

Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ТЕХНОЛОГИЯ WINDOWS PRESENTATION FOUNDATION

Курс лекций
для студентов специальности
1-40 01 03 Информатика и технологии программирования

Минск 2013

Содержание

1. Общая характеристика технологии WPF.....	3
2. Простейшее приложение WPF.....	4
3. XAML	6
4. Базовые концепции WPF	12
5. Варианты организации приложений WPF	19
6. Компоновка.....	26
7. Обзор элементов управления WPF	37
8. Ресурсы.....	51
9. Привязка данных	54
10. Работа с графикой	69
11. Стили и триггеры	84
12. Шаблоны	90
13. Списки и представления коллекций.....	95
Литература	108

1. Общая характеристика технологии WPF

Windows Presentation Foundation (WPF) – это технология для построения клиентских приложений Windows, являющаяся частью платформы .NET. WPF разработана как альтернатива технологии Windows Forms. Ниже перечислены основные особенности технологии WPF.

1. Собственные методы построения и рендеринга элементов. В Windows Forms классы для элементов управления делегируют функции отображения системным библиотекам, таким как user32.dll. В WPF любой элемент управления полностью строится (рисует) самой WPF. Для аппаратного ускорения рендеринга применяется технология DirectX (рис. 1).

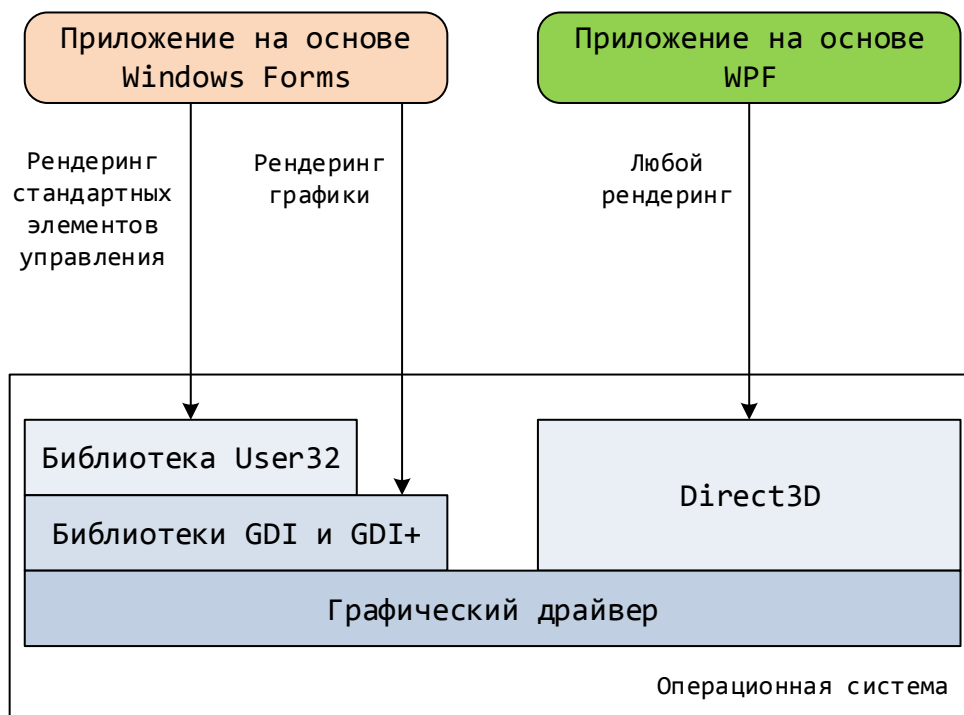


Рис. 1. Рендеринг в приложениях на основе Windows Forms и WPF.

2. Независимость от разрешения устройства вывода. Для указания размеров в WPF используется собственная единица измерения, равная 1/96 дюйма. Кроме этого, технология WPF ориентирована на использование не пиксельных, а векторных примитивов.

3. Декларативный пользовательский интерфейс. В WPF визуальное содержимое окна можно полностью описать на языке XAML. Это язык разметки, основанный на XML. Так как описание пользовательского интерфейса отделено от кода, дизайнеры могут использовать профессиональные инструменты (например, Microsoft Expression Blend), чтобы редактировать файлы XAML, улучшая внешний вид приложения. Применение XAML является предпочтительным, но не обязательным – приложения WPF можно создавать, используя только код.

4. Веб-подобная модель компоновки. WPF поддерживает гибкий визуальный поток, размещающий элементы управления на основе их содержимого. В

результате получается пользовательский интерфейс, который может быть адаптирован для отображения динамичного содержимого.

5. Стили и шаблоны. Стили стандартизируют форматирование и позволяют повторно использовать его по всему приложению. Шаблоны дают возможность изменить способ отображения любых элементов управления, даже таких основополагающих, как кнопки или поля ввода.

6. Анимация. В WPF анимация – неотъемлемая часть программного каркаса. Анимация определяется декларативными дескрипторами, и WPF запускает её в действие автоматически.

7. Приложения на основе страниц. В WPF можно строить приложения с кнопками навигации, которые позволяют перемещаться по коллекции страниц. Кроме этого, специальный тип приложения WPF – ХВАР – может быть запущен внутри браузера.

2. Простейшее приложение WPF

Построим простейшее однооконное приложение WPF. Для этого создадим файл `Program.cs` и поместим в него следующий код:

```
using System;
using System.Windows;

public class Program
{
    [STAThread]
    public static void Main()
    {
        var myWindow = new Window();
        myWindow.Title = "WPF Program";
        myWindow.Content = "Hello, world!";
        var myApp = new Application();
        myApp.Run(myWindow);
    }
}
```

Проанализируем этот код. Пространство имён `System.Windows` содержит классы `Window` и `Application`, описывающее окно и приложение соответственно. Точка входа помечена атрибутом `[STAThread]`. Это обязательное условие для любого приложения WPF, оно связано с моделью многопоточности WPF. В методе `Main()` создаётся и настраивается объект окна, затем создаётся объект приложения. Вызов метода `Run()` приводит к отображению окна и запуску цикла обработки событий (окно ждёт действий пользователя). Чтобы скомпилировать приложение, необходимо указать ссылки на стандартные сборки `PresentationCore.dll`, `PresentationFramework.dll`, `System.Xaml.dll` и `WindowsBase.dll`.

Отметим, что приложение допускает другую организацию. Вместо настройки объекта класса `Window` можно создать наследник этого класса и выполнить настройку в конструкторе наследника или в специальном методе:

```
// наследник класса Window, описывающий пользовательское окно
public class MainWindow : Window
{
    public MainWindow()
    {
        Title = "WPF Program";
        Content = "Hello, world";
    }
}
```

В Visual Studio приложениям WPF соответствует отдельный шаблон проекта. Этот шаблон ориентирован на использование XAML, поэтому в случае однооконного приложения будет создан следующий набор файлов:

- файл `MainWindow.xaml.cs` на языке C# и `MainWindow.xaml` на языке XAML описывают класс `MainWindow`, являющийся наследником класса `Window`;
- файлы `App.xaml.cs` и `App.xaml` описывают класс `App`, наследник класса `Application`.

Ниже приведён файл `MainWindow.xaml` для простейшего окна:

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF Program" Height="250" Width="400">

    <!-- содержимое окна -->
    Hello, world
</Window>
```

Visual Studio выполняет компиляцию проекта, созданного по шаблону WPF, в два этапа. Вначале для каждого файла XAML генерируется два файла, сохраняемых в подкаталогах `obj\Debug` или `obj\Release` (в зависимости от цели компиляции):

1. файл с расширением `*.baml` (BAML-файл) – двоичное представление XAML-файла, внедряемое в сборку в виде ресурса;
2. файл с расширением `*.g.cs` – разделяемый класс, который соответствует XAML-описанию. Этот класс содержит поля для всех именованных элементов XAML и реализацию метода `InitializeComponent()`, загружающего BAML-данные из ресурсов сборки. Кроме этого, класс содержит метод, подключающий все обработчики событий.

На втором этапе сгенерированные файлы компилируются вместе с исходными файлами C# в единую сборку (рис. 2).

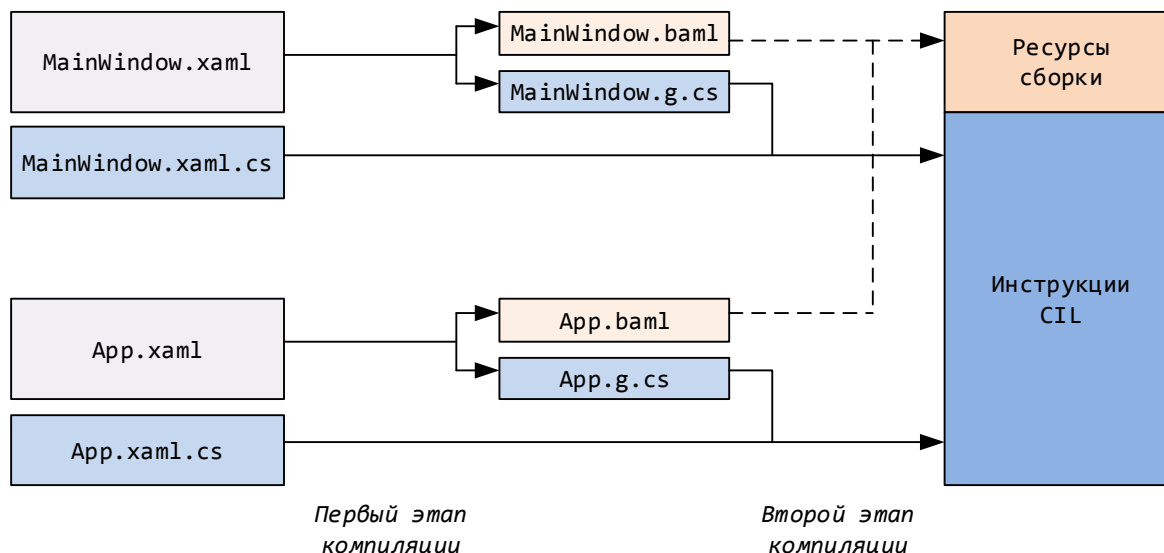


Рис. 2. Компиляция приложения WPF в Visual Studio.

3. XAML

Расширяемый язык разметки приложений (eXtensible Application Markup Language, XAML¹) – это язык для представления дерева объектов .NET, основанный на XML. Данные XAML превращаются в дерево объектов при помощи *анализатора XAML* (XAML parser). Основное назначение XAML – описание пользовательских интерфейсов в приложениях WPF. Однако XAML используется и в других технологиях, в частности, в Silverlight.

Рассмотрим основные правила XAML. Документ XAML записан в формате XML. Это означает, что имена элементов XAML чувствительны к регистру, нужна правильная вложенность элементов, а некоторые символы требуют особого обозначения (например, `&` – это символ `&`). Кроме этого, XAML по умолчанию игнорирует лишние пробельные символы (однако это поведение изменяется установкой у элемента атрибута `xml:space="preserve"`).

Объектные элементы XAML описывают объект некоторого типа платформы .NET и задают значения открытых свойств и полей объекта. Имя элемента указывает на тип объекта. Ниже приведено описание XAML для объекта класса `Button` (кнопка), а также эквивалентный код на языке C#:

```

<!-- определение объекта в XAML -->
<Button Width="100">
    I am a Button
</Button>

// определение объекта в коде
Button b = new Button();
b.Width = 100;
b.Content = "I am a Button";

```

¹ Произносится как ['zæməl].

Типы .NET обычно вложены в пространства имён. В XAML пространству имён .NET ставится в соответствие пространство имён XML. Для этого используется следующий синтаксис:

```
xmlns:префикс="clr-namespace:пространство-имён"
```

При необходимости указывается сборка, содержащая пространство имён:

```
xmlns:префикс="clr-namespace:пространство-имён;assembly=имя-сборки"
```

Для нужд WPF зарезервировано два пространства имён XML:

1. <http://schemas.microsoft.com/winfx/2006/xaml/presentation> — обычно является пространством имён по умолчанию (указывается без префикса) и соответствует набору пространств имён .NET с типами WPF (эти пространства имён имеют вид `System.Windows.*`).

2. <http://schemas.microsoft.com/winfx/2006/xaml> — отвечает пространству имён `System.Windows.Markup`, а также позволяет выделить директивы (указания) для анализатора XAML. Пространству имён анализатора XAML по традиции ставят в соответствие префикс `x`.

```
<!-- у корневого элемента Window заданы три пространства имён -->
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib">
```

Для установки значений свойств объекта в XAML можно использовать атрибуты XML, элементы свойств и содержимое элемента. При использовании *атрибутов* указывается имя свойства и значение свойства в виде строки:

```
<!-- задаём у кнопки красный фон -->
<Button Background="Red" />
```

Анализатор XAML применяет для преобразования строки в значение свойства специальные *конвертеры типов* (конвертеры не используются для строк, чисел и элементов перечислений). Приведённый выше фрагмент XAML эквивалентен следующему коду на C#:

```
// TypeConverter и TypeDescriptor определены в System.ComponentModel
var b = new Button();
TypeConverter convert = TypeDescriptor.GetConverter(typeof (Brush));
b.Background = (Brush) convert.ConvertFromInvariantString("Red");
```

Платформа .NET содержит более ста стандартных конвертеров. При необходимости можно разработать собственный конвертер, используя базовый класс `TypeConverter`.

Элемент *свойства* вложен в объектный элемент и имеет вид `<имя-типа.имя-свойства>`. Содержимое элемента свойства рассматривается как значение свойства (при необходимости применяются конвертеры). Обычно элементы свойств используются для значений, являющихся объектами.

```
<Button>
  <Button.Width>100</Button.Width>
  <Button.Background>Red</Button.Background>
</Button>
```

Тип, соответствующий объектному элементу, может быть помечен атрибутом `[ContentProperty]` с указанием имени *свойства содержимого*. В этом случае анализатор XAML рассматривает содержимое объектного элемента (за исключением элементов свойств) как значение для свойства содержимого. Например, в классе `ContentControl` (он является базовым для класса `Button`) свойством содержимого является `Content`:

```
[System.Windows.Markup.ContentProperty("Content")]
public class ContentControl
{
    public object Content { get; set; }
    // другие элементы класса ContentControl не показаны
}
```

Это означает, что следующие два фрагмента XAML эквивалентны:

```
<Button Content="Click me!" />

<Button>Click me!</Button>
```

Если тип реализует интерфейсы `IList` или `IDictionary`, при описании объекта этого типа в XAML дочерние элементы автоматически добавляются в соответствующую коллекцию. Например, свойство `Items` класса `ListBox` имеет тип `ItemCollection`, а этот класс реализует интерфейс `IList`:

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

Кроме этого, `Items` — это свойство содержимого для `ListBox`, а значит, приведённое XAML-описание можно упростить:

```
<ListBox>
  <ListBoxItem Content="Item 1" />
  <ListBoxItem Content="Item 2" />
</ListBox>
```


Во всех предыдущих примерах использовалось конкретное указание значения свойства. Механизм *расширений разметки* (markup extensions) позволяет вычислять значение свойства при преобразовании XAML в дерево объектов. Технически, любое расширение разметки – это класс, унаследованный от `System.Windows.Markup.MarkupExtension` и перекрывающий функцию `ProvideValue()`. Встретив расширение разметки, анализатор XAML генерирует код, который создаёт объект расширения разметки и вызывает `ProvideValue()` для получения значения. Приведём пример расширения разметки:

```
using System;
using System.Windows.Markup;

namespace MarkupExtensions
{
    public class ShowTimeExtension : MarkupExtension
    {
        public string Header { get; set; }

        public ShowTimeExtension() { }

        public override object ProvideValue(IServiceProvider sp)
        {
            return string.Format("{0}: {1}", Header, DateTime.Now);
        }
    }
}
```

Если расширение разметки используется в XAML как значение атрибута, то оно записывается в фигурных скобках (если имя расширения имеет суффикс *Extension*, этот суффикс можно не указывать). В фигурных скобках также перечисляются через запятую аргументы конструктора расширения и пары для настройки свойств расширения в виде *свойство=значение*.

```
<Window xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:local="clr-namespace:MarkupExtensions">
    <StackPanel>
        <Button Content="{local:ShowTime}" />
        <Button Content="{local:ShowTime Header=Time}" />
    </StackPanel>
</Window>
```

Расширения разметки могут применяться как значения элементов свойств:

```
<Button>
    <Button.Content>
        <local:ShowTime Header="Time" />
    </Button.Content>
</Button>
```

В табл. 1 представлены стандартные расширения разметки, доступные после подключения пространства имён `System.Windows.Markup`.

Таблица 1

Расширения разметки из `System.Windows.Markup`

Имя	Описание, пример использования
<code>x:Array</code>	Представляет массив. Дочерние элементы становятся элементами массива <pre><x:Array Type="{x:Type Button}"> <Button /> <Button /> </x:Array></pre>
<code>x:Null</code>	Представляет значение <code>null</code> <pre>Style="{x:Null}"</pre>
<code>x:Reference</code>	Используется для ссылки на ранее объявленный элемент <pre><TextBox Name="customer" /> <Label Target="{x:Reference customer}" /></pre>
<code>x:Static</code>	Представляет статическое свойство, поле или константу <pre>Height="{x:Static SystemParameters.IconHeight}"</pre>
<code>x:type</code>	Аналог применения оператора <code>typeof</code> из языка C#

Рассмотрим некоторые директивы анализатора XAML, применяемые в WPF. Анализатор генерирует код, выполняющий по документу XAML создание и настройку объектов. Действия с объектами (в частности, обработчики событий) обычно описываются в отдельном классе. Чтобы связать этот класс с документом XAML используется директива-атрибут `x:Class`. Этот атрибут применяется только к корневому элементу и содержит имя класса, являющегося наследником класса корневого элемента:

```
<!-- у корневого элемента Window задан атрибут x:Class -->
<Window x:Class="WpfApplication.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

Чтобы сослаться на объект в коде, этот объект должен иметь имя. Для указания имени объекта используется директива-атрибут `x:Name`:

```
<Button x:Name="btn" Content="Click me!" />
```

Заметим, что многие элементы управления WPF имеют свойство `Name`. Анализатор XAML использует соглашение, по которому задание свойства `Name` эквивалентно указанию директивы-атрибута `x:Name`.

Существует возможность встроить фрагмент кода в XAML-файл. Для этого используется директива-элемент `x:Code`. Такой элемент должен быть непосредственно вложен в корневой элемент, у которого имеется атрибут `x:Class`.

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

  <Button x:Name="btn" Click="btn_click" Content="Click me!" />
```

```

<x:Code>
  <![CDATA[
    void btn_click(object sender, RoutedEventArgs e)
    {
        btn.Content = "Inline Code Works!";
    }
  ]]>
</x:Code>
</Window>

```

Директива-атрибут `x:Key` применяется при описании дочерних элементов объекта-словаря, и позволяет указать ключ словаря для элемента:

```

<Window.Resources>
  <SolidColorBrush x:Key="borderBrush" Color="Red" />
  <SolidColorBrush x:Key="textBrush" Color="Black" />
</Window.Resources>

```

В .NET Framework 4.0 была представлена новая версия XAML, известная как *XAML 2009*. XAML 2009 пока не используется при описании интерфейсов WPF-приложений, поэтому только упомянем его основные особенности:

- полная поддержка универсальных типов (generics);
- работа с базовыми типами .NET (числа, строки) без подключения дополнительных пространств имён;
- создание объектов, используя вызов конструктора с параметрами;
- создание объектов путём вызова заданного фабричного метода;
- расширенное управление обработчиками событий.

Платформа .NET включает классы, образующие программный интерфейс для работы с XAML. Большинство классов принадлежит пространству имён `System.Xaml` и находится в одноимённой сборке. Статический класс `XamlServices` содержит методы для сериализации объектов в формате XAML. Классы `XamlReader`, `XamlWriter` и их наследники дают доступ к структуре XAML-данных. Класс `System.Windows.Markup.XamlReader` осуществляет загрузку XAML и порождает соответствующее дерево объектов.

```

public class Book
{
    public string Name { get; set; }
    public int ISBN { get; set; }
}

// сериализация объекта класса Book в XAML-формате
var book = new Book {Name = "First", ISBN = 123};
XamlServices.Save("book.xaml", book);

// создание WPF-окна на основе XAML-описания
Window window = null;
using (Stream s = File.OpenRead("MyWindow.xaml"))

```

```

{
    // предполагается, что тип корневого элемента известен
    window = (Window) System.Windows.Markup.XamlReader.Load(s);
}

```

4. Базовые концепции WPF

4.1. Иерархия классов

Типы, связанные с технологией WPF, сгруппированы в несколько сборок. Сборка `PresentationFramework.dll` содержит классы верхнего уровня – окна, панели, элементы управления. В этой сборке находятся типы для реализации высокоуровневых программных абстракций, например, стилей. Сборка `PresentationCore.dll` содержит базовые классы, от которых унаследованы все фигуры и элементы управления. В `WindowsBase.dll` описаны ещё более базовые ингредиенты, которые потенциально могут использоваться вне WPF. Кроме этого, частью WPF является библиотека `milcore.dll`, написанная на неуправляемом коде. Функции библиотеки `milcore.dll` транслируют визуальные элементы в примитивы `Direct3D`.

Рассмотрим основу иерархии классов WPF (рис. 3).

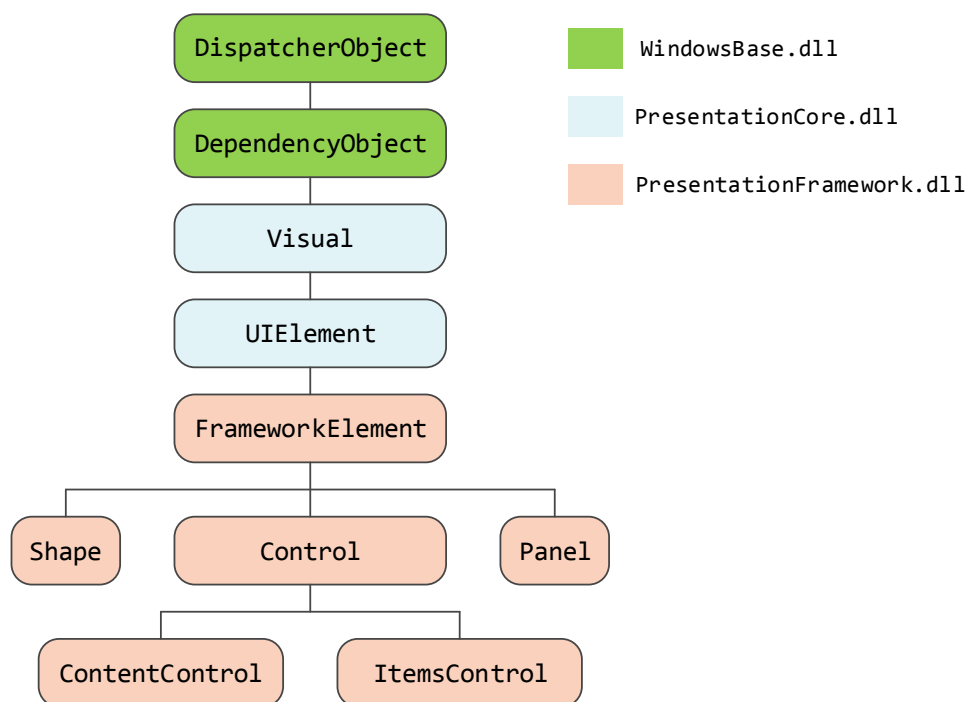


Рис. 3. Фундаментальные классы WPF.

1. `System.Threading.DispatcherObject`. Приложения WPF используют *однопоточную модель* (single-thread affinity, STA) – весь пользовательский интерфейс принадлежит единственному потоку. Чтобы содействовать работе модели STA, каждое приложение WPF управляется диспетчером, координирующим обработку сообщений. Будучи унаследованным от `DispatcherObject`, объект может удостовериться, выполняется ли его код в правильном потоке, и обратиться к диспетчеру, чтобы направить код в поток интерфейса.

2. `System.Windows.DependencyObject`. WPF поддерживает мощную модель *свойств зависимостей* (dependency property), которая положена в основу таких средств, как уведомления об изменениях, наследуемые значения по умолчанию и экономное хранение информации свойств. Наследуясь от `DependencyObject`, классы WPF получают поддержку свойств зависимости.

3. `System.Windows.Media.Visual`. Любой класс, унаследованный от `Visual`, обладает способностью отображаться в окне. Класс `Visual` инкапсулирует инструкции рисования, включая отсечения, прозрачность и настройки трансформации фигур. Этот класс также обеспечивает связь между управляемыми библиотеками WPF и библиотекой `milcore.dll`.

4. `System.Windows.UIElement`. Этот класс добавляет поддержку таких сущностей WPF, как *компоновка, ввод, фокус и события* (layout, input, focus, events – LIFE). В `UIElement` определён двухэтапный процесс измерения и организации компоновки. Этот класс вводит поддержку расширенной системы передачи событий, именуемой *маршрутизируемыми событиями* (routed events).

5. `System.Windows.FrameworkElement`. Класс `FrameworkElement` добавляет поддержку привязки данных, анимации, стилей и ресурсов. Этот класс также реализует некоторые абстрактные концепции из `UIElement`.

6. `System.Windows.Controls.Control`. *Элемент управления* (control) – это класс, который может взаимодействовать с пользователем. Примерами элементов управления являются `TextBox` (поле для ввода текста) и `Button` (кнопка). Класс `Control` добавляет к `FrameworkElement` свойства для установки шрифта, а также цветов переднего плана и фона. Но наиболее интересная деталь, которую он предоставляет – это поддержка *шаблонов*, которые позволяют заменять стандартный внешний вид элемента управления вашим собственным.

7. `System.Windows.Controls.ContentControl`. Это базовый класс для всех элементов управления, которые имеют отдельный фрагмент содержимого. Этот фрагмент может быть чем угодно – от обычной строки до панели компоновки, содержащей комбинацию фигур и других элементов управления.

8. `System.Windows.Controls.ItemsControl`. Родительский класс для всех элементов управления, которые отображают коллекцию каких-то единиц информации (например, `ListBox` и `TreeView`). Списковый элемент управления гибок – используя встроенные средства класса `ItemsControl`, можно трансформировать обычный `ListBox` в список переключателей, список флажков, ряд картинок или комбинацию разных элементов по своему выбору.

9. `System.Windows.Shapes.Shape`. От этого класса наследуются базовые фигуры графики, такие как `Rectangle`, `Polygon`, `Ellipse`, `Line` и `Path`. Эти фигуры могут быть использованы наряду с более традиционными визуальными элементами вроде кнопок и текстовых полей.

10. `System.Windows.Controls.Panel`. Это базовый класс для всех контейнеров компоновки – элементов, которые содержат в себе один или более дочерних элементов и упорядочивают их в соответствии с определёнными правилами размещения.

4.2. Свойства зависимостей и присоединённые свойства

Свойства зависимостей (dependency properties) – новая модель свойств, используемая в WPF для реализации следующих механизмов:

- уведомление об изменении значения свойства;
- сброс свойства к значению по умолчанию;
- наследование значения свойства дочерними объектами;
- эффективная модель хранения данных свойства;
- привязка свойства к данным произвольного объекта.

При вычислении значения любого свойства зависимостей WPF использует алгоритм, состоящий из следующих пяти шагов.

Шаг 1. Определение базового значения. Этот шаг подразумевает использование следующих источников, указанных в порядке убывания приоритета:

- локально указанное (в разметке или коде) значение;
- значение, полученное от триггера стиля;
- значение, полученное от триггера шаблона;
- значение, заданное в стиле;
- значение, полученное от триггера общего стиля;
- значение, заданное в общем стиле;
- значение, унаследованное от родительского элемента;
- значение по умолчанию.

Шаг 2. Вычисление значения. Если базовое значение является выражением привязки данных или ресурсов, производится вычисление выражения.

Шаг 3. Применение анимации. Если со свойством связана анимация, значение вычисляется по алгоритму этой анимации.

Шаг 4. Выполнение функции CoerceValueCallback. Эта функция может быть определена при регистрации свойства зависимостей. Её назначение – корректировка значения свойства на основании пользовательского алгоритма.

Шаг 5. Проверка значения. Если при регистрации свойства была задана функция `ValidateValueCallback`, она вызывается для проверки значения. В случае провала проверки генерируется исключение.

Рассмотрим схему описания свойства зависимостей в пользовательском типе. Эта схема включает три этапа:

1. Создание объекта, характеризующего метаданные свойства зависимостей. Этот этап является необязательным – для метаданных могут использоваться значения по умолчанию.

2. Регистрация свойства зависимостей.

3. Создание экземплярной оболочки для свойства зависимостей. Это также необязательный этап.

Объект метаданных свойства зависимостей – это экземпляр класса `PropertyMetadata` или его наследников. В табл. 2 представлены некоторые элементы такого наследника – класса `FrameworkPropertyMetadata`.

Элементы метаданных, доступные в `FrameworkPropertyMetadata`

Имя	Описание
<code>AffectsArrange</code> , <code>AffectsMeasure</code>	Если эти свойства установлены в <code>true</code> , свойство зависимостей влияет на расположение соседних элементов при компоновке
<code>AffectsRender</code>	Если установлено в <code>true</code> , свойство зависимостей может повлиять на способ рендеринга элемента
<code>BindsTwoWayByDefault</code>	Если установлено в <code>true</code> , свойство зависимостей использует по умолчанию двухстороннюю привязку данных
<code>Inherits</code>	Задаёт возможность наследования значения для свойства зависимостей дочерними элементами в дереве элементов
<code>IsAnimationProhibited</code>	Если установлено в <code>true</code> , свойство зависимостей не может использоваться в анимации
<code>IsNotDataBindable</code>	Если установлено в <code>true</code> , свойство зависимостей не может быть задано в выражении привязки данных
<code>DefaultValue</code>	Значение по умолчанию для свойства зависимостей
<code>CoerceValueCallback</code>	Обеспечивает обратный вызов, при котором производится попытка исправить значение свойства перед его проверкой
<code>PropertyChangedCallback</code>	Функция обратного вызова, который производится в случае изменения значения свойства

Любое свойство зависимостей должно быть зарегистрировано перед использованием. Для регистрации свойства необходимо создать статическое поле с типом `DependencyProperty`, которое инициализируется при объявлении или в статическом конструкторе. Экземпляр `DependencyProperty` возвращается статическим методом `DependencyProperty.Register()`. Простейшая перегрузка метода `Register()` требует задания имени и типа свойства зависимостей, а также типа владельца свойства. Кроме этого, может быть указан объект метаданных свойства зависимостей и функция обратного вызова `ValidateValueCallback`.

Ниже приведён фрагмент кода, демонстрирующий регистрацию свойства зависимостей в некотором классе. Обратите внимание на используемое соглашение об именах – суффикс *Property* в имени поля для свойства зависимостей.

```
public class User : DependencyObject
{
    public static readonly DependencyProperty NameProperty;

    static User()
    {
        NameProperty = DependencyProperty.Register(
            name: "Name",
            propertyType: typeof (string),
            ownerType: typeof (User),
            // метаданные содержат значение свойства по умолчанию
            typeMetadata: new PropertyMetadata("Empty"),
            validateValueCallback: IsNameValid);
    }
}
```

```

private static bool IsNameValid(object name)
{
    return !string.IsNullOrEmpty(name as string);
}

```

Для удобства использования свойства зависимостей можно создать экземплярную оболочку свойства. Аксессор и мутатор оболочки должны использовать методы класса `DependencyObject` `GetValue()` и `SetValue()` для чтения и установки значения свойства зависимостей. Дополнительную работу в аксессоре и мутаторе выполнять не рекомендуется, так как некоторые классы WPF могут обращаться к свойству зависимостей, минуя экземплярную оболочку.

