



**УФИМСКИЙ ГОСУДАРСТВЕННЫЙ
НЕФТЯНОЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

Визуальное программирование

Чиганова Наталья Викторовна

к.ф.-м.н., доцент кафедры
«Цифровые технологии и
моделирование»



5 и 6 семестр

5 семестр

18 ч. лекций

30 ч. лабораторные занятия

Зачет

Лекция 1. Введение.

Технология WPF

Визуальное программирование (от лат. visualis - зрительный) - это технология программирования, предусматривающая создание приложений с помощью наглядных средств.

Средствами визуального программирования обычно решают задачи построения пользовательского интерфейса и упрощения разработки приложения путем замены метода "написания программы" на метод **конструирования**.

Визуальное программирование, бесспорно, обладает достоинством **наглядного представления информации**.

Однако практически все визуальные средства нуждаются в дополнении функциями, которые не могут быть представлены в виде графических конструкций и требуют текстового выражения. Визуальные средства дополняются специальными программами - "скриптами", написанными на различных языках программирования.

Концепция визуального программирования реализована во многих современных средах разработки программных систем. Все ведущие фирмы, создающие средства для программирования и конструирования имеют системы, поддерживающие технологию визуального программирования.

Фирма Microsoft, разрабатывая концепцию .NET Framework, создала Visual Studio.NET Enterprise Architect, в которой реализовала все последние достижения в области программирования и в частности, в технологии визуального программирования.

Visual Studio.NET - полная многоязычная среда разработки для платформы Microsoft.NET. Visual Studio.NET предоставляет набор технологий, упрощающих создание, развертывание и последующее усовершенствование безопасных, масштабируемых и высокодоступных веб-приложений и веб-служб XML.

Технология WPF (Windows Presentation Foundation)

Приложения WPF основаны на **DirectX**.

Ключевая особенность рендеринга графики в WPF: используя WPF, значительная часть работы по отрисовке графики, как простейших кнопочек, так и сложных 3D-моделей, ложится на графический процессор на видеокарте, что также позволяет воспользоваться аппаратным ускорением графики.

Одной из важных особенностей является использование языка декларативной разметки интерфейса XAML, основанного на XML: вы можете создавать насыщенный графический интерфейс, используя или декларативное объявление интерфейса, или код на управляемых языках C#, либо совмещать и то, и другое.

