класса `Timeline`, могут быть доступны и некоторые другие свойства. Например, в классе `DoubleAnimation` вы можете специфицировать дополнительные свойства, перечисленные в табл. 35.5.

Таблица 35.5. Свойства класса `DoubleAnimation`

Свойство	Описание
`From` `To`	Устанавливая свойства `From` и `To`, вы можете специфицировать значения для запуска и завершения анимации.
`By`	Вместо определения стартового значения анимации, устанавливая свойство `By`, можно запускать анимацию с текущим значением связанного свойства и увеличивать его на указанное в `By` значение до завершения анимации.

Триггеры

Вместо использования триггера свойств можно определить триггер события, чтобы запустить анимацию. Следующий пример иллюстрирует создание анимации с забавной рожицей, знакомой вам по предыдущим примерам, у которой глаза будут перемещаться, как только случится событие щелчка на кнопке. Этот пример также демонстрирует возможность запуска анимации как из кода XAML, так и из отделенного кода.

На рис. 35.21 показан дизайн примера анимации нашей рожицы.

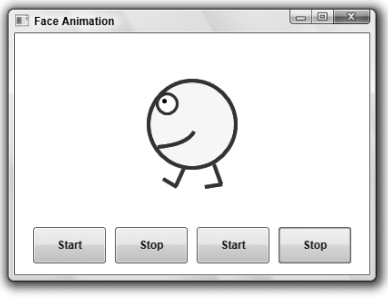

Рис. 35.21. Дизайн примера анимации

Внутри элемента `Window` определен элемент `DockPanel`, чтобы упорядочить картинку и кнопки. `Grid`, содержащий элемент `Canvas`, пристыкован вверху. Пристыковка внизу конфигурируется элементом `StackPanel`, содержащим четыре кнопки. Первые две из них используются для анимации глаз из отделенного кода; последние две используются для анимации глаз из XAML.

Анимация определена внутри раздела `<DockPanel.Triggers>`. Вместо триггера свойств используется триггер события. Первый триггер события срабатывает, как только возникает событие `Click` в кнопке `startButtonXAML`, что задано свойствами `RoutedEvent` и `SourceName`. Действие триггера определено элементом `BeginStoryboard`, с которого начинается содержащий его `Storyboard`. Элемент `BeginStoryboard` имеет определенное имя, поскольку оно необходимо для целей управления — действий паузы, продолжения и останова. Элемент `Storyboard` содержит две анимации. Первая изменяет значение позиции `Canvas.Left` глаза, а вторая изменяет значение `Canvas.Top`. Обе анимации имеют разные временные значения, что делает движение глаз очень интересным за счет различного поведения повторов.

Второй триггер события срабатывает, как только возникает событие `Click` кнопки `stopButtonXAML`. Здесь раскадровка останавливается элементом `StopStoryboard`, ссылающимся на запущенную раскадровку `beginMoveEye`.

```
<Window x:Class="AnimatedFace.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Face Animation" Height="300" Width="406">
<DockPanel>
  <Grid DockPanel.Dock="Top">
     <!-- Funny Face -->
     <Canvas Width="200" Height="200">
       <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
            Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
       <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
            Stroke="Blue" StrokeThickness="3" Fill="White" />
       <Ellipse Name="eye" Canvas.Left="67" Canvas.Top="72" Width="5"
            Height="5" Fill="Black" />
```

```
        <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
              Data="M 62,125 Q 95,122 102,108" />
        <Line Name="LeftLeg" X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue"
              StrokeThickness="4" />
        <Line Name="LeftFoot" X1="68" X2="83" Y1="160" Y2="169" Stroke="Blue"
              StrokeThickness="4" />
        <Line Name="RightLeg" X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue"
              StrokeThickness="4" />
        <Line Name="RightFoot" X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue"
              StrokeThickness="4" />
      </Canvas>
    </Grid>
    <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal">
      <Button Width="80" Height="40" Margin="20,5,5,5"
              Name="startAnimationButton">Start</Button>
      <Button Width="80" Height="40" Margin="5,5,5,5"
              Name="stopAnimationButton">Stop</Button>
      <Button Width="80" Height="40" Margin="5,5,5,5"
              Name="startButtonXAML">Start</Button>
      <Button Width="80" Height="40" Margin="5,5,5,5"
              Name="stopButtonXAML">Stop
      </Button>
    </StackPanel>
    <DockPanel.Triggers>
      <EventTrigger RoutedEvent="Button.Click" SourceName="startButtonXAML">
        <BeginStoryboard Name="beginMoveEye">
          <Storyboard Name="moveEye">
            <DoubleAnimation RepeatBehavior="Forever"
            DecelerationRatio=".8"
              AutoReverse="True" By="8" Duration="0:0:1"
              Storyboard.TargetName="eye"
              Storyboard.TargetProperty="(Canvas.Left)" />
            <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True" By="8"
              Duration="0:0:5" Storyboard.TargetName="eye"
              Storyboard.TargetProperty="(Canvas.Top)" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
      <EventTrigger RoutedEvent="Button.Click" SourceName="stopButtonXAML">
        <StopStoryboard BeginStoryboardName="beginMoveEye" />
      </EventTrigger>
      </DockPanel.Triggers>
    </DockPanel>
  </Window>
```

