Урок № 1

Введение в WPF, Контейнеры, Введение в элементы управления

Содержание

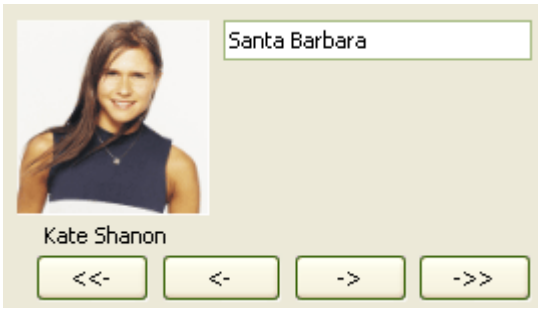
RadioButton
CheckBox
Текстовые_поля
TextBox
PasswordBox
Примеры использования элементов управления

1. Новые возможности WPF

WPF – это набор классов, предназначенный для создания пользовательских интерфейсов в интеллектуальных клиентских приложениях с поддержкой мультимедиа(звук и видео). **WPF** базируется на векторной графике и **DirectX**, приложение не использует оконные дескрипторы. Производительность **WPF** выше, чем у **GDI+** за счёт использования видеокарты.

Одно из основных преимуществ – возможность разделения труда между дизайнерами и разработчиками. До создания **WPF** дизайнер разрабатывал макет, а программист решал, как реализовать этот макет программными средствами. Зачастую это выглядело достаточно убого. Вот пример из статьи http://www.gotdotnet.ru/blogs/zxmd/6545/ . Дизайнер разработал макет:

Программист его «реализовал»:



Применение **WPF** позволяет одновременно решить 2 задачи:

1. Разгрузить программиста от процесса "рисования форм". Теперь ему не надо заботиться о внешнем виде программы, об этом позаботится дизайнер. Задача программиста будет заключаться только в описании бизнес логики приложения.

2. Отделение данных от их представления. Это означает, что в любой момент можно сменить внешний вид программы, при этом не затрагивая логику работы.

Это возможно благодаря использованию расширения **языка XML – XAML, XML Application Markup Language** (**произносится** [зэмл] или [замл]) **– XML** для разметки приложений, который широко используется в **.NET Framework 3.0.** В **WPF XAML** используется:

- - как язык разметки пользовательского интерфейса,
- - для определения элементов пользовательского интерфейса,
- - привязки данных,
- - поддержки событий и др. свойств.

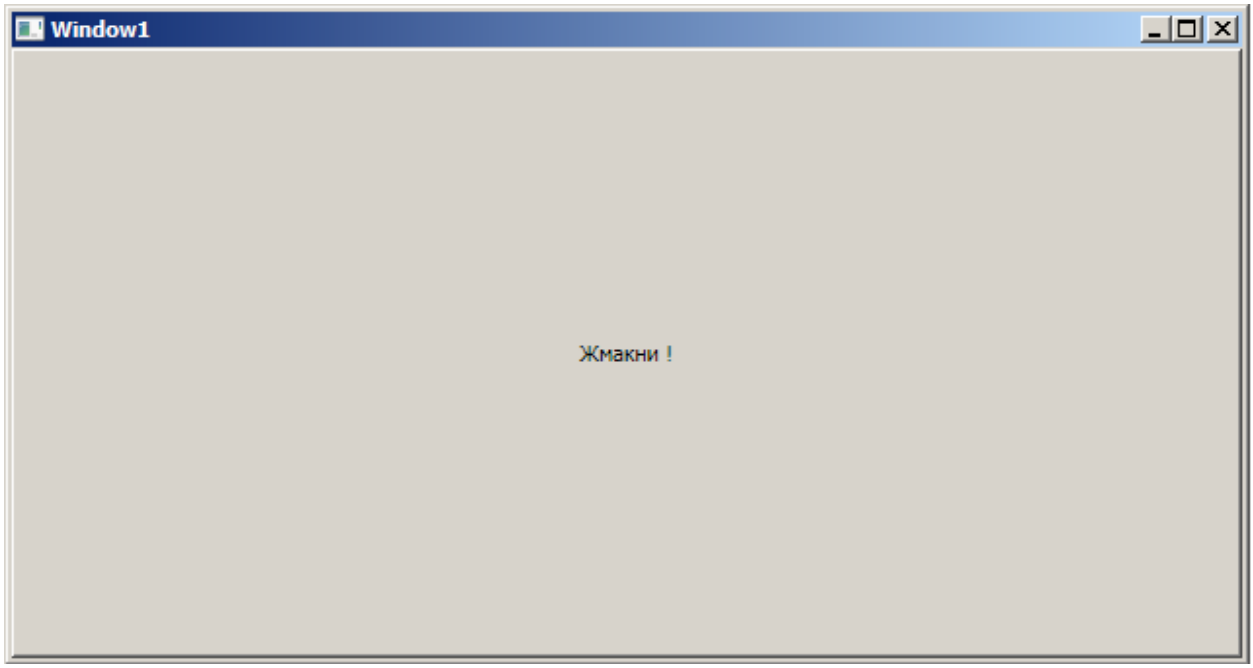
Варианты применения **XAML**-документа:

1. может быть внедрен в код приложения и интерпретируется исполняющей системой **WPF**;
2. может быть отдельно скомпилирован в **BAML (Binary Application Markup Language)** и добавлен в исполняемый файл в виде ресурса.

Вот пример простого приложения на **XAML**:

```
<Window x:Class="WpfApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="339" Width="639">
    <Grid>
        <Button Name="gm">
            Жмакни !
        </Button>
    </Grid>
</Window>
```

Вид приложения:



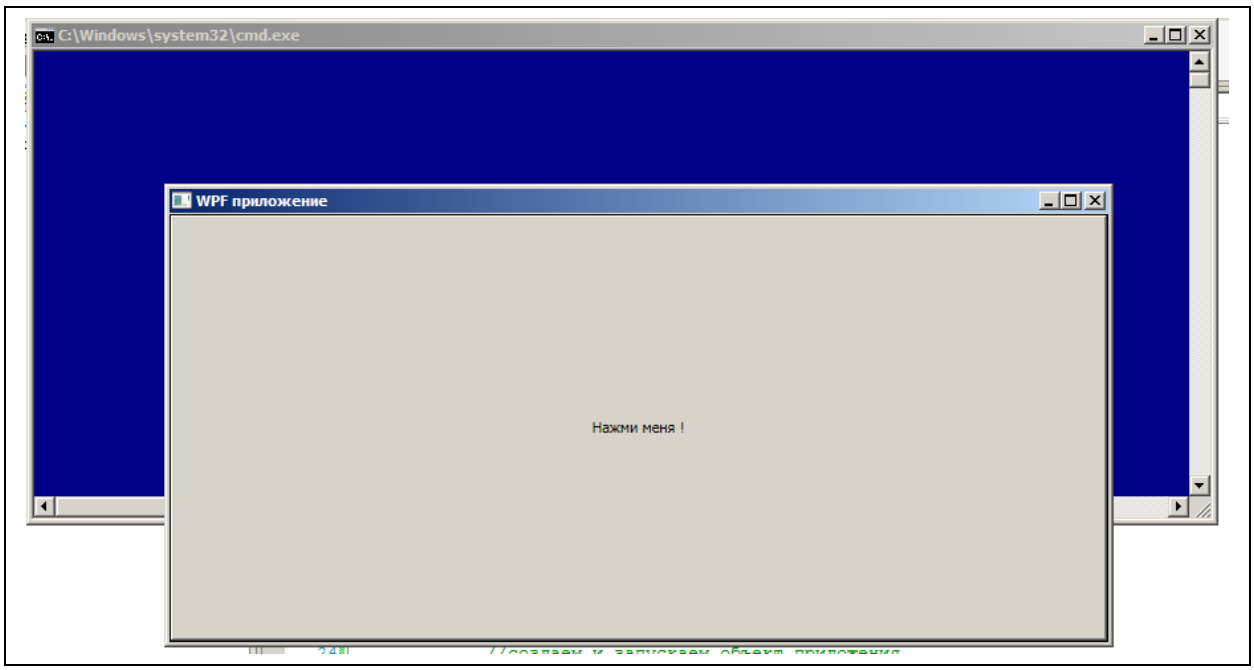
Приложение WPF может создаваться прямо в коде программы на C#. Создаем консольный проект, добавляем ссылки на сборки **WindowsBase**, **PresentationCore** и **PresentationFramework**. Запускается консольное окно и окно WPF- приложения

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace WPFConsole
{
    class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
            // создаем обьект окно WPF-приложения
            Window WinMain = new Window();
            // Создаем WPF-управляющий элемент
            Button klack = new Button();
            // связываем содержимое окна с управляющим элементом
            WinMain.Content = klack;

            WinMain.Title = "WPF приложение";
            klack.Content = "Нажми меня !";
            //создаем и запускаем объект приложения
            new Application().Run(WinMain);
        }
    }
}
```

Вот как выглядит результат:



2. Элементы и атрибуты.

Элементы **XAML** это **XML** - элементы, содержащие названия контейнеров, управляющих элементов или их свойства. Вот пример элемента Button

```
<Button>
          Жмакни !
      </Button>
```

Не правда ли, синтаксис хорошо знаком из **XML** ?

Элементы **XAML** могут использовать атрибуты для определения своих свойств:

<Button Content= «Нажми !» BackGround= «LightGreen» />

Свойство Content получает значение «Нажми !» , свойство BackGround – значение «LightGreen» .

Можно определять свойства элемента как дочерние элементы :

<Button>

```
        Нажми
        <Button.Background>
            LightGreen
        </Button.Background>
</Button>
```

3. Свойства зависимости.

Известно, что CLR дает возможность использовать в качестве членов классов свойства с параметрами или без. Например, в C# мы можем воспользоваться этим для создания **get**- и/или **set**- элементов для членов-данных и индексаторов с различным количеством параметров. Лаконичный синтаксис таких свойств весьма удобен и везде используется.

В контексте WPF ситуация обстоит таким образом, что в некоторых случаях функциональности обычных свойств недостаточно. Причины этого следующие:

- Иногда возникает необходимость наследования значений свойств в порядке отношений «родитель-ребенок» логического дерева. Например, если для родительской панели задан красный цвет фона, то такой цвет фона должны иметь и дочерние кнопки
- Часто у клиентов какого-либо класса возникает потребность в отслеживании событий о изменении какого-либо свойства, не имея информации о конкретном свойстве (например, нужно отслеживать значение цвета фона, не имея сведений о наличии события **BackColorChanged**). Одним из способов достижения этого является использование стандартного интерфейса **INotifyPropertyChanged**, который находится в пространстве имен **System.ComponentModel** и содержит единственное событие

PropertyChanged, аргументы которого включают имя измененного свойства в форме обычной строки.

- Встречаются случаи, когда для реализации определенной подсистемы необходимы механизмы работы на метауровне наподобие **Reflection**, но очень быстрые. Тогда обычно применяются не очень-то элегантные решения типа **Property Bags**, сделанные на ассоциативных массивах и другие подобные вещи.

В качестве архитектурно целостного и элегантного решения этих и некоторых других проблем WPF вводит концепцию **Dependency Properties** (свойства зависимости).

На базе Dependency Properties реализованы следующие подсистемы WPF:

- Стили
- **Data Binding**
- Наследование свойств в логическом дереве
- Анимация свойств
- Расширения разметки XAML
- **Layout** визуальных элементов и другое

Dependency Properties можно рассматривать с двух точек зрения – как их использовать (на клиентской стороне), и как их создавать (на серверной стороне).

Для того, чтобы использовать такое свойство, вам не нужно учитывать практически ничего – для клиента оно выглядит как обычное CLR-свойство. То есть, пока вам в ваших классах не нужна функциональность, предоставляемая подсистемой **Dependency Properties**, вам не нужно знать об этом совсем ничего.

Для того, чтобы создать **Dependency Property** в своем коде, нужно знать и уметь определенные вещи. Использование **Dependency Properties** накладывает на дизайн вашего кода некоторые ограничения. Во-первых, класс, который содержит **Dependency Property**, должен быть наследником класса **DependencyObject**. Во-вторых, такие свойства создаются по определенному несложному шаблону. Идея этого шаблона состоит в том, чтобы сначала зарегистрировать описание вашего свойства, используя определенные статические методы классов среды, а затем предоставить доступ клиентам к этому свойству при помощи фасада из обычного CLR-свойства. Чтобы не путать вас таким описанием, приведу простой пример кода.