```

public class User : DependencyObject
{
    // продолжение кода класса User

    public string Name
    {
        get { return (string) GetValue(NameProperty); }
        set { SetValue(NameProperty, value); }
    }
}

```

В классе `DependencyObject` определён метод `ClearValue()`, сбрасывающий свойство зависимостей в значение по умолчанию. Класс `DependencyProperty` имеет свойства для чтения, описывающие свойство зависимостей, и набор методов для регистрации свойств зависимостей и управления их метаданными.

Присоединённое свойство (attached property) определяется в одном типе, а применяется в объектах других типов. Характерный пример присоединённых свойств дают контейнеры компоновки. Например, контейнер `Canvas` определяет присоединённые свойства `Left`, `Right`, `Top` и `Bottom` для задания позиции элемента. Ниже приведён фрагмент XAML, описывающий абсолютно позиционированную кнопку (обратите внимание на синтаксис использования присоединённых свойств):

```

<Canvas>
    <Button Canvas.Left="30" Canvas.Top="40" />
</Canvas>

```

Технически, присоединённые свойства – особый вид свойств зависимостей. Для регистрации присоединённого свойства следует использовать метод `DependencyProperty.RegisterAttached()`. Экземплярная оболочка для присоединённого свойства не создаётся. Вместо этого нужно описать статические методы доступа и установки значения свойства. Имена методов должны включать префиксы *Set* и *Get* (это условие важно для анализатора XAML).


```

public class User : DependencyObject
{
    // опишем Name как присоединённое свойство
    public static DependencyProperty NameProperty =
        DependencyProperty.RegisterAttached("Name",
                                              typeof (string),
                                              typeof (User));

    public static void SetName(DependencyObject element,
                               object value)
    {
        element.SetValue(NameProperty, value);
    }

    public static string GetName(DependencyObject element)
    {
        return (string) element.GetValue(NameProperty);
    }
}

// код ниже эквивалентен следующему фрагменту XAML
// <Button User.Name="Alex" />
var b = new Button();
User.SetName(b, "Alex");

```

4.3. Маршрутизируемые события

Маршрутизируемые события (routed events) – модель событий WPF, созданная для использования в дереве визуальных элементов. При генерации маршрутизируемого события информация о нём может быть передана как родительским, так и дочерним элементам источника событий.

Реализация и использование маршрутизируемых событий имеет много общего со свойствами зависимостей. Рассмотрим в качестве примера реализацию события Click в стандартном элементе управления `Button`:

```

public class Button : ButtonBase
{
    // статическое поле для маршрутизируемого события
    public static RoutedEvent ClickEvent =
       EventManager.RegisterRoutedEvent("Click",
                                         RoutingStrategy.Bubble,
                                         typeof (RoutedEventHandler),
                                         typeof (Button));

    // экземплярная оболочка события
    public event RoutedEventHandler Click
    {
        add { AddHandler(ClickEvent, value); }
        remove { RemoveHandler(ClickEvent, value); }
    }
}

```

```

// внутренний метод для генерации события
protected override void OnClick(EventArgs e)
{
    RaiseEvent(new RoutedEventArgs(ClickEvent, this));
}
}

```

Методы `AddHandler()`, `RemoveHandler()` и `RaiseEvent()` – это методы класса `UIElement`. Листинг показывает, что при регистрации маршрутизируемого события используется метод `EventManager.RegisterRoutedEvent()`. Одним из аргументов метода является элемент перечисления `RoutingStrategy`, описывающего стратегию маршрутизации события:

`Tunnel` – событие генерируется в корневом элементе, затем в каждом дочернем элементе, пока не достигает элемента-источника.

`Bubble` – событие генерируется в элементе источнике, затем в каждом родительском элементе, вплоть до корня дерева элементов.

`Direct` – событие генерируется только в элементе-источнике.

Обработчики маршрутизируемых событий принимают аргумент `RoutedEventArgs`. Этот класс содержит следующие свойства: `Source` – источник события в логическом дереве элементов; `Handled` – при установке в `true` маршрутизация события в дереве прекращается; `RoutedEvent` – объект, описывающий маршрутизируемое событие.

Класс `UIElement` определяет множество маршрутизируемых событий, связанных с клавиатурой, мышью, стилусом. Большинство событий используют `Bubble`-стратегию. Многие `Bubble`-события имеют сопряжённое `Tunnel`-событие, которое генерируется перед `Bubble`-событием (`Tunnel`-событие отличает префикс *Preview* в названии).

4.4. Многопоточность в WPF

Элементы WPF, отображаемые в одном окне, обладают *потокowym родством* (thread affinity). Это означает, что поток, который их создал, владеет ими, а другие потоки не могут взаимодействовать с этими элементами напрямую. С набором элементов, обладающих потоковым родством, связан диспетчер, который принимает методы и ставит их на выполнение в очередь потока элементов. Чтобы элемент мог обладать потоковым родством и иметь доступ к диспетчеру, он должен наследоваться от класса `DispatcherObject`. Этот класс имеет всего три члена:

`Dispatcher` – свойство, возвращающее диспетчер потока;

`CheckAccess()` – метод возвращает `true`, если код выполняется в потоке элемента;

`VerifyAccess()` – если вызывающий код находится не в потоке элемента, данный метод генерирует исключение `InvalidOperationException`.

Диспетчер – это экземпляр класса `System.Windows.Threading.Dispatcher`. Метод диспетчера `BeginInvoke()` используется для того, чтобы спланировать код в качестве задачи для диспетчера:

```
// предположим, что это обработчик нажатия некоторой кнопки
private void btn_Click(object sender, RoutedEventArgs e)
{
    // запускаем новый метод в отдельном потоке
    new Thread(UpdateTextRight).Start();
}

private void UpdateTextRight()
{
    // обновим интерфейс - у диспетчера окна вызываем BeginInvoke()
    Dispatcher.BeginInvoke((Action)(() => txt.Text = "New text"));
}
```

Для организации в приложении WPF асинхронного выполнения можно использовать объект класса `System.ComponentModel.BackgroundWorker`. Основные элементы этого класса:

`RunWorkerAsync()` – метод стартует асинхронную операцию. Имеется перегруженная версия, принимающая аргумент асинхронной операции.

`CancelAsync()` – запрос на отмену асинхронной операции.

`DoWork` – обработчик этого события будет асинхронной операцией.

`RunWorkerCompleted` – это событие генерируется, если асинхронная операция завершилась, была отменена или прекращена из-за исключения.

`IsBusy` – булево свойство; позволяет узнать, выполняется ли операция.

5. Варианты организации приложений WPF

Приложения WPF допускают несколько способов организации. Самый распространённый вариант – приложение в виде набора окон. Альтернативой являются приложения на основе страниц.

5.1. Класс *Window*

Технология WPF использует для отображения стандартные окна операционной системы. Окно представлено классом `System.Windows.Window`. Это элемент управления со свойством содержимого `Content`. Обычно содержимым окна является контейнер компоновки.

Рассмотрим основные свойства класса `Window`. Внешний вид окна контролируют свойства `Icon`, `Title` (заголовок окна) и `WindowStyle` (стиль рамки). Начальная позиция окна управляется свойствами `Left` и `Top` (координаты левого верхнего угла окна) или свойством `WindowStartupLocation`, принимающим значения из одноимённого перечисления: `CenterScreen`, `CenterOwner`, `Manual`. Чтобы окно отображалось поверх всех окон, нужно установить свойство `Topmost`. Свойство `ShowInTaskbar` управляет показом иконки окна на панели задач, а свойство

TaskbarItemInfo типа System.Windows.Shell.TaskbarItemInfo даёт возможность настроить варианты отображения этой иконки.

Приведём пример описания окна – оно будет использоваться при рассмотрении контейнеров компоновки:

```
<Window x:Class="WpfLayout.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="220" Width="400"
  Icon="/WpfLayout;component/layout-icon.png"
  Title="WPF Layout Examples"
  WindowStartupLocation="CenterScreen"
  Topmost="True">

  <!-- здесь будем размещать содержимое окна -->
</Window>
```

Чтобы показать окно, используя код, нужно создать экземпляр окна и вызвать у него метод Show() или ShowDialog(). Первый метод отображает немодальное окно. Второй метод обычно применяется для модальных диалоговых окон – он ожидает закрытия окна и возвращает то значение, которое было установлено свойству окна DialogResult. Метод Hide() прячет окно, а метод Close() закрывает окно.

```
MyDialogWindow dialog = new MyDialogWindow();

// отображаем диалог и ждём, пока его не закроют
if (dialog.ShowDialog() == true)
{
    // диалог закрыли, нажав на ОК
}
```

Класс Window поддерживает отношение владения. Если отношение установлено, дочернее окно минимизируется или закрывается, когда соответствующие операции выполняет его владелец. Чтобы установить отношение, задайте у дочернего окна свойство Owner. Коллекция окна OwnedWindows содержит все дочерние окна (если они есть).

Опишем некоторые события, связанные с окном. Заметим, что класс FrameworkElement, который является базовым для класса Window, определяет события времени существования, генерируемые при инициализации (Initialized), загрузке (Loaded) или выгрузке (Unloaded) элемента. Событие Initialized генерируется после создания элемента и определения его свойств в соответствии с разметкой XAML. Это событие генерируется сначала вложенными элементами, затем их родителями. Событие Loaded возникает после того, как всё окно было инициализировано, и были применены стили и привязка данных. Событие Loaded сначала генерирует вмещающее окно, после чего его генерируют остальные вложенные элементы.

При изменении статуса активности окна генерируются события `Activated` и `Deactivated`. Событие `Closing` генерируется при попытке закрытия окна и даёт возможность отклонить эту операцию. Если закрытие окна все же происходит, генерируется событие `Closed`.

Ниже приведён листинг, демонстрирующий работу с некоторыми событиями окна. Обратите внимание – вместо назначения обработчиков событий используется перекрытие виртуальных методов, вызывающих события.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    protected override void OnClosing(CancelEventArgs e)
    {
        base.OnClosing(e);
        if (MessageBox.Show("Do you want to exit?", "Exit",
            MessageBoxButton.YesNo,
            MessageBoxImage.Question) == MessageBoxResult.No)
        {
            e.Cancel = true;
        }
    }

    protected override void OnClosed(EventArgs e)
    {
        base.OnClosed(e);
        // здесь можно сохранить состояние окна
    }

    protected override void OnInitialized(EventArgs e)
    {
        base.OnInitialized(e);
        // здесь можно восстановить состояние окна
    }
}
```

5.2. Класс *Application*

Класс `System.Windows.Application` помогает организовать точку входа для оконного приложения WPF. Этот класс содержит метод `Run()`, поддерживающий цикл обработки сообщений системы для указанного окна до тех пор, пока окно не будет закрыто:

```
Window myWindow = new Window();
Application myApp = new Application();
myApp.Run(myWindow);
```

Свойство `StartupUri` класса `Application` служит для указания главного окна приложения. Если главное окно задано, метод `Run()` можно вызывать без аргументов:

```
Application app = new Application();
app.StartupUri = new Uri("MainWindow.xaml", UriKind.Relative);
app.Run();
```

При разработке в Visual Studio для каждого оконного приложения WPF создается класс, наследуемый от `Application`, который разделён на XAML-разметку и часть с кодом. Именно в разметке XAML задаётся `StartupUri`:

```
<Application x:Class="WpfLayout.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
</Application>
```

Класс `Application` содержит несколько полезных свойств. При помощи статического свойства `Application.Current` можно получить ссылку на объект, представляющий текущее приложение. Коллекция `Windows` содержит все открытые окна приложения. Стартовое окно хранится в свойстве `MainWindow` (оно доступно для чтения и записи). Свойство `ShutdownMode` принимает значения из одноимённого перечисления и задаёт условие закрытия приложения. Словарь `Properties` позволяет хранить произвольную информацию с ключами любого типа и может использоваться для данных, разделяемых между окнами.

События класса `Application` включают `Startup` и `Exit`, `Activated` и `Deactivated`, а также событие `SessionEnding`, генерируемое при выключении компьютера или окончании сессии `Windows`. События обычно обрабатываются путём перекрытия виртуальных методов, вызывающих их генерацию.

5.3. Приложения на основе страниц

В WPF можно создавать приложения в виде набора страниц с возможностью навигации между страницами. Отдельная страница представлена объектом класса `System.Windows.Controls.Page`. Любая страница обслуживается навигационным контейнером, в качестве которого могут выступать объекты классов `NavigationWindow` или `Frame`.

Рассмотрим следующий пример. Создадим в Visual Studio WPF-приложение и добавим к проекту страницу `MainPage.xaml`:

```
<Page x:Class="WpfApplication.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  WindowHeight="220" WindowWidth="400"
  WindowTitle="Page Application">
  This is a simple page.
</Page>
```

В файле App.xaml установим StartupUri="MainPage.xaml", чтобы выполнение приложения началось со страницы MainPage.xaml. Так как в качестве StartupUri указана страница, а не окно, при запуске приложения автоматически будет создан новый объект **NavigationWindow** для выполнения роли навигационного контейнера. Класс **NavigationWindow** наследуется от **Window** и выглядит как обычное окно, за исключением кнопок навигации, которые отображаются сверху (показ этих кнопок можно отключить).

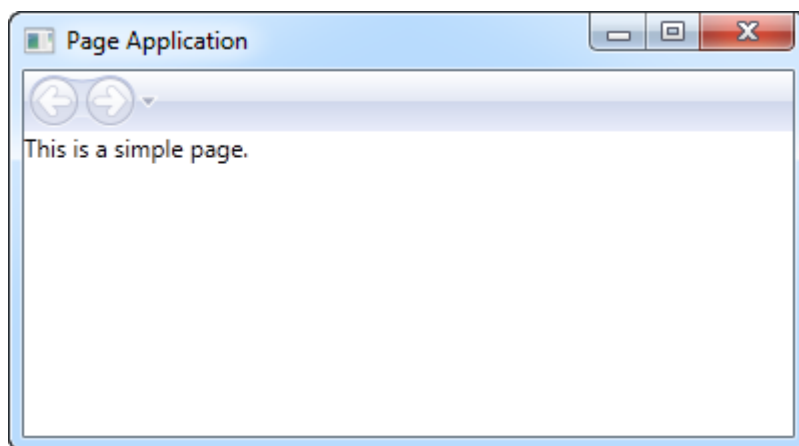


Рис. 4. Страница в контейнере **NavigationWindow**.

Класс **Page** похож на упрощённую версию класса **Window**. Основные свойства класса **Page** перечислены в табл. 3.

Таблица 3

Основные свойства класса **Page**

Имя	Описание
Background	Фон страницы, который задаётся с помощью объекта Brush
Content	Элемент, который отображается на странице. Обычно в роли такого элемента выступает контейнер компоновки
Foreground, FontFamily, FontSize	Цвет, вид и размер текста внутри страницы. Значения этих свойств наследуются элементами внутри страницы
WindowWidth, WindowHeight, WindowTitle	Свойства определяют внешний вид окна NavigationWindow , в которое упаковывается страница
NavigationService	Возвращает ссылку на объект NavigationService , который можно использовать для программной навигации
KeepAlive	Определяет, должен ли сохраняться объект страницы после перехода пользователя на другую страницу
ShowsNavigationUI	Определяет, должны ли в обслуживающем страницу контейнере отображаться навигационные элементы управления
Title	Устанавливает имя, которое должно применяться для страницы в хронологии навигации

Отметим, что в классе **Page** нет эквивалентов для методов **Hide()** и **Show()**, доступных в классе **Window**. Если потребуется показать другую страницу, придётся воспользоваться одним из видов навигации. Навигационные контейнеры и класс **NavigationService** поддерживают метод **Navigate()**, который удобно использовать для программной навигации. Метод **Navigate()** принимает в качестве

аргумента или объект страницы, или адрес страницы (обычно это имя XAML-файла):

```
// предполагается, что код расположен в методе страницы
// навигация по объекту страницы
PhotoPage nextPage = new PhotoPage();
this.NavigationService.Navigate(nextPage);

// или навигация по URI
var nextPageUri = new Uri("PhotoPage.xaml", UriKind.Relative);
this.NavigationService.Navigate(nextPageUri);
```

Гиперссылки – это простой способ декларативной навигации. В WPF гиперссылка представлена объектом класса `Hyperlink`, который внедряется в текстовый поток. Щелчки на ссылке можно обрабатывать при помощи события `Click`, или установить страницу перехода в свойстве `NavigateUri` (только если `Hyperlink` размещается на странице). При этом `NavigateUri` может указывать не только на объект `Page`, но и на веб-содержимое.

```
<TextBlock>
    Click <Hyperlink NavigateUri="PhotoPage.xaml">here</Hyperlink>
</TextBlock>
```

`NavigationWindow` поддерживает журнал показанных страниц и содержит две кнопки «назад» и «вперёд». Методы навигационного контейнера `GoBack()` и `GoForward()` позволяют перемещаться по журналу страниц, используя код.

Как было сказано выше, в качестве навигационного контейнера можно использовать элемент управления `Frame`. Он включает свойство `Source`, которое указывает на подлежащую отображению страницу. Так как `Frame` – обычный элемент управления, он может размещаться в заданном месте окна `Window` или страницы `Page`. В последнем случае образуются вложенные фреймы.

5.4. Приложения ХВАР

Приложения ХВАР (XAML Browser Application) – это приложения, которые выполняются внутри браузера. Приложения ХВАР являются полноценными приложениями WPF, но имеют несколько особенностей:

1. Приложение ХВАР отображается в окне браузера, но запускается в виде отдельного процесса, управляемого CLR. Следовательно, на компьютере клиента должен быть установлен .NET Framework версии 3.0 или выше.

2. Приложению ХВАР предоставляются те же разрешения, что и приложению .NET, которое запускается из интернета или локальной интрасети. Это означает, что по умолчанию приложение ХВАР не может записывать файлы, взаимодействовать с ресурсами компьютера, подключаться к базам данных или отображать полнофункциональные окна.

3. Приложения ХВАР не инсталлируются. При запуске приложения загружаются с сервера или локального диска и помещаются в кэш браузера.

Visual Studio предлагает для приложения ХВАР шаблон *WPF Browser Application*. Создав проект по этому шаблону, можно приступить к разработке страниц точно так же, как и при использовании *NavigationWindow*. Для развёртывания готового приложения ХВАР на веб-сервер нужно скопировать в виртуальный каталог следующие файлы:

Имя-Приложения.exe – файл содержит скомпилированный IL-код.

Имя-Приложения.exe.manifest – XML-документ с перечнем требований данного приложения (например, версий .NET-сборок).

Имя-Приложения.хвар. – точка входа в приложение; этот файл пользователь должен запросить в браузере, чтобы установить данное приложение ХВАР.

Самое сложное при создании приложения ХВАР – оставаться в рамках ограниченной модели безопасности. Один из простых способов узнать, разрешено ли выполнение того или иного действия – написать какой-то тестовый код и испытать его. Также все необходимые детали можно найти в документации по WPF.

5.5. Работа с панелью задач Windows 7

Опишем некоторые возможности по работе с панелью задач Windows 7, доступные в приложениях WPF. *Список переходов* (jump list) – это мини-меню, которое открывается при щелчке правой кнопкой мыши на иконке приложения в панели задач. Для работы со списком переходов в приложении WPF применяются классы *JumpList* (список переходов), *JumpPath* (путь к документу в списке переходов) и *JumpTask* (команда в списке) из пространства имён *System.Windows.Shell*:

```
// этот код размещён в конструкторе главного окна
// создаём и настраиваем команду для списка переходов
var jumpTask = new JumpTask();
jumpTask.Title = "Notepad";
jumpTask.Description = "Open Notepad";
jumpTask.ApplicationPath = @"%WINDIR%\system32\notepad.exe";
jumpTask.IconResourcePath = @"%WINDIR%\system32\notepad.exe";

// создаём сам список переходов, добавляем в него команду
var jumpList = new JumpList();
jumpList.JumpItems.Add(jumpTask);

// связываем список с текущим приложением
JumpList.SetJumpList(Application.Current, jumpList);
```

Класс *JumpTask* описывает команду, выполнение которой влечёт запуск заданного приложения. Класс имеет следующие свойства:

CustomCategory – категория, к которой относится команда.

Title – название команды;

Description – всплывающая подсказка;

IconResourcePath и *IconResourceIndex* – файл с иконкой и индекс иконки в файле для отображения в списке переходов;

ApplicationPath – путь к исполняемому файлу нужного приложения;
WorkingDirectory – рабочий каталог для приложения;
Arguments – аргументы командной строки для приложения.

У класса `JumpPath` для настройки доступны два свойства. Строка `CustomCategory` указывает категорию, к которой относится `JumpPath`. Свойство `Path` задаёт полный путь к файлу документа. При этом файл должен существовать, а его тип должен соответствовать типу файлов, за обработку которых отвечает данное приложение.

Для задания списка переходов в XAML используется присоединённое свойство `JumpList.JumpList`:

```
<Application x:Class="WpfLayout.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">

  <JumpList.JumpList>
    <JumpList>
      <JumpTask Title="Notepad" Description="Open Notepad"
        ApplicationPath="%WINDIR%\system32\notepad.exe"
        IconResourcePath="%WINDIR%\system32\notepad.exe" />
      <JumpPath Path="C:\Pictures\Cat.jpg" />
    </JumpList>
  </JumpList.JumpList>
</Application>
```

Класс `Window` имеет свойство `TaskbarItemInfo`, которое даёт доступ к иконке окна на панели задач. В следующем примере показан один из вариантов использования `TaskbarItemInfo` – на иконке отображается индикатор прогресса:

```
<Window.TaskbarItemInfo>
  <TaskbarItemInfo ProgressValue="0.3" ProgressState="Paused" />
</Window.TaskbarItemInfo>
```

6. компоновка

В WPF *компоновка* (layout) – это процесс размещения визуальных элементов на поверхности родительского элемента. Компоновка состоит из двух фаз:

1. *Фаза измерения* (measure). В этой фазе родительский контейнер запрашивает желаемый размер у каждого дочернего элемента, которые, в свою очередь, выполняют фазу измерения рекурсивно.

2. *Фаза расстановки* (arrange). Родительский контейнер сообщает дочерним элементам их истинные размеры и позицию, в зависимости от выбранного способа компоновки.

6.1. Размер и выравнивание

Рассмотрим некоторые свойства элементов WPF, связанные с процессом компоновки. Свойство `Visibility`, определённое в классе `UIElement`, управляет

видимостью элемента. Это свойство принимает значение из перечисления `System.Windows.Visibility`: `Visible` – элемент виден на визуальной поверхности; `Collapsed` – элемент не виден на визуальной поверхности и не участвует в процессе компоновки; `Hidden` – элемент не виден на визуальной поверхности, но участвует в процессе компоновки («занимает место»).

В классе `FrameworkElement` определён набор свойств, ответственных за размер, отступы и выравнивание отдельного элемента (табл. 4).

Таблица 4

Свойства размера, отступа и выравнивания

Имя	Описание
<code>HorizontalAlignment</code>	Определяет позиционирование дочернего элемента внутри контейнера компоновки, если доступно пространство по горизонтали. Принимает значения из одноимённого перечисления: <code>Center</code> , <code>Left</code> , <code>Right</code> , <code>Stretch</code>
<code>VerticalAlignment</code>	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по вертикали. Принимает значения из одноимённого перечисления: <code>Center</code> , <code>Top</code> , <code>Bottom</code> или <code>Stretch</code>
<code>Margin</code>	Добавляет пространство вокруг элемента. Это экземпляр структуры <code>System.Windows.Thickness</code> с отдельными компонентами для верхней, нижней, левой и правой стороны
<code>MinWidth</code> и <code>MinHeight</code>	Устанавливает минимальные размеры элемента. Если элемент слишком велик, он будет усечён
<code>MaxWidth</code> и <code>MaxHeight</code>	Устанавливает максимальные размеры элемента. Если контейнер имеет свободное пространство, элемент не будет увеличен сверх указанных пределов, даже если свойства <code>HorizontalAlignment</code> и <code>VerticalAlignment</code> установлены в <code>Stretch</code>
<code>Width</code> и <code>Height</code>	Явно устанавливают размеры элемента. Эта установка переопределяет значение <code>Stretch</code> для свойств <code>HorizontalAlignment</code> и <code>VerticalAlignment</code> . Однако размер не будет установлен, если выходит за пределы, заданные в <code>MinWidth</code> и <code>MinHeight</code>

При установке размеров в XAML можно указать единицу измерения: `px` (по умолчанию) – размер в единицах WPF (1/96 дюйма); `in` – размер в дюймах; `cm` – размер в сантиметрах; `pt` – размер в пунктах (1/72 дюйма).

```
<Button Width="100px" Height="0.5in" />
<Button Width="80pt" Height="2cm" />
```

В `FrameworkElement` свойства `Width` и `Height` установлены по умолчанию в значение `double.NaN`. Это означает, что элемент будет иметь такие размеры, которые нужны для отображения его содержимого. В разметке XAML значению `double.NaN` для свойств размера соответствует строка `"NaN"` или (более предпочтительно) строка `"Auto"`. Также в классе `FrameworkElement` определены свойства только для чтения `ActualWidth` и `ActualHeight`, содержащие действительные отображаемые размеры элемента после фазы расстановки.

Следующий пример демонстрирует компоновку с элементами, у которых установлены некоторые свойства размера и позиционирования. Обратите внимание на различные способы указания свойства `Margin`:

- одно значение: одинаковые отступы для всех четырёх сторон;
- два значения: отступы для левой/правой и верхней/нижней сторон;
- четыре числа: отступы для левой, верхней, правой и нижней стороны.

```
<StackPanel>
  <Button HorizontalAlignment="Left" Content="Button 1" />
  <Button HorizontalAlignment="Right" Content="Button 2" />
  <Button Margin="20" Content="Button 3" />
  <Button Margin="5,10,100,10" Content="Button 4" />
  <Button Margin="5,10" MaxWidth="100" Content="Button 5" />
</StackPanel>
```

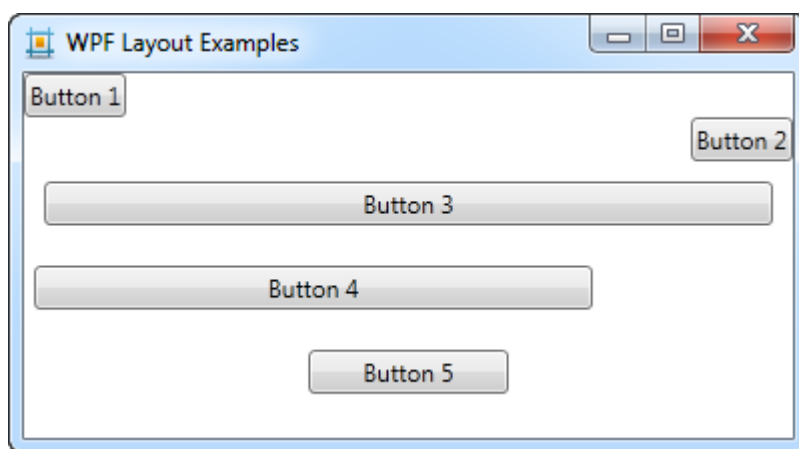


Рис. 5. Использование свойств размера и позиционирования.

В элементах управления, унаследованных от класса `Control`, определены свойства отступа и выравнивания для дочернего содержимого. За выравнивание отвечают свойства `HorizontalContentAlignment` и `VerticalContentAlignment`. Они поддерживают те же значения, что и свойства `HorizontalAlignment` и `VerticalAlignment`. Свойство `Padding` позволяет вставить пустое пространство между краями элемента управления и краями содержимого. Его тип и способ задания аналогичны свойству `Margin`.

6.2. Встроенные контейнеры компоновки

Контейнер компоновки – это класс, реализующий определённую логику компоновки дочерних элементов. Технология WPF предлагает ряд стандартных контейнеров компоновки:

1. `Canvas`. Позволяет элементам позиционироваться по фиксированным координатам.
2. `StackPanel`. Размещает элементы в горизонтальный или вертикальный стек. Контейнер обычно используется в небольших секциях сложного окна.

3. `WrapPanel`. Размещает элементы в сериях строк с переносом. Например, в горизонтальной ориентации `WrapPanel` располагает элементы в строке слева направо, затем переходит к следующей строке.

4. `DockPanel`. Выравнивает элементы по краю контейнера.

5. `Grid`. Встраивает элементы в строки и колонки невидимой таблицы.

Все контейнеры компоновки являются панелями, которые унаследованы от абстрактного класса `System.Windows.Controls.Panel`. Этот класс содержит несколько полезных свойств:

`Background` – кисть, используемая для рисования фона панели. Кисть нужно задать, если панель должна принимать события мыши (как вариант, это может быть прозрачная кисть).

`Children` – коллекция элементов, находящихся в панели. Это первый уровень вложенности – другими словами, это элементы, которые сами могут содержать дочерние элементы.

`IsItemsHost` – булево значение. Устанавливается в `true`, если панель используется для показа элементов в шаблоне спискового элемента управления.

`ZIndex` – присоединённое свойство класса `Panel` для задания высоты визуального слоя элемента. Элементы с большим значением `ZIndex` выводятся поверх элементов с меньшим значением.

`Canvas` – контейнер компоновки, реализующий позиционирование элементов путём указания фиксированных координат. Для задания позиции элемента следует использовать присоединённые свойства `Canvas.Left`, `Canvas.Right`, `Canvas.Top`, `Canvas.Bottom`. Эти свойства определяют расстояние от соответствующей стороны `Canvas` до ближайшей грани элемента¹. Использование контейнера `Canvas` демонстрируется в следующем примере:

```
<Canvas>
  <Button Background="Red" Content="Left=0, Top=0" />
  <Button Canvas.Left="20" Canvas.Top="20" Background="Orange"
    Content="Left=20, Top=20" />
  <Button Canvas.Right="20" Canvas.Bottom="20" Background="Yellow"
    Content="Right=20, Bottom=20" />
  <Button Canvas.Right="0" Canvas.Bottom="0" Background="Lime"
    Content="Right=0, Bottom=0" />
  <Button Canvas.Right="0" Canvas.Top="0" Background="Aqua"
    Content="Right=0, Top=0" />
  <Button Canvas.Left="0" Canvas.Bottom="0" Background="Magenta"
    Content="Left=0, Bottom=0" />
</Canvas>
```

¹ При одновременной установке `Canvas.Left` имеет преимущество перед `Canvas.Right`, а `Canvas.Top` – перед `Canvas.Bottom`.

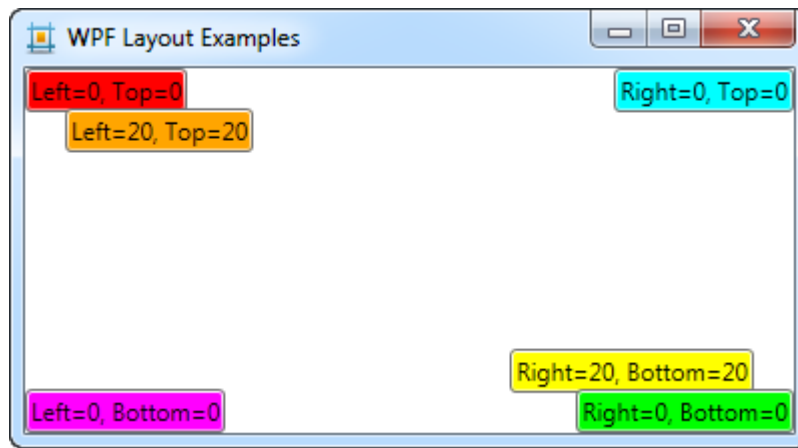


Рис. 6. Кнопки в контейнере `Canvas`.