Вместо запуска и остановки анимации непосредственно из триггеров событий в XAML вы можете легко управлять анимацией из отделенного кода. Кнопки `startAnimationButton` и `stopAnimationButton` имеют ассоциированные с ними обработчики событий `OnStartAnimation` и `OnStopAnimation`. Внутри обработчиков событий анимация запускается методом `Begin()` и останавливается методом `Stop()`. Для метода `Begin()` второй параметр устанавливается в `true`, чтобы позволить управлять анимацией с помощью запроса на останов.

```
public partial class Window1 : System.Windows.Window
{
 public Window1()
 {
  InitializeComponent();
  startAnimationButton.Click += OnStartAnimation;
  stopAnimationButton.Click += OnStopAnimation;
 }
```

```
    void OnStartAnimation(object sender, RoutedEventArgs e)
    {
     moveEye.Begin(eye, true);
    }

    void OnStopAnimation(object sender, RoutedEventArgs e)
    {
     moveEye.Stop(eye);
    }
   }
```

Теперь вы можете запустить приложение и наблюдать движение глаз рожицы, как только щелкнете на кнопке Start (Пуск).

Storyboard

Класс `Storyboard` унаследован от базового класса `Timeline`, но может содержать в себе множество временных шкал. Класс `Storyboard` может быть использован для управления временными шкалами. В табл. 35.6 описаны методы класса `Storyboard`.

Таблица 35.6. Методы класса `Storyboard`

Метод	Описание
`Begin()`	Метол `Begin()` запускает анимацию, ассоциированную с раскадровкой.
`BeginAnimation()`	Методом `BeginAnimation()` можно запустить одну анимацию для свойства зависимости.
`CreateClock()`	Метод `CreateClock()` возвращает объект, который можно использовать для управления анимацией.
`Pause()` `Resume()`	Методами `Pause()` и `Resume()` можно приостанавливать и возобновлять анимацию.
`Seek()`	С помощью метода `Seek()` можно совершить прыжок во времени и переместить точку выполнения анимации на определенный временной интервал.
`Stop()`	Методом `Stop()` останавливает отсчет времени и анимацию.

Класс `EventTrigger` позволяет определить действия, которые нужно выполнить при возникновении события. В табл. 35.7 перечислены свойства этого класса.

Таблица 35.7. Свойства класса `EventTrigger`

Свойство	Описание
`RoutedEvent`	Через свойство `RoutedEvent` вы можете определить событие, в ответ на которое должен стартовать триггер, например, событие `Click` для кнопки.
`SourceName`	Свойство `SourceName` задает элемент WPF, к которому должно подключаться событие.

Классы `TriggerAction`, которые можно поместить внутрь `EventTrigger`, перечислены в табл. 35.8. В примере, который вы видели ранее, описаны действия `BeginStoryboard` и `StopStoryboard`, но в этой таблице присутствует и ряд других.

Таблица 35.8. Классы `TriggerAction`

Класс	Описание
`SoundPlayerAction`	С помощью `SoundPlayerAction` можно воспроизвести файл `.wav`.
`BeginStoryboard`	`BeginStoryboard` запускает анимацию, определенную `Storyboard`.
`PauseStoryboard`	`PauseStoryboard` приостанавливает анимацию.
`ResumeStoryboard`	`ResumeStoryboard` возобновляет приостановленную анимацию.
`StopStoryboard`	`StopStoryboard` останавливает запущенную анимацию.
`SeekStoryboard`	Посредством `SeekStoryboard` можно изменить текущий момент воспроизведения анимации.
`SkipStoryboardToFill`	`SkipStoryboardToFill` перемещает точку исполнения анимации таким образом, чтобы захватить заданный период в конце.
`SetStoryboardSpeedRatio`	С помощью `SetStoryboardSpeedRatio` можно изменить скорость воспроизведения анимации.

Добавление средств 3-D к WPF

В этом разделе мы представим введение в средства 3-D WPF. Здесь вы найдете необходимую информацию, чтобы начать работать с этими средствами.

Пространство имен для 3-D — это `System.Windows.Media.Media3D`.

Чтобы понять 3-D в WPF, важно значит отличие координатной системы. На рис. 35.22 показана координатная система WPF 3-D. Начальная точка находится в центре. Положительные значения по оси x откладываются вправо, а отрицательные — влево. Ось y — вертикальная, с положительными значениями вверху и отрицательными внизу. Ось z определяет положительные значения в направлении наблюдателя.

Наиболее важные классы и их функциональность описаны в табл. 35.9.