```
namespace Test
{
    public class Employee : DependencyObject
    {
        public readonly static DependencyProperty NameProperty =
            DependencyProperty.Register("Name",
                                        typeof(string),
                                        typeof(Employee));


        public string Name
        {
            get { return (string)GetValue(NameProperty); }
            set
            {
                if (Name == value)
                    return;

                SetValue(NameProperty, value);
            }
        }
    }
}
```

Регистрация

Фасад

Видно, что на статическом уровне мы сначала регистрируем описание свойства, а затем предоставляем фасад для него при помощи обычного синтаксиса. Все довольно просто. Конечно, при разработке могут потребоваться и более сложный синтаксис регистрации, но в подавляющем большинстве случаев это выглядит так, как показано выше.

Таким образом, мы получаем видимое извне "обычное" свойство, а также тот факт, что описание этого свойства зарегистрировано во внутренних механизмах WPF и эта регистрационная информация может быть использована для поддержки описанной выше функциональности.

В заключение хочется подчеркнуть следующее. Далеко не все свойства в WPF – **Dependency Properties**. Этот механизм используется только там, где это нужно. Таким образом, для прикладного программиста должно стать естественным правило использовать **Dependency Properties** только при необходимости.

Свойства зависимости в отличие от обычных свойств связаны со значениями других компонентов WPF, в которые входят:

- ресурсы
- привязка данных
- стили
- анимации
- переопределения метаданных
- наследование значения свойства
- интеграция конструктора WPF

При использовании XAML свойство зависимости регистрируется при указании свойства. Например, здесь свойство Background элемента Button зависит от значения ресурса <StackPanel.Resourses> в котором определяется свойство (Property = «Background») и его значение (Value = «Red»). Регистрация зависимости происходит указанием ключа и целевого типа <Style x:Key= «{x:Type Button}» TargetType= «{x:Type Button}».

```
<StackPanel>
  <StackPanel.Resources>
```

```
    <Style x:Key="{x:Type Button}" TargetType="{x:Type Button}">
     <Setter Property="Background" Value="Red"/>
    </Style>
  </StackPanel.Resources>
  <Button Background="Green">I am NOT red!</Button>
  <Button>I am styled red</Button>
</StackPanel>
```

Указанное локально значение <Button Background="Green"> имеет приоритет над свойством зависимости. Вот результат этой программы:

Присоединенные свойства

Все элементы WPF получают также свойства тех элементов, внутри которых они расположены. Эти свойства называются присоединенными свойствами и доступны через префикс – имя родительского элемента(Canvas.Top):

```
    <Canvas>
            <Button Canvas.Top="30" Canvas.Left="40" Height="46" Width="80">
                Ещё !
            </Button>
        </Canvas>
```

В коде на C# присоединенные свойства становятся доступны через включение элемента в родительский и методы родительского элемента, которые именуются в виде **Set<имя свойства>** и **Get<имя свойства>** (SetTop):

```
            // создаем родительский объект
            Canvas cv = new Canvas();
            // связываем содержимое окна с управляющим элементом
            WinMain.Content = cv;
            Button b2 = new Button();
            // включаем кнопку в родительский элемент
            cv.Children.Add(b2);
            b2.Content = "Я в канве";
            // меняем свойства дочернего элемента методами родительского
            Canvas.SetTop(b2, 30);
            Canvas.SetLeft(b2, 50);
```

Суть разделения труда между дизайнером и разработчиком состоит в том, что дизайнер проектирует внешний вид приложения на **XAML** в одном файле, а программист разрабатывает функциональность на **C#** в другом(или нескольких других). Такое разделение представления и логики функционирования по разным файлам получило название **Code Behind**. При этом в качестве Content управляющих элементов могут выступать и текст, и изображение, и видеоклип. В **WPF** такое разделение является основой технологии. Рассмотрим пример :

4. Пример простой программы

Как же производится соединение **XAML** и C# ? Так же, как добавляются обработчики событий на **JavaScript** к элементам **HTML**-странички , так же добавляются обработчики и в **XAML** , с той разницей, что обработчик формируется средой редактирования при двойном щелчке:

XAML : <Button … Click="Button_Click">
И в C# : private void Button_Click(object sender, RoutedEventArgs e) {…}

Значением атрибута Click становится имя метода-обработчика, вызываемого при событии Click. Сам метод находится в коде C#. Итак две кнопки, два двойных щелчка, два обработчика. Код **XAML** :

```
<Window x:Class="WpfApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="484" Width="775">
   <!-- Элемент Canvas Определяет область, в рамках которой можно явно
расположить дочерние
    элементы путем использования координат, являющихся относительными к
области Canvas.
    -->
        <Canvas>
            <!-- Вложенный элемент Button присоединяет свойства внешнего
элемента
            Canvas.Top и Canvas.Left
              ( и другие -тоже ) -->
```

```
            <Button Canvas.Top="0" Canvas.Left="0" Height="443" Width="260"
Name="gm" Click="gm_Click">
            Жмакни !
            <!-- свойства элемента могут быть заданы атрибутами или
вложенными элементами -->
            <Button.Background>
                <!-- построение градиентной заливки фона -->
                <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                    <!-- точки разбиения градиентной заливки -->
                    <GradientStop Color="Yellow" Offset="0.0"/>
                    <GradientStop Color="Magenta" Offset="0.35"/>
                    <GradientStop Color="Cyan" Offset="0.55"/>
                    <GradientStop Color="LightGreen" Offset="1.0"/>
                </LinearGradientBrush>
            </Button.Background>
        </Button>
        <!-- Еще один вложенный элемент Button присоединяет свойства
внешнего элемента
            Canvas.Top и Canvas.Left( и другие -тоже ) -->
        <Button Canvas.Top="0" Canvas.Left="257" Height="443" Width="496"
Click="Button_Click">
            <!-- MediaElemen представляет элемент управления, содержащий
аудио и/или видео.
            LoadedBehavior - поведение загрузки MediaState для файла
мультимедиа. Это свойство зависимости.
            -->
            <MediaElement Name="mediaElement1" Source="..\..\Begemotik2.avi"
LoadedBehavior="Manual"  />
            </Button>

        </Canvas>
  <!--  </Grid> -->

</Window>
```

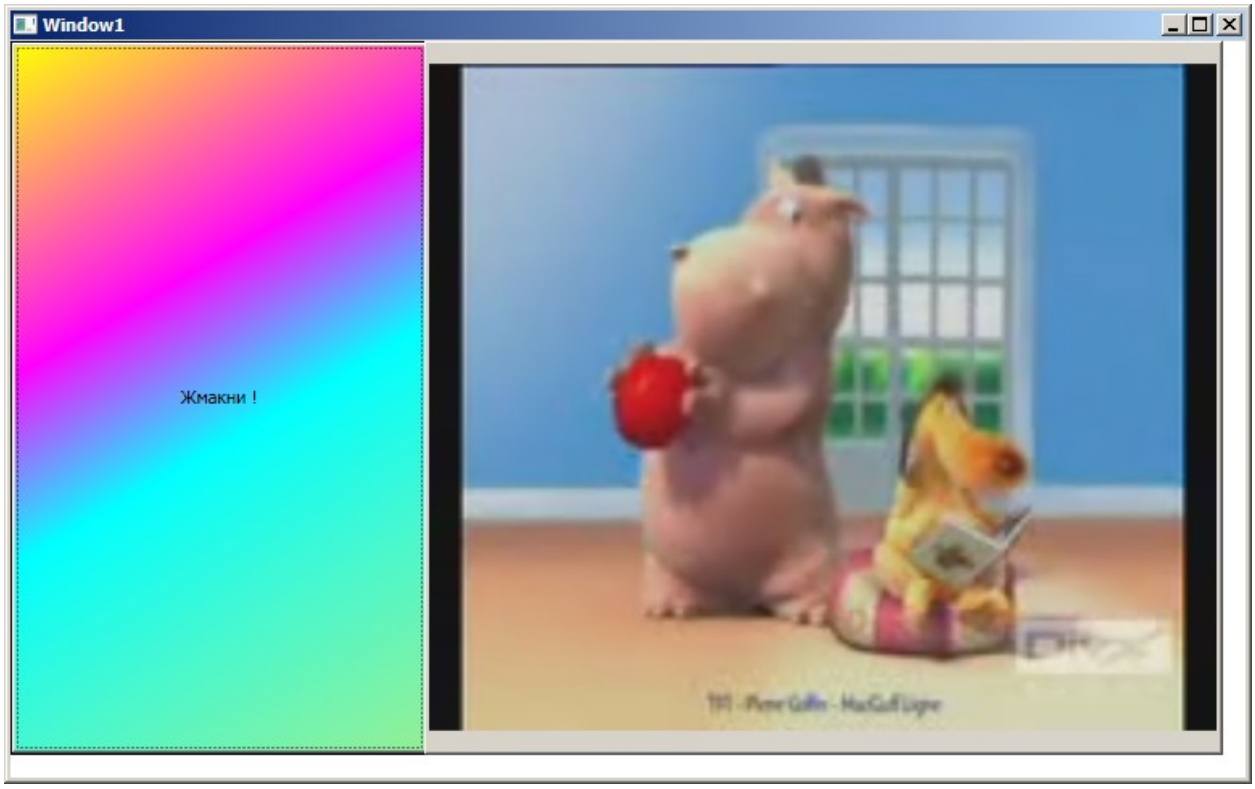
Логика реакции программы на действия пользователя (C#):