`StackPanel` – популярный контейнер компоновки, который размещает дочерние элементы последовательно, по мере их объявления в контейнере. Свойство `Orientation` управляет направлением размещения дочерних элементов и принимает значения из одноимённого перечисления: `Vertical` (по умолчанию) или `Horizontal`.

```
<StackPanel Orientation="Horizontal">
  <Button Background="Red" Content="One" />
  <Button Background="Orange" Content="Two" />
  <Button Background="Yellow" Content="Three" />
  <Button Background="Lime" Content="Four" />
  <Button Background="Aqua" Content="Five" />
  <Button Background="Magenta" Content="Six" />
</StackPanel>
```

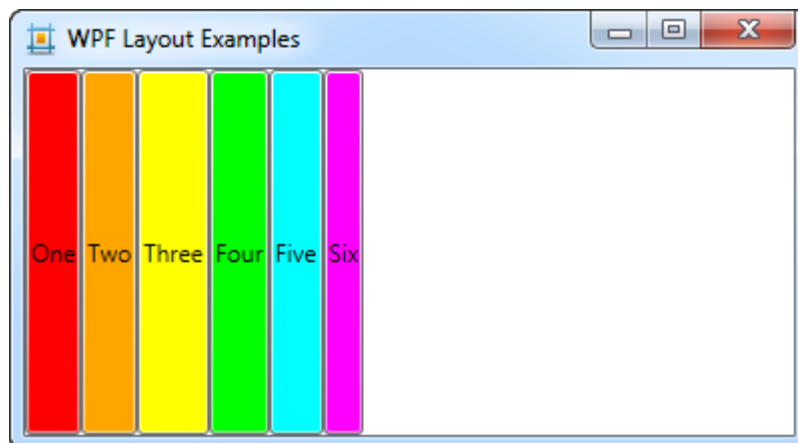


Рис. 7. `StackPanel` с горизонтальной ориентацией.

`WrapPanel` – это контейнер компоновки, который во многом аналогичен `StackPanel`. Однако `WrapPanel` использует автоматический перенос элементов, для которых не хватает вертикального (горизонтального) пространства, в новый столбец (строку). `WrapPanel` поддерживает несколько свойств настройки:

`Orientation` – свойство аналогично одноимённому свойству у `StackPanel`, но по умолчанию использует значение `Horizontal`.

`ItemHeight` – единая мера высоты для всех дочерних элементов. В рамках заданной единой высоты каждый дочерний элемент располагается в соответствии со своим свойством `VerticalAlignment` или усекается.

`ItemWidth` – единая мера ширины для всех дочерних элементов. В рамках заданной единой ширины каждый дочерний элемент располагается в соответствии со своим свойством `HorizontalAlignment` или усекается.

По умолчанию, свойства `ItemHeight` и `ItemWidth` не заданы (имеют значение `double.NaN`). В этой ситуации ширина столбца (высота строки) определяется по самому широкому (самому высокому) дочернему элементу.

```
<WrapPanel ItemHeight="80">
  <Button Width="60" Height="40" Background="Red" Content="One" />
  <Button Width="60" Height="20" Background="Orange"
    Content="Two" />
  <Button Width="60" Height="40" Background="Yellow"
    Content="Three" />
  <Button Width="60" Height="20" VerticalAlignment="Top"
    Background="Lime" Content="Four" />
  <Button Width="60" Height="40" Background="Aqua" Content="Five" />
  <Button Width="60" Height="20" Background="Magenta"
    Content="Six" />
</WrapPanel>
```

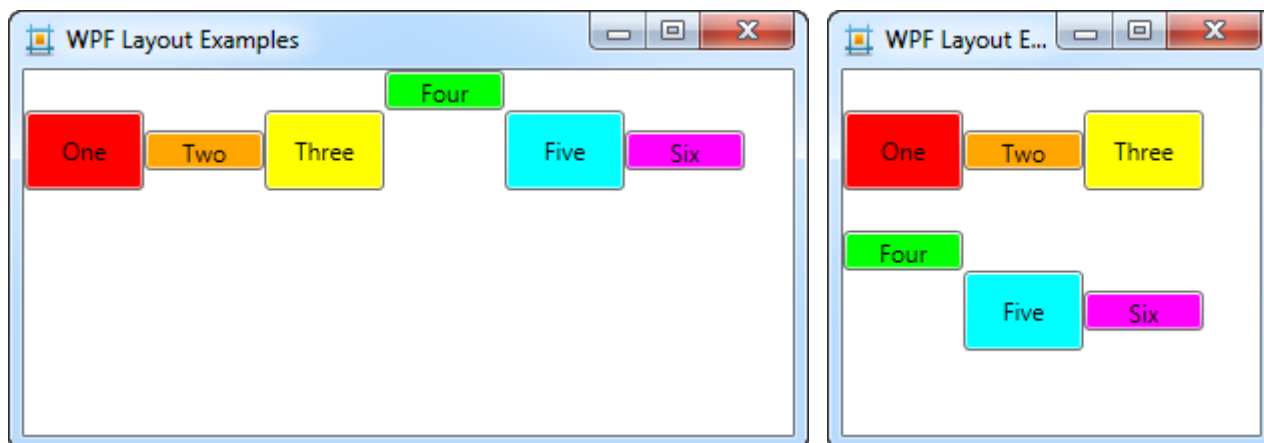


Рис. 8. Элементы на `WrapPanel` для разной ширины окна.

`DockPanel` (док) реализует *примыкание* (docking) дочерних элементов к одной из своих сторон. Примыкание настраивается при помощи присоединённого свойства `DockPanel.Dock`, принимающего значения `Left`, `Top`, `Right` и `Bottom` (перечисление `System.Windows.Controls.Dock`). Примыкающий элемент растягивается на всё свободное пространство дока по вертикали или горизонтали (в зависимости от стороны примыкания), если у элемента явно не задан размер. Порядок дочерних элементов в доке имеет значение. Если в `DockPanel` свойство `LastChildFill` установлено в `true` (это значение по умолчанию), последний дочерний элемент занимает всё свободное пространство дока.

```

<DockPanel>
  <Button DockPanel.Dock="Top" Background="Red" Content="1 (Top)" />
  <Button DockPanel.Dock="Left" Background="Orange"
    Content="2 (Left)" />
  <Button DockPanel.Dock="Right" Background="Yellow"
    Content="3 (Right)" />
  <Button DockPanel.Dock="Bottom" Background="Lime"
    Content="4 (Bottom)" />
  <Button Background="Aqua" Content="5 (Fill)" />
</DockPanel>

```

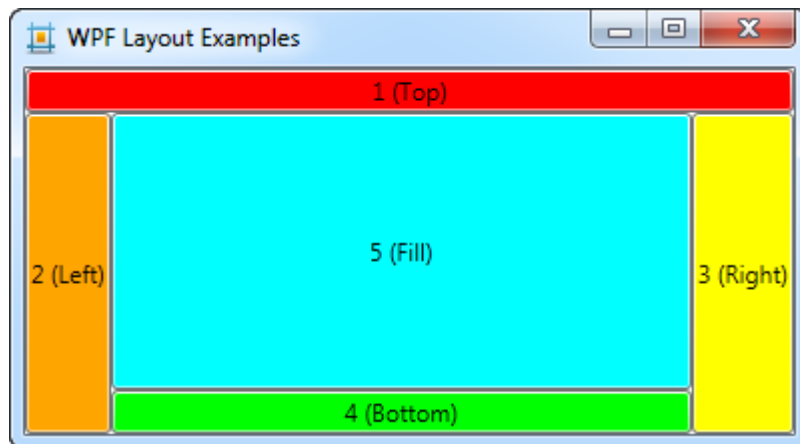


Рис. 9. Док с набором кнопок.

Grid – один из наиболее гибких и широко используемых контейнеров компоновки. Он организует пространство как таблицу с настраиваемыми столбцами и строками. Дочерние элементы размещаются в указанных ячейках таблицы.

```

<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Button Grid.Column="0" Grid.Row="0"
    Background="Red" Content="One" />
  <Button Grid.Column="1" Grid.Row="0" Width="60"
    Background="Orange" Content="Two" />
  <Button Grid.Column="2" Grid.Row="0"
    Background="Yellow" Content="Three" />
  <Button Grid.Column="0" Grid.Row="1"
    Background="Lime" Content="Four" />
  <Button Grid.Column="1" Grid.Row="1" Grid.ColumnSpan="2"
    Background="Aqua" Content="Five" />
</Grid>

```

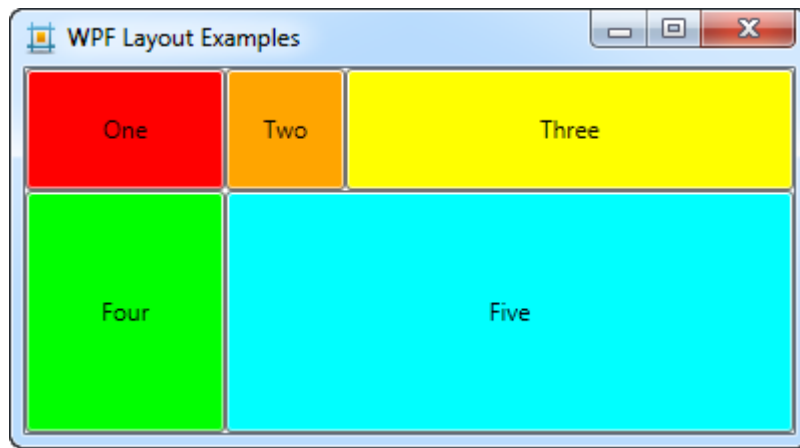



Рис. 10. Демонстрация контейнера `Grid`.

При настройке `Grid` необходимо задать набор столбцов и строк с помощью коллекций `ColumnDefinitions` и `RowDefinitions`. Для столбцов может быть указана ширина, для строк – высота. При этом допустимо использовать абсолютное значение, подбор по содержимому или *пропорциональный размер*. В последнем случае применяется символ * и необязательный коэффициент пропорциональности. В примере, приведённом выше, высота второй строки равна удвоенной высоте первой строки, независимо от высоты окна.

Дочерние элементы связываются с ячейками `Grid` при помощи присоединённых свойств `Grid.Column` и `Grid.Row`. Если несколько элементов расположены в одной ячейке, они наслаиваются друг на друга. Один элемент может занять несколько ячеек, определяя значения для присоединённых свойств `Grid.ColumnSpan` и `Grid.RowSpan`.

Контейнер `Grid` позволяет изменять размеры столбцов и строк при помощи перетаскивания, если используется элемент управления `GridSplitter`. Вот несколько правил работы с этим элементом:

1. `GridSplitter` должен быть помещён в ячейку `Grid`. Лучший подход заключается в резервировании специального столбца или строки для `GridSplitter` со значениями `Height` или `Width`, равными `Auto`.

2. `GridSplitter` всегда изменяет размер всей строки или столбца, а не отдельной ячейки. Чтобы сделать внешний вид `GridSplitter` соответствующим такому поведению, растяните его по всей строке или столбцу, используя присоединённые свойства `Grid.ColumnSpan` или `Grid.RowSpan`.

3. Изначально `GridSplitter` настолько мал, что его не видно. Дайте ему минимальный размер. Например, в случае вертикальной разделяющей полосы установите `VerticalAlignment` в `Stretch`, а `width` – в фиксированный размер.

4. Выравнивание `GridSplitter` определяет, будет ли разделительная полоса горизонтальной (используемой для изменения размеров строк) или вертикальной (для изменения размеров столбцов). В случае горизонтальной разделительной полосы установите `VerticalAlignment` в `Center` (что принято по умолчанию). В случае вертикальной разделительной полосы установите `HorizontalAlignment` в `Center`.

Ниже приведён фрагмент разметки, использующей горизонтальный `GridSplitter`:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <GridSplitter Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="3"
    HorizontalAlignment="Stretch" Height="3"
    VerticalAlignment="Center" />
</Grid>
```

Классы `RowDefinition` и `ColumnDefinition` обладают строковым свойством `SharedSizeGroup`, при помощи которого строки или столбцы объединяются в *группу, разделяющую размер*. Это означает, что при изменении размера одного элемента группы (строки или столбца), другие элементы автоматически получают такой же размер. Разделение размеров может быть выполнено как в рамках одного контейнера `Grid`, так и между несколькими `Grid`. В последнем случае необходимо установить свойство `Grid.IsSharedSizeScope` в значение `true` для внешнего контейнера компоновки:

```
<Grid IsSharedSizeScope="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="myGroup" />
    <ColumnDefinition />
    <ColumnDefinition SharedSizeGroup="myGroup" />
  </Grid.ColumnDefinitions>

  <!-- определение элементов Grid -->
</Grid>
```

6.3. Прокрутка и декорирование содержимого

Необходимость в прокрутке возникает, если визуальное содержимое выходит за границы родительского элемента¹. *Панель прокрутки* – это элемент управления содержимым `ScrollView`. Его свойства `VerticalScrollBarVisibility` и `HorizontalScrollBarVisibility` управляют видимостью полос прокрутки и принимают значение из перечисления `ScrollBarVisibility`:

¹ Классы, унаследованные от `UIElement`, обладают булевым свойством `ClipToBounds`. Если в родительском элементе это свойство установлено в `true`, визуальное содержимое дочерних элементов отсекается при выходе за границы родителя.

1. **Visible** – полоса прокрутки видима, даже если в ней нет необходимости.
2. **Auto** – полоса прокрутки появляется только тогда, когда содержимое не помещается в визуальных границах панели прокрутки.
3. **Hidden** – полоса прокрутки не видна, но прокрутку можно выполнить в коде или используя клавиатуру.
4. **Disabled** – полоса прокрутки не видна, прокрутку выполнить нельзя.

Элемент **ScrollView** имеет методы для программной прокрутки. Вертикальная прокрутка выполняется при помощи методов **LineUp()**, **LineDown()**, **PageUp()**, **PageDown()**, **ScrollToHome()**, **ScrollToEnd()**, а горизонтальная прокрутка – при помощи **LineLeft()**, **LineRight()**, **PageLeft()**, **PageRight()**, **ScrollToHorizontalOffset()**, **ScrollToLeftEnd()**, **ScrollToRightEnd()**.

Одной из особенностей **ScrollView** является возможность участия содержимого в процессе прокрутки. Такое содержимое должно быть представлено объектом, реализующим интерфейс **IScrollInfo**. Кроме этого, необходимо установить свойство **ScrollView.CanContentScroll** в значение **true**. Интерфейс **IScrollInfo** реализуют всего несколько элементов. Одним из них является контейнер **StackPanel**. Его реализация интерфейса **IScrollInfo** обеспечивает *логическую прокрутку*, которая осуществляет переход от элемента к элементу, а не от строки к строке.

```
<ScrollView CanContentScroll="True">
  <StackPanel>
    <Button Height="100" Content="1"/>
    <Button Height="100" Content="2"/>
    <Button Height="100" Content="3"/>
    <Button Height="100" Content="4"/>
  </StackPanel>
</ScrollView>
```

Декораторы обычно служат для того, чтобы графически разнообразить и украсить область вокруг объекта. Все декораторы являются наследниками класса **System.Windows.Controls.Decorator**. Большинство декораторов – это специальные классы, предназначенные для использования вместе с определёнными элементами управления. Есть два общих декоратора, применять которые имеет смысл при создании пользовательских интерфейсов: **Border** и **Viewbox**.

Декоратор **Border** принимает вложенное содержимое и добавляет к нему фон или рамку. Для управления **Border** можно использовать свойства размера и отступа, а также некоторые особые свойства:

Background – задаёт фон декоратора с помощью объекта **Brush**.

BorderBrush, **BorderThickness** – свойства задают цвет и ширину рамки. Чтобы показать рамку, нужно задать оба свойства. У рамки можно отдельно настроить ширину каждой из четырёх сторон.

CornerRadius – позволяет закруглить углы рамки. Можно отдельно настроить радиус закругления каждого угла.

```

<Border Margin="20" Padding="10" VerticalAlignment="Top"
        Background="LightYellow" BorderBrush="SteelBlue"
        BorderThickness="10,5,10,5" CornerRadius="5">
    <StackPanel>
        <Button Margin="5" Content="One" />
        <Button Margin="5" Content="Two" />
        <Button Margin="5" Content="Three" />
    </StackPanel>
</Border>

```

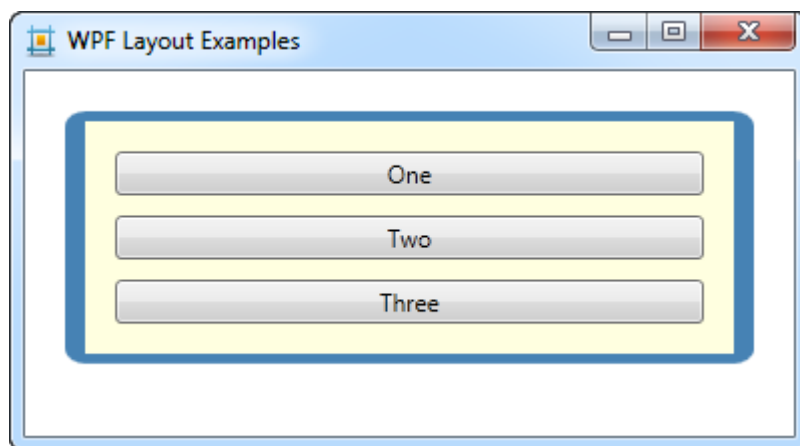


Рис. 11. Декоратор `Border`.

Декоратор `Viewbox` масштабирует своё содержимое так, чтобы оно умещалось в этом декораторе. По умолчанию `Viewbox` выполняет масштабирование, которое сохраняет пропорции содержимого. Это поведение можно изменить при помощи свойства `Stretch`. Например, если свойству присвоить значение `Fill`, содержимое внутри `Viewbox` будет растянуто в обоих направлениях. Кроме этого, можно использовать свойство `StretchDirection`. Оно управляет масштабированием, когда содержимое достаточно мало (или слишком велико), чтобы уместиться в `Viewbox`.

7. Обзор элементов управления WPF

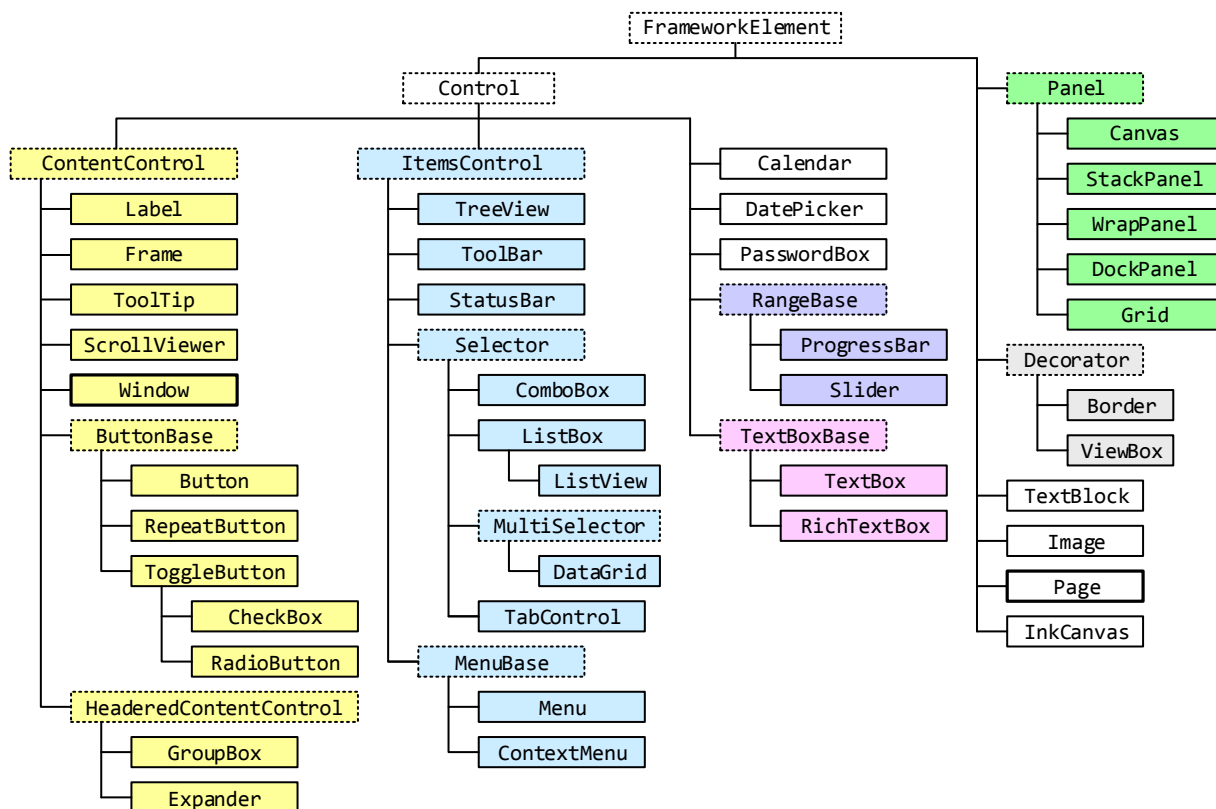


Рис. 12. Стандартные элементы WPF.

Формально, к элементам управления относятся классы, унаследованные от класса `Control`. Рассмотрим некоторые собственные свойства этого класса, разбив их на группы:

1. Внешний вид. Свойство `Template` задаёт шаблон, полностью определяющий внешний вид элемента управления.

2. Позиционирование содержимого. За позиционирование ответственны свойства `Padding`, `HorizontalContentAlignment` и `VerticalContentAlignment`.

3. Цвета и окантовка.

`Foreground` – кисть для отображения содержимого;

`Background` – кисть для фона элемента;

`BorderBrush` и `BorderThickness` – кисть и ширина окантовки.

4. Шрифт содержимого.

`FontFamily` – имя семейства шрифтов (например, `"Arial"`);

`FontSize` – размер шрифта в единицах WPF;

`FontStyle` – наклон текста. Нужный набор можно получить из статических свойств класса `FontStyles`: `Normal`, `Italic` или `Oblique`;

`FontWeight` – вес (т.е. толщина) текста. Предварительно заданный набор можно получить, используя статические свойства класса `FontWeight`;

`FontStretch` – величина, на которую растягивается или сжимается текст.

5. Табуляция. Целочисленное свойство `TabIndex` и булево `IsTabStop`.

7.1. Элементы управления содержимым

Элементы управления содержимым – это элементы, допускающие размещение единственного дочернего элемента, представляющего их содержимое. Элементы управления содержимым наследуются от класса `ContentControl`. У этого класса имеется объектное свойство `Content` и булево свойство `HasContent`. Если в `Content` помещается объект-наследник `UIElement`, для отображения вызывается метод `OnRender()` этого объекта. Для объектов других типов делается попытка применить шаблон данных. Если шаблон данных не задан в свойстве `ContentTemplate` или в ресурсах приложения, у дочернего элемента вызывается метод `ToString()`, а результат обрамляется в элемент `TextBlock`.

Элементы управления содержимым можно разбить на три подгруппы: *кнопки, простые контейнеры, контейнеры с заголовком*.

Все кнопки наследуются от абстрактного класса `ButtonBase`. Этот класс содержит событие `Click`, булево свойство `IsPressed` и свойство `ClickMode` – оно определяет момент генерирования `Click`. Значениями `ClickMode` являются элементы одноимённого перечисления: `Release` (по умолчанию), `Press` и `Hover`.

Элемент управления `Button` определяет обычную кнопку, которая может быть кнопкой по умолчанию или кнопкой отмены. Это поведение контролируется булевыми свойствами `IsDefault` и `IsCancel`. Кнопка по умолчанию срабатывает при нажатии на `<Enter>`, кнопка отмены – при нажатии `<Esc>`.

Элемент `RepeatButton` – кнопка, непрерывно генерирующая событие `Click`, если на ней задержать нажатие. Частота генерации события `Click` определяется свойствами `Delay` и `Interval` (свойства содержат целое число миллисекунд).

`ToggleButton` – кнопка с «залипанием». Будучи нажатой, она остаётся в таком состоянии, пока по ней не произведут повторное нажатие. Текущее состояние кнопки определяется свойством `IsChecked` с типом `bool?` (если свойство `IsThreeState` установлено в `true`, нажатия на кнопку заставляют `IsChecked` меняться по схеме `true-null-false`). При изменении свойства `IsChecked` кнопка генерирует события `Checked`, `Unchecked` и `Indeterminate`.

Флажок-переключатель формально относится к кнопкам и представлен классом `CheckBox`. Этот класс является прямым наследником `ToggleButton` с переписанным шаблоном внешнего вида.

Элемент управления `RadioButton` также унаследован от `ToggleButton`. Особенность `RadioButton` – поддержка группового поведения, при котором установка одного переключателя сбрасывает остальные. По умолчанию к одной группе относятся переключатели, имеющие общий родительский контейнер. Можно задать строковое свойство `GroupName`, чтобы определить собственный способ выделения группы переключателей.

```
<StackPanel>
  <RadioButton Margin="2" GroupName="Gender" Content="Male"
               IsChecked="True" />
  <RadioButton Margin="2" GroupName="Gender" Content="Female" />
```

```

<RadioButton Margin="2" GroupName="Status" Content="Single"
             IsChecked="True" />
<RadioButton Margin="2" GroupName="Status" Content="Married" />
<RadioButton Margin="2" GroupName="Status" Content="Divorced" />
<RadioButton Margin="2" GroupName="Status" Content="Widower" />

<CheckBox Margin="2" Content="Has children" />

<Button Margin="8" Click="btn_Click" Content="Save Profile" />
</StackPanel>

```

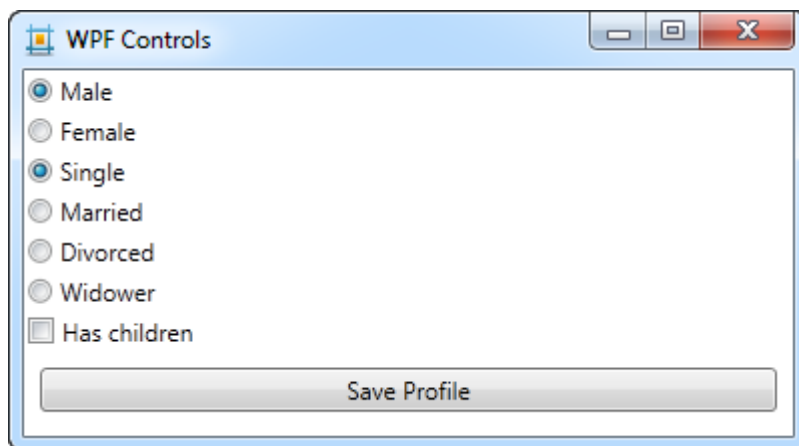


Рис. 13. Демонстрация различных кнопок.

Перейдём к разбору простых контейнеров, которые (согласно своему названию) предназначены для простого обрамления дочернего содержимого. Один из простых контейнеров – элемент `Label`. Обычно он используется для представления текста, но в WPF его содержимое может быть любым. Особенность `Label` – поддержка «горячих клавиш», при помощи которых фокус ввода передаётся заданному элементу управления. Покажем типичный пример такого использования `Label` (обратите внимание на синтаксис задания свойства `Target` и на применение символа подчёркивания для указания «горячей клавиши» U):

```

<Label Target="{x:Reference userName}" Content="_User:" />
<TextBox x:Name="userName" />

```

Элемент управления `ToolTip` отображает своё содержимое в виде всплывающей подсказки. `ToolTip` может содержать любые элементы управления, но с ними нельзя взаимодействовать. `ToolTip` определяет события `Open` и `Close`, а также несколько свойств для настройки своего поведения и внешнего вида. Особенность `ToolTip` в том, что этот элемент нельзя поместить в дерево элементов. Вместо этого требуется задавать одноимённое свойство, определённое в классе `FrameworkElement`. Дополнительные параметры отображения всплывающей подсказки можно настроить при помощи свойств класса `ToolTipService`.

```

<CheckBox Margin="10" Content="Simple CheckBox"
          ToolTipService.VerticalOffset="15">

```



```

<CheckBox.ToolTip>
  <StackPanel>
    <Label FontWeight="Bold" Background="Blue" Foreground="White"
      Content="The CheckBox" />
    <TextBlock TextWrapping="Wrap" Width="200">
      CheckBox is a familiar control.
      In WPF it is a ToggleButton styled differently.
    </TextBlock>
  </StackPanel>
</CheckBox.ToolTip>
</CheckBox>

```

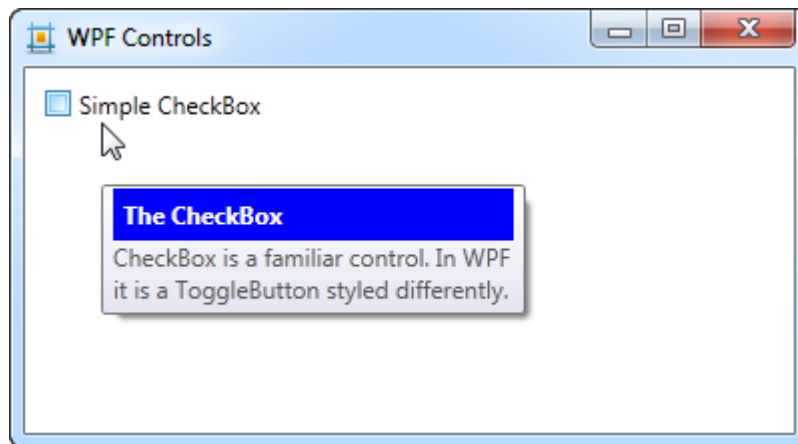


Рис. 14. Сложная всплывающая подсказка.

Элемент управления `Frame` предназначен для изолирования своего содержимого от остальной части пользовательского интерфейса. Его свойство `Source` указывает на отображаемую HTML-страницу или XAML-страницу:

```

<Frame Source="http://www.pinvoke.net" />

```

Последняя группа элементов управления содержимым – контейнеры с заголовком. Эти элементы наследуются от класса `HeaderedContentControl`, который добавляет объектное свойство `Header` к классу `ContentControl`.

Простейший контейнер с заголовком представлен элементом `GroupBox`. Следующий пример демонстрирует использование этого элемента управления для группировки нескольких флажков:

```

<GroupBox Margin="5" Header="Grammar">
  <StackPanel>
    <CheckBox Content="Check grammar as you type" />
    <CheckBox Content="Hide grammatical errors" />
    <CheckBox Content="Check grammar with spelling" />
  </StackPanel>
</GroupBox>

```

Подчеркнём, что в заголовок контейнера может быть помещено любое содержимое – например, кнопка:


```

<GroupBox Margin="5">
  <GroupBox.Header>
    <Button Margin="5" Content="Grammar" />
  </GroupBox.Header>
  <StackPanel>
    <CheckBox Content="Check grammar as you type" />
    <CheckBox Content="Hide grammatical errors" />
    <CheckBox Content="Check grammar with spelling" />
  </StackPanel>
</GroupBox>

```

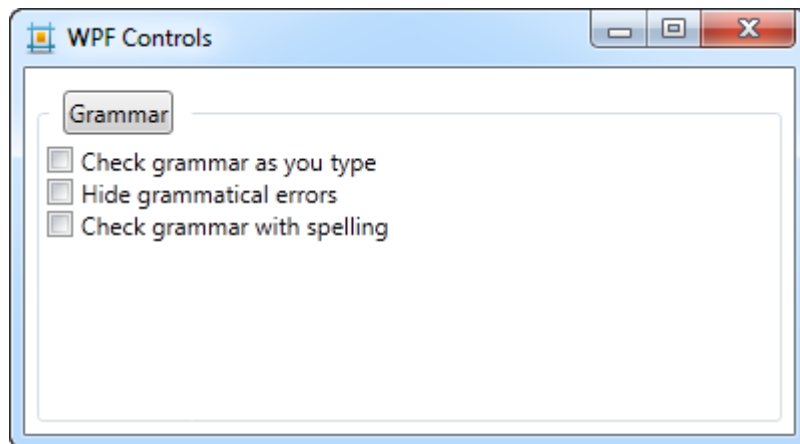


Рис. 15. Контейнер **GroupBox** с кнопкой в заголовке.

Элемент управления **Expander** напоминает **GroupBox**, но содержит в заголовке особую кнопку, которая позволяет спрятать и показать содержимое контейнера. Можно настроить направление разворачивания контейнера.