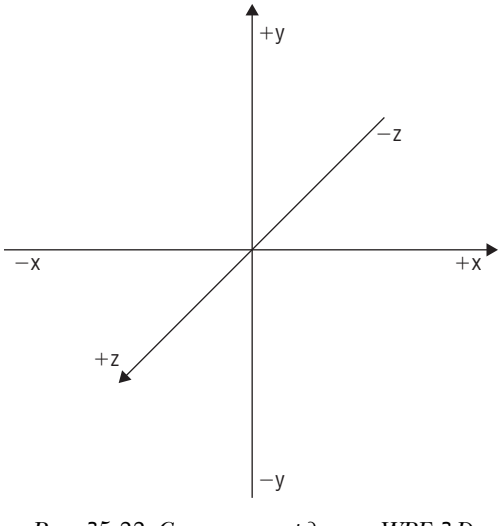

Рис. 35.22. Система координат WPF 3-D

Таблица 35.9. Классы 3-D

Класс	Описание
`ViewPort3D`	`ViewPort3D` определяет поверхность отображения объектов 3-D. Этот элемент содержит все визуальные элементы для трехмерного рисования.
`ModelVisual3D`	`ModelVisual3D` содержится в `ViewPort3D` и содержит все визуальные элементы. Вы можете назначить трансформацию всей модели сразу.
`GeometryModel3D`	`GeometryModel3D` содержится внутри `ModelVisual3D` и состоит из сетки и материала.
`Geometry3D`	`Geometry3D` — абстрактный базовый класс для определения геометрических фигур. Конкретный класс, производный от `Geometry3D` — это `MeshGeometry3D`. С помощью `MeshGeometry3D` вы можете определять позиции треугольников для построения трехмерной модели
`Material`	`Material` — абстрактный базовый класс для определения лицевой и изнаночной поверхности треугольников, определенный свойством `MeshGeometry3D.Material` и содержащийся внутри `GeometryModel3D`. В .NET 3.5 определено несколько классов материалов, такие как `DiffuseMaterial`, `EmissiveMaterial` и `SpecularMaterial`. В зависимости от типа материала освещение вычисляется по-разному. `EmissiveMaterial` предполагает такое вычисление освещения, которое исходит из того, что материал издает свет цвета кисти. `DiffuseMaterial` издает рассеянный свет, а `SpecularMaterial` определяет модель материала с зеркальным отражением. С классом `MaterialGroup` вы можете создавать комбинации, состоящие из разных материалов.
`Light`	`Light` — абстрактный базовый класс, описывающий освещение. Конкретные его реализации: `AmbientLight`, `DirectionalLight`, `PointLight` и `SpotLight`. Класс `AmbientLight` — неестественный свет, равномерно освещает сцену. При таком свете вы не увидите граней. `DirectionalLight` определяет направленный свет. Примером может служить солнечный свет. Свет поступает с одной стороны, и здесь вы можете видеть грани и тени. `PointLight` — свет, поступающий из определенной точки и распространяющийся от нее во всех направлениях. `SpotLight` светит в определенном направлении. Этот свет определяет конус освещенности, так что вы можете получить отдельную ярко освещенную область.
`Camera`	`Camera` — абстрактный базовый класс для описания камеры, используемой для отображения трехмерной сцены на двумерную. Конкретные его реализации: `PerspectiveCamera`, `OrthographicCamera` и `MatrixCamera`. При использовании `PerspectiveCamera` трехмерные объекты выглядят тем меньше, чем дальше они находятся. Это отличает его от `OrthographicCamera`, где расстояние объекта от камеры не влияет на его размер. С помощью `MatrixCamera` вы можете определять вид и трансформацию матрицы.
`Transform3D`	`Transform3D` — абстрактный базовый класс для описания трехмерных трансформаций. Конкретные реализации: `RotateTransform3D`, `ScaleTransform3D`, `TranslateTransform3D`, `MatrixTransform3D` и `Transform3DGroup`. Класс `TranslateTransform3D` позволяет трансформировать объект в направлениях x, y и z. `ScaleTransform3D` позволяет изменять размер объекта. С помощью класса `RotateTransform3D` вы можете вращать объект вокруг осей x, y и z. Посредством `Transform3DGroup` можно комбинировать другие трансформации.

Треугольник

Этот раздел мы начнем с простого примера 3-D. Модель 3-D состоит из треугольников, поэтому простейшая будет состоять из одного треугольника. Треугольник определяется свойством `Position` объекта `MeshGeometry3D`. Все три его точки имеют одинаковую координату z, равную –4, а координаты x/y равны –1 –1, 1 –1, и 0 1. Свойство `TriangleIndices` задает порядок позиций в порядке против часовой стрелки. Этим свойством вы можете определять видимую сторону треугольника. Одна сторона треугольника показывает цвет, заданный в свойстве `Material` класса `GeometryModel3D`, а другая сторона показывает свойство `BackMaterial`.