```
namespace WpfApplication1
{
    /// <summary>
    /// Логика взаимодействия для Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
        // обработчик события нажатия на серую кнопку
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            // Следующая строчка добавлена самостоятельно
            MessageBox.Show("Нажата серая кнопка");
        }
        // обработчик события нажатия на цветную кнопку
        private void gm_Click(object sender, RoutedEventArgs e)
        {
```

```
            // Следующая строчка добавлена самостоятельно
            MessageBox.Show("Нажата цветная кнопка");
        }

    }
```

Внешний вид приложения :



Окончательный вариант приложения доступен в папке Source с примерами.

5. Контейнеры

Множество усилий при создании пользовательского интерфейса тратится на организацию содержимого, чтобы оно было привлекательным, практичным и гибким. Особенно при адаптации элементов интерфейса к разным размерам окна.

Определимся с используемыми терминами. **Компоновка** элементов управления - это модель организации пользовательского интерфейса. **Контейнер** – это элемент wpf, который может содержать другие элементы как дочерние

В WPF мы разделяем компоновку элементов интерфейса, используя разные контейнеры. Каждый контейнер имеет свою собственную логику компоновки — некоторые складывают элементы в стопку, другие распределяют их по ячейкам сетки и т.д. В WPF упор делается на создание компоновок, которые могут адаптироваться к изменяющемуся содержимому, разным языкам и широкому разнообразию размеров окон. В этом разделе мы познакомимся, как работает модель компоновки WPF, и начнем использовать базовые контейнеры компоновки. Также мы рассмотрим пример компоновки — чтобы изучить основы компоновки WPF на практике.

Окно WPF может содержать только один элемент. Например, элемент управления - кнопку. Но современный пользовательский интерфейс должен быть богаче, чем просто одна кнопка ☺. Чтобы разместить более одного элемента и создать более практичный пользовательский интерфейс, вам нужно поместить в окно контейнер и затем добавлять элементы управления в этот контейнер.

В WPF компоновка определяется используемым контейнером. Существует несколько контейнеров для компоновки:

StackPanel	Размещает элементы в горизонтальный или вертикальный стек. Этот контейнер компоновки обычно используется в небольших секциях крупного более сложного окна.
WrapPanel	Размещает элементы в сериях строк с переносом. В горизонтальной ориентации **WrapPanel** располагает элементы в строке слева направо, затем переходит к следующей строке. В вертикальной ориентации **WrapPanel** располагает элементы сверху вниз, используя дополнительные колонки для дополнения оставшихся элементов.
DockPanel	Выравнивает элементы по краю контейнера.
Grid	Выстраивает элементы в строки и колонки невидимой таблицы. Это один из наиболее гибких и широко используемых контейнеров компоновки.

UniformGrid	Помещает элементы в невидимую таблицу, устанавливая одинаковый размер для всех ячеек. Данный контейнер компоновки используется не часто.
Canvas	Позволяет элементам позиционироваться абсолютно - по фиксированным координатам. Этот контейнер компоновки более всего похож на компоновщик **Windows Forms**, но не предусматривает средств привязки и стыковки. В результате это неподходящий выбор для окон переменного размера.

Окно WPF-приложения должно соответствовать описанным ниже ключевым принципам:

• Элементы (такие как элементы управления) не должны иметь явно установленных размеров. Вместо этого они растут, чтобы вместить их содержимое. Например, кнопка увеличивается, когда вы добавляете в нее текст. Вы можете ограничить элементы управления приемлемыми размерами, устанавливая максимальное и минимальное их значение.

• Элементы не указывают свою позицию в экранных координатах. Вместо этого они упорядочиваются своим контейнером на основе размера, порядка и (необязательно) другой информации, специфичной для контейнера компоновки. Если вы хотите добавить пробел между элементами, то используете для этого свойство **Margin**.

• Контейнеры компоновки "разделяют" доступное пространство между своими дочерними элементами. Они пытаются предоставить каждому элементу его предпочтительный размер (на основе его содержимого), если позволяет свободное пространство. Они могут также выделять дополнительное пространство одному или более дочерним элементам.

• Контейнеры компоновки могут быть вложенными. Типичный пользовательский интерфейс начинается с **Grid** - наиболее развитого контейнера, и содержит другие контейнеры компоновки, которые организуют меньшие группы элементов, такие как текстовые поля с

метками, элементы списка, пиктограммы в панели инструментов, колонки кнопок и т.д.

Компоновка WPF происходит за две стадии: стадия измерения и стадия расстановки. На стадии измерения контейнер выполняет проход в цикле по дочерним элементам и опрашивает их предпочтительные размеры. На стадии расстановки контейнер помещает дочерние элементы в соответствующие позиции. Конечно, элемент не может всегда иметь свой предпочтительный размер — иногда контейнер недостаточно велик, чтобы обеспечить его. В этом случае контейнер должен усекать такой элемент для того, чтобы вместить его в видимую область. Как вы убедитесь, часто можно избежать такой ситуации, устанавливая минимальный размер окна.

Все контейнеры компоновки WPF являются панелями, которые унаследованы от абстрактного класса **System.Windows.Controls**:

Класс **Panel** добавляет набор членов, включая общедоступные свойства:

Имя	Описание
Background	Получает или задает объект **Brush**, который используется для заполнения области между границами объекта **Panel**. Это свойство зависимости.
Children	Получает коллекцию **UIElementCollection** дочерних элементов этого объекта **Panel**.
IsItemsHost	Получает или задает значение, которое указывает, что

	этот объект **Panel** является контейнером для элементов пользовательский интерфейс, созданных объектом **ItemsControl** (такие как узлы из **TreeView** или элементы списка **ListBox**).
ZIndex	Получает или задает значение, которое представляет порядок в плоскости **z**, в которой будет выведен элемент.

Базовый класс **Panel** — это не что иное, как начальная точка для построения других более специализированных классов. WPF предлагает ряд производных от **Panel** классов, которые вы можете использовать для организации компоновки. Рассмотрим их подробнее.

Grid

Grid — наиболее мощный контейнер компоновки в WPF. Большая часть того, что вы можете достичь с другими элементами управления компоновкой, также возможна и в **Grid**. Контейнер **Grid** является идеальным инструментом для разбиения вашего окна на меньшие области, которыми вы можете управлять с помощью других панелей.

Фактически **Grid** настолько удобен, что когда вы добавляете новый документ XAML для окна в Visual Studio, он автоматически добавляет дескрипторы **Grid** в качестве контейнера первого уровня, вложенного внутрь корневого элемента Window.

Grid располагает элементы в невидимой сетке строк и колонок. Хотя в отдельную ячейку этой сетки можно поместить более одного элемента (и тогда они перекрываются), обычно имеет смысл помещать в каждую ячейку по одному элементу. Конечно, этот элемент сам может быть другим контейнером компоновки, который организует свою собственную группу содержащихся в нем элементов управления.

Создание компоновки на основе **Grid** — двухшаговый процесс. Сначала вы выбираете количество колонок и строк, которые вам нужны. Затем назначаете соответствующую строку и колонку каждому содержащемуся элементу, тем самым помещая его в правильное место.

Вы создаете колонки и строки, заполняя объектами коллекции **Grid.ColumnDefinitions** и **Grid.RowDefinitions**. Например, если вы решите, что вам нужно две строки и три колонки, вы должны добавить следующие дескрипторы:

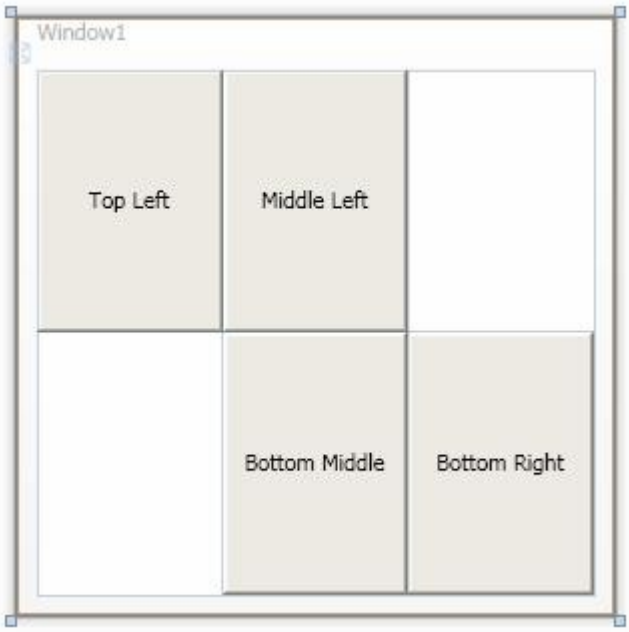
```
<Grid ShowGridLines="True">
<Grid.RowDefinitions>
<RowDefinition></RowDefinition>
<RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition></ColumnDefinition>
<ColumnDefinition></ColumnDefinition>
<ColumnDefinition></ColumnDefinition>
</Grid.ColumnDefinitions>
</Grid>
```

Как демонстрирует этот пример, не обязательно указывать какую-либо информацию в элементах **RowDefinition** или **ColumnDefinition**. Если вы оставите их пустыми (как показано здесь), то **Grid** поровну разделит пространство между всеми строками и колонками. В данном примере каждая ячейка будет одного и того же размера, который зависит от размера включающего окна:

Чтобы поместить индивидуальные элементы в ячейку, вы используете прикрепленные свойства **Row** и **Column**. Оба эти свойства принимают числовое значение индекса, начиная с 0. Например, вот как вы можете создать частично заполненную кнопками сетку:

```
<Grid ShowGridLines="True">
. . . .
<Button Grid.Row="0" Grid.Column="0">Top Left</Button>
<Button Grid.Row="0" Grid.Column="1">Middle Left</Button>
<Button Grid.Row="1" Grid.Column="2">Bottom Right</Button>
<Button Grid.Row="1" Grid.Column="1">Bottom Middle</Button>
</Grid>
```

Ниже показано, как эта простая сетка выглядит. Обратите внимание, что свойство **ShowGridLines** установлено в **true**, так что вы можете видеть границы между колонками и строками.



Grid предоставляет базовый набор свойств компоновки. Это значит, что вы можете добавлять поля вокруг содержимого ячейки, можете менять режим изменения размера, чтобы элемент не рос, заполняя ячейку целиком, а также вы можете выравнивать элемент по одному из граней ячейки. Если вы заставите элемент иметь размер, превышающий тот, что может вместить ячейка, часть содержимого будет отсечена.

DockPanel

Эта панель растягивает элементы управления вдоль одной из внешних границ. Простейший способ визуализировать это — вообразить линейку инструментов, которая присутствует в верхней части многих приложений Windows. Такие линейки инструментов прикрепляются к вершине окна. Если вы прикрепите кнопку к вершине **DockPanel**, она растянется на всю ширину **DockPanel**, но получит высоту, которая ей потребуется (на основе ее содержимого и свойства **MinHeight**). Если же вы прикрепите кнопку к левой стороне контейнера, ее высота будет

растянута для заполнения контейнера, но ширина будет установлена по необходимости.

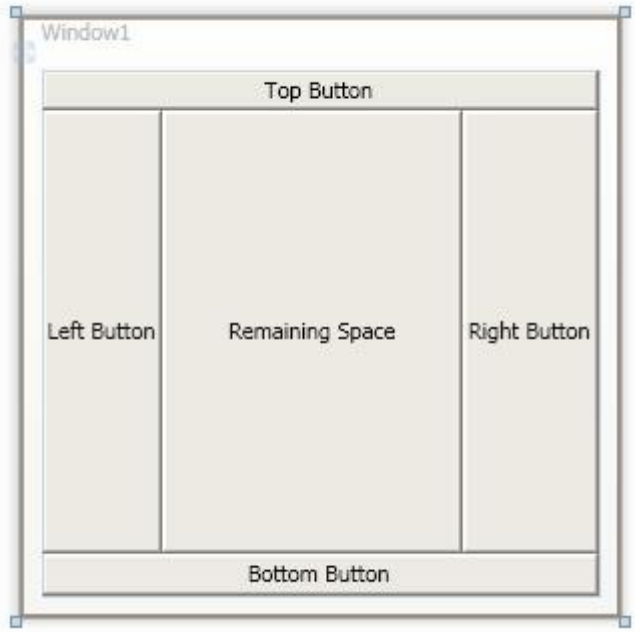
Возникает очевидный вопрос: как дочерние элементы выбирают сторону, к которой они хотят стыковаться? Ответ— через прикрепленное свойство по имени **Dock**, которое может быть установлено в **Left**, **Right**, **Top** или **Bottom**. Каждый элемент, помещаемый внутри **DockPanel**, автоматически получает это свойство.

Ниже приведен пример, который помещает одну кнопку на каждую сторону **DockPanel**:

```
<DockPanel LastChildFill="True">
<Button DockPanel.Dock="Top">Top Button</Button>
<Button DockPanel.Dock="Bottom">Bottom Button</Button>
<Button DockPanel.Dock="Left">Left Button</Button>
<Button DockPanel.Dock="Right">Right Button</Button>
<Button>Remaining Space</Button>
</DockPanel>
```

В этом примере также устанавливается **LastChildFill** в **true**, что указывает **DockPanel** о необходимости отдать оставшееся пространство последнему элементу.

Результат показан ниже:



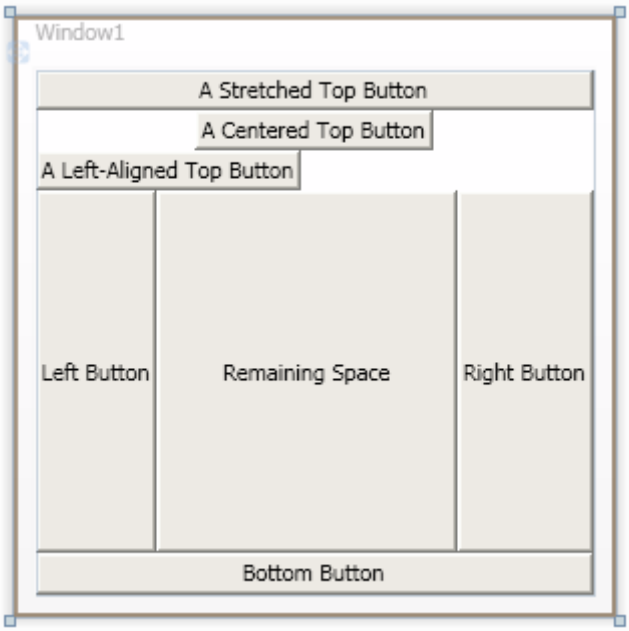
При такой стыковке элементов управления важен порядок. В данном примере верхняя и нижняя кнопки получают всю ширину **DockPanel**. поскольку они стыкованы первыми. Когда затем стыкуются левая и правая кнопки, они помещаются между первыми двумя. Если поступить наоборот, то левая и правая кнопки получат полную высоту сторон панели, а верхняя и нижняя станут уже, потому что им придется размешаться уже между боковыми кнопками.

Вы можете стыковать несколько элементов к одной стороне. В этом случае элементы просто выстраиваются вдоль этой стороны в том порядке, в котором они объявлены в вашей разметке. И, если вам не нравится поведение в отношении растяжения и промежуточных пробелов, вы можете подкорректировать свойства **Margin**, **HorizontalAlignment** и **VerticalAlignment**. Ниже приведена модифицированная версия предыдущего примера:

```
<DockPanel LastChildFill="True">
<Button DockPanel.Dock="Top">A Stretched Top Button</Button>
<Button DockPanel.Dock="Top" HorizontalAlignment="Center">
A Centered Top Button</Button>
<Button DockPanel.Dock="Top" HorizontalAlignment="Left">
A Left-Aligned Top Button</Button>
```

```
<Button DockPanel.Dock="Bottom">Bottom Button</Button>
<Button DockPanel.Dock="Left">Left Button</Button>
<Button DockPanel.Dock="Right">Right Button</Button>
<Button>Remaining Space</Button>
</DockPanel>
```

Поведение в отношении стыковки остается прежним. Сначала стыкуются верхние кнопки, затем — нижняя кнопка и оставшееся пространство делится между боковыми кнопками, а последняя кнопка помещается в середине. Ниже можно видеть полученное в результате окно:



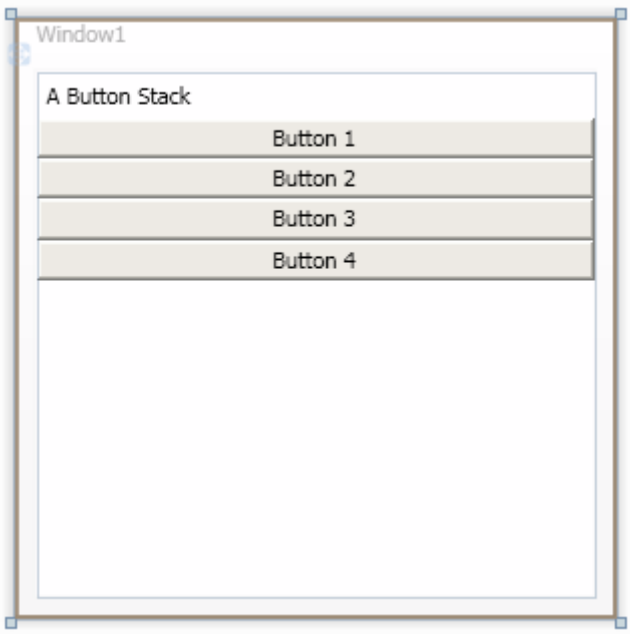
StackPanel

StackPanel — один из простейших контейнеров компоновки. Он просто укладывает свои дочерние элементы в одну строку или колонку. Например, рассмотрим следующее окно, которое содержит стек из трех кнопок:

```
<StackPanel>
<Label>A Button Stack</Label>
```

```
<Button>Button 1</Button>
<Button>Button 2</Button>
<Button>Button 3</Button>
<Button>Button 4</Button>
</StackPanel>
```

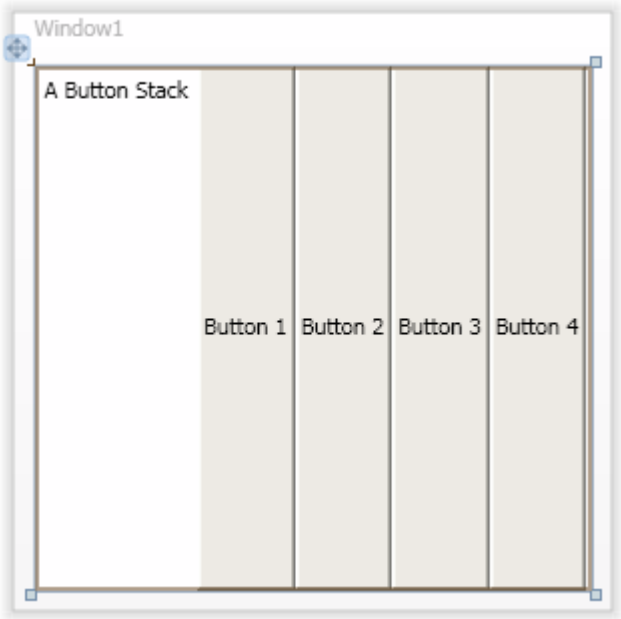
Ниже показано полученное в результате окно:



По умолчанию **StackPanel** располагает элементы сверху вниз, устанавливая высоту каждого из них такой, которая необходима для отображения его содержимого. В данном примере это значит, что размер меток и кнопок устанавливается достаточно большим для комфортабельного размещения текста внутри них. Все элементы растягиваются на полную ширину **StackPanel**, которая равна ширине окна. Если вы расширите окно, **StackPanel** также расширится, и кнопки растянутся, чтобы заполнить ее. **StackPanel** может также использоваться для упорядочивания элементов в горизонтальном направлении посредством установки свойства **Orientation**:

```
<StackPanel Orientation="Horizontal">
```

Теперь элементы получают свою минимальную ширину (достаточную чтобы вместить их текст) и растягиваются до полной высоты, чтобы заполнить содержащую их панель:



Хотя компоновка определяется контейнером, дочерние элементы тоже могут сказать свое слово. Фактически панели компоновки взаимодействуют со своими дочерними элементами через небольшой набор свойств компоновки, перечисленных в таблице:

Наименование	Описание
HorizontalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по горизонтали. Вы можете выбрать **Center**, **Left**, **Right** или **Stretch**.
VerticalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по вертикали. Вы можете выбрать **Center**, **Top**, **Bottom** или **Stretch**.
Margin	Добавляет немного места вокруг элемента. Свойство

	Margin — это экземпляр структуры **System.Windows.Thickness**, с отдельными компонентами для верхней, нижней, левой и правой граней.
MinWidth и **MinHeight**	Устанавливает минимальные размеры элемента. Если элемент слишком велик, чтобы поместиться в его контейнер компоновки, он будет усечен.
MaxWidth и **MaxHeight**	Устанавливает максимальные размеры элемента. Если контейнер имеет свободное пространство, элемент не будет увеличен сверх указанных пределов, даже если свойства **HorizontalAlignment** и **VerticalAlignment** установлены в **Stretch**.
Width и **Height**	Явно устанавливают размеры элемента. Эта установка переопределяет значение **Stretch** для свойств **HorizontalAlignment** и **VerticalAlignment**. Однако этот размер не будет установлен, если выходит за пределы, заданные в **MinWidth**, **MinHeight**, **MaxWidth** и **MaxHeight**.

Canvas

Canvas позволяет размещать элементы, используя точные координаты, что вообще-то является плохим выбором при проектировании богатых управляемых данными форм и стандартных диалоговых окон, но ценным инструментом, если вам нужно построить нечто другое (вроде поверхности рисования для инструмента построения диаграмм). **Canvas** также является наиболее легковесным из контейнеров компоновки. Это объясняется тем, что он не включает в себя никакой сложной логики компоновки, согласовывающей размерные предпочтения своих дочерних элементов. Вместо этого он просто располагает их в указанных вами позициях с точными размерами, которые вам нужны.

Чтобы позиционировать элемент на **Canvas**, вы устанавливаете прикрепленные свойства **Canvas.Left** и **Canvas.Top**. Свойство **Canvas.Left** задает количество единиц измерения между левой гранью вашего элемента и левой границей **Canvas**. Свойство **Canvas.Top** устанавливает количество единиц измерения между вершиной вашего элемента и левой границей **Canvas**. Эти значения задаются в независимых от устройства единицах измерения, которые соответствуют обычным пикселям, когда системная установка DPI составляет 96 dpi.

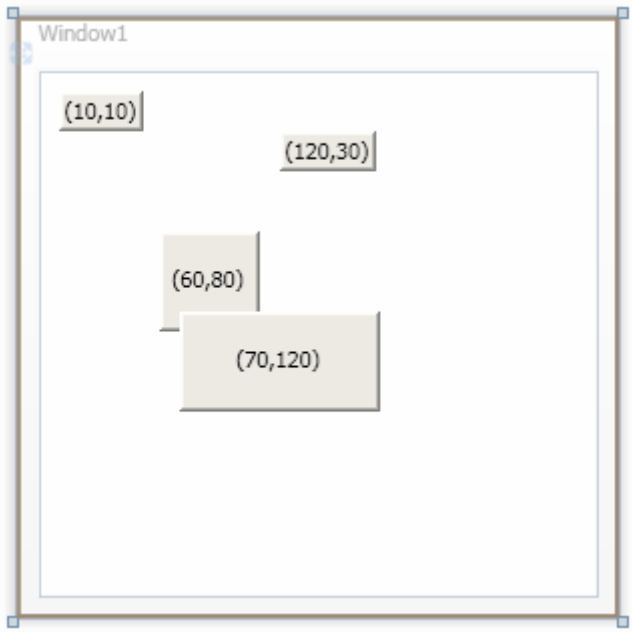
Дополнительно вы можете устанавливать размер вашего элемента явно, используя его свойства **Width** и **Height**. Это чаще применяется при использовании **Canvas**, чем с другими панелями, потому что **Canvas** не имеет собственной логики компоновки.

Если вы не устанавливаете свойства **Width** и **Height**, ваш элемент получит желательный для него размер. Другими словами, он станет достаточно большим, чтобы вместить свое содержимое.

Ниже приведен пример простого **Canvas**, включающего четыре кнопки:

```
<Canvas>
<Button Canvas.Left="10" Canvas.Top="10">(10,10)</Button>
<Button Canvas.Left="120" Canvas.Top="30">(120,30)</Button>
<Button Canvas.Left="60" Canvas.Top="80" Width="50"
Height="50">(60,80)</Button>
<Button Canvas.Left="70" Canvas.Top="120" Width="100"
Height="50">(70,120)</Button>
</Canvas>
```

Вот как выглядит результат:



Если вы измените размеры окна, то **Canvas** растянется для заполнения всего доступного пространства, но ни один из элементов управления на его поверхности не изменит своего положения и размера. **Canvas** не включает никаких средств привязки или стыковки, которые имеются в координатных компоновках Windows Forms.

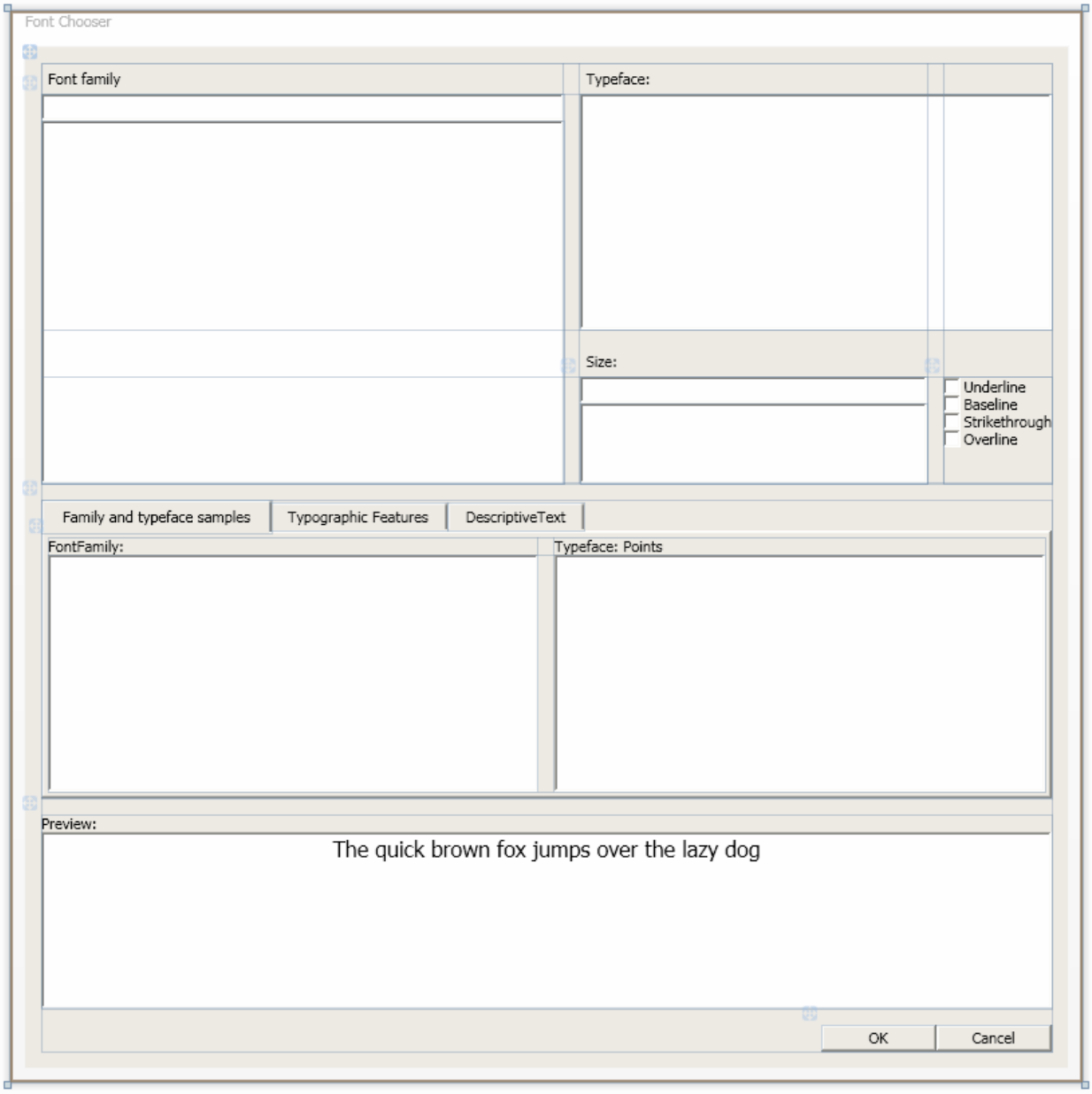
ContentElement

ContentElement – особый класс элементов, конечный контент которых не определен. Поэтому в качестве контента в нем служит его единственный под-элемент.

Например, класс **ContentButton** – реализует элемент «кнопка». Но так как в кнопке может быть текст или рисунок, или то и другое вместе (или еще что-то), то контент его неопределен. **ContentElement** является отправной точкой элементов содержимого, представляющего собой отдельные части содержимого, которое можно помещать в поток документа.

6. Практический пример сложного каркаса приложения с использование разных контейнеров

Вот как может выглядеть диалоговое окно выбора шрифтов в текстовом редакторе:



Рассмотрим его XAML код. Мы видим, что родительским контейнером является <Grid>, разделенный на четыре строки в соотношении 4:3:2:auto. Далее идет описание первой строки родительского <Grid>, в которую вложен дочерний <Grid>, в свою очередь разделенный на пять

столбцов и три строки. Код достаточно документирован, дальнейший анализ кода выполните самостоятельно.

```
<Window x:Class="FontDialogSample.FontChooser"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:app="clr-namespace:FontDialogSample"
Title="Font Chooser"
Width="800" Height="800"
Background="{x:Static SystemColors.ControlBrush}"
>
<Grid Margin="12">
<Grid.RowDefinitions>
<RowDefinition Height="4*"/>
<RowDefinition Height="3*"/>
<RowDefinition Height="2*"/>
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>

<!-- Row 0: Selection controls -->
<Grid Grid.Column="0" Grid.Row="0">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="3*"/>
<ColumnDefinition Width="12"/>
<ColumnDefinition Width="2*"/>
<ColumnDefinition Width="12"/>
<ColumnDefinition Width="Auto"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>

<!-- Families -->
<Label Grid.Column="0" Grid.Row="0" Content="_Font family"/>
<DockPanel Grid.Column="0" Grid.Row="1" Grid.RowSpan="3">
<TextBox DockPanel.Dock="Top" Name="fontFamilyTextBox"/>
```

```
<ListBox Name="fontFamilyList"/>
</DockPanel>

<!-- Family Typefaces (weight, style, stretch) -->
<Label Grid.Column="2" Grid.Row="0" Grid.ColumnSpan="3" Content="_Typeface:"/>
<ListBox        Grid.Column="2"        Grid.Row="1"        Grid.ColumnSpan="3"
Name="typefaceList"/>

<!-- Font sizes -->
<Label Grid.Column="2" Grid.Row="2" Margin="0,12,0,0" Content="_Size:"/>
<DockPanel Grid.Column="2" Grid.Row="3">
<TextBox DockPanel.Dock="Top" Name="sizeTextBox"/>
<ListBox Height="60" Name="sizeList"/>
</DockPanel>

<!-- Text decorations -->
<DockPanel Grid.Column="4" Grid.Row="3">
<CheckBox DockPanel.Dock="Top" Content="Underline" Name="underlineCheckBox"/>
<CheckBox DockPanel.Dock="Top" Content="Baseline" Name="baselineCheckBox"/>
<CheckBox             DockPanel.Dock="Top"             Content="Strikethrough"
Name="strikethroughCheckBox"/>
<CheckBox Content="Overline" Name="overlineCheckBox"/>
</DockPanel>
</Grid>

<!-- Row 1: Tab control with family typeface samples, etc. -->
<TabControl Grid.Column="0" Grid.Row="1" Margin="0,12,0,0" Name="tabControl">
<TabItem Header="Family and typeface samples" Name="samplesTab">
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition/>
<ColumnDefinition Width="12"/>
<ColumnDefinition/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition/>
</Grid.RowDefinitions>