```

<Expander Header="Grammar" ExpandDirection="Right">
  <StackPanel Margin="10">
    <CheckBox Content="Check grammar as you type" />
    <CheckBox Content="Hide grammatical errors" />
    <CheckBox Content="Check grammar with spelling" />
  </StackPanel>
</Expander>

```

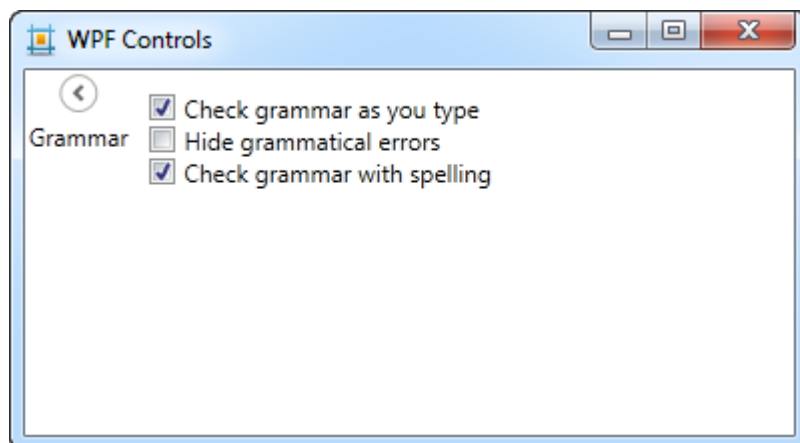


Рис. 16. Контейнер **Expander**, раскрытый вправо.

7.2. Списковые элементы управления

Списковые элементы управления (далее для краткости – *списки*) могут хранить и представлять коллекцию элементов. Для большинства списков доступна гибкая настройка внешнего вида, а также функции фильтрации, группировки и сортировки данных, однако в данном параграфе рассматриваются только простейшие аспекты работы со списками.

Все списки наследуются от класса `ItemsControl`, который определяет несколько полезных свойств:

`Items` – коллекция типа `ItemCollection`, сохраняющая элементы списка (свойство доступно только для чтения).

`ItemsSource` – свойство позволяет связать список с набором данных. Свойство доступно для установки во время выполнения приложения.

`HasItems` – булево свойство только для чтения. Показывает, имеются ли элементы в списке.

`DisplayMemberPath` – строковое свойство, определяющее отображаемое значение для элемента списка. `DisplayMemberPath` поддерживает синтаксис, который называется *путь к свойству*. Путь к свойству напоминает запись свойства агрегированного объекта, но без операций приведения типа. Например, если у кнопки `btn` задать в качестве содержимого строку, путь к свойству в виде `"btn.Content[0]"` будет означать первый символ строки.

`ItemStringFormat` – строка форматирования для элемента списка.

`AlternationCount` и `AlternationIndex` – свойства для настройки чередования визуальных стилей для элементов списка.

В разметке XAML дочерние объекты списка автоматически рассматриваются как его элементы. Правила отображения отдельного элемента аналогичны правилам, применяемым при показе содержимого `ContentControl`.

Списковые элементы управления можно разделить на три подгруппы: *селекторы, меню, списки без категории*. Селекторы допускают индексацию и выбор элементов списка. К селекторам относятся элементы управления `ListBox`, `ComboBox`, `TabControl`, `ListView` и `DataGrid`. Все селекторы наследуются от класса `Selector`, который предоставляет следующие свойства и события:

`SelectedItem` – первый выбранный элемент списка.

`SelectedIndex` – индекс выбранного элемента (-1, если ничего не выбрано).

`SelectedValuePath` – путь к свойству для значения выбранного элемента.

`SelectedValue` – значение выбранного элемента. Если не установлено свойство `SelectedValuePath`, это значение совпадает с `SelectedItem`.

`SelectionChanged` – событие, генерируемое при изменении `SelectedItem`.

В классе `Selector` определены два присоединённых булевых свойства, используемых с элементами списка, – `IsSelected` и `IsSelectionActive`. Второе свойство является свойством только для чтения и позволяет узнать, установлен ли на выделенном элементе фокус. Кроме этого, класс `Selector` имеет два присоединённых события – `Selected` и `Unselected`.

Элемент управления `ListBox` отображает список и допускает множественный выбор элементов. Эта возможность настраивается при помощи свойства `SelectionMode`, принимающего значения из одноимённого перечисления:

1. `Single` (по умолчанию) – можно выбрать только один элемент списка.
2. `Multiple` – можно выбрать несколько элементов. Щелчок по невыбранному элементу выбирает его, и наоборот.
3. `Extended` – множественный выбор, используя клавиши `<Shift>` и `<Ctrl>`.

У `ListBox` имеется коллекция `SelectedItem`s, содержащая выбранные элементы списка. Каждый элемент в `ListBox` представлен объектом класса `ListBoxItem`, в котором определено булево свойство `IsSelected` и события `Selected` и `Unselected`. В разметке XAML `ListBoxItem` можно не использовать. Анализатор XAML выполняет самостоятельное «оборачивание» дочерних элементов в объекты `ListBoxItem` (замечание справедливо и для других списков).

```
<ListBox SelectionMode="Multiple" SelectedIndex="2">
  <TextBlock Margin="5" Text="Text Block" />
  <Button Margin="5" Content="Button" />
  <CheckBox Margin="5" Content="Check Box" />
  <RadioButton Margin="5" Content="Radio Button" />
  <Label Margin="5" Content="Label" />
</ListBox>
```

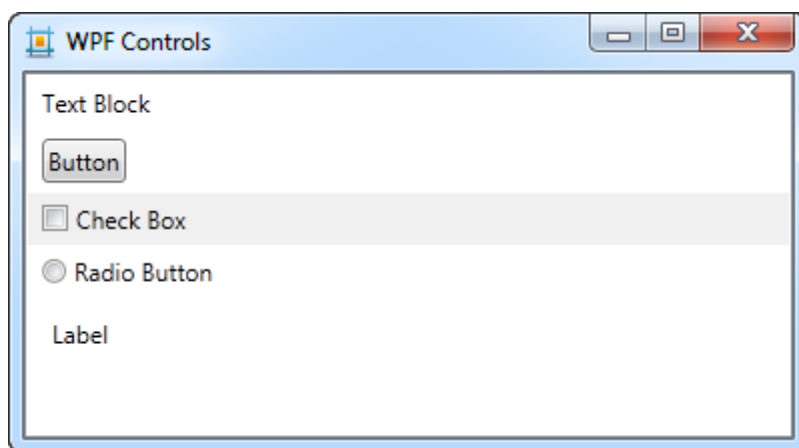


Рис. 17. Элемент управления `ListBox`.

Класс `ComboBox` позволяет выбрать один элемент из списка, отображая текущий выбор и раскрывая список элементов по требованию. Отдельный элемент в `ComboBox` представлен объектом `ComboBoxItem` (унаследованным от `ListBoxItem`). Для описания состояния списка служит булево свойство `IsDropDownOpen` и связанные события `DropDownOpened` и `DropDownClosed`. Свойства `IsEditable` и `IsReadOnly` могут использоваться, чтобы подсвечивать и отображать элемент списка при вводе текста, связанного с элементом. Этот текст определяется в элементе списка при помощи присоединённого свойства `TextSearch.Text` или в `ComboBox` при помощи свойства `TextSearch.TextPath`.

Элемент управления `TabControl` – это простой набор страниц с закладками. По умолчанию закладки расположены сверху. Их положение можно изменить,

используя свойство `TabStripPlacement`, принимающее значение из перечисления `System.Windows.Controls.Dock`. Каждая страница в `TabControl` – это объект класса `TabItem`. Класс `TabItem` – это контейнер с заголовком:

```
<TabControl SelectedIndex="1" TabStripPlacement="Left">
  <TabItem Header="Tab 1">Content for Tab 1</TabItem>
  <TabItem Header="Tab 2">Content for Tab 2</TabItem>
  <TabItem Header="Tab 3">Content for Tab 3</TabItem>
</TabControl>
```

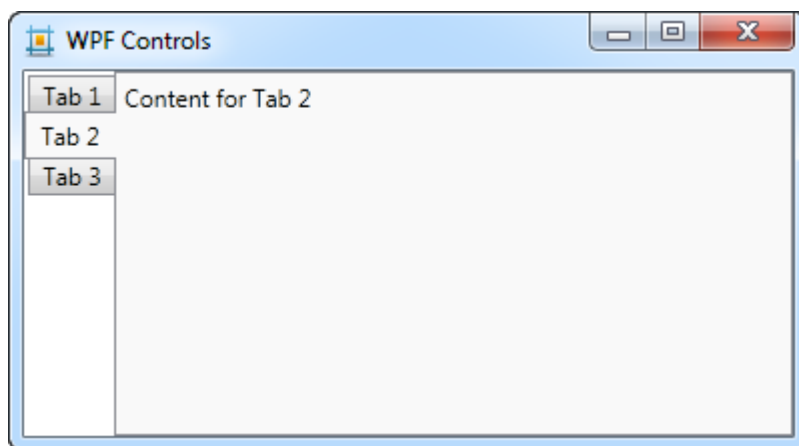


Рис. 18. Элемент управления `TabControl`.

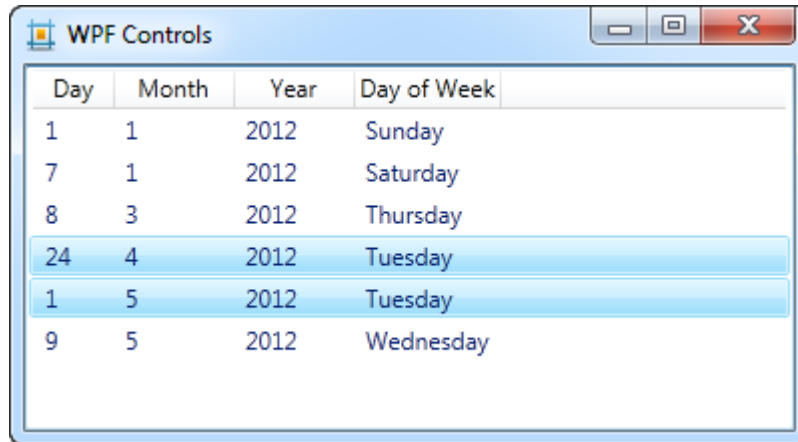
Класс `ListView` унаследован от `ListBox`. Этот список позволяет настроить свой внешний вид при помощи свойства `View` с типом `ViewBase`. WPF поставляется с одним наследником класса `ViewBase` – классом `GridView`. При помощи `GridView` элемент управления `ListView` отображает данные в виде таблицы с колонками. Каждая колонка – это объект класса `GridViewColumn`, помещённый в коллекцию `GridView.Columns`. У колонки настраивается ширина, заголовок, отображаемые в колонке значения (при помощи привязки данных). `GridView` также поддерживает изменение размеров колонок и их перетаскивание во время работы приложения.

```
<ListView xmlns:sys="clr-namespace:System;assembly=mcorlib">
  <ListView.View>
    <GridView>
      <GridViewColumn Header="Day" Width="40"
        DisplayMemberBinding="{Binding Day}"/>
      <GridViewColumn Header="Month" Width="60"
        DisplayMemberBinding="{Binding Month}"/>
      <GridViewColumn Header="Year" Width="60"
        DisplayMemberBinding="{Binding Year}"/>
      <GridViewColumn Header="Day of Week"
        DisplayMemberBinding="{Binding DayOfWeek}"/>
    </GridView>
  </ListView.View>
  <sys:DateTime>1/1/2012</sys:DateTime>
  <sys:DateTime>1/7/2012</sys:DateTime>
```

```

<sys:DateTime>3/8/2012</sys:DateTime>
<sys:DateTime>4/24/2012</sys:DateTime>
<sys:DateTime>5/1/2012</sys:DateTime>
<sys:DateTime>5/9/2012</sys:DateTime>
</ListView>

```



Day	Month	Year	Day of Week
1	1	2012	Sunday
7	1	2012	Saturday
8	3	2012	Thursday
24	4	2012	Tuesday
1	5	2012	Tuesday
9	5	2012	Wednesday

Рис. 19. `ListView` показывает праздники первого полугодия 2012 года.

Таблица `DataGrid` позволяет отображать и редактировать данные. Внешний вид `DataGrid` определяют колонки, которые хранятся в коллекции `Columns`. Колонки бывают нескольких классов (каждый унаследован от `DataGridColumn`):

1. `DataGridTextColumn` – отображает текст или поле ввода;
2. `DataGridHyperlinkColumn` – отображает гиперссылку;
3. `DataGridCheckBoxColumn` – передаёт булевы значения как `CheckBox`;
4. `DataGridComboBoxColumn` – отображает текст при просмотре и выпадающий список при редактировании;
5. `DataGridTemplateColumn` – колонка использует для отображения шаблоны, указанные в свойствах `CellTemplate` и `CellEditingTemplate`.

Колонки `DataGrid` могут быть описаны вручную или генерироваться автоматически на основе отображаемых данных. Каждая колонка допускает гибкую настройку своего внешнего вида. Ниже представлен пример использования элемента `DataGrid`:

```

<DataGrid x:Name="dataGrid" AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTextColumn Header="Name" Binding="{Binding Name}"/>
    <DataGridHyperlinkColumn Header="Website"
      Binding="{Binding Site}" />
    <DataGridCheckBoxColumn Header="Has kids?"
      Binding="{Binding HasKids}" />
  </DataGrid.Columns>
</DataGrid>

public class Person
{
  public string Name { get; set; }
}

```

```

    public Uri Site { get; set; }
    public bool HasKids { get; set; }
}

// код в конструкторе MainWindow - заполняем DataGrid
var uri = new Uri("http://www.eden.com");
var adam = new Person {Name = "Adam", Site = uri, HasKids = true};
var eve = new Person {Name = "Eve", Site = uri, HasKids = true};
dataGrid.ItemsSource = new List<Person> {adam, eve};

```

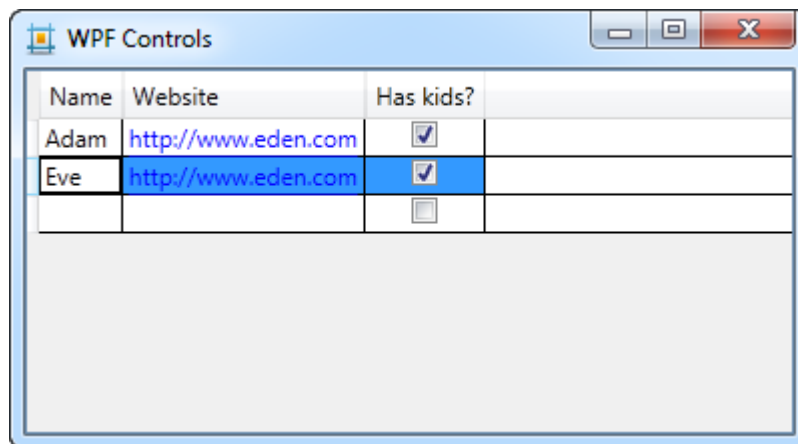


Рис. 20. `DataGrid` в режиме редактирования.

Перейдём к рассмотрению элементов управления из категории меню. Класс `Menu` может содержать коллекцию любых объектов, но ожидается, что будут использованы объекты `MenuItem` и `Separator`. `Separator` – простой элемент, представляющий разделитель пунктов меню. `MenuItem` – это контейнер с заголовком (наследник `HeaderedItemsControl`). В случае `MenuItem` заголовок определяет отображаемый пункт меню, а свойство содержимого `Items` может содержать элементы подменю. `MenuItem` поддерживает «горячие клавиши». Этот класс также содержит свойства `Icon` и `IsCheckable` и определяет события `Checked`, `Unchecked`, `SubmenuOpened`, `SubmenuClosed`, `Click`.

```

<DockPanel LastChildFill="False">
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">
      <MenuItem Header="_New..." />
      <MenuItem Header="_Open..." />
      <Separator />
      <MenuItem Header="Sen_d To">
        <MenuItem Header="Mail Recipient" />
        <MenuItem Header="My Documents" />
      </MenuItem>
    </MenuItem>
    <MenuItem Header="_Edit" />
    <MenuItem Header="_View" />
  </Menu>
</DockPanel>

```

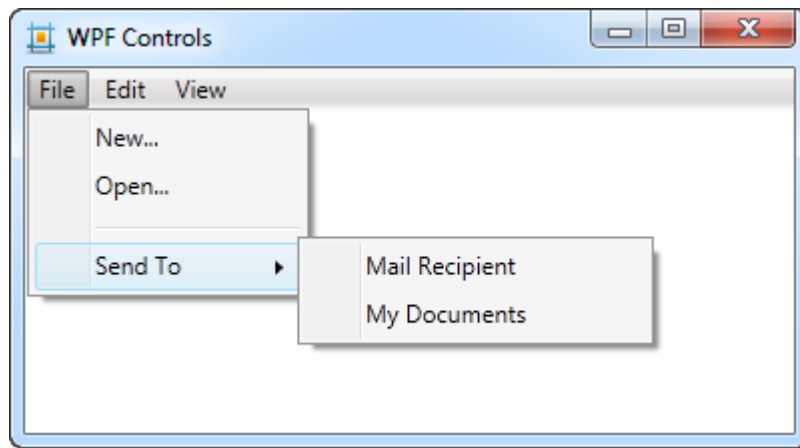


Рис. 21. Пример меню.

Класс [Ribbon](#) (пространство имён `System.Windows.Controls.Ribbon` и одноимённая сборка) позволяет создать *ленту* – элемент интерфейса с меню, панелями инструментов и вкладками. Для описания структуры ленты используются специальные классы¹ (например, [RibbonButton](#) или [RibbonMenuItem](#)).

Класс [ContextMenu](#) является контейнером объектов [MenuItem](#) и служит для представления контекстного меню некоторого элемента. В классе [ContextMenu](#) определены свойства для настройки места появления контекстного меню относительно элемента, с которым оно связано. Объект [ContextMenu](#) нельзя поместить в дерево элементов, а следует задавать как значение одноимённого свойства, определённого в классе [FrameworkElement](#).

Рассмотрим списковые элементы без категории. Элемент управления [TreeView](#) отображает иерархию данных в виде дерева с узлами, которые могут быть свёрнуты и раскрыты. Каждый элемент в [TreeView](#) является объектом [TreeViewItem](#). Класс [TreeViewItem](#) напоминает [MenuItem](#) – это тоже контейнер с заголовком `Header` и коллекцией дочерних элементов `Items`. Класс [TreeViewItem](#) содержит булевы свойства `IsExpanded` и `IsSelected`, а также события `Expanded`, `Collapsed`, `Selected`, `Unselected`.

```
<TreeView>
  <TreeViewItem Header="ItemsControl">
    <TreeViewItem Header="Selector">
      <TreeViewItem Header="ListBox" />
      <TreeViewItem Header="ComboBox" />
      <TreeViewItem Header="TabControl" />
    </TreeViewItem>
    <TreeViewItem Header="HeaderedItemsControl" >
      <TreeViewItem Header="MenuItem" />
      <TreeViewItem Header="TreeViewItem" />
    </TreeViewItem>
  </TreeViewItem>
</TreeView>
```

¹ Класс [RibbonWindow](#) описывает окно специального вида, используемое совместно с лентой.

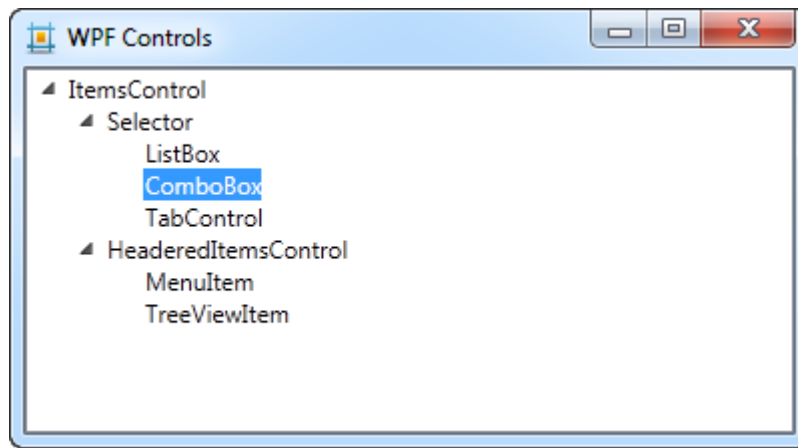


Рис. 22. Простой **TreeView**.

Элемент управления **ToolBar** представляет панель инструментов. Он просто группирует кнопки, разделители (**Separator**) и другие дочерние элементы на одной панели. Хотя **ToolBar** можно разместить в произвольном месте, обычно один или несколько таких объектов размещают внутри элемента **ToolBarTray**, что позволяет перетаскивать и переупорядочить панели инструментов. Элемент управления **StatusBar** группирует элементы подобно **ToolBar**. Обычно элемент **StatusBar** размещается в нижней части окна.

7.3. Прочие элементы управления

Рассмотрим оставшиеся стандартные элементы управления, разбив их на следующие подгруппы: *текстовые элементы, элементы для представления диапазона, элементы для работы с датами.*

К текстовым элементам относятся **TextBlock**, **TextBox**, **RichTextBox**, **PasswordBox**. **TextBlock** формально не является элементом управления, так как унаследован непосредственно от **FrameworkElement**. Он предназначен для отображения небольшой порции текста. Для текста доступны настройки шрифта, выравнивания, переноса, а сам текст может включать теги XAML XPS.

```
<StackPanel>
  <!-- текст в разметке -->
  <TextBlock x:Name="tb1" Text="Simple TextBlock" />
  <TextBlock x:Name="tb2" TextWrapping="Wrap" FontSize="20">
    <Bold>TextBlock</Bold> is designed to be
    <Italic>lightweight.</Italic>
  </TextBlock>
</StackPanel>

// текст в коде
tb1.Text = "Simple TextBlock";

tb2.Inlines.Clear();
tb2.Inlines.Add(new Bold(new Run("TextBlock")));
tb2.Inlines.Add(new Run(" is designed to be "));
tb2.Inlines.Add(new Italic(new Run("lightweight.")));
```

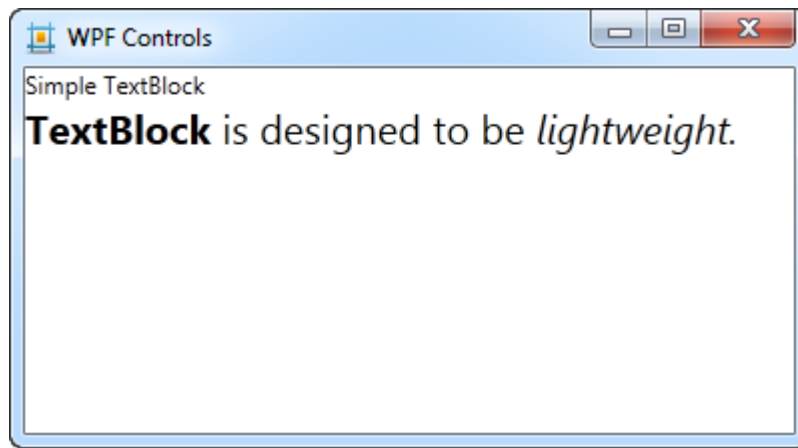


Рис. 23. Демонстрация возможностей `TextBlock`.

Элемент управления `TextBox` служит для отображения и ввода текста, заданного в строковом свойстве `Text`. `TextBox` определяет события `TextChanged` и `SelectionChanged`, а также содержит несколько свойств и методов для работы с частью введённого текста. Для управления переносом текста следует использовать свойство `TextWrapping`. Чтобы пользователь мог ввести несколько строк текста, нажимая `<Enter>`, установите свойство `AcceptsReturn` в `true`. Элемент управления `TextBox` поддерживает проверку правописания (для английского, испанского, французского и немецкого языка). Для этого необходимо установить свойство зависимости `SpellCheck.IsEnabled` в `true`.

```
<TextBox SpellCheck.IsEnabled="True" Text="Error" />
```

Элемент управления `RichTextBox` – это «продвинутая» версия `TextBox`. Многие свойства у этих элементов общие, так как они унаследованы от одного базового класса `TextBoxBase`. Содержимое `RichTextBox` сохраняется в свойстве `Document` типа `FlowDocument`, который создан для поддержки XPS.

Элемент управления `PasswordBox` предназначен для ввода паролей. Можно сказать, что это упрощённая версия `TextBox` – не поддерживаются вырезание и копирование текста, не генерируются события `TextChanged` и `SelectionChanged`. В `PasswordBox` введённый пароль сохраняется в свойстве `Password`, а при изменении текста генерируется событие `PasswordChanged`. Символ, который отображается вместо букв пароля, настраивается при помощи свойства `PasswordChar`.

Элементы для представления диапазона `ProgressBar` и `Slider` хранят и отображают в некой форме числовое значение, попадающее в заданный диапазон. Оба элемента унаследованы от класса `RangeBase`, имеющего свойства `Value`, `Minimum`, `Maximum` (все – типа `double`) и событие `ValueChanged`.

`ProgressBar` обычно используют для визуализации процесса выполнения длительной операции. Класс `ProgressBar` добавляет к `RangeBase` два свойства: `IsIndeterminate` – если установить это свойство в `true`, будет показываться непрерывная бегущая полоска; `Orientation` – размещение `ProgressBar` (`Horizontal` или `Vertical`).

Элемент управления **Slider** – это слайдер (ползунок) с возможностью ручной установки значения из диапазона. У слайдера имеется свойство **Orientation** и несколько свойств, управляющих метками (например, **TickPlacement**). Кроме этого, слайдер позволяет задать выделенный диапазон при помощи свойств **IsSelectionRangeEnabled**, **SelectionStart** и **SelectionEnd**.

```
<StackPanel>
  <ProgressBar Value="80" Height="20" Margin="10"/>
  <Slider Maximum="30" Value="25"
    TickPlacement="BottomRight" TickFrequency="2"
    IsSelectionRangeEnabled="True"
    SelectionStart="10" SelectionEnd="20"/>
  <Slider Height="100" Maximum="30" HorizontalAlignment="Center"
    Orientation="Vertical" />
</StackPanel>
```

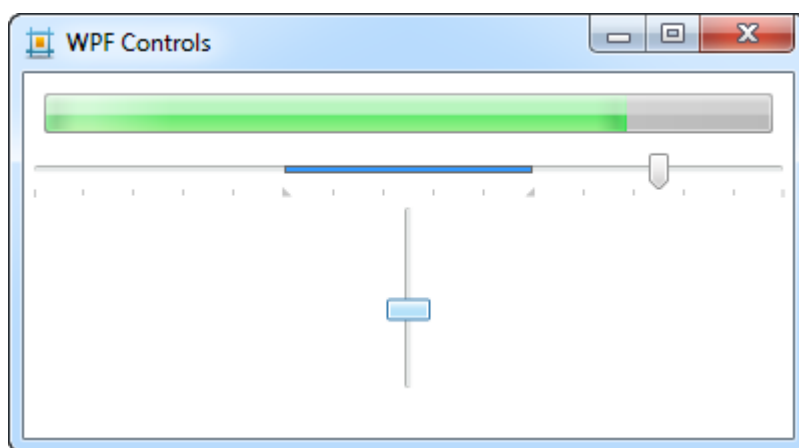


Рис. 24. Элемент **ProgressBar** и два слайдера.

Элементами для работы с датами являются **Calendar** и **DatePicker**. **Calendar** отображает небольшой календарь с возможностью клавиатурной навигации и выбора. Этот класс имеет несколько полезных свойств:

DisplayMode – режим отображения календаря (Year, Month, Decade);

SelectionMode – режим выбора дат (SingleDate, SingleRange, MultipleRange, None);

BlackoutDates – коллекция дат, которые не могут быть выбраны;

DisplayDate – текущая отображаемая дата;

DisplayDateStart и **DisplayDateEnd** – задают доступный диапазон дат;

IsTodayHighlighted – подсветка текущей даты;

SelectedDate и **SelectedDates** – выбранная дата или коллекция дат.

Элемент **DatePicker** позволяет задать дату, набирая её с клавиатуры или применяя выпадающий элемент **Calendar**. Большинство свойств **DatePicker** служит для настройки этого внутреннего календаря. Свойство **Text** содержит введенный в **DatePicker** текст. Если этот текст нельзя конвертировать в дату, генерируется событие **DateValidationError** (что по умолчанию приводит к исключительной ситуации).

```

<StackPanel Orientation="Horizontal">
    <Calendar DisplayMode="Month" DisplayDate="1/1/2012"
        DisplayDateEnd="12/31/2012">
        <Calendar.BlackoutDates>
            <CalendarDateRange Start="1/1/2012" End="1/10/2012" />
        </Calendar.BlackoutDates>
    </Calendar>

    <DatePicker Margin="20,0" VerticalAlignment="Top"
        SelectedDateFormat="Long" SelectedDate="1/1/2012"
        DisplayDateStart="1/1/12" DisplayDateEnd="12/31/12"
        FirstDayOfWeek="Monday" />
</StackPanel>

```

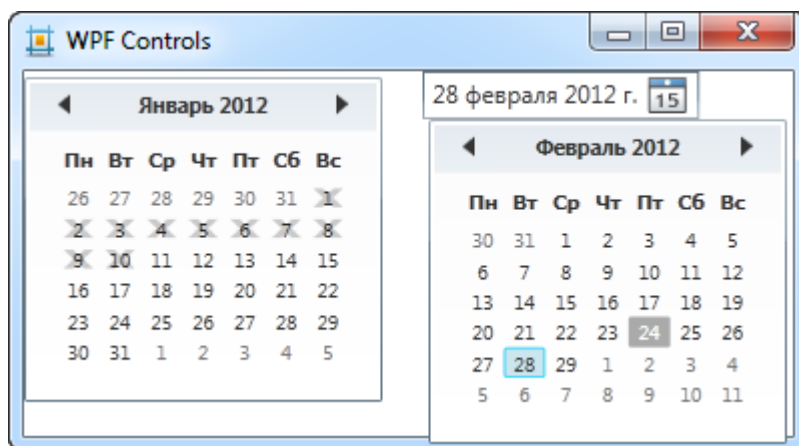


Рис. 25. Элементы управления `Calendar` и `DatePicker`.

8. Ресурсы

Платформа .NET поддерживает инфраструктуру для работы с *ресурсами* – информационными фрагментами приложения, представляющими изображения, таблицы строк или иные данные. WPF расширяет базовые возможности .NET, предоставляя поддержку двух видов ресурсов – двоичных и логических.

8.1. Двоичные ресурсы

Двоичный ресурс в WPF – это традиционный ресурс с точки зрения платформы .NET. Обычно на этапе разработки двоичный ресурс представлен в виде файла в составе проекта. Такой файл может быть внедрён в сборку .NET или существовать в виде отдельного компонента, логически связанного со сборкой. Это поведение настраивается в Visual Studio в окне свойств файла. Установите свойство `Build Action` в значение `Resource` для внедрения ресурса, или в `Content` для закрепления связи между отдельным файлом ресурса и сборкой.

Для доступа к двоичным ресурсам в WPF обычно используется универсальный идентификатор ресурса в формате упакованного URI¹. *Упакованный URI*

¹ Если файл ресурса не связывался со сборкой при компиляции, можно использовать обычный URI (путь к файлу на локальной или сетевой машине, адрес в интернет).

имеет вид `pack://контейнер/путь`. Ниже представлены типичные примеры упакованных URI:

`pack://application:,,,/img.png` – ресурс `img.png` (изображение), внедрённый в текущую сборку, или файл `img.png`, ассоциированный со сборкой при компиляции.

`pack://application:,,,/images/img.png` – аналогично предыдущего URI, но в проекте Visual Studio файл `img.png` располагался в подкаталоге `images`.

`pack://application:,,,/MyAssembly;component/img.png` – ресурс `img.png`, внедрённый в сборку `MyAssembly.dll`.

`images/img.png` – относительный упакованный URI для ссылки на ресурс, связанный с текущей сборкой.

Для работы с URI служит класс `System.Uri`. Рассмотрим примеры связывания элемента `Image` с изображением, представленным в виде ресурса:

```
// этот объект описывает абсолютный URI
var absoluteUri = new Uri("pack://application:,,,/images/img.png");

// этот объект описывает относительный URI
var relativeUri = new Uri("images/img.png", UriKind.Relative);

// создадим элемент Image
// и свяжем его с bitmap-изображением, используя абсолютный URI
var picture = new Image();
picture.Source = new BitmapImage(absoluteUri);
```

В разметке XAML ссылки на ресурсы обычно задаются простыми строками, так как имеется стандартный конвертер из строки в объект URI:

```
<Image x:Name="picture" Source="images/img.png" />
```

8.2. Логические ресурсы

Логические (объектные) ресурсы – это произвольные объекты, ассоциированные с элементом WPF. Классы `FrameworkElement` и `FrameworkContentElement` определяют словарь `Resources` с типом `System.Windows.ResourceDictionary`. Этот словарь хранит коллекцию логических ресурсов элемента. С коллекцией `Resources` можно работать как в коде, так и в разметке XAML. В последнем случае объект, помещаемый в ресурс, должен обладать конструктором без параметров. Кроме этого, для определения ключа словаря в разметке требуется использовать атрибут `x:Key` из пространства имён анализатора XAML:

```
<!-- определение ресурсов для кнопки в XAML -->
<Button x:Name="btn" Content="OK">
    <Button.Resources>
        <SolidColorBrush x:Key="background" Color="Yellow" />
        <SolidColorBrush x:Key="borderBrush" Color="Red" />
    </Button.Resources>
</Button>
```

```
// эквивалентное определение ресурсов для кнопки в коде
btn.Resources.Add("background", new SolidColorBrush(Colors.Yellow));
btn.Resources.Add("borderBrush", new SolidColorBrush(Colors.Red));
```

Чтобы сослаться на ресурс в разметке XAML, необходимо использовать расширения разметки `StaticResourceExtension` или `DynamicResourceExtension`. При этом указывается ключ ресурса. Анализатор XAML выполняет поиск ресурса по ключу, просматривая коллекцию ресурсов элемента, затем ресурсы родительского элемента, и так далее. Вот почему ресурсы обычно объявляются на уровне родительского окна или объекта `Application`.