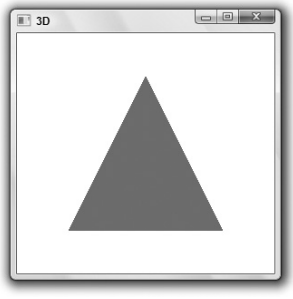

Рис. 35.23. Результирующий треугольник

Камера, используемая для отображения сценария, позиционируется в точке с координатами 0,0,0, и ориентирована в направлении 0,0,–8. Смещение позиции камеры влево приводит к перемещению прямоугольника вправо и наоборот. Изменение позиции y камеры приводит к увеличению и уменьшению прямоугольника. Свет, используемый в этой сцене — `AmbientLight` — освещает всю сцену белым цветом. На рис. 35.23 можно видеть конечный результат.

```
<Window x:Class="Triangle3D.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="3D" Height="300" Width="300">
  <Grid>
    <Viewport3D>
      <Viewport3D.Camera>
        <PerspectiveCamera Position="0 0 0" LookDirection="0 0 -8" />
      </Viewport3D.Camera>
      <ModelVisual3D>
        <ModelVisual3D.Content>
          <AmbientLight Color="White" />
        </ModelVisual3D.Content>
      </ModelVisual3D>
      <ModelVisual3D>
        <ModelVisual3D.Content>
          <GeometryModel3D>
            <GeometryModel3D.Geometry>
              <MeshGeometry3D
                   Positions="-1 -1 -4, 1 -1 -4, 0 1 -4"
                   TriangleIndices="0, 1, 2" />
            </GeometryModel3D.Geometry>
            <GeometryModel3D.Material>
              <MaterialGroup>
                <DiffuseMaterial>
                  <DiffuseMaterial.Brush>
                    <SolidColorBrush Color="Red" />
                  </DiffuseMaterial.Brush>
                </DiffuseMaterial>
              </MaterialGroup>
            </GeometryModel3D.Material>
          </GeometryModel3D>
        </ModelVisual3D.Content>
      </ModelVisual3D>
    </Viewport3D>
  </Grid>
</Window>
```

Изменение освещения

На рис. 35.23 просто показан простой треугольник — этот результат вы можете получить с меньшими усилиями, используя 2-D. Однако отсюда вы можете продолжить, используя средства 3-D. Например, изменяя свет от равномерного (ambient) к локальному (spotlight) с элементом `SpotLight`, вы можете немедленно изменить внешний вид треугольника. При локальном освещении вы определяете позицию источника света и его направление. Указывая в качестве позиции координаты –1 1 2, вы помещаете источник света над левым углом треугольника на высоту треугольника. Отсюда свет направлен вниз и влево. Вы можете видеть, как при этом выглядит треугольник, на рис. 35.24.

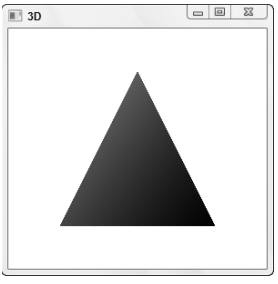

Рис. 35.24. Изменение освещения треугольника

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <SpotLight Position="-1 1 -2" Color="White"
         Direction="-1.5, -1, -5" />
  </ModelVisual3D.Content>
</ModelVisual3D>
```

Добавление текстур

Вместо использования кисти сплошного цвета для материала треугольника вы можете использовать другу кисть, такую как `LinearGradientBrush`, как показано в следующем коде XAML. Элемент `LinearGradientBrush`, определенный с `DiffuseMaterial`, задает цвета конечных точек градиента — желтый, оранжевый, красный, синий и фиолетовый. Чтобы отобразить двухмерную поверхность такого объекта, как кисть, на трехмерную геометрию, должно быть установлено свойство `TextCoordinates`. Свойство `TextCoordinates` определяет коллекцию точек 2-D, которые отображаются на позиции 3-D. На рис. 35.25 показаны двухмерные координаты кисти из примера приложения. Первая позиция в треугольнике, –1 –1, отображается на координаты кисти 0 1; позиция 1 –1, представляющая нижний правый угол, отображается на точку 1 1 кисти, имеющую фиолетовый цвет, а 0 1 отображается на 0.5 0. На рис. 35.26 изображен треугольник из материала градиентной кисти, снова с рассеянным светом.

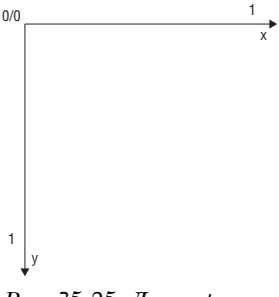

Рис. 35.25. Двухмерные координаты кисти

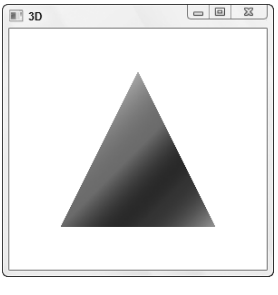

Рис. 35.26. Треугольник из материала градиентной кисти

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <GeometryModel3D>
      <GeometryModel3D.Geometry>
        <MeshGeometry3D
        Positions="-1 -1 -4, 1 -1 -4, 0 1 -4"
        TriangleIndices="0, 1, 2"
        TextureCoordinates="0 1, 1 1, 0.5 0" />
      </GeometryModel3D.Geometry>
      <GeometryModel3D.Material>
        <MaterialGroup>
          <DiffuseMaterial>
            <DiffuseMaterial.Brush>
```

```
            <LinearGradientBrush StartPoint="0,0"
               EndPoint="1,1">
              <GradientStop Color="Yellow" Offset="0" />
              <GradientStop Color="Orange" Offset="0.25" />
              <GradientStop Color="Red" Offset="0.50" />
              <GradientStop Color="Blue" Offset="0.75" />
              <GradientStop Color="Violet" Offset="1" />
            </LinearGradientBrush>
          </DiffuseMaterial.Brush>
        </DiffuseMaterial>
      </MaterialGroup>
    </GeometryModel3D.Material>
  </GeometryModel3D>
 </ModelVisual3D.Content>
</ModelVisual3D>
```