<TextBlock Grid.Column="0" Grid.Row="0">
```

```
<Run>FontFamily: </Run>
<Run Name="fontFamilyNameRun"/>
</TextBlock>
<RichTextBox
Grid.Column="0" Grid.Row="1"
IsReadOnly="True"
VerticalScrollBarVisibility="Auto"
Name="fontFamilySamples"
/>
<TextBlock Grid.Column="2" Grid.Row="0">
<Run>Typeface: </Run>
<Run Name="typefaceNameRun">Points</Run>
</TextBlock>
<RichTextBox
Grid.Column="2" Grid.Row="1"
IsReadOnly="True"
VerticalScrollBarVisibility="Auto"
Name="typefaceSamples"
/>
</Grid>
</TabItem>
<TabItem Header="Typographic Features" Name="typographyTab">
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="8"/>
<ColumnDefinition/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition/>
</Grid.RowDefinitions>

<!-- Feature list of left-hand side of typography tab -->
<Label Grid.Column="0" Grid.Row="0" Content="Feature"/>
<ListBox Grid.Column="0" Grid.Row="1" Name="featureList" MinWidth="192"/>

<!-- Column headers on right-hand side of typography tab -->
<Grid Grid.Column="2" Grid.Row="0">
<Grid.ColumnDefinitions>
```

```
<ColumnDefinition Width="1"/>
<ColumnDefinition Width="96"/>
<ColumnDefinition/>
</Grid.ColumnDefinitions>
<Label Grid.Column="1" Content="Selection"/>
<Label Grid.Column="2" Content="Sample"/>
</Grid>

<!-- Feature page -->
<Border         Grid.Column="2"         Grid.Row="1"         BorderThickness="1"
BorderBrush="{x:Static SystemColors.ControlDarkDarkBrush}">
<ScrollViewer VerticalScrollBarVisibility="Auto">
<Grid Name="typographyFeaturePage">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="96"/>
<ColumnDefinition/>
<!-- The grid contents for each feature are filled in programmatically -->
</Grid.ColumnDefinitions>
</Grid>
</ScrollViewer>
</Border>
</Grid>
</TabItem>
<TabItem Header="DescriptiveText" Name="descriptiveTextTab">
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="3*"/>
<ColumnDefinition Width="8"/>
<ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition/>
</Grid.RowDefinitions>

<Label Grid.Column="0" Grid.Row="0" Content="Font description:"/>
<RichTextBox
Grid.Column="0" Grid.Row="1"
IsReadOnly="True"
VerticalScrollBarVisibility="Auto"
```

```
Name="fontDescriptionBox"
/>

<Label Grid.Column="2" Grid.Row="0" Content="License:"/>
<TextBox
Grid.Column="2" Grid.Row="1"
IsReadOnly="True"
TextWrapping="WrapWithOverflow"
VerticalScrollBarVisibility="Auto"
Name="fontLicenseBox"
/>
</Grid>
</TabItem>
</TabControl>

<!-- Row 2: Preview text -->
<DockPanel Grid.Column="0" Grid.Row="2" Margin="0,12,0,0">
<TextBlock DockPanel.Dock="Top">Preview:</TextBlock>
<TextBox
FontSize="16" AcceptsReturn="True"
TextAlignment="Center"
TextWrapping="Wrap"
VerticalScrollBarVisibility="Auto"
Name="previewTextBox"
>
The quick brown fox jumps over the lazy dog
</TextBox>
</DockPanel>

<!-- Row 3: OK and Cancel buttons -->
<StackPanel     Grid.Column="0"     Grid.Row="3"     Orientation="Horizontal"
HorizontalAlignment="Right" Margin="0,12,0,0">
<Button Content="OK" Width="86" IsDefault="True" Click="OnOKButtonClicked"/>
<Button Content="Cancel" Width="86" Click="OnCancelButtonClicked"/>
</StackPanel>

</Grid>
</Window>
```

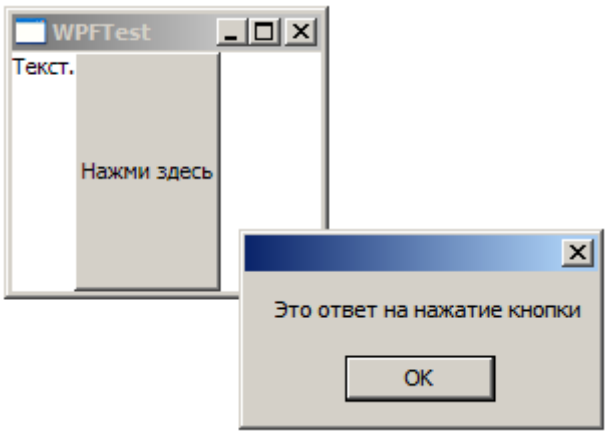
Полностью код приложения находится в папке **Source/fontdialog**.

7. Основы взаимодействия с элементами управления

Использование XAML является основным способом декларативного программирования в среде WPF. Вот небольшой пример кода XAML, который используется для определения окна с текстом и кнопкой, визуально «прилепленными» к друг другу:

```
<Window x:Class="WPFTest.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPFTest" Height="440" Width="515"
    >
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="Текст."/>
        <Button Name="TestButton" Content="Нажми здесь" Click="TestButtonClick"/>
    </StackPanel>

</Window>
```



На рисунке показаны основные элементы этого простого примера. Весь документ занимает тэг Window, который декларативно определяет окно, его свойства и содержимое. **Атрибут (1)** задает имя так называемого **Code-behind** класса, то есть класса, который будет содержать процедурную программную логику для данной разметки. Понятие **Code-behind** классов позаимствовано из ASP.NET, где оно имеет практически ту же семантику и широко используется. **Code-behind** класс для данной разметки мы рассмотрим здесь чуть позже.

Атрибуты (2) содержат описания того, какие XML-пространства имен используются в данном документе. Те два пространства имен, которые мы видим на рисунке, используются в WPF практически всегда; их использование учтено в шаблонах для создания соответствующих элементов проектов Visual Studio. Первое пространство имен содержит описание стандартных элементов WPF и используется по-умолчанию, второе – описание специальных расширений XAML и по соглашению имеет псевдоним x. Также XAML предлагает специальный синтаксис для подключения пространств имен .NET; это выглядит следующим образом:

```
<Window x:Class="WPFTest.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:WPFTest="clr-namespace:WPFTest"
    xmlns:CoolControls="clr-namespace:ControlSprings.CoolControls; assembly=CoolControlsLibrary"
    Title="WPFTest" Height="440" Width="515"
    >
    <Window.Resources>
        <ObjectDataProvider ObjectType="{x:Type WPFTest:MyCustomDataClass}"/>
    </Window.Resources>

    <StackPanel Orientation="Horizontal">
        <CoolControls:GroovyButton Text="Groovy!"/>
    </StackPanel>

</Window>
```

Как видно, для того, чтобы подключить CLR-пространство имен к вашему документу, необходимо назначить ему псевдоним, указать собственно его имя и имя сборки. Для того чтобы это работало, ссылка на эту сборку должна быть добавлена в проект. Для того чтобы добавить ссылку на пространство имен из текущей сборки, нужно просто опустить часть «;assembly=XXX».

Атрибуты (3) содержат описание некоторых свойств окна; здесь это заголовок, ширина и высота. Здесь все довольно просто.

Далее находятся тэги, которые описывают содержимое окна. Непосредственным логическим «ребенком» окна является панель типа **StackPanel (4),** которая используется для обеспечения

последовательного горизонтального (как в данном случае) или вертикального размещения дочерних элементов.

В эту панель вложены два стандартных элемента управления: **текстовый блок (5)** и **кнопка (6)**. Описание кнопки задает ее имя (которое будет доступно как имя члена класса типа **Button** из **Code-behind класса**) и подразумевает наличие обработчика события **Click (7)** также в этом процедурно описанном классе.

Прежде чем мы перейдем к рассмотрению code-behind класса, можно задавать имена только для тех элементов, для которых они нужны (как в случае с кнопкой). Известно, что в Windows Forms ситуация сложилась по другому – даже для самого незначительных сплиттера и панели нужно задавать имена и чаще всего они имеют вид вроде splitter1. Здесь же легко применить всегда полезный принцип – если надо что-то назвать – назови, а если не надо – то и не беспокойся.

```
using System.Windows;

namespace WPFTest
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void TestButtonClick(object sender, RoutedEventArgs e)
        {
            MessageBox.Show(this, "Это ответ на нажатие кнопки");
        }
    }
}
```

На рисунке выше показан вид **code-behind** класса для описанной разметки. Выглядит очень похоже на описание такого же класса для Windows Forms. А теперь давайте перейдем к анализу конкретных элементов управления.

8. Статический текст. Класс Label

В WPF элемент управления обычно описывается как элемент, который может получать фокус и принимать данные, вводимые пользователем — в качестве примера можно привести текстовое поле или кнопку. **Label** считается элементом управления, поскольку он поддерживает мнемонические команды (клавиши быстрого доступа, передающие фокус связанным элементам управления).

Элементы управления содержимым (content control) — это специализированный тип элементов управления, которые могут хранить (и отображать) какую-то порцию содержимого. С технической точки зрения элемент управления содержимым является элементом управления, который может включать один вложенный элемент. Этим он отличается от контейнера компоновки, который может хранить сколь угодно много вложенных элементов.

Как было сказано выше, все контейнеры компоновки WPF являются наследниками класса **Panel**, что позволяет им хранить множество элементов. Точно так же, все элементы управления содержимым являются наследниками абстрактного класса **ContentControl**. Иерархия классов показана ниже:

System.Object
  System.Windows.Threading.DispatcherObject
    System.Windows.DependencyObject
      System.Windows.Media.Visual
        System.Windows.UIElement
          System.Windows.FrameworkElement
            System.Windows.Controls.Control
              System.Windows.Controls.ContentControl
                System.Windows.Controls.Frame
                System.Windows.Controls.GroupItem
                System.Windows.Controls.HeaderedContentControl
                System.Windows.Controls.Label
                System.Windows.Controls.ListBoxItem
                System.Windows.Controls.Primitives.ButtonBase
                System.Windows.Controls.Primitives.StatusBarItem
                System.Windows.Controls.ScrollViewer
                System.Windows.Controls.ToolTip
                System.Windows.Controls.UserControl
                System.Windows.Window

Простейшим элементом управления содержимым является **Label** — метка. Как и любой другой элемент управления содержимым, она принимает одиночную порцию содержимого, которую вы хотите поместить внутри нее. Отличительной чертой элемента **Label** является его поддержка мнемонических команд— клавиш быстрого доступа, которые передают фокус связанному элементу управления.

Для обеспечения поддержки этой функции элемент управления Label предлагает свойство **Target**. Чтобы задать это свойство, вам необходимо воспользоваться выражением привязки, которое будет указывать на другой элемент управления. Ниже показан синтаксис, который нужно использовать для этой цели:

```
<Label Target="{Binding ElementName=txtA}">Choose _A</Label>
<TextBox Name="txtA"></TextBox>
<Label Target="{Binding ElementName=txtB}">Choose _B</Label>
<TextBox Name="txtB"></TextBox>
```

Символ подчеркивания в тексте метки указывает на клавишу быстрого доступа. (Если вы действительно хотите, чтобы в метке отображался символ подчеркивания, нужно добавить два таких символа.) Все мнемонические команды работают при одновременном нажатии клавиши **<Alt>** и заданной вами клавиши быстрого доступа.

Например, если в данном примере пользователь нажмет комбинацию **<Alt+A>**, то первая метка передаст фокус связанному элементу управления, которым в данном случае является **txtA**. Точно так же нажатие комбинации **<Alt+B>** приводит к передаче фокуса элементу управления txtB.

9. Кнопки

Button

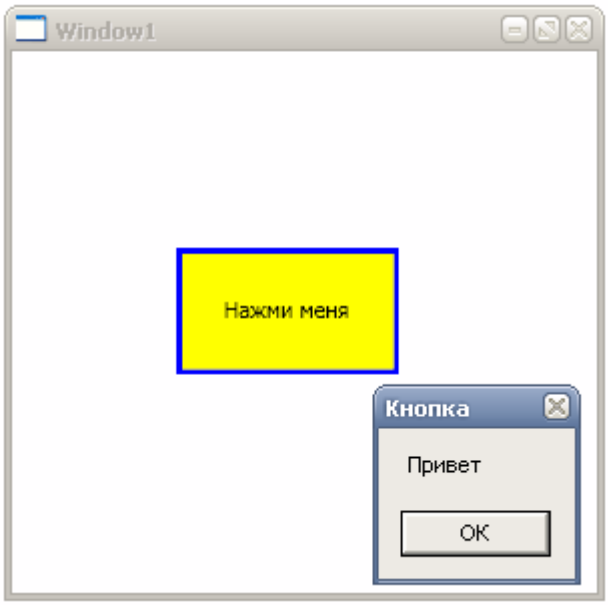
Элемент управления **Button** реагирует на пользовательский ввод с мыши, клавиатуры, пера или другого устройства ввода и инициирует событие **Click**. Элемент управления **Button** является основным компонентом пользовательского интерфейса, который может содержать как простое содержимое, например, текст, так и сложное, например, изображения и элементы управления **Panel**. Иерархия наследования представлена ниже:

System.Object
  System.Windows.Threading.DispatcherObject
    System.Windows.DependencyObject
      System.Windows.Media.Visual
        System.Windows.UIElement
          System.Windows.FrameworkElement
            System.Windows.Controls.Control
              System.Windows.Controls.ContentControl
                System.Windows.Controls.Primitives.ButtonBase
                  System.Windows.Controls.Button

Ниже представлен код XAML, создающий кнопку и в качестве реакции на ее нажатие выводящий диалоговое окно:

```
<Button Name="btn1" Background="Yellow"
BorderBrush="Blue" BorderThickness="1"
Click="OnClick1" Margin="82,98,99,109">
Нажми меня
</Button>
```

Внешний вид примера приведен ниже:



Полностью код приложения находится в папке **Source/button**.

CheckBox и RadioButton

Кнопки **CheckBox** и **RadioButton** — это кнопки другого вида. Они являются потомками класса **ToggleButton**:

System.Object
System.Windows.Threading.DispatcherObject
System.Windows.DependencyObject
System.Windows.Media.Visual
System.Windows.UIElement
System.Windows.FrameworkElement
System.Windows.Controls.Control
System.Windows.Controls.ContentControl
System.Windows.Controls.Primitives.ButtonBase
System.Windows.Controls.Primitives.ToggleButton
System.Windows.Controls.CheckBox
System.Windows.Controls.RadioButton

Это означает, что пользователь может включать и выключать их (отсюда и наличие слова **toggle**). В случае **CheckBox** включение элемента управления означает отметку в нем флажка. Класс **CheckBox** не добавляет никаких членов, поэтому базовый интерфейс **CheckBox** определяется в классе **ToggleButton**. Более того, **ToggleButton** добавляет свойство **IsChecked**. Свойство **IsChecked** может принимать обнуляемое булевское значение — другими словами, оно может

принимать значения **true**, **false** или **null**. Очевидно, что **true** представляет отмеченный флажок, а **false**— пустое место. Значение null используется для представления промежуточного состояния, которое отображается в виде затененного окошка. Промежуточное состояние обычно служит для того, чтобы представить значения, которые не были заданы, или области, в которых существует некоторые разногласия. Например, если у вас имеется флажок, который позволяет применять полужирный шрифт в текстовом приложении, а текущий выбор включает как полужирный, так и обычный текст, вы можете присвоить флажку значение **null**, чтобы отображать промежуточное состояние.

Чтобы присвоить значение **null** в разметке WPF, нужно использовать расширение разметки **Null**, как показано ниже:

```
<CheckBox IsChecked="{x:Null}">A check box in indeterminate state</CheckBox>
```

Наряду со свойством **IsChecked** класс **ToggleButton** добавляет свойство **IsThreeState**, которое определяет, может ли пользователь вводить флажок в промежуточное состояние. Если свойство **IsThreeState** будет иметь значение **false** (оно присваивается по умолчанию), то при щелчке флажок будет менять свое состояние между "отмечен" и "не отмечен", а промежуточное состояние можно задать только с помощью кода. Если свойство **Threestate** будет иметь значение **true**, то щелчки на флажке будут по очереди давать три возможных состояния.

Класс **ToggleButton** определяет также три события, которые возникают, когда флажок принимает одно из определенных состояний: **Checked**, **Unchecked** и **Intermediate**.

В большинстве случаев проще всего внедрить эту логику в один из обработчиков событий, обрабатывая событие **Click**, наследуемое от класса **ButtonBase**. Событие **Click** возникает всякий раз, когда кнопка меняет свое состояние.

RadioButton тоже является наследником класса **ToggleButton** и использует то же свойство **IsChecked** и те же события **Checked**.

Unchecked и **Intermediate**. Вместе с ними **RadioButton** добавляет свойство **GroupName**, которое позволяет управлять расположением переключателей в группах.

Обычно переключатели группируются их контейнером. Это означает, что если вы поместите три элемента управления **RadioButton** в панели **StackPanel**, они сформируют группу, из которой вы сможете выбрать только один из них. С другой стороны, если вы поместите комбинацию переключателей в две разных панели **StackPanel**, вы получите две независимые группы.

Свойство **GroupName** позволяет переопределить это поведение. Вы можете использовать его для того, чтобы создать несколько групп в одном и том же контейнере, или же чтобы создать одну группу, которая будет охватывать множество контейнеров. В любом случае нужно просто присвоить всем переключателям, принадлежащим друг другу, имя одной и той же группы. Рассмотрим пример:

```
<StackPanel>
<GroupBox Margin="5">
<StackPanel>
<RadioButton>Group 1</RadioButton>
<RadioButton>Group 1</RadioButton>
<RadioButton>Group 1</RadioButton>
<RadioButton Margin="0,10, 0, 0" GroupName="Group2">Group 2</RadioButton>
</StackPanel>
</GroupBox>
<GroupBox Margin="5">
<StackPanel>
<RadioButton>Group 3</RadioButton>
<RadioButton>Group 3</RadioButton>
<RadioButton>Group 3</RadioButton>
<RadioButton Margin="0,10, 0, 0" GroupName="Group2">Group 2</RadioButton>
</StackPanel>
</GroupBox>
</StackPanel>
```

Здесь мы видим два контейнера, вмещающих переключатели, и три группы. Последний переключатель внизу каждого группового окна является частью третьей группы. Для упаковки переключателей нет необходимости применять контейнер **GroupBox**, хотя, как правило, именно он и используется. Этот контейнер отображает рамку и надпись, которую можно применить к группе кнопок.

10. Текстовые поля TextBox, PasswordBox

Тестовые элементы управления могут заключать в себе только ограниченный тип содержимого. **TextBox** всегда хранит строку (она определяется в свойстве **Text**). **PasswordBox** тоже содержит строку текста (она определяется в свойстве **Password**), однако использует **SecureStnng** для защиты от некоторых типов атак.

Обычно, элемент управления **TextBox** хранит одну строку текста. (Вы можете ограничить допустимое количество символов с помощью свойства **MaxLength**.) Однако часто бывают случаи, когда приходится создавать многострочное текстовое окно для работы с большим объемом

содержимого. Для этой цели нужно воспользоваться свойством **TextWrapping**, присвоив ему значение **Wrap** или **WrapWithOverflow**. При значении **Wrap** текст будет всегда обрываться на краю элемента управления, даже если при этом слишком большое слово будет разбито на два. **WrapWithOverflow** позволяет растянуть несколько строк за границы правого края, если алгоритм разрыва строки не может найти подходящее место (например, пробел или дефис) для разбиения строки.

Чтобы на самом деле увидеть в текстовом окне множество строк, оно должно иметь подходящие размеры. Вместо того чтобы задавать жестко кодированную высоту (которая не подойдет для разных размеров шрифтов и может вызвать проблемы с компоновкой), вы можете использовать свойства **MinLines** и **MaxLines**. Свойство **MinLines** определяет минимальное количество строк, которые должны отображаться в текстовом окне. Например, если этому свойству присвоить значение 2, то текстовое окно примет высоту, равную высоте минимум двух строк текста. **MaxLines** задает максимальное количество отображаемых строк. Даже если текстовое окно будет расширено до таких размеров, чтобы уместиться в своем контейнере (например, строка **Grid** с пропорциональными размерами или последний элемент в **DockPanel**), оно не превысит заданный лимит.

Свойства **MinLines** и **MaxLines** не влияют на количество содержимого, которое вы можете поместить в текстовом окне. Они просто помогают придать подходящие размеры текстовому окну. В своем коде вы можете проверить свойство **LineCount**, чтобы узнать точно, сколько строк умещается в текстовом окне.

Если ваше текстовое окно поддерживает ввод текста, то нужно позаботиться о том, чтобы пользователь мог вводить больше текста, чем может быть отображено в видимых строках. По этой причине обычно имеет смысл добавить постоянно отображаемую линейку прокрутки (или отображаемую по запросу), присвоив свойству **VerticalScrollBarVisibility** значение **Visible** или **Auto**. (Вы можете

также задать свойство **HorizontalScrollBarVisibility**, чтобы отображать реже используемую горизонтальную полосу прокрутки.)

Иногда приходится делать так, чтобы пользователь мог вводить жесткие возвраты в многострочном текстовом окне, нажимая клавишу **<Enter>**. (Обычно при нажатии клавиши <**Enter**> в текстовом окне активизируется кнопка, используемая по умолчанию.) Чтобы текстовое окно поддерживало клавишу **<Enter>**, присвойте свойству **AcceptsReturn** значение **true**. Можно также задать свойство **AcceptsTab**, чтобы пользователь мог вставлять символы табуляции. В противном случае при нажатии клавиши **<Tab>** будет передан фокус следующему элементу управления, заданному в последовательности перехода с помощью клавиши табуляции.

Класс **TextBox** включает также набор методов, которые позволяют программно перемещаться по текстовому содержимому небольшими или крупными шагами. К этим методам относятся **LineUp()**, **LineDown(**), **PageUp()**, **PageDown()**, **ScrollToHome()**, **ScrollToEnd()** и **ScrollToLine()**.

Иногда необходимо создавать текстовые окна исключительно для отображения текста. В этом случае свойству **IsReadOnly** потребуется присвоить значение true, чтобы исключить возможность редактирования текста в поле. Этот подход предпочтительнее блокирования текстового окна путем присваивания свойству **IsEnabled** значения **fals**e, поскольку заблокированное текстовое окно отображает текст, выделенный серым цветом (такой текст трудно читать), не поддерживает выделение текста (или копирование в буфер обмена) и его прокрутку.

Известно, что можно выделять текст в любом текстовом окне, щелкая кнопкой мыши и перемещая ее указатель, или, удерживая нажатой клавишу <**Shift**>, выделять его с помощью клавиш управления курсором. Класс **TextBox** дает возможность определять или изменять выделенный в данный момент текст программным образом, используя свойства **SelectionStart**, **SelectionLength** и **SelectedText**. Свойство

SelectionStart определяет позицию, начиная с нуля, в которой будет осуществляться выделение текста. Например, если этому свойству присвоить значение 10, то первым выделенным символом будет одиннадцатый символ в текстовом окне. Свойство **SelectionLength** задает общее количество выделенных символов. (Значение, равное нулю, свидетельствует о том, что не было выделено ни одного символа.) Свойство **SelectedText** позволяет быстро проверить или изменить выделенный текст в текстовом окне. Вы можете отреагировать на изменение выделения с помощью события **SelectionChanged**.

Класс **TextBox** включает свойство **AutoWordSelection**, позволяющее управлять поведением выделения. Если ему присвоить значение true, то в текстовом окне будет выделяться по одному слову одновременно.

Элемент управления **TextBox** обладает еще несколькими специализированными возможностями. Наиболее интересной является проверка орфографии, при которой нераспознанные слова подчеркиваются волнистой линией красного цвета. Пользователь может щелкнуть правой кнопкой мыши на нераспознанном слове и выбрать из списка правильный вариант. Чтобы добавить функцию проверки орфографии в элемент управления **TextBox**, вам нужно задать свойство зависимостей **SpellCheck.IsEnabled**, как показано ниже:

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

Функция проверки орфографии является специфической для WPF и не зависит от любого другого программного обеспечения (например, Office). Функция проверки орфографии определяет необходимый словарь на основании выбранного пользователем языка ввода на клавиатуре. Выбрать словарь можно с помощью свойства **Language** элемента управления **TextBox**, которое происходит от класса **FrameworkElement**, или посредством задания атрибута **xml:lang** в элементе **<TextBox**>.

Другой полезной функцией является **Undo**, которая позволяет

отменить последние изменения. Функцию **Undo** можно реализовать программно (посредством метода **Undo()**), с помощью комбинации клавиш <**Ctrl+Z**>, а также при условии, что свойство **CanUndo** не будет иметь значение **False**.

Программно манипулируя текстом в текстовом окне, вы можете использовать методы **BeginChange()** и **EndChange()**, чтобы сгруппировать серию действий, которые **TextBox** обработает как один "блок" изменений. Впоследствии эти действия можно будет отменить за один раз.

Элемент управления **PasswordBox** выглядит подобно элементу управления **TextBox**, однако он отображает строку, содержащую символы-кружочки, скрывающие собой настоящие символы. (С помощью свойства **PasswordChar** можно выбрать другую маску символов.) Кроме того, **PasswordBox** не поддерживает работу с буфером обмена, поэтому вы не сможете скопировать текст, который содержится в этом элементе управления.

По сравнению с **TextBox**, класс **PasswordBox** имеет более простой интерфейс. Как и класс **TextBox**, он предлагает свойство **MaxLength**, методы **Clear()**, **Paste()** и **SelectAll()**, а также событие **PasswordChanged**, которое возникает в случае изменения текста. Главное отличие этого элемента управления от **TextBox** кроется внутри. Несмотря на то что вы можете задать текст и прочитать его как обычную строку с помощью свойства **Password**, внутренне элемент управления **PasswordBox** использует исключительно объект **System.Security.SecureString**. **SecureString** — это исключительно текстовый объект, подобный обычной строке. Разница заключается в способе его хранения в памяти. **SecureString** хранится в памяти в зашифрованном виде. Ключ, который используется для расшифровки строки, генерируется псевдослучайным образом и хранится в порции памяти, которая никогда не записывается на диск. Поэтому даже если произойдет поломка вашего компьютера, злоумышленник не сможет

проверить страничный файл, чтобы извлечь данные пароля. В самом лучшем случае он найдет всего лишь зашифрованную форму.

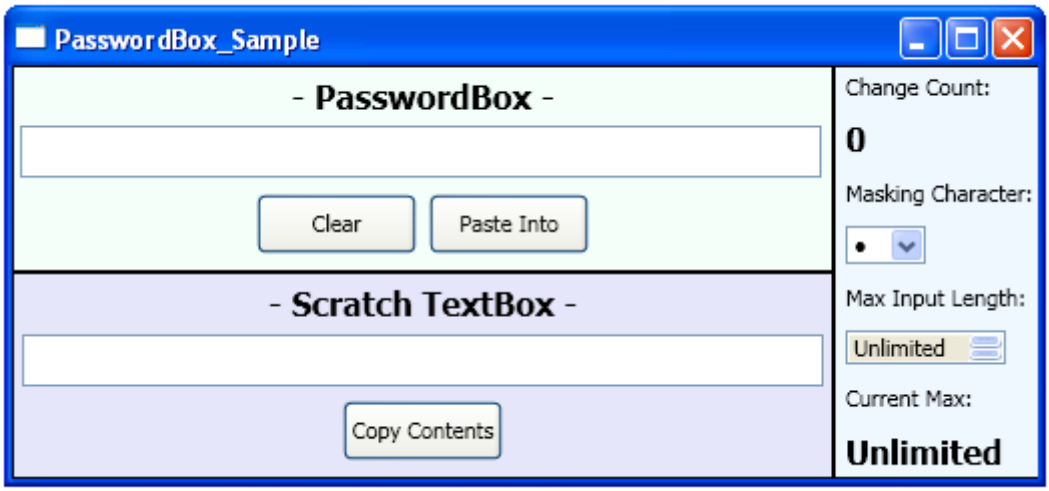
Класс **SecuteString** также имеет средство освобождения по запросу. Когда вы вызываете метод **SecureString.Dispose()**, данные пароля, находящиеся в памяти, перезаписываются. Это дает гарантию, что вся информация о пароле будет стерта из памяти, и никто не сможет ею воспользоваться. **PasswordBox** вызывает метод **Dispose()** для хранимой внутри строки **SecureString** при уничтожении элемента управления.

11. Примеры использования элементов управления

В примере **Source/passwordbox** демонстрируются основные возможности и функциональность класса PasswordBox, такие как:

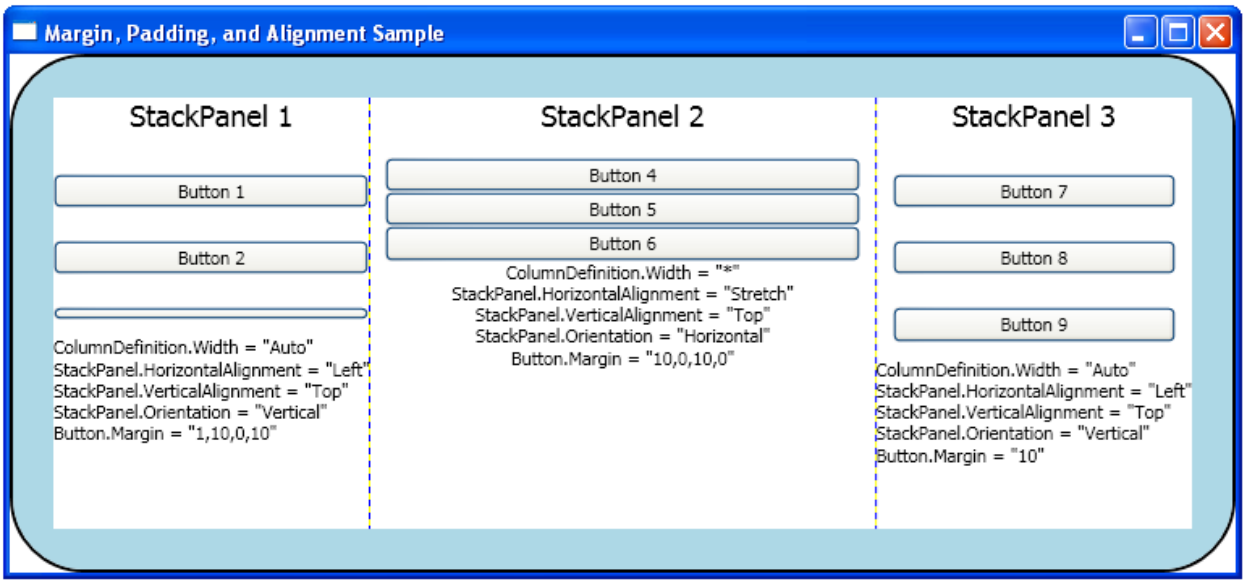
- Выбор символа маски из предопределенного списка символов.
- Задание максимальной длины пароля от 6 до 255 знаков, или указание отсутствия ограничения.
- Ввод текста в рабочий элемент **TextBox** и использование кнопки **Копировать**, чтобы поместить текст в системный буфер обмена.
- Использование кнопки **Вставит**ь в для вставки в элемент **PasswordBox** с помощью метода **Paste**.
- Очистка содержимого **PasswordBox** путем нажатия кнопки **Очистить** (что вызывает метод **Clear**).
- Просмотр числа изменений пароля, содержащегося в элементе **PasswordBo**x (соответствует числу возникновений события **PasswordChanged**).

Внешний вид приложения выглядит следующим образом:



XAML код примера подробно откомментирован.

В примере **Source/marginpaddingalignment** демонстрируется использование свойств макета для точного размещения дочерних элементов. Этот пример интересен тем, что элементы управления создаются программно, а не декларативно (на XAML). Код примера подробно откомментирован. Внешний вид приложения выглядит следующим образом:



12. Домашнее задание

1. Создать приложение – калькулятор. Внешний вид калькулятора может быть похож на стандартный калькулятор Windows, или иметь авторский ☺ дизайн.