```
<Window xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Window.Resources>
        <SolidColorBrush x:Key="borderBrush" Color="Red" />
    </Window.Resources>

    <Button Content="OK" BorderBrush="{StaticResource borderBrush}" />
</Window>
```

Статическая ссылка на ресурс сохраняет объект ресурса, а динамическая — ключ объекта. Поэтому при динамическом использовании любая модификация ресурса (даже после применения) отразится на целевом объекте. Ниже показан пример статического и динамического применения ресурсов в коде:

```
// статическое применение ресурса с поиском по дереву элементов
btn.Background = (Brush) btn.FindResource("background");

// статическое применение ресурса из заданной коллекции ресурсов
btn.Background = (Brush) window.Resources["background"];

// динамическое применение ресурса
btn.SetResourceReference(Button.BackgroundProperty, "background");
```

Определение логических ресурсов в XAML часто выносится в отдельный *файл ресурсов*:

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

    <Image x:Key="logo" Source="logo.jpg" />
</ResourceDictionary>
```

Для того чтобы объединить ресурсы в файлах с коллекцией `Resources` некоего элемента, следует использовать свойство `MergedDictionaries` класса `ResourceDictionary`:


```

<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="file1.xaml" />
      <ResourceDictionary Source="file2.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>

```

9. Привязка данных

9.1. Базовые концепции привязки данных

Привязка данных (data binding) – это отношение, которое используется для извлечения информации из *источника данных* и установки свойства зависимости в *целевом объекте*. Целевой объект обычно является элементом управления. Источником данных может быть произвольный объект .NET. Привязка способна (при определённых условиях) автоматически обновлять целевое свойство при изменении данных источника.

Очевидно, что при определении привязки нужно указать **целевое свойство**, **источник данных** и **правило извлечения информации** из источника. Кроме этого, можно выполнить настройку следующих параметров:

1. Направление привязки. Привязка может быть *однонаправленной* (целевое свойство меняется при изменении данных источника), *двунаправленной* (изменения в источнике и целевом объекте влияют друг на друга) и *от цели к источнику* (источник данных обновляется при изменении целевого свойства).

2. Условие обновления источника. Если привязка двунаправленная или от цели к источнику, можно настроить момент обновления источника данных.

3. Конвертеры значений. Привязка может выполнять автоматическое преобразование данных при их перемещении от источника к целевому объекту и наоборот.

4. Правила проверки данных. Привязка может включать правила проверки данных на корректность и поведение, выполняемое при нарушении этих правил.

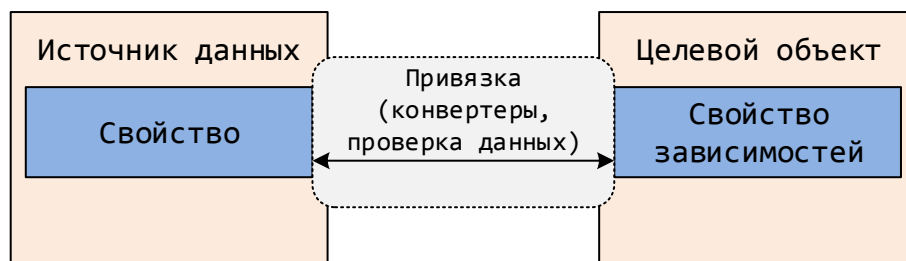


Рис. 26. Концепция привязки данных.

В WPF привязка данных описывается либо декларативно, либо в коде. В любом случае используется объект класса `System.Windows.Data.Binding` (унаследованный от `MarkupExtension`). В табл. 5 приведено описание основных свойств этого класса.

Свойства класса `Binding`

Имя свойства	Описание
<code>Converter</code>	Конвертер для преобразования данных при привязке
<code>ConverterCulture</code>	Культура, передаваемая в конвертер
<code>ConverterParameter</code>	Произвольный объект, передаваемый в конвертер
<code>Delay</code>	Время задержки (в миллисекундах) операции обновления источника при изменении целевого свойства
<code>ElementName</code>	Имя элемента в дереве объектов, который будет источником данных для привязки
<code>FallbackValue</code>	Значение для целевого свойства, которое будет использоваться, если операция привязки закончилась неудачей
<code>IsAsync</code>	При установке в <code>true</code> привязка выполняется асинхронно
<code>Mode</code>	Направление привязки (перечисление <code>BindingMode</code>): <code>OneWay</code> – от источника к целевому свойству; <code>TwoWay</code> – от источника к целевому свойству и наоборот; <code>OneTime</code> – целевое свойство устанавливается на основе данных источника, затем изменения в источнике игнорируются; <code>OneWayToSource</code> – источник обновляется при изменении целевого свойства; <code>Default</code> – используется направление привязки, заданное в метаданных свойства зависимостей
<code>NotifyOnSourceUpdated</code>	При установке в <code>true</code> при передаче информации от источника к целевому объекту генерируется событие <code>SourceUpdated</code>
<code>NotifyOnTargetUpdated</code>	При установке в <code>true</code> при передаче информации от целевого объекта к источнику генерируется событие <code>TargetUpdated</code>
<code>NotifyOnValidationError</code>	Если равно <code>true</code> , при наличии ошибок проверки данных у целевого объекта генерируется присоединённое событие <code>Error</code>
<code>Path</code>	Путь к информации в источнике данных. Аргумент конструктора класса <code>Binding</code>
<code>RelativeSource</code>	Путь к источнику, задаваемый относительно целевого объекта
<code>Source</code>	Источник данных
<code>StringFormat</code>	Форматирование для извлекаемых данных строкового типа
<code>TargetNullValue</code>	Значение, которое будет использоваться для целевого свойства, если из источника извлекается <code>null</code>
<code>ValidatesOnDataErrors</code>	Булево значение: указывает на использование объектом-источником интерфейса <code>IDataErrorInfo</code>
<code>ValidatesOnExceptions</code>	Булево значение: указывает на необходимость рассматривать исключения как ошибки проверки данных
<code>ValidatesOnNotifyDataErrors</code>	Булево значение: указывает на использование объектом-источником интерфейса <code>INotifyDataErrorInfo</code>
<code>ValidationRules</code>	Коллекция объектов, определяющих правила проверки данных
<code>UpdateSourceExceptionFilter</code>	Метод обработки исключений, генерируемых при проверке данных

Имя свойства	Описание
UpdateSourceTrigger	Задаёт момент обновления источника при изменениях в целевом свойстве (перечисление <code>UpdateSourceTrigger</code>): PropertyChanged – немедленно при изменении; LostFocus – при потере целевым элементом фокуса ввода; Explicit – при вызове метода <code>UpdateSource()</code> ; Default – по значению, заданному в метаданных свойства зависимостей при помощи <code>DefaultUpdateSourceTrigger</code>
XPath	Выражение XPath. Может использоваться, если источник привязки возвращает XML

Класс `Binding` позволяет описывать привязку в XAML или в коде. В следующем фрагменте разметки XAML привязка соединяет два свойства разных элементов управления – при перемещении слайдера автоматически меняется размер шрифта текста:

```
<StackPanel>
  <Slider x:Name="slider" Minimum="1" Maximum="40" />
  <TextBlock x:Name="text" Text="Sample Text"
    FontSize="{Binding ElementName=slider, Path=Value}" />
</StackPanel>
```

Рассмотрим этот же пример, реализованный при помощи кода:

```
// создаём и настраиваем объект Binding
var binding = new Binding();
binding.ElementName = "slider";
binding.Path = new PropertyPath("Value");

// устанавливаем привязку
text.SetBinding(TextBlock.FontSizeProperty, binding);
```

Класс `System.Windows.Data.BindingOperations` предоставляет набор статических методов для управления привязкой в коде. В частности, метод `GetBinding()` позволяет получить привязку для указанного целевого объекта и его свойства зависимостей, метод `SetBinding()` устанавливает привязку, а метод `ClearBinding()` удаляет привязку.

```
// удаляем установленную привязку для элемента text
BindingOperations.ClearBinding(text, TextBlock.FontSizeProperty);
```

9.2. Источники и поставщики данных

Источник данных для привязки может быть задан при помощи следующих свойств объекта `Binding`: `ElementName`, `Source`, `RelativeSource`. Эти свойства являются взаимоисключающими – установив одно из них, остальные следует сбросить, иначе при выполнении привязки генерируется исключение.

Строковое свойство `ElementName` удобно использовать, чтобы определить в качестве источника данных элемент, который принадлежит одному логическому

дереву вместе с целевым объектом. Этот элемент должен иметь имя, оно и указывается в `ElementName`. Кроме установки `ElementName`, у привязки обычно задается свойство `Path`, определяющее путь к извлекаемым из элемента данным. Свойство `Path` имеет тип `System.Windows.PropertyPath`. При работе с `Path` в разметке XAML следует иметь в виду, что существует стандартный конвертер из строки в объект `PropertyPath`. Класс `Binding` имеет перегруженный конструктор, принимающий строку, трактуемую как значение свойства `Path`.

```
<!-- указали ElementName и явно задали Path -->
<TextBlock Text="{Binding ElementName=slider, Path=Value}" />
<!-- Path задан как аргумент конструктора Binding -->
<TextBlock Text="{Binding Value, ElementName=slider}" />
```

Свойство `Path` формируется по следующим правилам. В простейшем случае `Path` содержит имя свойства в источнике данных: `Path=PropertyName`. Для указания свойств агрегируемых объектов используется разделитель-точка: `Path=Property.Subproperty`. Для ссылки на присоединённое свойство это свойство записывается в скобках: `Path=(DockPanel.Dock)`. При работе с индексатором индексы записываются в квадратных скобках: `Path=Data[0]`.

Свойство `Source` даёт возможность указать в качестве источника привязки произвольный объект. В следующем примере привязка выполняется к объекту класса `Person`, сохранённому в словаре ресурсов окна.

```
// класс Person объявлен в пространстве имён WpfBinding
public class Person
{
    public string Name { get; set; }
    public double Age { get; set; }
}

<Window x:Class="WpfBinding.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfBinding"
    Height="200" Width="400" Title="WPF Binding">
    <Window.Resources>
        <local:Person x:Key="smith" Name="Mr. Smith" Age="27.3" />
    </Window.Resources>

    <Grid Margin="10">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="40" />
            <RowDefinition Height="40" />
        </Grid.RowDefinitions>
```

```

<TextBlock Grid.Column="0" Grid.Row="0" Text="Name" />
<TextBox Grid.Column="1" Grid.Row="0"
    Text="{Binding Name, Source={StaticResource smith}}" />

<TextBlock Grid.Column="0" Grid.Row="1" Text="Age" />
<TextBox Grid.Column="1" Grid.Row="1"
    Text="{Binding Age, Source={StaticResource smith}}" />
</Grid>
</Window>

```

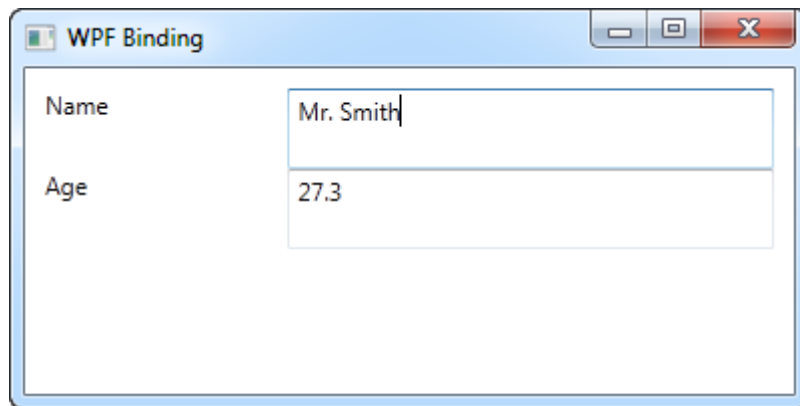


Рис. 27. Демонстрация привязки к объекту.

Свойство привязки `RelativeSource` позволяет задать источник данных на основе отношения к целевому объекту. Например, можно использовать это свойство для привязки элемента к самому себе или к элементу, находящемуся на неизвестное число шагов вверх в логическом дереве элементов. Для установки свойства используется объект `System.Windows.Data.RelativeSource`. Главным элементом этого объекта является свойство `Mode`, принимающее значение из перечисления `RelativeSourceMode`:

`Self` – привязка целевого элемента к самому себе.

`FindAncestor` – привязка к предку в логическом дереве. Для спецификации типа предка нужно установить свойство `AncestorType`. Можно использовать целочисленное свойство `AncestorLevel`, чтобы пропустить определённое число вхождений предка.

`PreviousData` – привязка к предыдущему объекту в коллекции.

`TemplatedParent` – привязка к элементу, для которого применён шаблон (этот режим работает только для привязки внутри шаблона элемента управления или шаблона данных).

Приведём несколько примеров использования `RelativeSource`. Заметим, что конструктор класса `RelativeSource` имеет перегруженные версии, принимающее в качестве аргументов значение для свойства `Mode` и, возможно, значения для `AncestorType` и `AncestorLevel`. Класс `RelativeSource` унаследован от `MarkupExtension`, а значит, его можно использовать как расширение разметки.

```

<!-- привязка к заголовку окна, в котором находится TextBlock -->
<TextBlock Text="{Binding Title,
                    RelativeSource={RelativeSource Mode=FindAncestor,
                    AncestorType={x:Type Window}}}" />

<!-- перепишем, используя аргументы конструктора RelativeSource -->
<TextBlock Text="{Binding Title, RelativeSource=
                    {RelativeSource FindAncestor, Window, 1}}" />

<!-- привязка элемента к самому себе -->
<TextBlock Text="{Binding Width,
                    RelativeSource={RelativeSource Self}}" />

```

Если при описании привязки не задано ни одно из свойств `ElementName`, `Source`, `RelativeSource`, источник данных определяется на основе значения свойства `DataContext` целевого объекта. Если у целевого объекта это свойство не установлено, будет выполнен поиск по дереву элементов для нахождения первого родительского `DataContext`, не равного `null`.

Изменим пример, использовавшийся для демонстрации свойства привязки `Source`, чтобы применялся `DataContext`:

```

<!-- описание окна и его ресурсов опущено для краткости -->
<Grid Margin="10" DataContext="{StaticResource smith}">
  <!-- описание структуры Grid опущено для краткости -->

  <TextBlock Grid.Column="0" Grid.Row="0" Text="Name" />
  <TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Name}" />

  <TextBlock Grid.Column="0" Grid.Row="1" Text="Age" />
  <TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Age}" />
</Grid>

```

Технология WPF поддерживает механизм *поставщиков данных* (data providers), чтобы упростить декларативную настройку доступа к источникам данных. В настоящее время имеется два поставщика: `ObjectDataProvider` получает информацию, вызывая заданный метод, а `XmlDataProvider` доставляет данные непосредственно из источника XML. Все поставщики данных унаследованы от класса `System.Windows.Data.DataSourceProvider`.

Для применения `ObjectDataProvider` у него нужно настроить следующие свойства: `ObjectType` — тип, который содержит метод извлечения данных; `MethodName` — имя метода, возвращающего данные (это может быть как экземплярный, так и статический метод). Дополнительно можно указать аргументы для передачи конструктору типа (коллекция `ConstructorParameters`) и список аргументов метода извлечения данных (коллекция `MethodParameters`). Если булево значение `IsAsynchronous` установлено в `true`, выполнения метода извлечения данных происходит асинхронно.

```

<!-- используются следующие пространства имён
xmlns:local="clr-namespace:WpfBinding"
xmlns:system="clr-namespace:System;assembly=mscorlib" -->
<Window.Resources>
  <ObjectDataProvider x:Key="provider"
    ObjectType="{x:Type local:PersonDatabase}"
    MethodName="Read">
    <ObjectDataProvider.MethodParameters>
      <system:Int32>42</system:Int32>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
</Window.Resources>

// класс PersonDatabase объявлен в пространстве имён WpfBinding
public class PersonDatabase
{
    public Person Read(int id)
    {
        return new Person();
    }
}

```

Поставщик `XmlDataProvider` предлагает простой способ извлечения XML-данных из отдельного файла, местоположения в Интернете или ресурса приложения. Чтобы использовать `XmlDataProvider`, он должен быть определён и ориентирован на нужный файл или URI за счёт установки свойства `Source`. Для фильтрации данных можно применить свойство `XPath`. По умолчанию `XmlDataProvider` загружает XML-содержимое асинхронно, если только явно не установить свойство `IsAsynchronous` в `false`.

```

<!-- показано описание ресурсов некоего окна
файл store.xml содержит XML-элементы Products -->
<Window.Resources>
  <XmlDataProvider x:Key="provider" Source="/store.xml"
    XPath="/Products" />
</Window.Resources>

```

9.3. Обновление данных и направление привязки

Ранее было отмечено, что привязка способна автоматически обновлять целевое свойство при изменении источника данных. Чтобы реализовать этот функционал, можно поступить одним из следующих способов:

1. Сделать свойство источника данных свойством зависимостей.
2. Генерировать при изменении источника событие *Имя-СвойстваChanged*.
3. Реализовать в источнике интерфейс `INotifyPropertyChanged` (пространство имён `System.ComponentModel`), содержащий событие `PropertyChanged`. Это событие нужно генерировать при изменении данных, указывая имя свойства.

Проведём модификацию класса `Person`, используя третий подход:

```

public class Person : INotifyPropertyChanged
{
    private string _name;
    private double _age;

    public string Name
    {
        get { return _name; }
        set { SetProperty(ref _name, value, "Name"); }
    }

    public double Age
    {
        get { return _age; }
        set { SetProperty(ref _age, value, "Age"); }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void SetProperty<T>(ref T field, T value, string name)
    {
        if (!EqualityComparer<T>.Default.Equals(field, value))
        {
            field = value;
            var handler = PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(name));
            }
        }
    }
}

```

Для того чтобы изменения в целевом свойстве попадали в источник данных, нужно установить режим привязки в `BindingMode.TwoWay` или `BindingMode.OneWayToSource`. При необходимости можно также задать свойство привязки `UpdateSourceTrigger` — момент обновления источника данных. Если `UpdateSourceTrigger` установить в `UpdateSourceTrigger.Explicit`, требуется написать код, вызывающий для обновления метод `UpdateSource()` у объекта `BindingExpression`, связанного с элементом:

```

// tbxAge - это поле ввода, привязанное к свойству возраста
BindingExpression be =
    tbxAge.GetBindingExpression(TextBox.TextProperty);
be.UpdateSource();

```


9.4. Конвертеры значений

Конвертер значений отвечает за преобразование данных при переносе из источника в целевое свойство и за обратное преобразование в случае двунаправленной привязки.

Для создания конвертера значений требуется выполнить четыре шага.

1. Создать класс, реализующий интерфейс `IValueConverter`.
2. Применить к классу атрибут `[ValueConversion]` и специфицировать исходный и целевой типы данных (необязательный шаг).
3. Реализовать метод `Convert()`, выполняющий преобразование данных от источника к целевому объекту.
4. Реализовать метод `ConvertBack()`, делающий обратное преобразование.

Приведём пример простого конвертера, преобразующего вещественное число в кисть WPF:

```
[ValueConversion(typeof (double), typeof (Brush))]  
public class DoubleToColorConverter : IValueConverter  
{  
    public object Convert(object value, Type targetType,  
                          object parameter, CultureInfo culture)  
    {  
        return (double) value > 18 ? Brushes.Blue : Brushes.Red;  
    }  
  
    public object ConvertBack(object value, Type targetType,  
                             object parameter, CultureInfo culture)  
    {  
        // это конвертер для однонаправленной привязки  
        return null;  
    }  
}
```

Чтобы ввести в действие конвертер значений, нужно использовать свойство привязки `Converter` и, при необходимости, свойства `ConverterParameter` и `ConverterCulture`. Как несложно понять, значения последних двух свойств передаются методам конвертера в качестве аргументов.

Используем конвертер `DoubleToColorConverter` в примере привязки для объекта `Person`. Объект конвертера разместим в ресурсах окна. Предполагаем, что объект-источник присваивается свойству окна `DataContext` в коде:

```
<Window x:Class="WpfBinding.MainWindow"  
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        xmlns:local="clr-namespace:WpfBinding"  
        Height="200" Width="400" Title="WPF Binding">  
    <Window.Resources>  
        <local:DoubleToColorConverter x:Key="dc" />  
    </Window.Resources>
```

```

<Grid Margin="10">
    <!-- описание структуры Grid опущено для краткости -->

    <TextBlock Grid.Column="0" Grid.Row="0" Text="Name" />
    <TextBox Grid.Column="1" Grid.Row="0"
        Text="{Binding Name, Mode=TwoWay}" />

    <TextBlock Grid.Column="0" Grid.Row="1" Text="Age" />
    <TextBox Grid.Column="1" Grid.Row="1"
        Text="{Binding Age, Mode=TwoWay}"
        Foreground="{Binding Age,
            Converter={StaticResource dc}}" />

</Grid>
</Window>

```

В состав WPF входит ряд конвертеров для нескольких распространённых сценариев привязки. Один из них называется `BooleanToVisibilityConverter` и преобразует перечисление `Visibility` в тип `bool?`.

С помощью конвертеров значений можно реализовать преобразование нескольких значений для создания единственного конвертированного значения. Для этого необходимо:

1. Использовать привязку `MultiBinding` вместо `Binding`;
2. Создать конвертер, реализующий интерфейс `IMultiValueConverter`.

Класс `MultiBinding` схож с классом `Binding` – оба класса унаследованы от `System.Windows.Data.BindingBase`. Класс `MultiBinding` группирует привязки в коллекцию `Bindings`.

```

<!-- TextBlock будет показывать имя и возраст Person -->
<TextBlock>
    <TextBlock.Text>
        <MultiBinding StringFormat="{0} - {1}">
            <Binding Path="Name" />
            <Binding Path="Age" />
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>

```

Как и `IValueConverter`, интерфейс `IMultiValueConverter` определяет методы `Convert()` и `ConvertBack()`. Отличие в том, что методам передаётся массив значений и массив типов вместо единственного значения и типа. Массивы значений и типов формируются на основе привязок в коллекции `Bindings`.

9.5. Проверка данных

Для привязки, которая передаёт информацию в источник данных (режимы `BindingMode.TwoWay` и `BindingMode.OneWayToSource`) можно выполнить проверку данных перед помещением их в источник. Если данные не проходят проверку, то источник не обновляется.

Первый способ организации проверки заключается в создании и применении *проверочных правил*. Каждое проверочное правило – это наследник класса `System.Windows.Controls.ValidationRule` с перекрытым методом `Validate()`. Этот метод получает проверяемое значение и информацию о культуре, а возвращает объект `ValidationResult` с результатом проверки. Ниже приведён пример правила для проверки возраста:

```
public class AgeRule : ValidationRule
{
    public double MaxAge { get; set; }

    public AgeRule()
    {
        MaxAge = 100;
    }

    public override ValidationResult Validate(object value,
                                             CultureInfo culture)
    {
        double age;
        if (!double.TryParse((string) value, out age))
        {
            return new ValidationResult(false, "Cannot parse");
        }
        if ((age < 0) || (age > MaxAge))
        {
            return new ValidationResult(false, "Out of range");
        }
        return new ValidationResult(true, null);
    }
}
```

Проверочные правила помещают в коллекцию `ValidationRules`, имеющуюся у каждой привязки:

```
<!-- изменим предыдущие примеры привязки -->
<TextBox Grid.Column="1" Grid.Row="1">
    <TextBox.Text>
        <Binding Path="Age" Mode="TwoWay">
            <Binding.ValidationRules>
                <local:AgeRule MaxAge="120"/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Второй способ организации проверки основан на генерации исключительной ситуации в методе установки свойства источника данных:

```
// фрагмент класса Person: определение свойства Age
public double Age
{
    get { return _age; }
    set
    {
        if (value < 0)
        {
            throw new ArgumentException("Age < 0");
        }
        _age = value;
    }
}
```

Исключительные ситуации, возникающие при передаче информации в источник, отлавливаются при помощи специального встроенного правила `ExceptionValidationRule`. Вместо использования `ExceptionValidationRule` можно установить свойство привязки `ValidatesOnExceptions` в значение `true`.

Третий способ организации проверки основан на реализации источником данных интерфейса `System.ComponentModel.DataAnnotations.IDataErrorInfo`. Интерфейс `IDataErrorInfo` содержит два элемента: строковое свойство `Error` и строковый индексатор с ключом-строкой. Свойство `Error` – это общее описание ошибок объекта. Индексатор принимает имя свойства и возвращает соответствующую детальную информацию об ошибке. Ключевая идея в том, что вся логика обработки ошибок централизована в индексаторе.

Ниже приведён пример класса, реализующего `IDataErrorInfo`:

```
public class Person : IDataErrorInfo
{
    public string Name { get; set; }
    public double Age { get; set; }
    public string this[string columnName]
    {
        get
        {
            switch (columnName)
            {
                case "Name":
                    return string.IsNullOrEmpty(Name) ?
                        "Name cannot be empty" : null;
                case "Age":
                    return ((Age < 0) || (Age > 100)) ?
                        "Incorrect age" : null;
                default:
                    return null;
            }
        }
    }
}
```

```

        // это свойство не используется в WPF
        public string Error
        {
            get { return null; }
        }
    }
}

```

Чтобы заставить WPF использовать интерфейс `IDataErrorInfo` для проверки ошибок при модификации свойства, нужно добавить встроенное правило `DataErrorValidationRule` в коллекцию `ValidationRules`. В качестве альтернативы использованию `DataErrorValidationRule` можно установить свойство привязки `ValidatesOnDataErrors` в значение `true`.

Четвёртый вариант организации проверки основан на реализации источником данных интерфейса `System.ComponentModel.INotifyDataErrorInfo`. Этот интерфейс похож на гибрид интерфейсов `INotifyPropertyChanged` и `IDataErrorInfo`. Его булево свойство `HasErrors` указывает на наличие ошибок объекта. Метод `GetErrors()` позволяет получить список ошибок по строке с именем свойства. Событие `ErrorsChanged` необходимо генерировать при изменении списка ошибок.

Рассмотрим пример класса, реализующего `INotifyDataErrorInfo`:

```

public class Person : INotifyDataErrorInfo
{
    // поля объекта
    private string _name;
    private double _age;

    // коллекция ошибок
    private Dictionary<string, List<string>> _errors =
        new Dictionary<string, List<string>>();

    // свойства объекта
    public string Name
    {
        get { return _name; }
        set
        {
            if (NameIsValid(value) && _name != value)
            {
                _name = value;
            }
        }
    }

    public double Age
    {
        get { return _age; }
    }
}

```

```

        set
        {
            if (AgeIsValid(value) && _age != value)
            {
                _age = value;
            }
        }
    }

    // реализация интерфейса INotifyDataErrorInfo
    public event EventHandler<DataErrorsChangedEventArgs>
        ErrorsChanged;

    public bool HasErrors
    {
        get { return _errors.Count > 0; }
    }

    public IEnumerable GetErrors(string property)
    {
        if (string.IsNullOrEmpty(property) ||
            !_errors.ContainsKey(property))
        {
            return null;
        }
        return _errors[property];
    }

    // два метода проверки правильности свойств
    private bool NameIsValid(string name)
    {
        if (string.IsNullOrEmpty(name))
        {
            AddError("Name", "Name cannot be empty");
            return false;
        }
        RemoveError("Name", "Name cannot be empty");
        return true;
    }

    private bool AgeIsValid(double age)
    {
        if ((age < 0) || (age > 100))
        {
            AddError("Age", "Incorrect age");
            return false;
        }
        RemoveError("Age", "Incorrect age");
        return true;
    }

```

```

// метод для добавления ошибки в коллекцию ошибок
private void AddError(string property, string error)
{
    if (!_errors.ContainsKey(property))
    {
        _errors[property] = new List<string>();
    }
    if (!_errors[property].Contains(error))
    {
        _errors[property].Add(error);
        RaiseErrorsChanged(property);
    }
}

// метод для удаления ошибки из коллекции ошибок
private void RemoveError(string property, string error)
{
    if (_errors.ContainsKey(property) &&
        _errors[property].Contains(error))
    {
        _errors[property].Remove(error);
        if (_errors[property].Count == 0)
        {
            _errors.Remove(property);
        }
        RaiseErrorsChanged(property);
    }
}

// метод для генерации события
private void RaiseErrorsChanged(string property)
{
    if (ErrorsChanged != null)
    {
        ErrorsChanged(this,
            new DataErrorsChangedEventArgs(property));
    }
}
}

```

Чтобы заставить WPF использовать интерфейс `INotifyDataErrorInfo` для проверки ошибок при модификации свойства, нужно добавить встроенное правило `NotifyDataErrorValidationRule` в коллекцию `ValidationRules`. В качестве альтернативы использованию `NotifyDataErrorValidationRule` можно установить свойство привязки `ValidatesOnNotifyDataErrors` в значение `true`.

При нарушении любого правила проверки выполняются следующие шаги:

1. В целевом объекте присоединённое свойство `Validation.HasError` устанавливается в `true`.

2. Создаётся объект `ValidationError` с информацией об ошибке и добавляется в присоединённую коллекцию `Validation.Errors` целевого объекта.

3. Если свойство привязки `NotifyOnValidationError` установлено в `true`, в целевом объекте инициируется присоединённое событие `Validation.Error`.

При возникновении ошибки также изменяется визуальное представление целевого элемента управления. Шаблон элемента заменяется шаблоном, определённым в свойстве `Validation.ErrorTemplate`. Например, в текстовом поле новый шаблон окрашивает контур поля в красный цвет.

10. Работа с графикой

10.1. Представление цвета в WPF

WPF позволяет работать с двумя цветовыми моделями:

1. *RGB* – распространённая цветовая модель, в которой каждый компонент цвета (красный, зелёный, синий) представлен одним байтом. Дополнительно может использоваться однобайтовый *альфа-канал*, чтобы задать прозрачность цвета (0 – полностью прозрачный, 255 – полностью непрозрачный).

2. *scRGB* – в этой модели каждый компонент цвета (альфа-канал, красный, зелёный, синий) представлен с помощью 16- или 32-битных вещественных чисел в диапазоне от 0 до 1.

Структура `System.Windows.Media.Color` хранит информацию о цвете. Свойства структуры позволяют прочесть или задать отдельную цветовую компоненту в любой из двух цветовых моделей, а статические методы – создать цвет на основе компонент или произвести простейшие операции с цветом:

```
Color c1 = Color.FromRgb(10, 20, 30);
Color c2 = Color.FromArgb(250, 10, 20, 32);
Color c3 = Color.FromScRgb(0.4f, 0.5f, 0.7f, 0.2f);
Color c4 = (c1 + c2)*2;
byte red = c4.R;
bool flag = Color.AreClose(c1, c2);
```

Класс `System.Windows.Media.Colors` содержит набор именованных цветов в виде статических свойств для чтения. Класс `System.Windows.SystemColors` предоставляет аналогичный набор для стандартных цветов системы:

```
Color c1 = Colors.IndianRed;
Color c2 = SystemColors.ControlColor;
```

При установке цвета в разметке XAML применяются следующие форматы:

Имя-цвета – одно из имён свойств в классе `Colors`;

#rgb или *#rrggbb* – аналог вызова `Color.FromRgb(0xrr, 0xgg, 0xbb)`;

#argb или *#aarrggbb* – аналог `Color.FromArgb(0xaa, 0xrr, 0xgg, 0xbb)`;

sc# a r g b – аналог `Color.FromScRgb(a, r, g, b)` (числа $\in [0,1]$).