Вы можете добавить к материалам текст или другие элементы управления аналогичным образом. Чтобы сделать это, нужно просто создать `VisualBrush` *с элементами, которые вы хотите нарисовать.* `VisualBrush` *обсуждается в главе 34.*

Трехмерный объект

Теперь обратимся к реальному трехмерному объекту — ящику. Ящик сделан из пяти прямоугольников: задней, передней, левой, правой стенок и дна. Каждый прямоугольник состоит из двух треугольников, потому что именно треугольник — основа сетки. В WPF и 3-D термин *сетка* (mesh) используется для описания треугольных примитивов, используемых для построения трехмерных фигур.

Ниже приведен код прямоугольника передней стенки ящика, состоящей из двух треугольников. Позиции углов треугольников устанавливаются в порядке против часовой стрелки, как определено в TriangleIndices. Фронтальная поверхность прямоугольника рисуется красной кистью; задняя сторона — серой кистью. Обе эти кисти относятся к типу SolidColorBrush и определены в ресурсах Window.

```
<!-- Передняя стенка -->
<GeometryModel3D>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D
         Positions="-1 -1 1, 1 -1 1, 1 1 1, 1 1 1,
                   -1 1 1, -1 -1 1"
         TriangleIndices="0 1 2, 3 4 5" />
  </GeometryModel3D.Geometry>
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="{StaticResource redBrush}" />
  </GeometryModel3D.Material>
  <GeometryModel3D.BackMaterial>
    <DiffuseMaterial Brush="{StaticResource grayBrush}" />
  </GeometryModel3D.BackMaterial>
</GeometryModel3D>
```

Другой прямоугольник очень похож — отличаются только координаты вершин. Ниже показан код XAML левой стенки ящика:

```
<!-- Левая стенка -->
<GeometryModel3D>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D
         Positions="-1 -1 1, -1 1 1, -1 -1 -1, -1 -1 -1, -1 1 1, -1 1 -1"
         TriangleIndices="0 1 2, 3 4 5" />
  </GeometryModel3D.Geometry>
```

```
  <GeometryModel3D.Material>
    <DiffuseMaterial Brush="{StaticResource redBrush}" />
  </GeometryModel3D.Material>
  <GeometryModel3D.BackMaterial>
    <DiffuseMaterial Brush="{StaticResource grayBrush}" />
  </GeometryModel3D.BackMaterial>
</GeometryModel3D>
```

Код примера определяет отдельный объект `GeometryModel3D` для каждой стенки ящика. Это сделано лишь для того, чтобы код был понятнее. До тех пор, пока один и тот же материал используется для каждой стороны, можно определить сетку, содержащую все 10 треугольников для всех сторон ящика.

Все треугольники комбинируются внутри одной группы `Model3DGroup`, так что одна трансформация может быть выполнена для всех сторон ящика:

```
<!-- Модель -->
<ModelVisual3D>
  <ModelVisual3D.Content>
  <Model3DGroup>

  <!-- Элементы GeometryModel3D для каждой стенки ящика -->
  </Model3DGroup>
```

Со свойством `Transform` элемента `Model3DGroup` все геометрии внутри группы могут быть трансформированы совместно. Ниже приведен пример применения трансформации `RotateTransform3D`, определенной как `AxisAngleRotation3D`. Чтобы вращать ящик во время выполнения, свойство `Angle` привязывается к значению элемента управления `Slider`.

```
<!-- Трансформация завершенной модели -->
<Model3DGroup.Transform>
  <RotateTransform3D CenterX="0" CenterY="0" CenterZ="0">
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D x:Name="axisRotation"
           Axis="0, 0, 0"
           Angle="{Binding Path=Value,
                  ElementName=axisAngle}" />
    </RotateTransform3D.Rotation>
  </RotateTransform3D>
</Model3DGroup.Transform>
```