```
<Button Background="Red" />
<Button Background="#64A" />
```

```
<Button Background="#FF00674A" />
<Button Background="sc# 0.1 0.1 0.5 0.3" />
```

Ради справедливости отметим, что в приведённом выше примере на самом деле используется не цвет, а соответствующая кисть `SolidColorBrush`:

```
<!-- эквивалент <Button Background="Red" /> -->
<Button>
  <Button.Background>
    <SolidColorBrush Color="Red" />
  </Button.Background>
</Button>
```

10.2. Кисти

Кисть – это объект, используемый для заполнения фона, переднего плана, границы, линии. Любая кисть является потомком абстрактного класса `System.Windows.Media.Brush`. Имеется несколько стандартных классов кистей:

1. `SolidColorBrush` – закрашивает область сплошным цветом;
2. `LinearGradientBrush` – закрашивает область, используя линейное градиентное заполнение, изображающее плавный переход от одного цвета к другому;
3. `RadialGradientBrush` – рисует область, используя радиальный градиент.
4. `ImageBrush` – рисует область, используя изображение, которое может растягиваться, масштабироваться или многократно повторяться.
5. `VisualBrush` – заполняет область, используя объект `Visual`.
6. `DrawingBrush` – рисует область, используя объект `Drawing`.

Кисть `SolidColorBrush` – самая простая из кистей. Во всех предыдущих примерах разметки использовалась именно она. Свойство кисти `Color` определяет цвет сплошной заливки. Анализатор XAML способен автоматически создать объект `SolidColorBrush` на основе строки с представлением цвета. Также отметим, что в классе `SystemColors` задан набор статических кистей `SolidColorBrush`, соответствующих системным цветам.

```
<Button Background="{x:Static SystemColors.ControlBrush}" />
```

Кисть `LinearGradientBrush` создаёт заполнение, которое представляет собой переход от одного цвета к другому. Ниже приведён простейший пример градиента, который закрашивает фон окна по диагонали от синего цвета в левом верхнем углу к белому цвету в правом нижнем углу:

```
<Window.Background>
  <LinearGradientBrush>
    <GradientStop Color="Blue" Offset="0" />
    <GradientStop Color="White" Offset="1" />
  </LinearGradientBrush>
</Window.Background>
```

Градиент в `LinearGradientBrush` строится по следующим правилам. Вокруг заполняемой области очерчивается виртуальный прямоугольник, у которого левый верхний угол имеет координаты (0, 0), а правый нижний – (1, 1). В этих координатах при помощи свойств `StartPoint` и `EndPoint` задаётся *вектор градиента* (по умолчанию `StartPoint=0,0` и `EndPoint=1,1`)¹. Коллекция `GradientStops`, являющаяся свойством содержимого для `LinearGradientBrush`, содержит *опорные точки градиента* – объекты `GradientStop` с указанием на цвет и смещение относительно вектора градиента (0 – начало вектора, 1 – конец вектора):

```
<Window.Background>
  <LinearGradientBrush StartPoint="0.5,0" EndPoint="1,1">
    <GradientStop Color="Blue" Offset="0" />
    <GradientStop Color="Azure" Offset="0.3" />
    <GradientStop Color="White" Offset="1" />
  </LinearGradientBrush>
</Window.Background>
```

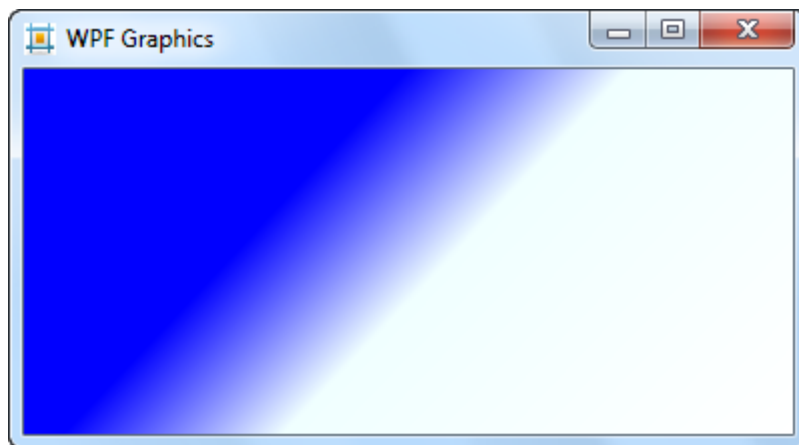


Рис. 28. Демонстрация градиентной кисти.

Свойство градиентной кисти `SpreadMethod` принимает значения из перечисления `GradientSpreadMethod` и управляет тем, как будет распространяться градиент за пределы вектора градиента. По умолчанию используется значение `Pad` – области вне градиента заполняются соответствующим сплошным цветом. Допустимы также значения `Reflect` (для обращения градиента) и `Repeat` (для дублирования той же цветовой последовательности).

Кисть `RadialGradientBrush` работает подобно `LinearGradientBrush`, но использует градиент, который исходит из начальной точки в радиальном направлении. Координаты начальной точки задаёт свойство `GradientOrigin`, которое по умолчанию равно (0.5,0.5). Градиент распространяется до границы *круга градиента*, который описывается тремя свойствами: `Center`, `RadiusX` и `RadiusY`. В классе `RadialGradientBrush` имеются свойства `MappingMode` и `SpreadMethod`.

¹ Если свойство `LinearGradientBrush.MappingMode` установить в `MappingMode.Absolute`, для вектора градиента будут использоваться абсолютные координаты.

```
<RadialGradientBrush RadiusX="1" RadiusY="1"
    GradientOrigin="0.7,0.3">
    <GradientStop Color="White" Offset="0" />
    <GradientStop Color="DodgerBlue" Offset="1" />
</RadialGradientBrush>
```



Рис. 29. Использование `RadialGradientBrush` для фона окна.

Кисть `ImageBrush` позволяет заполнить область изображением, которое указывается в свойстве `ImageSource`. Изображение может быть либо битовым (формата BMP, PNG, GIF, JPEG, ICON), либо векторным. В первом случае картинка идентифицируется URI, который обычно указывает на двоичный ресурс сборки. Во втором случае используется объект `DrawingImage`.

Для настройки `ImageBrush` можно использовать следующие свойства:

`Stretch` – правило растяжения картинки, если она не совпадает с заполняемой областью: `None` – не растягивать, `Fill` – заполнить, исказив пропорции, `Uniform` – сохранить пропорции и заполнить, сколько получится, `UniformToFill` – сохранить пропорции и заполнить всё, обрезав лишнее;

`AlignmentX` и `AlignmentY` – правила выравнивания картинки, если она меньше заполняемой области;

`Viewbox` – фрагмент для кисти, который будет вырезан из картинки;

`ViewboxUnits` – способ определения координат `Viewbox` (`Absolute` – абсолютные, `RelativeToBoundingBox` – относительные при помощи виртуального прямоугольника);

`Viewport` – фрагмент закрашиваемой области, на который отображается картинка кисти. Свойство применяется, когда картинкой нужно «замостить» большую область.

`ViewportUnits` – способ определения координат `Viewport`;

`TileMode` – способ заполнения картинкой кисти большой области: `None` – без заполнения, `Tile` – простое заполнение, `FlipX`, `FlipY`, `FlipXY` – заполнения с отражением по указанной оси.

В следующем примере кисть `ImageBrush` используется для заполнения фона окна. Из изображения иконки приложения вырезается четверть, которая повторяется на фоне двадцать раз (четыре строки по пять фрагментов):

```

<Window.Background>
  <ImageBrush ImageSource="layout.jpg"
    Viewbox="0,0 0.5,0.5"
    Viewport="0,0 0.2,0.25"
    TileMode="Tile" />
</Window.Background>

```

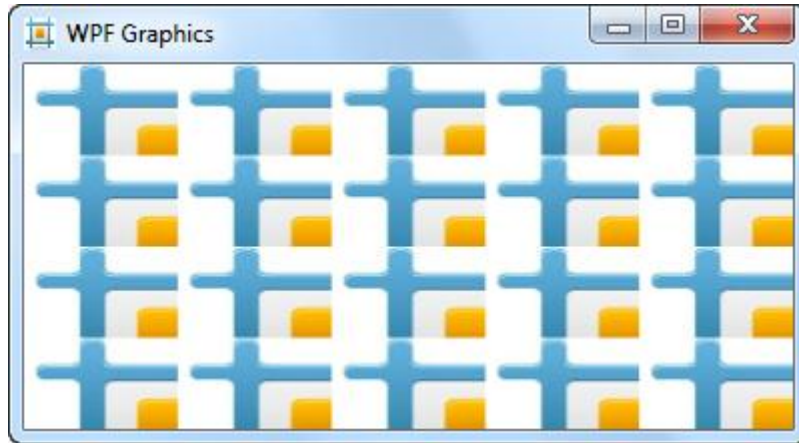


Рис. 30. Кисть `ImageBrush`.

Кисть `VisualBrush` позволяет брать визуальное содержимое элемента и использовать его для заполнения. Например, с помощью `VisualBrush` можно скопировать внешний вид кнопки. Однако такая «кнопка» не будет реагировать на нажатия или получать фокус – это просто копия внешнего вида элемента. Интересно, что `VisualBrush` не просто копирует визуальное представление, а отслеживает изменения в копируемом элементе. В следующем примере поверхность окна меняется, когда пользователь редактирует текст в поле ввода:

```

<!-- контейнер Canvas вложен в окно -->
<Canvas>
  <Canvas.Background>
    <VisualBrush Visual="{Binding ElementName=txt}"
      Viewport="0,0 0.5,0.5"
      TileMode="Tile" />
  </Canvas.Background>
  <TextBox Canvas.Left="20" Canvas.Top="20"
    Name="txt" FontSize="16" Width="150" />
</Canvas>

```

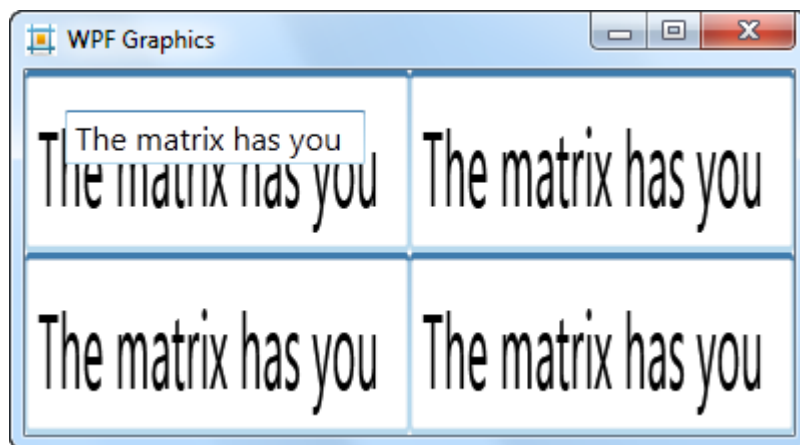


Рис. 31. Кисть `VisualBrush`.

Кисть `DrawingBrush` использует для заполнения области объект `Drawing`, помещённый в одноимённое свойство кисти. Класс `Drawing` представляет векторные двумерные рисунки. Отметим, что кисти `ImageBrush`, `VisualBrush` и `DrawingBrush` унаследованы от общего предка – класса `TileBrush`. Этот класс определяет свойства, связанные с заполнением области картинкой (`Viewbox`, `Viewport`, `TileMode`).

10.3. Трансформации

Трансформация – это заданное изменение координатной системы, в которой отображается элемент. Описание таких трансформаций на плоскости, как масштабирование, отражение и поворот, можно выполнить в терминах числовых матриц размером 2x2. Чтобы представить в матричной форме операцию сдвига координатной системы, используют однородные координаты.

Однородными координатами точки (x, y) является тройка вида $(x, y, 1)$. Если дана тройка чисел (a, b, c) , соответствующая точка в однородных координатах находится после применения нормировки – деления на c : $(a/c, b/c, 1)$. Тройки $(a, b, 0)$ описывают в однородных координатах *бесконечно удалённую точку*.

В терминах однородных координат основные трансформации можно выразить следующим образом:

1. Масштабирование (включая отражения): $(x, y, 1) \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$.
2. Поворот на угол φ : $(x, y, 1) \begin{pmatrix} \cos \varphi & \sin \varphi & 0 \\ -\sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$.
3. Сдвиг: $(x, y, 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{pmatrix}$.

Комбинация трансформаций выполняется как умножение матриц.

В WPF трансформации представлены классами, унаследованными от абстрактного класса `System.Windows.Media.Transform`. Набор предопределённых трансформаций перечислен в табл. 6.

Классы трансформаций

Имя класса	Описание	Важные свойства
<code>TranslateTransform</code>	Смещает координатную систему на указанную величину	X, Y
<code>RotateTransform</code>	Поворачивает координатную систему вокруг заданной точки	Angle, CenterX, CenterY
<code>ScaleTransform</code>	Масштабирует координатную систему. Можно применять разную степень масштабирования по измерениям X и Y	ScaleX, ScaleY, CenterX, CenterY
<code>SkewTransform</code>	Деформирует координатную систему, наклоняя её оси на заданное число градусов	AngleX, AngleY, CenterX, CenterY
<code>MatrixTransform</code>	Выполняет трансформацию, используя указанную матрицу вида $\begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ dx & dy & 1 \end{bmatrix}$	Matrix
<code>TransformGroup</code>	Комбинирует несколько трансформаций. Порядок трансформаций имеет значение	Children

Укажем некоторые способы задания трансформаций:

1. Класс `UIElement` определяет свойства `RenderTransform` и `RenderTransformOrigin`. `RenderTransform` – это трансформация, выполняемая после процесса компоновки непосредственно перед отображением элемента. `RenderTransformOrigin` задаёт стартовую (неподвижную) точку трансформации. По умолчанию это точка имеет координаты (0, 0) (координаты точки относительные, в терминах виртуального ограничивающего прямоугольника).

2. Класс `FrameworkElement` содержит свойство `LayoutTransform` для трансформации, применяемой до процесса компоновки.

3. Класс `Brush` имеет свойства `RelativeTransform` и `Transform`, позволяющие выполнить трансформацию кисти до и после её применения.

Следующий пример демонстрирует использование трансформаций.

```
<StackPanel Orientation="Horizontal">
  <Button Height="40" Width="70" Content="Translate">
    <Button.RenderTransform>
      <TranslateTransform X="10" Y="-10"/>
    </Button.RenderTransform>
  </Button>
  <Button Height="40" Width="70" Content="Rotate">
    <Button.LayoutTransform>
      <RotateTransform Angle="-45" />
    </Button.LayoutTransform>
  </Button>
  <Button Height="40" Width="70" Content="Scale">
    <Button.LayoutTransform>
      <ScaleTransform ScaleX="-1" ScaleY="1.5" />
    </Button.LayoutTransform>
  </Button>
</StackPanel>
```



```

<Button Height="40" Width="70" Content="Skew">
    <Button.RenderTransform>
        <SkewTransform AngleX="20" AngleY="0" />
    </Button.RenderTransform>
</Button>
<Button Height="40" Width="70" Content="Matrix">
    <Button.LayoutTransform>
        <MatrixTransform Matrix="1,0.5,1,-1,20,10" />
    </Button.LayoutTransform>
</Button>
</StackPanel>

```

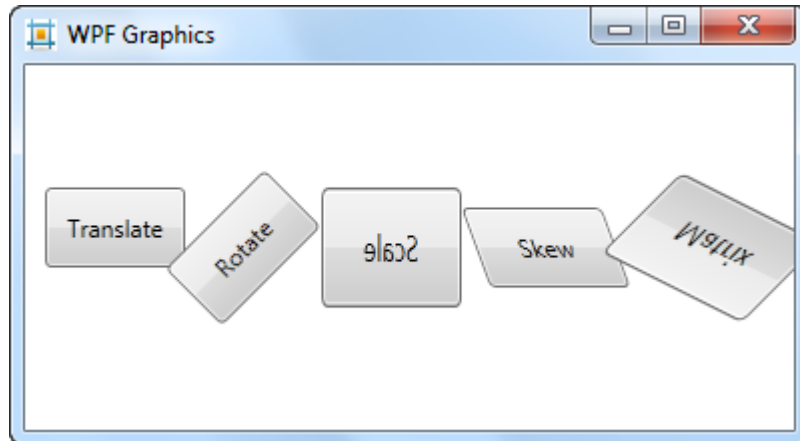


Рис. 32. Кнопки, подвергшиеся трансформации.

10.4. Эффекты

WPF поддерживает применение к элементам *эффектов*, таких как размытие или отбрасывание тени. Эффекты описываются классами, унаследованными от `System.Windows.Media.Effects.Effect`: `BlurEffect` – эффект размытия; `DropShadowEffect` – эффект тени; `ShaderEffect` – эффект, определённый на языке HLSL (*пиксельные шейдеры*).

Для применения эффектов класс `UIElement` определяет свойство `Effect`. Следующая разметка демонстрирует эффекты `BlurEffect` и `DropShadowEffect`:

```

<StackPanel Orientation="Horizontal">
    <Button Margin="30" Height="40" Width="80" Content="Blur">
        <Button.Effect>
            <BlurEffect Radius="2" RenderingBias="Quality" />
        </Button.Effect>
    </Button>

    <Button Margin="30" Height="40" Width="80" Content="Shadow" >
        <Button.Effect>
            <DropShadowEffect ShadowDepth="5" Direction="300"
                             Color="Blue" />
        </Button.Effect>
    </Button>
</StackPanel>

```

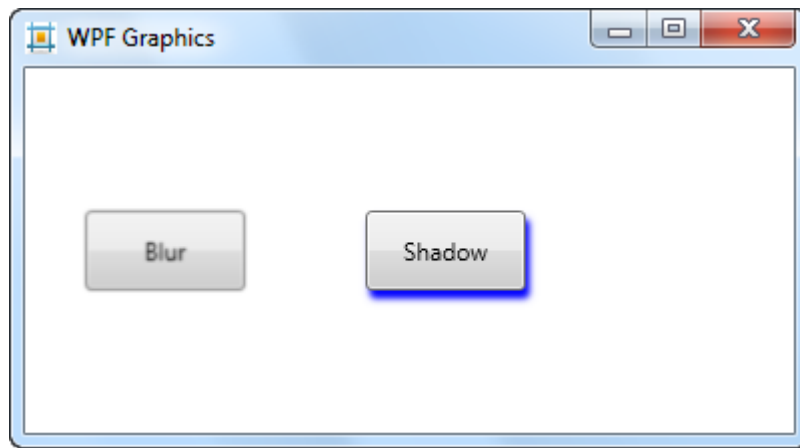


Рис. 33. Применение эффектов размытия и тени.

10.5. Прозрачность

В WPF поддерживается истинная прозрачность. Каждый элемент и кисть содержат свойство `Opacity`, определяющее степень прозрачности и принимающее вещественные значения из диапазона $[0, 1]$. Например, `Opacity=0.9` создаёт эффект 90% видимости и 10% прозрачности. Также можно использовать цвет (и соответствующую сплошную кисть) с альфа-каналом, меньшим максимума.

Все элементы содержат свойство `OpacityMask`, которое принимает любую кисть. Альфа-канал кисти определяет степень прозрачности (другие цветовые компоненты значения не имеют). Применение `OpacityMask` с кистями, содержащими градиентный переход от сплошного к прозрачному цвету создаёт эффект постепенного «исчезновения» поверхности. Если поместить в `OpacityMask` кисть `DrawingBrush`, можно создать прозрачную область заданной формы.

В следующем примере `OpacityMask` используется в сочетании с `VisualBrush` для создания популярного эффекта отражения. По мере набора текста `VisualBrush` рисует ниже отражение этого текста, закрашивая прямоугольник с установленным `OpacityMask` для постепенного затухания отражения.

```
<StackPanel>
  <TextBox Name="txt" FontSize="20" FontWeight="Bold" />
  <Rectangle RenderTransformOrigin="1,0.5" Height="40">
    <Rectangle.Fill>
      <VisualBrush Visual="{Binding ElementName=txt}" />
    </Rectangle.Fill>
    <Rectangle.OpacityMask>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0.3" Color="Transparent" />
        <GradientStop Offset="1" Color="#44000000" />
      </LinearGradientBrush>
    </Rectangle.OpacityMask>
    <Rectangle.RenderTransform>
      <ScaleTransform ScaleY="-1" />
    </Rectangle.RenderTransform>
  </Rectangle>
</StackPanel>
```

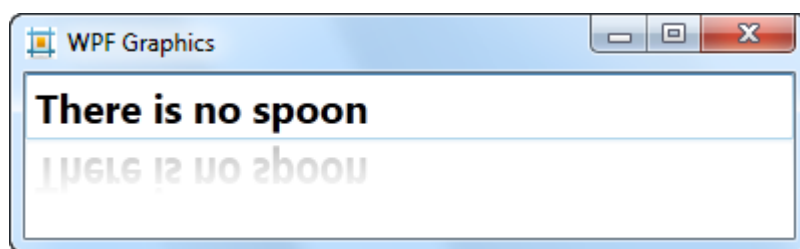


Рис. 34. Прозрачность для эффекта отражения.

10.6. Фигуры

Фигуры – это элементы WPF, представляющие простые линии, эллипсы, прямоугольники и многоугольники. Все фигуры наследуются от абстрактного класса `System.Windows.Shapes.Shape`. Этот класс определяет набор важных свойств, перечисленных в табл. 7.

Таблица 7

Свойства класса `Shape`

Имя	Описание
<code>Fill</code>	Кисть, рисующая поверхность фигуры
<code>Stroke</code>	Кисть, рисующая границу фигуры
<code>StrokeThickness</code>	Ширина границы в единицах WPF
<code>StrokeStartLineCap</code> , <code>StrokeEndLineCap</code>	Контур начала и конца линии
<code>StrokeDashArray</code> , <code>StrokeDashOffset</code> , <code>StrokeDashCap</code>	Свойства позволяют создавать пунктир. Можно управлять размером и частотой пунктира, а также контуром, ограничивающим начало и конец каждого фрагмента пунктира
<code>StrokeLineJoin</code> , <code>StrokeMiterLimit</code>	Контур углов фигуры
<code>Stretch</code>	Способ заполнения фигурой доступного пространства
<code>DefiningGeometry</code>	Объект <code>Geometry</code> для описания геометрии фигуры. Геометрия описывает координаты и размер фигуры без учёта поддержки событий клавиатуры и мыши
<code>GeometryTransform</code>	Позволяет применить к фигуре трансформацию
<code>RenderedGeometry</code>	Геометрия, описывающая фигуру после трансформации

Перейдём к рассмотрению конкретных фигур. `Rectangle` и `Ellipse` представляют прямоугольник и эллипс. Размеры этих фигур определяются свойствами `Height` и `Width`, а свойства `Fill` и `Stroke` делают фигуру видимой. Класс `Rectangle` имеет свойства `RadiusX` и `RadiusY` для создания закруглённых углов.

Фигура `Line` – это отрезок, соединяющий две точки. Начальная и конечная точки устанавливаются свойствами `X1` и `Y1` (начало) и `X2` и `Y2` (конец). Координаты отрезка отсчитываются относительно верхнего левого угла контейнера, в котором он содержится. Если отрезок рисуется в `Canvas`, установка присоединённых свойств позиции задаст смещение координатной системы рисования. Чтобы сделать отрезок видимым, нужно задать его свойство `Stroke`.

Класс `Polyline` позволяет рисовать последовательность связанных отрезков. Для этого необходимо задать набор точек в свойстве-коллекции `Points`. В XAML для `Points` можно использовать лаконичный строчный синтаксис, просто

перечислив точки координат через пробел или запятую. Класс `Polygon` подобен `Polyline`, но при рисовании добавляет финальный отрезок, соединяющий начальную и конечную точки коллекции `Points`.

```
<!-- некоторые фигуры, изображённые на холсте Canvas -->
<Canvas>
  <!-- прямоугольник -->
  <Rectangle Canvas.Top="10" Canvas.Left="80"
    Width="120" Height="80" RadiusX="30" RadiusY="20"
    Fill="Azure" Stroke="Coral" />

  <!-- эллипс -->
  <Ellipse Canvas.Bottom="10" Canvas.Right="10"
    Width="80" Height="60"
    Fill="Aqua" Stroke="Olive" StrokeThickness="4" />

  <!-- линия -->
  <Line X1="150" Y1="170" X2="360" Y2="10" Stroke="Blue" />

  <!-- полигон -->
  <Polygon Points="10,115 70,60 70,170"
    Fill="Green" Stroke="Purple" StrokeThickness="2" />
</Canvas>
```

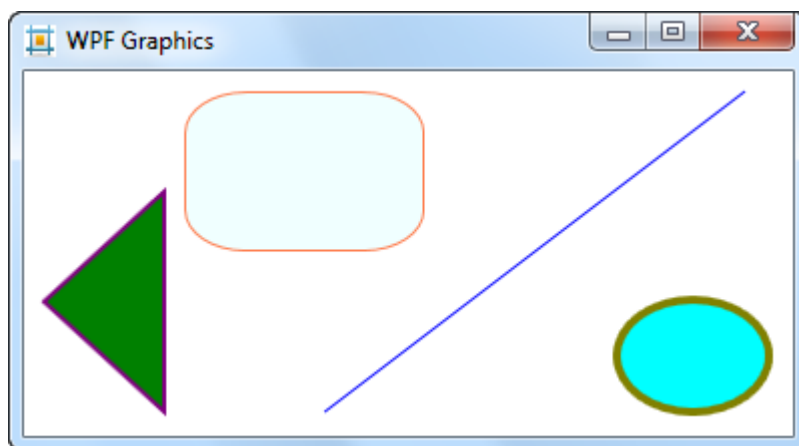


Рис. 35. Прямоугольник, эллипс, линия, полигон.

Рисуя фигуры `Line` и `Polyline`, можно указать форму начальной и конечной точки линии, используя свойства `StartLineCap` и `EndLineCap`. Изначально эти свойства установлены в `Flat`, что означает немедленное завершение линии в её конечных координатах. К другим возможным вариантам относятся `Round` (линия закругляется), `Triangle` (сводит обе стороны линии в точку) и `Square` (завершает линию чёткой гранью). Все значения, кроме `Flat`, добавляют линии длину – половину толщины линии дополнительно.

Все фигуры позволяют изменять вид и форму своих углов через свойство `StrokeLineJoin`. Значение по умолчанию – `Miter` – использует чёткие грани, `Bevel` обрезает угол в точке сопряжения, а `Round` плавно закругляет его.

Линии (включая границы фигур) могут быть изображены пунктиром. При этом в свойстве-массиве `StrokeDashArray` указываются длины сплошных и прерванных сегментов (пробелов). Эти значения интерпретируются относительно толщины линии.

```
<Canvas>
  <Line X1="30" Y1="30" X2="200" Y2="30"
        Stroke="Red" StrokeThickness="40"
        StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />

  <Polyline Points="60,140 80,90 100,150 120,90 140,140"
            Stroke="Blue" StrokeThickness="20"
            StrokeLineJoin="Round" />

  <Rectangle Canvas.Top="20" Canvas.Left="260"
            Width="80" Height="120"
            Stroke="Navy" StrokeThickness="4"
            StrokeDashArray="2 1.5" />
</Canvas>
```

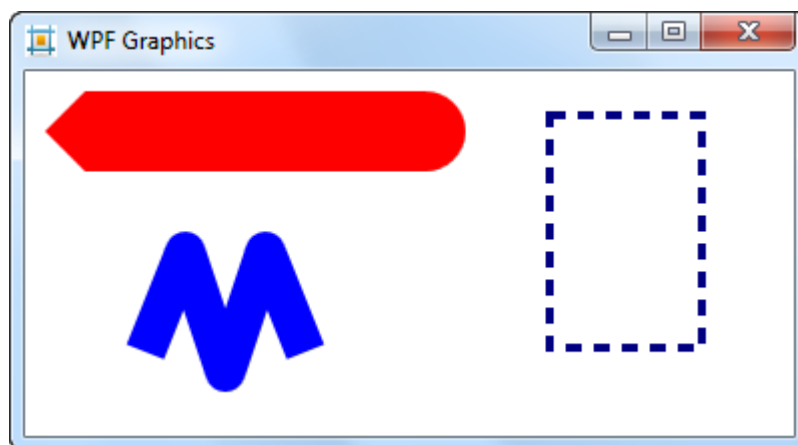


Рис. 36. Концы линий, форма углов и штриховка.

Фигура `Path` предназначена для отображения произвольной геометрии. Этот класс имеет свойство `Data` с типом `Geometry`. При задании свойства используется один из наследников класса `Geometry`:

`RectangleGeometry` – прямоугольник.

`EllipseGeometry` – эллипс.

`LineGeometry` – прямая линия.

`GeometryGroup` – связывает набор объектов `Geometry` в единый путь.

`CombinedGeometry` – объединяет две геометрии в единую фигуру, при этом можно указать способ комбинирования составляющих.

`PathGeometry` – сложная фигура, состоящая из дуг, кривых и линий.

`StreamGeometry` – неизменяемый облегченный эквивалент `PathGeometry`.

Классы `LineGeometry`, `RectangleGeometry` и `EllipseGeometry` отображаются непосредственно на фигуры `Line`, `Rectangle` и `Ellipse`. Ниже для сравнения приведены эквивалентные фрагменты разметок:

```

<!-- фигура Rectangle -->
<Rectangle Fill="Yellow" Stroke="Blue" Width="100" Height="50" />

<!-- фигура Path -->
<Path Fill="Yellow" Stroke="Blue">
  <Path.Data>
    <RectangleGeometry Rect="0,0 100,50" />
  </Path.Data>
</Path>

```

Класс `GeometryGroup` предоставляет возможность для объединения нескольких геометрий в единое целое. В следующем примере при помощи `GeometryGroup` создаётся фигура в виде прямоугольника с отверстием:

```

<Canvas>
  <TextBlock Canvas.Top="50" Canvas.Left="20"
    FontSize="25" FontWeight="Bold" Text="Hello There" />
  <Path Canvas.Top="10" Canvas.Left="10"
    Margin="5" Fill="Yellow" Stroke="Blue">
    <Path.Data>
      <GeometryGroup>
        <RectangleGeometry Rect="0,0 120,100" />
        <EllipseGeometry Center="50,50" RadiusX="40" RadiusY="30" />
      </GeometryGroup>
    </Path.Data>
  </Path>
</Canvas>

```

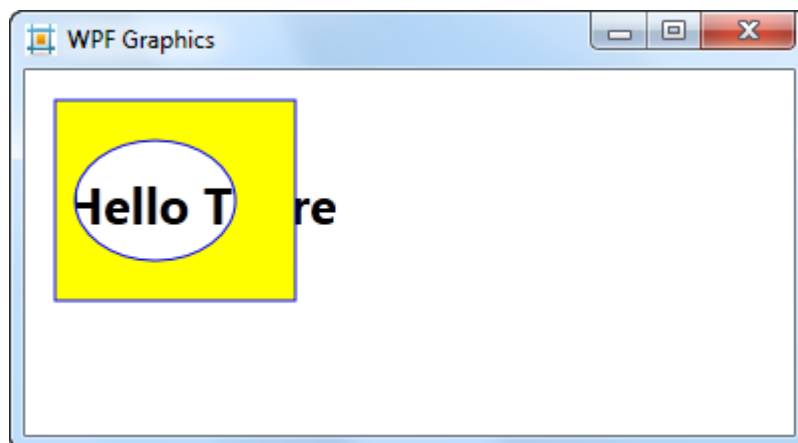


Рис. 37. Демонстрация `GeometryGroup`.

Класс `GeometryGroup` отлично работает при создании фигур посредством рисования одной и последующего «вычитания» другой изнутри первой. Для сложных случаев совмещения фигур предназначен класс `CombinedGeometry`. Он принимает две геометрии в свойствах `Geometry1` и `Geometry2` и комбинирует их способом, указанным в свойстве `GeometryCombineMode` (имеющем тип одноимённого

перечисления). Ниже приведён фрагмент разметки, демонстрирующий объединение геометрий, и рисунок, показывающий все четыре способа комбинирования.

```
<Path Fill="Gold" Stroke="Blue">
  <Path.Data>
    <CombinedGeometry GeometryCombineMode="Union">
      <CombinedGeometry.Geometry1>
        <RectangleGeometry Rect="0,0 100,80" />
      </CombinedGeometry.Geometry1>
      <CombinedGeometry.Geometry2>
        <EllipseGeometry Center="85,40" RadiusX="70" RadiusY="30" />
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </Path.Data>
</Path>
```

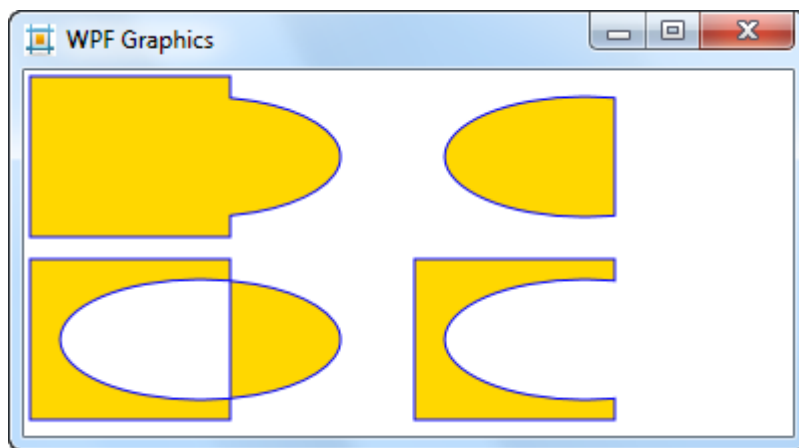


Рис. 38. Комбинирование геометрий: Union, Intersect, Xor, Exclude.

Класс `PathGeometry` описывает геометрию при помощи объектов `PathFigure`, хранящихся в коллекции `Figures`. Объект `PathFigure` представляет непрерывный набор связанных отрезков и кривых линий. У класса `PathFigure` четыре ключевых свойства:

`StartPoint` – точка начала первой линии фигуры (объект `Point`).

`Segments` – коллекция объектов `PathSegment`, используемых для рисования фигуры. Есть несколько типов сегментов, унаследованных от `PathSegment`: `LineSegment`, `ArcSegment`, `BezierSegment`, `QuadraticBezierSegment`, `PolyLineSegment`, `PolyBezierSegment`, `PolyQuadraticBezierSegment`.

`IsClosed` – если установлено в `true`, то добавляется отрезок, соединяющий начальную и конечную точки `PathFigure`.

`IsFilled` – если равно `true`, фигура заполняется кистью `Path.Fill`.

Использование объектов `PathGeometry`, `PathFigure`, `PathSegment` при описании сложной геометрии может быть чрезвычайно многословно. В WPF имеется краткий альтернативный синтаксис, которые позволяет представить детализированные фигуры в меньшем объёме кода разметки. Этот синтаксис называют *геометрическим мини-языком*. Выражения геометрического мини-языка – это

строки, содержащие серии команд. Каждая *команда* – это одна буква, за которой необязательно следуют несколько параметров (например, координаты). Команда отделяется пробелом. В табл. 8 перечислены все команды.

Таблица 8

Команды геометрического мини-языка

Команда, параметры	Описание
F <i>значение</i>	Устанавливает свойство <code>FillRule</code> – правило «заливки» контура. Эта команда должна стоять в начале строки (если она вообще будет применяться)
M <i>x,y</i>	Устанавливает начальную точку <code>PathFigure</code> . Любое выражение геометрического языка начинается либо с команды M, либо с пары команд F и M. Команда M внутри выражения служит для перемещения начала координатной системы
L <i>x,y</i>	Создаёт <code>LineSegment</code> до указанной точки
H <i>x</i>	Создаёт горизонтальный отрезок до указанного значения <i>x</i>
V <i>y</i>	Создаёт вертикальный отрезок до указанного значения <i>y</i>
A <i>radiusX,radiusY degrees isLargeArc isClockwise x,y</i>	Создаёт <code>ArcSegment</code> до заданной точки. Указываются радиусы эллипса, описывающего дугу, угол дуги в градусах и булевы флаги настройки дуги
C <i>x1,y1 x2,y2 x,y</i>	Создаёт <code>BezierSegment</code> до указанной точки, используя контрольные точки <i>x1,y1</i> и <i>x2,y2</i>
Q <i>x1,y1 x,y</i>	Создаёт <code>QuadraticBezierSegment</code> до указанной точки с одной контрольной точкой <i>x1,y1</i>
S <i>x1,y1 x,y</i>	Создаёт гладкий <code>BezierSegment</code> до указанной точки, с одной контрольной точкой <i>x1,y1</i>
Z	Завершает текущую <code>PathFigure</code> и устанавливает <code>IsClosed</code> в <code>true</code>

Следующая разметка определяет фигуру в виде треугольника:

```
<Path Stroke="Blue">
  <Path.Data>
    <PathGeometry>
      <PathFigure IsClosed="True" StartPoint="10,100">
        <LineSegment Point="100,100" />
        <LineSegment Point="100,50" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

Чтобы нарисовать тот же треугольник при помощи геометрического мини-языка, нужна такая разметка:

```
<Path Stroke="Blue" Data="M 10,100 L 100,100 L 100,50 Z" />
```

11. Стили и триггеры

Стиль – это коллекция значений свойств, которые могут быть применены к элементу. В WPF стили играют ту же роль, которую CSS играет в HTML-разметке. Подобно CSS, стили WPF позволяют определять общий набор характеристик форматирования и применять их по всему приложению для обеспечения согласованности. Стили могут работать автоматически, предназначаться для элементов конкретного типа и каскадироваться через дерево элементов.

Рассмотрение стилей начнём с конкретного примера, в котором определяется и применяется стиль для кнопок.

```
<StackPanel Orientation="Horizontal" Margin="30">
  <StackPanel.Resources>
    <Style x:Key="buttonStyle">
      <Setter Property="Button.FontSize" Value="22" />
      <Setter Property="Button.Background" Value="Purple" />
      <Setter Property="Button.Foreground" Value="White" />
      <Setter Property="Button.Height" Value="80" />
      <Setter Property="Button.Width" Value="80" />
      <Setter Property="Button.RenderTransformOrigin"
        Value=".5,.5" />
      <Setter Property="Button.RenderTransform">
        <Setter.Value>
          <RotateTransform Angle="10" />
        </Setter.Value>
      </Setter>
    </Style>
  </StackPanel.Resources>

  <Button Style="{StaticResource buttonStyle}">1</Button>
  <Button Style="{StaticResource buttonStyle}">2</Button>
  <Button Style="{StaticResource buttonStyle}">3</Button>
</StackPanel>
```

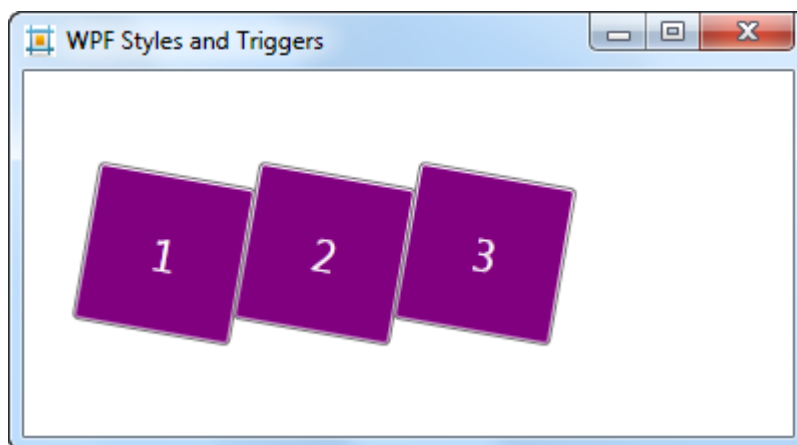


Рис. 39. Кнопки, к которым применён стиль.

Любой стиль в WPF – это объект класса `System.Windows.Style`. Основным свойством стиля является коллекция `Setters` (свойство содержимого), в которой

каждый элемент задаёт значение для некоторого свойства зависимостей (стили работают только со свойствами зависимостей). Чтобы задать значение для свойства зависимостей, нужно указать имя этого свойства. В нашем примере имя включает префикс – класс элемента. Если требуется применить один стиль к визуальным элементам разных типов, для префикса используется имя общего типа-предка:

```
<StackPanel Orientation="Horizontal" Margin="30">
  <StackPanel.Resources>
    <Style x:Key="controlStyle">
      <Setter Property="Control.FontSize" Value="22" />
      <Setter Property="Control.Background" Value="Purple" />
      <Setter Property="Control.Foreground" Value="White" />
      <Setter Property="Control.Height" Value="80" />
      <Setter Property="Control.Width" Value="80" />
    </Style>
  </StackPanel.Resources>

  <Button Style="{StaticResource controlStyle}" Content="1" />
  <TextBox Style="{StaticResource controlStyle}" Text="Hello" />
  <Expander Style="{StaticResource controlStyle}" Content="3" />
</StackPanel>
```

Свойство стиля `TargetType` позволяет указать конкретный тип, к которому применяется стиль. В этом случае префикс в установщиках свойств использовать необязательно:

```
<!-- можно записать TargetType="{x:Type Button}" -->
<Style x:Key="baseStyle" TargetType="Button">
  <Setter Property="FontSize" Value="22" />
  <Setter Property="Background" Value="Purple" />
  <Setter Property="Foreground" Value="White" />
</Style>
```

Определяя стиль, можно построить его на основе стиля-предка. Для этого следует воспользоваться свойством стиля `BasedOn`. Стили также могут содержать локальные логические ресурсы (коллекция `Resources`):

```
<Style x:Key="baseStyleWH" TargetType="Button"
      BasedOn="{StaticResource baseStyle}">
  <Style.Resources>
    <sys:Double x:Key="size">80</sys:Double>
  </Style.Resources>
  <Setter Property="Height" Value="{StaticResource size}" />
  <Setter Property="Width" Value="{StaticResource size}" />
</Style>
```

Кроме установки свойств зависимостей, стиль позволяет задать обработчики событий. Для этого применяется объект `EventSetter`:

```
<Style x:Key="buttonStyle" TargetType="Button">
  <Setter Property="FontSize" Value="22" />
  <EventSetter Event="MouseEnter" Handler="button_MouseEnter" />
</Style>
```

Как показывает первый пример параграфа, для применения стилей в элементе управления следует установить свойство `Style`, которое определено в классе `FrameworkElement`¹. Хотя стиль можно определить в самом элементе управления, обычно для этого используются ресурсы окна или приложения. Если при описании стиля в ресурсе задать `TargetType`, но не указывать ключ, стиль будет автоматически применяться ко всем элементам указанного типа.

```
<StackPanel Orientation="Horizontal" Margin="30">
  <StackPanel.Resources>
    <Style TargetType="Button">
      <Setter Property="FontSize" Value="22" />
      <Setter Property="Background" Value="Purple" />
      <Setter Property="Foreground" Value="White" />
      <Setter Property="Height" Value="80" />
      <Setter Property="Width" Value="80" />
    </Style>
  </StackPanel.Resources>

  <!-- эта кнопка будет иметь указанный выше стиль -->
  <Button Content="One" />
  <!-- у этой кнопки такой же стиль, но задана своя ширина -->
  <Button Width="140" Content="Two" />
  <!-- убрали стиль у кнопки, присвоив Style значение null -->
  <Button Style="{x:Null}" Content="Three" />
</StackPanel>
```

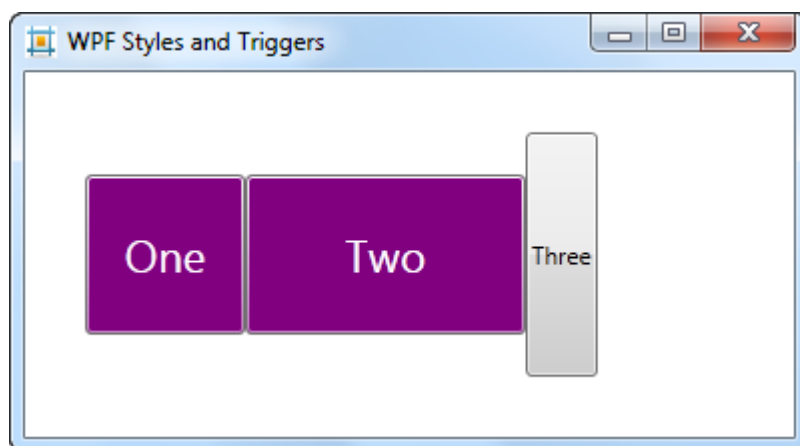


Рис. 40. Различные варианты применения стиля.

¹ Свойство `Style` – это не единственное «стилевое» свойство. Класс `FrameworkElement` имеет свойство `FocusVisualStyle` – стиль, применяемый при получении фокуса ввода. Класс `ItemsControl` имеет свойство `ItemContainerStyle` – стиль, применяемый к каждому элементу списка.

При описании стилей часто применяются триггеры. *Триггер* – это способ описания реакции на изменение, альтернативный применению обработчиков событий. Триггеры имеют два достоинства. Во-первых, их легко задать декларативно. Во-вторых, для триггера указывается только условие старта, условие завершения действия триггера указывать не надо.

Каждый триггер является экземпляром класса, который наследуется от `System.Windows.TriggerBase`. В WPF доступно пять видов триггеров:

1. *Триггер свойства* – класс `Trigger`. Это триггер самого простого типа. Он следит за появлением изменений в свойстве зависимостей.

2. *Триггер события* – класс `EventTrigger`. Триггер следит за наступлением указанного события. Триггеры событий применяются в анимации.

3. *Триггер данных* – класс `DataTrigger`. Этот триггер работает со связыванием данных. Он похож на триггер свойства, но следит за появлением изменений не в свойстве зависимостей, а в любых связанных данных.

4. *Мультитриггер* – класс `MultiTrigger`. Триггер объединяет несколько триггеров свойств. Он стартует, если выполняются все условия объединяемых триггеров.

5. *Мультитриггер данных* – класс `MultiDataTrigger`. Подобен мультитриггеру, но объединяет несколько триггеров данных.

Для декларации и хранения триггеров у классов `FrameworkElement`, `Style`, `DataTemplate` и `ControlTemplate` имеется коллекция `Triggers`. Правда, `FrameworkElement` поддерживает только триггеры событий.

Рассмотрим применение триггеров свойств при описании стиля. У триггера свойства настраиваются:

- имя свойства, с которым связан триггер;
- значение свойства, которое задаёт условие старта триггера;
- коллекция `Setters`, описывающая действие триггера.

Взяв за основу первый пример этого параграфа, модифицируем стиль при помощи триггера так, чтобы при перемещении над кнопкой указателя мыши происходил поворот кнопки:

```
<Style x:Key="buttonStyle" TargetType="Button">
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="RenderTransform">
        <Setter.Value>
          <RotateTransform Angle="10" />
        </Setter.Value>
      </Setter>
    </Trigger>
  </Style.Triggers>

  <Setter Property="FontSize" Value="22" />
  <Setter Property="Background" Value="Purple" />
  <Setter Property="Foreground" Value="White" />
</Style>
```

```

<Setter Property="Height" Value="80" />
<Setter Property="Width" Value="80" />
<Setter Property="RenderTransformOrigin" Value=".5,.5" />
</Style>

```

В следующем примере триггер используется для того, чтобы изменить внешний вид `TextBox` при наступлении ошибки проверки. Обратите внимание на использование привязки данных и работу с присоединёнными свойствами.

```

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

```

Ещё один пример демонстрирует использование триггера совместно со свойствами `AlternationIndex` и `ItemContainerStyle` класса `ItemsControl` для организации чередования внешнего вида строк в элементе `ListBox`:

```

<StackPanel>
  <StackPanel.Resources>
    <Style x:Key="AltRowStyle" TargetType="ItemsControl">
      <Setter Property="Background" Value="LightGray"/>
      <Setter Property="Foreground" Value="White"/>
      <Style.Triggers>
        <Trigger Property="ItemsControl.AlternationIndex" Value="1">
          <Setter Property="Background" Value="White"/>
          <Setter Property="Foreground" Value="Black"/>
        </Trigger>
      </Style.Triggers>
    </Style>
  </StackPanel.Resources>

  <ListBox AlternationCount="2" Margin="20"
    ItemContainerStyle="{StaticResource AltRowStyle}">
    <ListBoxItem>Item 1</ListBoxItem>
    <ListBoxItem>Item 2</ListBoxItem>
    <ListBoxItem>Item 3</ListBoxItem>
    <ListBoxItem>Item 4</ListBoxItem>
    <ListBoxItem>Item 5</ListBoxItem>
  </ListBox>
</StackPanel>

```

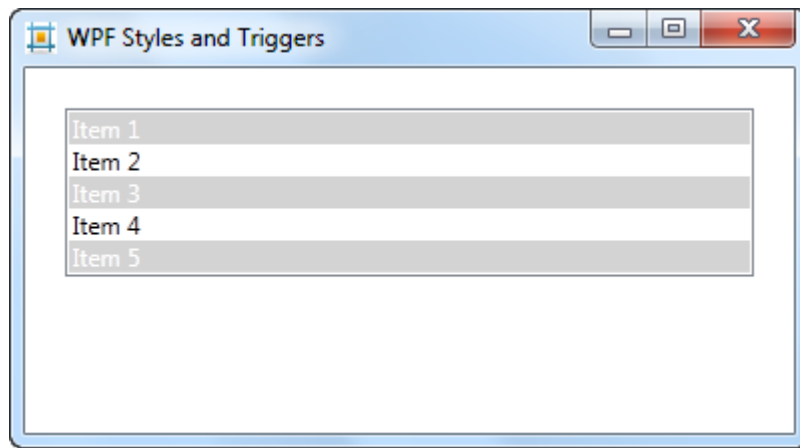


Рис. 41. `ListBox` с чередующимися стилями строк.

В стилях могут применяться не только триггеры свойств, но и триггеры данных, которые позволяют отследить изменения в обычном свойстве любого объекта .NET (а не только в свойствах зависимостей). Вместо свойства `Property` в триггерах данных применяется свойство `Binding`, содержащее привязку данных. Ниже демонстрируется пример стиля с триггером данных – элемент управления `TextBox` делается неактивным, если в него ввести "disabled"¹.

```
<StackPanel>
  <StackPanel.Resources>
    <Style TargetType="TextBox">
      <Setter Property="Margin" Value="10" />
      <Setter Property="Height" Value="30" />
      <Setter Property="Background"
        Value="{Binding Text,
          RelativeSource={RelativeSource Self}}" />
    <Style.Triggers>
      <DataTrigger Binding="{Binding Text,
        RelativeSource={RelativeSource Self}}"
        Value="disabled">
        <Setter Property="IsEnabled" Value="False" />
      </DataTrigger>
    </Style.Triggers>
  </Style>
</StackPanel.Resources>

  <TextBox Text="Enter color name or disabled" />
  <TextBox Text="Enter color name or disabled" />
  <TextBox Text="Enter color name or disabled" />
</StackPanel>
```

¹ В примере можно использовать обычный триггер свойства, ведь `TextBox.Text` – свойство зависимостей.

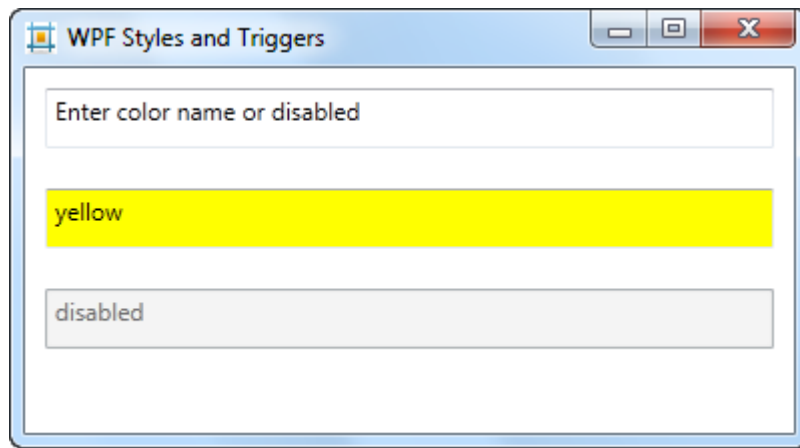


Рис. 42. Демонстрация работы триггера данных.

В предыдущих примерах коллекция `Triggers` содержала только один триггер. В эту коллекцию можно поместить любое количество триггеров – они будут работать независимо друг от друга. Если необходимо создать триггер, стартующий при наступлении совокупности изменений, следует воспользоваться классом `MultiTrigger` (или `MultiDataTrigger` – для триггеров данных). Ниже описан триггер стиля, при применении которого кнопка поворачивается, если над ней находится указатель мыши, и она имеет фокус ввода.

```
<Style.Triggers>
  <MultiTrigger>
    <MultiTrigger.Conditions>
      <Condition Property="IsMouseOver" Value="True" />
      <Condition Property="IsFocused" Value="True" />
    </MultiTrigger.Conditions>
    <Setter Property="RenderTransform">
      <Setter.Value>
        <RotateTransform Angle="10" />
      </Setter.Value>
    </Setter>
  </MultiTrigger>
</Style.Triggers>
```

12. Шаблоны

Шаблоны – фундаментальная концепция технологии WPF. Шаблоны обеспечивают настраиваемое представление (внешний вид) для элементов управления и произвольных объектов, отображаемых как содержимое элементов.

12.1. Шаблоны элементов управления

Большинство элементов управления имеют *внешний вид* и *поведение*. Рассмотрим кнопку: её внешним видом является область для нажатия, а поведением – событие `Click`, которое вызывается в ответ на нажатие кнопки. WPF эффективно разделяет внешний вид и поведение, благодаря концепции *шаблона элемента управления*. Шаблон элемента управления полностью определяет визуальную структуру элемента. Шаблон переопределяем – в большинстве случаев

это обеспечивает достаточную гибкость и освобождает от необходимости написания пользовательских элементов управления.

Рассмотрим создание пользовательского шаблона для кнопки. Шаблон элемента управления – это экземпляр класса `System.Windows.ControlTemplate`. Основным свойством шаблона является свойство содержимого `VisualTree`, которое содержит визуальный элемент, определяющий внешний вид шаблона. В элементах управления ссылка на шаблон устанавливается через свойство `Template`. С учётом вышесказанного первая версия шаблона для кнопки будет описана следующей разметкой:

```
<Button Content="Sample" Width="120" Height="80" Padding="15">
  <Button.Template>
    <ControlTemplate>
      <Border BorderBrush="Blue" BorderThickness="4"
        CornerRadius="2" Background="Gold" />
    </ControlTemplate>
  </Button.Template>
</Button>
```

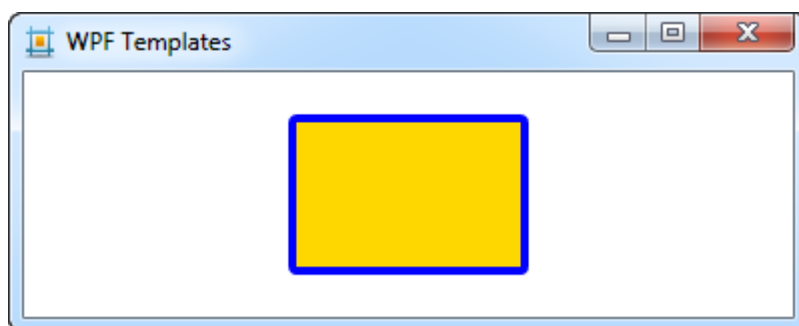


Рис. 43. Первая версия шаблона для кнопки.

Самый большой недостаток шаблона заключается в том, что он не отображает содержимое кнопки (свойство `Content`). Исправим это. У шаблона может быть установлено свойство `TargetType`. Оно содержит тип элемента управления, являющегося целью шаблона. Если это свойство установлено, при описании `VisualTree` для ссылки на содержимое элемента управления можно использовать объект `ContentPresenter` (для элементов управления содержимым) или объект `ItemsPresenter` (для списков).

```
<Button Content="Sample" Width="120" Height="80" Padding="15">
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <Border BorderBrush="Blue" BorderThickness="4"
        CornerRadius="2" Background="Gold">
        <ContentPresenter />
      </Border>
    </ControlTemplate>
  </Button.Template>
</Button>
```

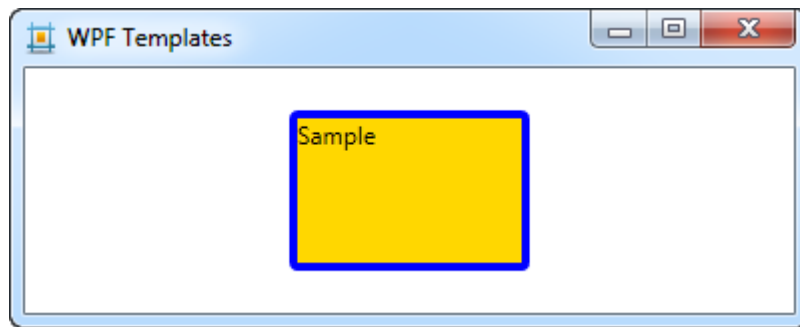


Рис. 44. Шаблон, отображающий контент.

Вторая версия шаблона не учитывает отступ, заданный на кнопке при помощи свойства `Padding`. Чтобы исправить это, используем привязку данных. В шаблонах допустим особый вид привязки – `TemplateBinding`. Эта привязка извлекает информацию из свойства элемента управления, являющегося целью шаблона.

```
<Button Content="Sample" Width="120" Height="80" Padding="15">
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <Border BorderBrush="Blue" BorderThickness="4"
        CornerRadius="2" Background="Gold">
        <ContentPresenter Margin="{TemplateBinding Padding}" />
      </Border>
    </ControlTemplate>
  </Button.Template>
</Button>
```

В шаблонах элементов управления часто используются триггеры. Например, для кнопки при помощи триггеров можно реализовать изменение внешнего вида при нажатии или при перемещении указателя мыши.

```
<Button Content="Sample" Width="120" Height="80" Padding="15">
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <Border Name="brd" BorderBrush="Blue" BorderThickness="4"
        CornerRadius="2" Background="Gold">
        <ContentPresenter Margin="{TemplateBinding Padding}" />
      </Border>
      <ControlTemplate.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
          <Setter TargetName="brd" Property="Background"
            Value="Yellow" />
        </Trigger>
      </ControlTemplate.Triggers>
    </ControlTemplate>
  </Button.Template>
</Button>
```

Обратите внимание – элемент триггера **Setter** содержит установку свойства **TargetName**. Как ясно из контекста, **TargetName** используется, чтобы обратиться к именованному дочернему элементу визуального представления. Этот дочерний элемент должен быть описан до триггера.

Заметим, что, как правило, шаблоны элементов управления описываются в ресурсах окна или приложения. Часто шаблон объявляют в стиле элемента управления. Это позволяет создать эффект «шаблона по умолчанию» (в отличие от стиля, шаблон нельзя указать в ресурсах без ключа).

```
<Window.Resources>
  <Style TargetType="Button">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="Button">
          <!--содержимое шаблона не изменилось-->
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>
```

Наш шаблон для кнопки прост. Однако во многих случаях шаблоны выглядят сложнее. Ниже перечислены некоторые из характеристик более сложных шаблонов¹:

1. Шаблоны включают элементы управления, которые иницируют встроенные команды или обработчики событий.
2. Шаблоны используют именованные элементы, имена у которых обычно начинаются с префикса *PART_*. При создании шаблона следует проверять наличие всех этих именованных элементов, поскольку класс элемента управления может включать код, который манипулирует непосредственно этими элементами (например, присоединяет обработчики событий).
3. Шаблоны включают вложенные элементы управления, которые могут иметь свои собственные шаблоны.

12.2. Шаблоны данных

Шаблон данных – механизм для настройки отображения объектов заданного типа. Любой шаблон данных – это объект `System.Windows.DataTemplate`. Основное свойство шаблона данных – `VisualTree`. Оно содержит визуальный элемент, определяющий внешний вид шаблона. При формировании `VisualTree` обычно используется привязка данных для извлечения информации из объекта, для которого применяется шаблон. Сам шаблон данных, как правило, размещают в ресурсах окна или приложения.

¹ В составе Expression Blend имеется демонстрационный проект SimpleStyles, который предоставляет коллекцию простых, отлаженных шаблонов для всех стандартных элементов управления WPF. SimpleStyles – это отправная точка для создания собственных шаблонов.

Рассмотрим пример использования шаблонов данных. Пусть имеется объект класса `Person`, описанный в ресурсах окна и являющийся содержимым окна:

```
// класс Person объявлен в пространстве имён WpfTemplates
public class Person
{
    public string Name { get; set; }
    public double Age { get; set; }
}

<Window x:Class="WpfTemplates.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfTemplates"
    Width="400" Height="160" Title="WPF Templates">
    <Window.Resources>
        <local:Person x:Key="smith" Name="Mr. Smith" Age="27.3" />
    </Window.Resources>
    <Window.Content>
        <StaticResourceExtension ResourceKey="smith" />
    </Window.Content>
</Window>
```

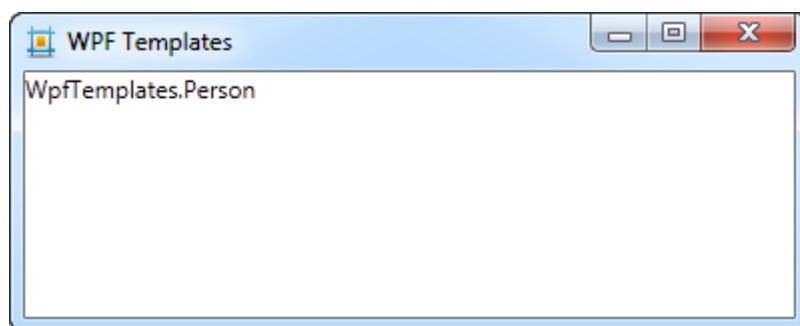


Рис. 45. Показ объекта `Person` без шаблона данных.