Чтобы увидеть наш ящик, понадобится камера. Мы используем здесь камеру `PerspectiveCamera`, чтобы ящик уменьшался по мере удаления от нее. Позиция и ориентация камеры могут быть установлены во время выполнения.

```
<!-- Камера -->
  <Viewport3D.Camera>
    <PerspectiveCamera x:Name="camera"
         Position="{Binding Path=Text,
              ElementName=textCameraPosition}"
         LookDirection="{Binding Path=Text,
              ElementName=textCameraDirection}" />
  </Viewport3D.Camera>
```

Приложение использует два разных источника света. Один из них —DirectionalLight:

```
<!-- Направленный свет -->
<ModelVisual3D>
  <ModelVisual3D.Content>
    <DirectionalLight Color="White" x:Name="directionalLight">
      <DirectionalLight.Direction>
        <Vector3D X="1" Y="2" Z="3" />
```

```
      </DirectionalLight.Direction>
    </DirectionalLight>
  </ModelVisual3D.Content>
</ModelVisual3D>
```

Другой источник света — `SpotLight`. С этим источников света можно подсветить определенную часть ящика. `SpotLight` определяет свойства `InnerConeAngle` и `OuterConeAngle` для задания освещенной области:

```
<!-- Точечный свет -->
<ModelVisual3D>
  <ModelVisual3D.Content>
    <SpotLight x:Name="spotLight"
    InnerConeAngle="{Binding Path=Value,
                    ElementName=spotInnerCone}"
    OuterConeAngle="{Binding Path=Value,
                    ElementName=spotOuterCone}"
    Color="#FFFFFF"
    Direction="{Binding Path=Text, ElementName=spotDirection}"
    Position="{Binding Path=Text, ElementName=spotPosition}"
    Range="{Binding Path=Value, ElementName=spotRange}" />
  </ModelVisual3D.Content>
</ModelVisual3D>
```

При запуске приложения вы можете изменять угол поворота ящика, камеры и источников света, как показано на рис. 35.27.

> Создать трехмерную модель, состоящую только из прямоугольников и треугольников, довольно просто. Но более сложные модели вам не придется создавать вручную — для этого можно использовать один из нескольких инструментов. Инструменты 3-D для WPF вы найдете по адресу `www.codeplex/3DTools`.

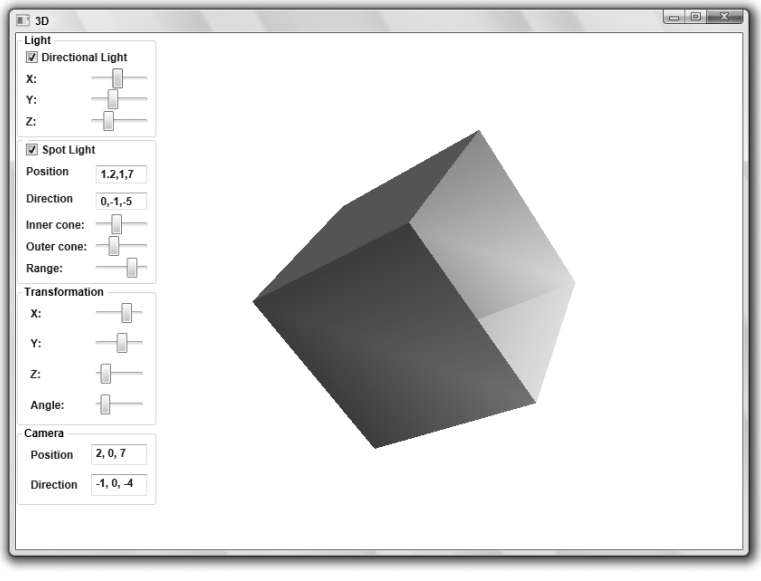

Рис. 35.27. Пример приложения с ящиком

Интеграция с Windows Forms

Вместо того чтобы переписывать пользовательский интерфейс с нуля на WPF, вы можете использовать существующие элементы управления Windows Forms внутри приложения WPF, а также создавать новые элементы управления WPF для применения в приложениях Windows Forms. Наилучший способ интеграции Windows Forms и WPF — создание элементов управления и интеграция элементов управления в приложения, использующие другую технологию.

Интеграция Windows Forms и WPF сталкивается с серьезным препятствием. Если вы интегрируете Windows Forms с WPF, то элементы управления Windows Forms выглядят так, как они выглядели в прежние времена. Элементы управления и приложения Windows Forms не могут иметь современный внешний вид WPF. Поэтому с точки зрения пользовательского интерфейса может быть лучше полностью переписать пользовательский интерфейс.

Чтобы интегрировать Windows Forms и WPF, вам понадобятся классы из пространства имен `System.Windows.Forms.Integration`, находящиеся в сборке `WindowsFormsIntegration`.

Элементы управления WPF в приложениях Windows Forms

Вы можете использовать элементы управления WPF внутри приложения Windows Forms. WPF-элемент — это нормальный класс .NET. Однако вы не можете использовать его напрямую из кода Windows Forms; элемент управления WPF не является элементом управления Windows Forms. Интеграция может быть осуществлена с применением класса-оболочки `ElementHost` из пространства имен `System.Windows.Forms.Integration`. Класс `ElementHost` — это элемент управления Windows Forms, поскольку он унаследован от `System.Windows.Forms.Control` и может использоваться подобно любому другому элементу управления Windows Forms в приложении Windows Forms. `ElementHost` принимает и управляет элементами управления WPF.

Начнем с простейшего элемента управления WPF. С помощью Visual Studio 2008 вы можете создать библиотеку WPF User Control Library. Наш пример элемента управления унаследован от базового класса `UserControl` и содержит сетку (grid) и кнопку со специальным содержимым.

```
<UserControl x:Class="WPFControl.UserControl1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
<Grid>
  <Button>
     <Canvas Height="230" Width="230">
       <Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
            Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
       <Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
            Stroke="Blue" StrokeThickness="3" Fill="White" />
       <Ellipse Canvas.Left="70" Canvas.Top="75" Width="5" Height="5" Fill="Black" />
       <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
            Data="M 62,125 Q 95,122 102,108" />
       <Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue" StrokeThickness="4" />
       <Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue" StrokeThickness="4" />
       <Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue" StrokeThickness="4" />
       <Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue" StrokeThickness="4" />
     </Canvas>
  </Button>
</Grid>
</UserControl>
```

Вы можете создать приложение Windows Forms, выбрав шаблон приложения Windows Forms. Поскольку проект пользовательского элемента управления WPF находится в том же решении, что и приложение Windows Forms, вы можете перетащить пользовательский элемент управления WPF из панели инструментов на поверхность проектирования приложения Windows Forms. Это добавит ссылку на сборки `PresentationCore`, `PresentationFramework`, `WindowsBase`, `WindowsFormsIntegration` и, конечно же, на сборку, содержащую элемент управления WPF.

Внутри сгенерированного дизайнером кода вы найдете переменную, ссылающуюся на пользовательский элемент WPF и объект типа `ElementHost`, который служит для него оболочкой:

```
private System.Windows.Forms.Integration.ElementHost elementHost1;
private WPFControl.UserControl1 userControl11;
```

В методе `InitializeComponent` вы можете видеть инициализацию объекта и присвоение экземпляра элемента управления WPF свойству `Child` класса `ElementHost`:

```
private void InitializeComponent()
{
  this.elementHost1 = new
       System.Windows.Forms.Integration.ElementHost();
  this.userControl11 = new WPFControl.UserControl1();
  this.SuspendLayout();
  //
  // elementHost1
  //
  this.elementHost1.Location = new System.Drawing.Point(39, 44);
  this.elementHost1.Name = "elementHost1";
  this.elementHost1.Size = new System.Drawing.Size(259, 229);
  this.elementHost1.TabIndex = 0;
  this.elementHost1.Text = "elementHost1";
  this.elementHost1.Child = this.userControl11;
  //...
}
```

Запустив это приложение Windows Forms, вы увидите рядом внутри формы элемент управления WPF и обычный элемент управления Windows Forms, как показано на рис. 35.28.

Конечно же, вы можете добавлять методы, свойства и события в элемент управления WPF и использовать их точно так же, как и во всех других элементах управления.

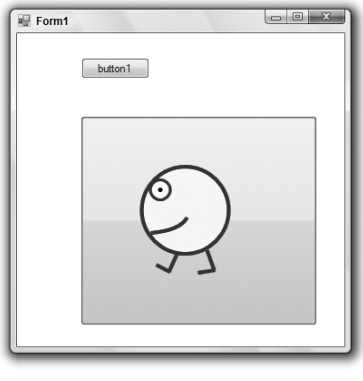

Рис. 35.28. Пример применения элементов управления WPF в приложении Windows Forms

Элементы управления Windows Forms в приложениях WPF

Вы можете интегрировать Windows Forms и WPF также и в противоположном направлении: помещая элемент управления Windows Forms в приложение WPF. Как и в случае использования класса `ElementHost` для создания оболочки для элемента управления WPF внутри приложения Windows Forms, здесь вам также понадобится класс, позволяющий поместить элемент управления Windows Forms в среду приложения WPF. Этот класс называется `WindowsFormsHost` и находится он в той же сборке `WindowsFormsIntegration`. Класс `WindowsFormsHost` унаследован от базовых классов `HwndHost` и `FrameworkElement`, а потому может быть использован как элемент WPF.