Определим для `Person` шаблон данных. У класса `DataTemplate` имеется свойство `DataType`, устанавливающее тип, к которому будет применяться шаблон. Если задано это свойство, шаблон будет использоваться в любой ситуации, где нужно отобразить объект.

```
<!-- остальное описание окна не изменилось -->
<Window.Resources>
    <local:Person x:Key="smith" Name="Mr. Smith" Age="27.3" />
    <DataTemplate DataType="{x:Type local:Person}">
        <Border Name="bord" BorderBrush="Aqua" BorderThickness="2"
            CornerRadius="3" Padding="10" Margin="5">
            <TextBlock FontSize="20" FontWeight="Bold"
                Text="{Binding Name}" />
        </Border>
    </DataTemplate>
</Window.Resources>
```

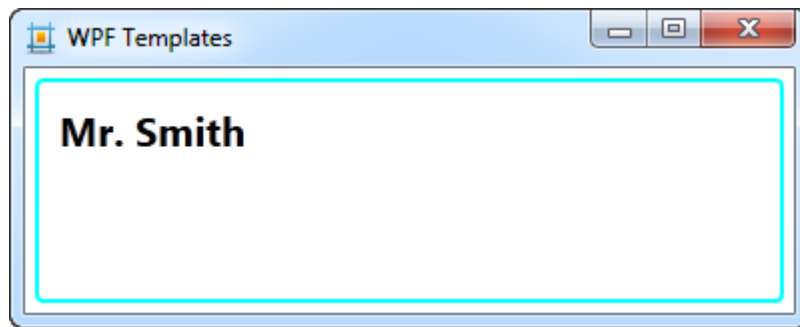


Рис. 46. Показ объекта `Person` при помощи шаблона.

В шаблон данных можно поместить триггеры данных. Следующий пример показывает использование триггера для того, чтобы изменить цвет окантовки для объектов `Person` со значением `Age=1`:

```
<!-- эта разметка - часть шаблона DataTemplate -->
<DataTemplate.Triggers>
  <DataTrigger Binding="{Binding Age}" Value="1">
    <Setter TargetName="bord" Property="BorderBrush" Value="Red" />
  </DataTrigger>
</DataTemplate.Triggers>
```

При работе с иерархическими элементами управления (например, `TreeView`) вместо шаблона данных на основе `DataTemplate` следует использовать `HierarchicalDataTemplate`. У такого шаблона имеется свойство `ItemsSource`, которое нужно связать с дочерней коллекцией, и свойство `ItemTemplate` – дочерний шаблон данных (`DataTemplate` или `HierarchicalDataTemplate`).

13. Списки и представления коллекций

В примерах параграфа будут использоваться классы `Person` и `Repository`.

```
// классы объявлены в пространстве имён Domain
public class Person : INotifyPropertyChanged, IDataErrorInfo
{
    private string _name;
    private double _age;

    public string Name
    {
        get { return _name; }
        set { SetProperty(ref _name, value, "Name"); }
    }

    public double Age
    {
        get { return _age; }
        set { SetProperty(ref _age, value, "Age"); }
    }
}
```

```

public event PropertyChangedEventHandler PropertyChanged;

public string this[string columnName]
{
    get
    {
        switch (columnName)
        {
            case "Name":
                return string.IsNullOrEmpty(Name) ?
                    "Name cannot be empty" : null;
            case "Age":
                return Age < 0 ? "Incorrect age" : null;
        }
        return null;
    }
}

public string Error
{
    get { return null; }
}

private void SetProperty<T>(ref T field, T value, string name)
{
    if (!EqualityComparer<T>.Default.Equals(field, value))
    {
        field = value;
        var handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(name));
        }
    }
}

public static class Repository
{
    public static List<Person> ReadAll()
    {
        var trinity = new Person {Name = "Trinity", Age = 32};
        var smith = new Person {Name = "Agent Smith", Age = 39};
        var brown = new Person {Name = "Agent Brown", Age = 39};
        var jones = new Person {Name = "Agent Jones", Age = 39};
        var neo = new Person {Name = "Neo", Age = 37};
        return new List<Person> {trinity, smith, brown, jones, neo};
    }
}

```


13.1. Привязка к коллекции

Построим приложение для просмотра и редактирования данных, описанных выше. Используем возможности привязки. В левой части окна разместим `ListBox`, установив его свойство `DisplayMemberPath`. Заполнение `ListBox` выполним в коде. В правой части окна расположим текстовые поля для редактирования выбранного элемента списка и кнопку для добавления элемента. Обратите внимание, как в разметке задана привязка к выбранному элементу списка.

```
<Window x:Class="WpfLists.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="400" Height="220" Title="WPF Lists">
  <StackPanel Orientation="Horizontal">
    <ListBox Name="lst" DisplayMemberPath="Name" Width="200" />

    <Grid Margin="5" Width="170"
      DataContext="{Binding SelectedItem, ElementName=lst}">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="7" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="11" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="11" />
        <RowDefinition Height="25" />
      </Grid.RowDefinitions>

      <TextBlock Grid.Row="0" Grid.Column="0" Text="Name" />
      <TextBox Grid.Row="0" Grid.Column="2"
        Text="{Binding Name, Mode=TwoWay,
          ValidatesOnDataErrors=True}" />

      <TextBlock Grid.Row="2" Grid.Column="0" Text="Age" />
      <TextBox Grid.Row="2" Grid.Column="2"
        Text="{Binding Age, Mode=TwoWay,
          ValidatesOnDataErrors=True}" />

      <Button Grid.Row="4" Grid.Column="2" Content="Add"
        Click="Button_Click" />
    </Grid>
  </StackPanel>
</Window>
```

```
// класс окна
public partial class MainWindow : Window
{
    private readonly List<Person> src;

    public MainWindow()
    {
        InitializeComponent();
        src = Repository.ReadAll();
        lst.ItemsSource = src;
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        src.Add(new Person {Name = "Morpheus", Age = 38});
    }
}
```

Запуск приложения выявляет следующий недостаток – `ListBox` не обновляет свой внешний вид при добавлении элемента. Чтобы включить отслеживание изменений, нужно в качестве источника данных использовать коллекцию с интерфейсом `INotifyCollectionChanged`. Существует единственная стандартная коллекция, реализующая этот интерфейс – `ObservableCollection<T>`.

```
// переписанный фрагмент конструктора
// тип поля src также нужно изменить
src = new ObservableCollection<Person>(Repository.ReadAll());
```

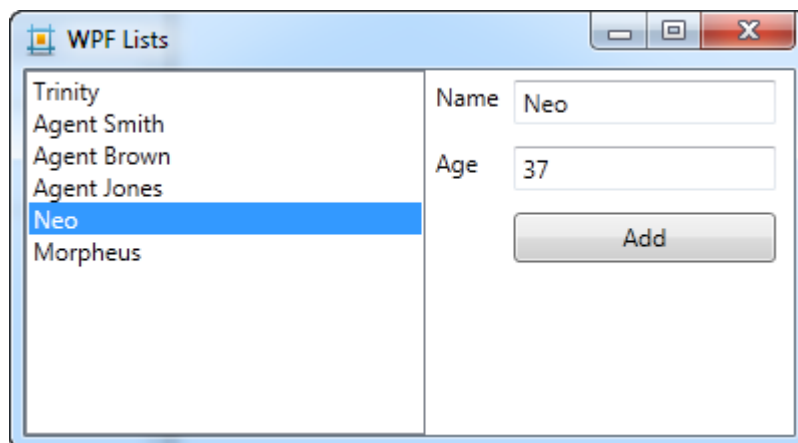


Рис. 47. Просмотр и редактирование коллекции.

13.2. Настройка внешнего вида списков

Класс `ItemsControl` обладает набором свойств, позволяющих гибко настроить внешний вид любого списка:

`Template` – шаблон списка как элемента управления. Чтобы отображать данные, помещённые в список, шаблон должен содержать `ItemsPresenter` или контейнер компоновки, у которого свойство `IsItemsHost` равно `true`.

`ItemsPanel` – контейнер компоновки для элементов списка.

`ItemContainerStyle` – стиль контейнера, обрамляющего элемент списка.

`ItemContainerStyleSelector` – объект класса-наследника `StyleSelector` с методом выбора стиля для `ItemContainerStyle`.

`ItemTemplate` – шаблон данных, используемый для отображения отдельного элемента списка.

`ItemTemplateSelector` – объект класса-наследника `DataTemplateSelector` с методом выбора шаблона для `ItemTemplate`.

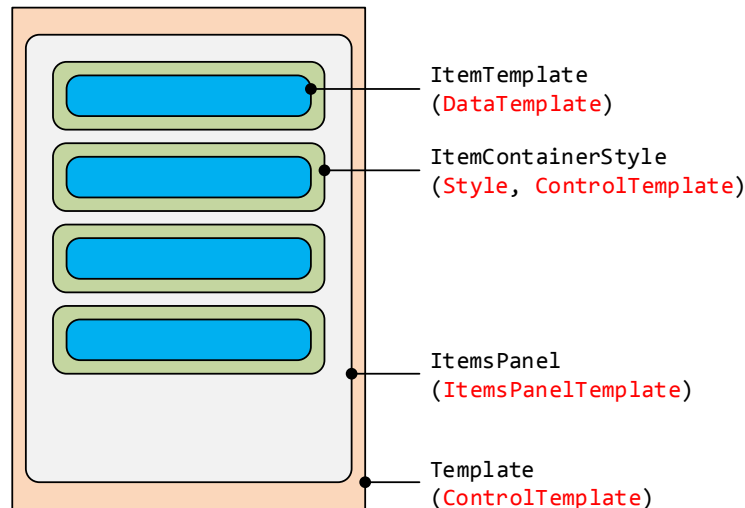


Рис 48. Структура списка.

Если у списка установлено свойство `ItemContainerStyle`, то он передаст этот стиль каждому своему элементу при его создании (в случае `ListBox` каждый элемент – это объект `ListBoxItem`). Ниже показана разметка и эффект применения `ItemContainerStyle` для списка. Обратите внимание на триггер, срабатывающий при выделении элемента.

```
<!-- помещаем стиль в ресурсы окна -->
<Window.Resources>
    <!-- этот ресурс нужен для изменения цвета выбранного элемента -->
    <SolidColorBrush x:Key="{x:Static SystemColors.HighlightBrushKey}"
        Color="DarkRed" />

    <Style x:Key="normal" TargetType="ListBoxItem">
        <Setter Property="Background" Value="LightSteelBlue" />
        <Setter Property="Margin" Value="3" />
        <Setter Property="Padding" Value="3" />
        <Style.Triggers>
            <Trigger Property="IsSelected" Value="True">
                <Setter Property="FontSize" Value="14" />
                <Setter Property="FontWeight" Value="Bold" />
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
```

```
<!-- изменения в описании ListBox -->
<ListBox Name="lst" DisplayMemberPath="Name" Width="200"
        ItemContainerStyle="{StaticResource normal}" />
```

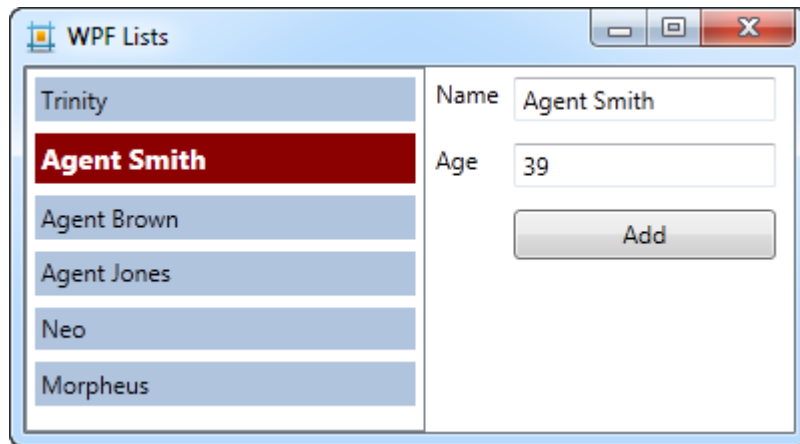


Рис. 49. Список с установленным свойством ItemContainerStyle.

При помощи *селекторов стилей* реализуется возможность менять стиль элемента списка на основе элемента данных. Селектор стилей – класс, унаследованный от `StyleSelector`, с переопределенным методом `SelectStyle()`. Для рассматриваемого примера создадим селектор стилей, выделяющий «агентов».

```
public class AgentStyleSelector : StyleSelector
{
    public Style NormalStyle { get; set; }
    public Style AgentStyle { get; set; }

    public override Style SelectStyle(object item,
                                     DependencyObject container)
    {
        var person = (Person) item;
        return person.Name.StartsWith("Agent") ? AgentStyle :
                                                    NormalStyle;
    }
}
```

Чтобы селектор стилей работал, нужно построить два стиля, а также создать и инициализировать экземпляр `AgentStyleSelector`.

```
<Window.Resources>
    <!-- стили простые, без триггеров -->
    <Style x:Key="normal" TargetType="ListBoxItem">
        <Setter Property="Background" Value="LightSteelBlue" />
        <Setter Property="Margin" Value="3" />
        <Setter Property="Padding" Value="3" />
    </Style>

    <Style x:Key="agent" TargetType="ListBoxItem">
```

```

        <Setter Property="Background" Value="Aqua" />
        <Setter Property="Margin" Value="3" />
        <Setter Property="Padding" Value="3" />
    </Style>
</Window.Resources>

<!-- настройка ListBox -->
<ListBox Name="lst" DisplayMemberPath="Name" Width="200">
    <ListBox.ItemContainerStyleSelector>
        <WpfLists:AgentStyleSelector
            NormalStyle="{StaticResource normal}"
            AgentStyle="{StaticResource agent}" />
    </ListBox.ItemContainerStyleSelector>
</ListBox>

```

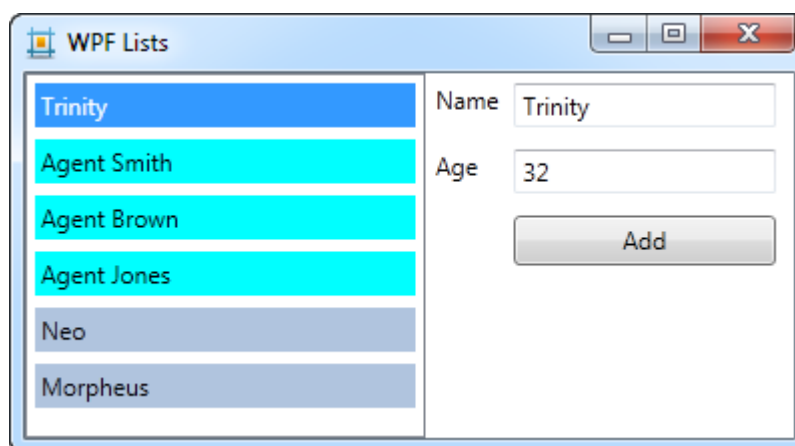


Рис. 50. Селектор стилей в действии.

Процесс выбора стилей выполняется, когда список привязывается в первый раз. Это становится проблемой, когда в результате редактирования какой-то элемент данных перемещается из одной категории стиля в другую. В такой ситуации нужно заново применить стили. При этом проверяется и обновляется каждый элемент в списке – процесс, не занимающий много времени в списках малых и средних размеров. Простой подход состоит в удалении селектора стиля установкой свойства `ItemContainerStyleSelector` в `null`, с последующим его переназначением:

```

StyleSelector selector = lst.ItemContainerStyleSelector;
lst.ItemContainerStyleSelector = null;
lst.ItemContainerStyleSelector = selector;

```

Можно организовать запуск этого кода автоматически в ответ на определённые изменения, обрабатывая такие события, как `PropertyChanged` (объявлено в `INotifyPropertyChanged`) или `Binding.SourceUpdated`.

Изменить состав отображаемой в элементе списка информации помогают шаблоны данных. Для установки шаблона данных можно использовать свойство списка `ItemTemplate` или свойство шаблона `DataTemplate` (глобальный шаблон).

У списка также имеется свойство `ItemTemplateSelector` для задания *селектора шаблонов* (работает так же, как и селектор стилей).

```
// селектор шаблона данных
public class AgentTemplateSelector : DataTemplateSelector
{
    public DataTemplate NormalTemplate { get; set; }
    public DataTemplate AgentTemplate { get; set; }

    public override DataTemplate SelectTemplate(object item,
                                                DependencyObject container)
    {
        var person = item as Person;
        return person == null ? null :
            (person.Name.StartsWith("Agent") ? AgentTemplate :
            NormalTemplate);
    }
}

<!-- описание шаблонов данных в ресурсах -->
<Window.Resources>
    <DataTemplate x:Key="normal">
        <Border Width="170" BorderBrush="Blue" BorderThickness="1"
            CornerRadius="2" Padding="3" Margin="3">
            <StackPanel>
                <TextBlock FontSize="12" FontWeight="Bold"
                    Text="{Binding Name}" />
                <TextBlock Text="{Binding Age}" />
            </StackPanel>
        </Border>
    </DataTemplate>

    <DataTemplate x:Key="agent">
        <Border Width="170" BorderBrush="Red" BorderThickness="3"
            CornerRadius="2" Padding="3" Margin="3">
            <TextBlock FontSize="16" FontWeight="Bold"
                Text="{Binding Name}" />
        </Border>
    </DataTemplate>
</Window.Resources>

<!-- настройка ListBox для использования шаблонов данных -->
<ListBox Name="lst" Width="200">
    <ListBox.ItemTemplateSelector>
        <WpfLists:AgentTemplateSelector
            NormalTemplate="{StaticResource normal}"
            AgentTemplate="{StaticResource agent}" />
    </ListBox.ItemTemplateSelector>
</ListBox>
```

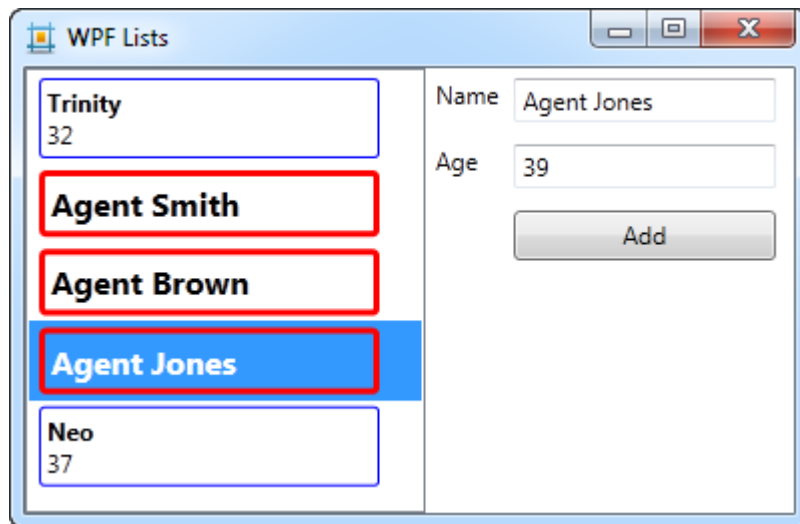


Рис. 51. Список с шаблонами данных.

Стили элемента, шаблоны данных и селекторы дают достаточный контроль над всеми аспектами представления элемента. Однако они не позволяют изменить организацию элементов относительно друг друга. Независимо от того, какие шаблоны и стили применяются, `ListBox` поместит каждый элемент в отдельную горизонтальную строку и сложит строки друг на друга в стопку, формируя список.

Эту компоновку можно изменить, заменив контейнер, который используется списком для организации своих дочерних элементов. Для этого необходимо установить свойство `ItemsPanel`. В следующем фрагменте разметки применяется `WrapPanel` в качестве оболочки вокруг доступной ширины элемента управления `ListBox`:

```
<!-- показана только настройка свойства ItemsPanel -->
<ListBox Name="lst" Width="400"
    ScrollViewer.HorizontalScrollBarVisibility="Disabled">
    <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapPanel />
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
</ListBox>
```

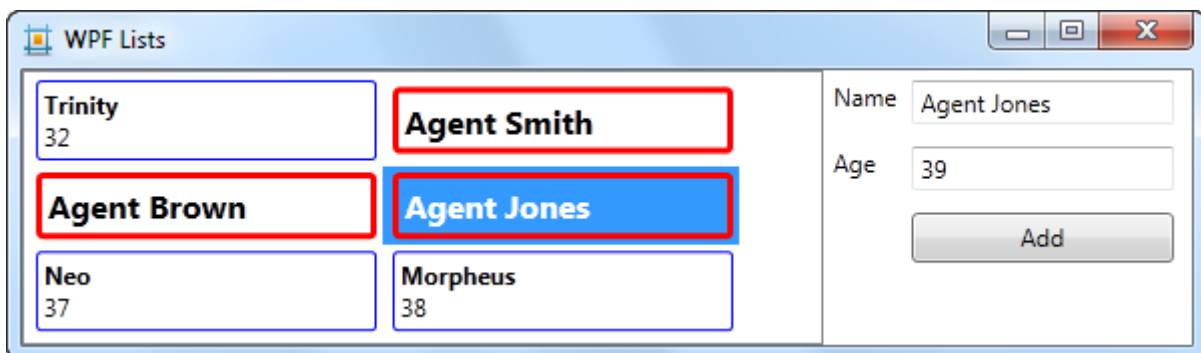


Рис. 52. Изменение контейнера компоновки элементов.

Большинство списочных элементов управления используют в качестве контейнера для компоновки [VirtualizingStackPanel](#). Этот контейнер гарантирует эффективную обработку больших списков привязанных данных. Он создаёт только те элементы, которые необходимы для отображения набора текущих видимых элементов.

13.3. Представления коллекций

При выполнении привязки коллекции к списковому элементу управления WPF использует специальный объект, созданный на основе коллекции и называемый *представлением коллекции* (collection view). Представление позволяет фильтровать, сортировать и группировать данные коллекции и даёт возможность навигации по коллекции.

Представления построены на основе интерфейса [ICollectionView](#) из пространства имён `System.ComponentModel`. Этот интерфейс унаследован от [IEnumerable](#) и [INotifyCollectionChanged](#). Его элементы описаны в табл. 9.

Таблица 9

Элементы интерфейса [ICollectionView](#)

Имя	Описание
CanFilter, CanGroup, CanSort	Булевы свойства, которые указывают на поддержку представлением фильтрации, группировки и сортировки
Contains()	Метод для проверки вхождения элемента в представление
Culture	Объект, описывающий региональные стандарты (используется при сортировке представления)
CurrentChanging, CurrentChanged	События, генерируемые при изменении позиции элемента
CurrentItem, CurrentPosition	Текущий элемент представления и его позиция
Filter	Функция фильтрации, описанная как Predicate<object>
GroupDescriptions	Коллекция объектов GroupDescription , описывающих условия группировки данных
Groups	Коллекция групп, имеющихся в представлении
IsCurrentAfterLast, IsCurrentBeforeFirst	Булевы свойства, указывающие на то, что текущий элемент представления вышел за пределы инкапсулируемой коллекции
IsEmpty	Свойство равно true , если представление не содержит данных
MoveCurrentTo(), MoveCurrentToPosition(), MoveCurrentToFirst(), MoveCurrentToLast(), MoveCurrentToNext(), MoveCurrentToPrevious()	Эти методы предназначены для изменения позиции текущего элемента представления
Refresh()	Метод для повторного создания представления по коллекции
SortDescriptions	Коллекция объектов SortDescription , описывающих критерии сортировки данных представления
SourceCollection	Коллекция, инкапсулируемая представлением

Существует четыре стандартных класса, реализующих `ICollectionView`: `CollectionView`, `ListCollectionView`, `BindingListCollectionView` (все три из пространства имён `System.Windows.Data`) и класс `ItemCollection` (пространство имён `System.Windows.Controls`). Выбор класса для представления диктуется источником данных:

1. Если источник данных реализует интерфейс `IBindingList`, создаётся объект `BindingListCollectionView`. Это происходит, когда источником является объект `DataTable`. Представления `BindingListCollectionView` не поддерживают фильтрацию через свойство `Filter`, но определяют специальное строковое свойство `CustomFilter`.

2. Если источник данных реализует `IList`, создаётся объект представления `ListCollectionView`.

3. Если источник данных реализует только интерфейс `IEnumerable`, создаётся простейший объект представления `CollectionView`. Представления этого типа не поддерживают группировку.

Реализуем при помощи представлений фильтр по возрасту. В случае, когда списковый элемент управления уже связан с коллекцией данных, получить представление можно из свойства `Items` списка (тип этого свойства — `ItemCollection`). Добавим в окно кнопку установки фильтра и кнопку сброса фильтра. Обработчики кнопок показаны ниже:

```
// обработчик для кнопки установки фильтра
private void btnAge_Click(object sender, RoutedEventArgs e)
{
    // получаем представление у списка lst
    ICollectionView view = lst.Items;

    // устанавливаем фильтр, используя лямбда-выражение
    view.Filter = p => ((Person) p).Age > 33;
}

// обработчик для кнопки сброса фильтра
private void btnReset_Click(object sender, RoutedEventArgs e)
{
    ICollectionView view = lst.Items;

    // чтобы сбросить фильтр, достаточно присвоить ему null
    view.Filter = null;
}
```

Используем представление, чтобы выполнить сортировку по возрасту. Отдельный критерий сортировки описывается при помощи структуры `System.ComponentModel.SortDescription`. При этом указывается имя поля сортировки и направление сортировки (по возрастанию или по убыванию). Представление хранит все критерии сортировки в коллекции `SortDescriptions`. Ниже показан код обработчика для кнопки сортировки:

```
private void btnSort_Click(object sender, RoutedEventArgs e)
{
    ICollectionView view = lst.Items;
    view.SortDescriptions.Add(new SortDescription("Age",
        ListSortDirection.Ascending));
}
```

Заметим, что если используется представление типа `ListCollectionView`, сортировку можно выполнить при помощи свойства `CustomSort`, которое принимает объект `IComparer`.

Представления позволяют группировать данные коллекции. Для этого нужно добавить объекты `PropertyGroupDescription` (пространство имён `System.Windows.Data`) в коллекцию представления `GroupDescriptions`:

```
private void btnGroup_Click(object sender, RoutedEventArgs e)
{
    ICollectionView view = lst.Items;
    var desc = new PropertyGroupDescription {PropertyName = "Age"};
    view.GroupDescriptions.Add(desc);
}
```

Когда используется группировка, для каждой группы создаётся отдельный объект `GroupItem`, и все эти объекты добавляются в список. Сделать группы видимыми можно, отформатировав элемент `GroupItem`. Класс `ItemsControl` имеет свойство `GroupStyle`, которое предоставляет коллекцию объектов `GroupStyle`. Каждый объект `GroupStyle` описывает стиль для отдельного уровня группировки. Несмотря на имя, класс `GroupStyle` стилем не является. Он представляет собой удобный пакет, содержащий несколько полезных параметров для конфигурирования `GroupItem`:

`ContainerStyle` – стиль для элемента `GroupItem`;

`ContainerStyleSelector` – селектор стиля для `GroupItem`;

`HeaderTemplate` – шаблон для отображения в начале каждой группы;

`HeaderTemplateSelector` – селектор шаблона `HeaderTemplate`;

`HidesIfEmpty` – булево свойство, используемое для сокрытия пустых групп.

Чтобы добавить заголовок для группы, нужно установить свойство `GroupStyle.HeaderTemplate`. Свойство является обычным шаблоном данных. Если в шаблоне будет использоваться привязка данных, её нужно выполнять относительно предназначенного для группы объекта `PropertyGroupDescription`.

```
<ListBox Name="lst" Width="240">
  <ListBox.GroupStyle>
    <GroupStyle>
      <GroupStyle.HeaderTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Name}"
            FontSize="14" FontWeight="Bold" Padding="3"
            Foreground="White" Background="Green" />
        </DataTemplate>
      </GroupStyle.HeaderTemplate>
    </GroupStyle>
  </ListBox.GroupStyle>
</ListBox>
```

```

        </DataTemplate>
    </GroupStyle.HeaderTemplate>
</GroupStyle>
</ListBox.GroupStyle>
</ListBox>

```

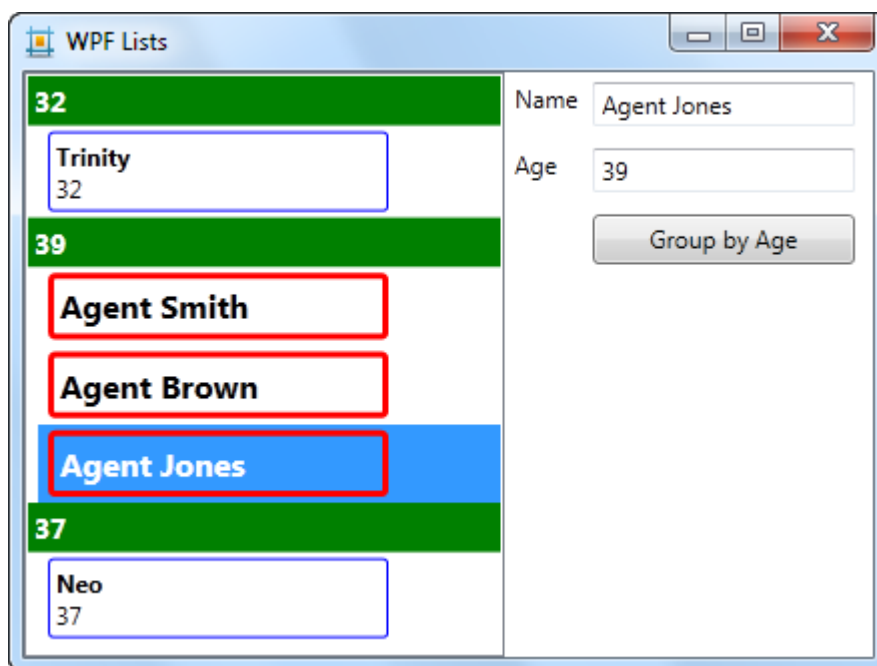


Рис. 53. Группировка списка.

Класс `System.Windows.Data.CollectionViewSource` — вспомогательный класс для работы с представлениями, который удобно использовать в разметке XAML¹. Двумя наиболее важными свойствами этого класса являются свойство `View`, которое упаковывает объект представления, и свойство `Source`, которое указывает на источник данных. У него также имеются свойства `SortDescriptions` и `GroupDescriptions` для сортировки и группировки коллекции, и событие `Filter`, которое можно использовать для выполнения фильтрации. Статический метод `CollectionViewSource.GetDefaultView()` позволяет получить представление для указанной коллекции.

¹ Если планируется использование объекта `CollectionViewSource`, привязка должна выполняться непосредственно к этому объекту, а не к его свойству `View`.

Литература

1. Андерсон, К. Основы Windows Presentation Foundation / К. Андерсон. – М.: ДМК Пресс, Спб.: БХВ-Петербург, 2008. – 432 с.: ил.
2. Мак-Дональд, М. WPF 4: Windows Presentation Foundation в .NET 4.0 с примерами на C# 2010 для профессионалов / М. Мак-Дональд. – М. : Издат. дом «Вильямс», 2011. – 1024 с.
3. Натан, А. WPF 4. Подробное руководство / А. Натан. – Спб.: Символ-Плюс, 2012. – 880 с.
4. Петцольд, Ч. Microsoft Windows Presentation Foundation / Ч. Петцольд. – М.: Издательство «Русская Редакция»; СПб.: Питер, 2008. – 944 с.: ил.
5. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – 6-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1312 с.: ил.