Для такой интеграции сначала создается библиотека Windows Control Library. Добавьте элементы `TextBox` и `Button` к форме, используя визуальный дизайнер. Чтобы изменить свойство `Text` кнопки, в отделенный код добавляется свойство `ButtonText`:

```
public partial class UserControl1 : UserControl
 {
 public UserControl1()
 {
  InitializeComponent();
 }
 public string ButtonText
 {
  get { return button1.Text; }
  set { button1.Text = value; }
 }
}
```

В приложении WPF вы можете добавить объект `WindowsFormsHost` из панели инструментов дизайнера. Это добавит ссылки на сборки `WindowsFormsIntegration`, `System.Windows.Forms`, а также на сборку элемента управления Windows Forms. Чтобы использовать элемент управления Windows Forms из XAML, вам нужно добавить псевдоним пространства имен XML, ссылающийся на пространство имен .NET. Поскольку сборка, содержащая элемент управления Windows Forms, находится отдельно от сборки приложения WPF, вы также должны добавить имя сборки в псевдоним пространства имен. Элемент управления Windows Forms теперь может содержаться внутри элемента `WindowsFormsHost`, как показано ниже. Вы можете присвоить значение свойству `ButtonText` непосредственно из XAML, подобно элементам .NET.

```
<Window x:Class="WPFApplication.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:winforms="clr-namespace:Wrox.ProCSharp.WPF;assembly=WindowsFormsControl"
Title="WPF Interop Application" Height="300" Width="300"
>
<Grid>
  <Grid.RowDefinitions>
     <RowDefinition />
     <RowDefinition />
  </Grid.RowDefinitions>
  <WindowsFormsHost Grid.Row="0" Height="180">
     <winforms:UserControl1 x:Name="myControl" ButtonText="Click Me!" />
  </WindowsFormsHost>
  <StackPanel Grid.Row="1">
     <TextBox Margin="5,5,5,5" Width="140"
     Height="30"></TextBox>
     <Button Margin="5,5,5,5" Width="80" Height="40">
     WPF Button</Button>
  </StackPanel>
</Grid>
</Window>
```

На рис. 35.29 показано, как выглядит приложение WPF. Разумеется, элемент управления Windows Forms по-прежнему выглядит как элемент управления Windows Forms и не обладает возможностями изменения размеров и стилизации, которые предоставляет WPF.

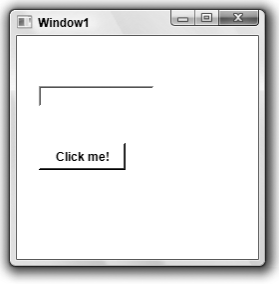

Рис. 35.29. Пример применения элементов управления Windows Forms в приложении WPF

Браузерное приложение WPF

Visual Studio 2008 имеет еще один шаблон проекта WPF — WPF Browser Application. Такое приложение может выполняться внутри Internet Explorer, но при этом на клиентской системе должна быть инсталлирована соответствующая версия .NET Framework. Здесь вы привносите средства "толстого клиента" в браузер. Однако для WPF Browser Application на клиентской системе необходима .NET Framework и поддерживается только браузер Internet Explorer.

При создании проекта этого типа создается файл XBAP (XAML Browser Application). XBAP — это XML-файл, определяющий приложение и сборки, в него входящие, для развертывания ClickOnce.

Приложение XABP — это приложение ограниченного доверия. Вы можете использовать только тот код .NET, который доступен с учетом прав доступа Internet.

ClickOnce рассматривается в главе 16.

Браузерные приложения WPF отличаются от Silverlight, определяющее подмножество WPF, которое не требует инсталляции .NET Framework на клиентской системе. Silverlight требует дополнения (add-in) к браузеру и поддерживает различные браузеры и различные операционные системы. Silverlight 1.0 не предусматривает программирования с использованием .NET; вы можете применять только JavaScript для программного доступа к элементам XAML. Silverlight 1.1 будет поддерживать .NET Microframework.

Резюме

В настоящей главе рассматривались еще некоторые средства WPF.

Привязка данных WPF сделала значительный скачок вперед по сравнению с тем, что было в Windows Forms. Вы можете привязать любое свойство класса .NET к свойству элемента WPF. Режим привязки определяет ее направление. Вы можете привязывать объекты .NET и списки, а также определять шаблон данных для создания внешнего вида по умолчанию для класса .NET с шаблоном данных.

Привязка команд делает возможным отображение кода обработчика на меню и панели инструментов. Вы также увидели, как легко реализовать копирование и вставку текста в WPF, поскольку обработчик команд для этой технологии уже включен в элемент управления `TextBox`.

Анимация позволяет пользователю динамически изменять каждое свойство элемента WPF. Анимации могут сдержанными и не раздражающими, при этом обеспечивая более высокую отзывчивость пользовательского интерфейса и его привлекательность для пользователя.

WPF также позволяет легко отображать трехмерные изображения на двухмерную поверхность экрана. Вы увидели в этой главе, как создается трехмерная модель и ее представление посредством разных источников света и камер.

В этой и предыдущей главах был представлен обзор WPF и приведено достаточно информации для того, чтобы приступить к использованию этой технологии. Дополнительную информацию о WPF вы найдете в книгах, посвященных этой технологии, например, *WPF: Windows Presentation Foundation в .NET 3.5 с примерами на C# 2008 для профессионалов, 2-е издание* (ИД "Вильямс", 2008 г.).