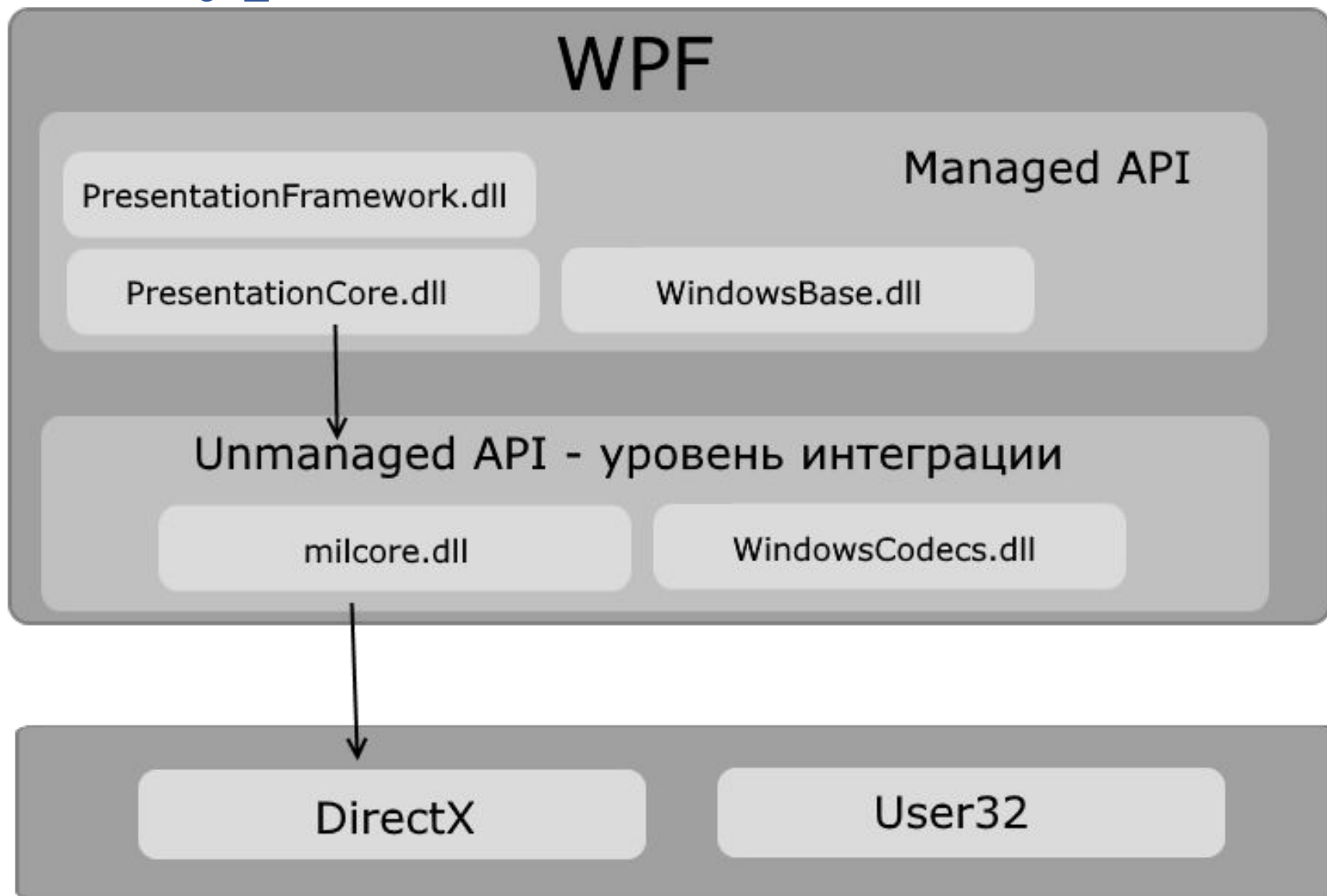
Преимущества WPF

- Использование традиционных языков .NET-платформы - C# и VB.NET для создания логики приложения
- Возможность **декларативного определения** графического интерфейса с помощью специального языка разметки XAML, основанном на xml и представляющем альтернативу программному созданию графики и элементов управления, а также возможность комбинировать XAML и C#/VB.NET
- **Независимость от разрешения экрана:** поскольку в WPF все элементы измеряются в независимых от устройства единицах, приложения на WPF легко масштабируются под разные экраны с разным разрешением.
- Новые возможности - создание трехмерных моделей, привязка данных, использование таких элементов, как стили, шаблоны, темы и др.
- Хорошее **взаимодействие с WinForms**, благодаря чему, например, в приложениях WPF можно использовать традиционные элементы управления из WinForms.
- **Богатые возможности** по созданию различных приложений: это и мультимедиа, и двухмерная и трехмерная графика, и богатый набор встроенных элементов управления, а также возможность самим создавать новые элементы, создание анимаций, привязка данных, стили, шаблоны, темы и многое другое
- **Аппаратное ускорение графики** - вне зависимости от того, работаете ли вы с 2D или 3D, графикой или текстом, все компоненты приложения транслируются в объекты, понятные Direct3D, и затем визуализируются с помощью процессора на видеокарте, что повышает производительность, делает графику более плавной.
- Создание приложений под множество ОС семейства Windows - от Windows XP до Windows 10

WPF имеет определенные **ограничения**

- Для создания приложений с большим количеством трехмерных изображений, прежде всего игр, лучше использовать другие средства - DirectX или специальные фреймворки, такие как Unity.
- Также стоит учитывать, что по сравнению с приложениями на Windows Forms объем программ на WPF и потребление ими памяти в процессе работы в среднем несколько выше.

Архитектура WPF



WPF разбивается на два уровня: managed API и unmanaged API (уровень интеграции с DirectX).

Managed API (управляемый API-интерфейс) содержит код, исполняемый под управлением общеязыковой среды выполнения .NET - Common Language Runtime.

Этот API описывает основной функционал платформы WPF и состоит из следующих компонентов:

- **PresentationFramework.dll**: содержит все основные реализации компонентов и элементов управления, которые можно использовать при построении графического интерфейса
- **PresentationCore.dll**: содержит все базовые типы для большинства классов из PresentationFramework.dll
- **WindowsBase.dll**: содержит ряд вспомогательных классов, которые применяются в WPF, но могут также использоваться и вне данной платформы

Unmanaged API используется для интеграции вышележащего уровня с DirectX:

- **milcore.dll**: собственно обеспечивает интеграцию компонентов WPF с DirectX. Данный компонент написан на неуправляемом коде (C/C++) для взаимодействия с DirectX.
- **WindowsCodecs.dll**: библиотека, которая предоставляет низкоуровневую поддержку для изображений в WPF.

Создание проекта

В меню **File** (**Файл**) выберем пункт **New** (**Создать**) -> **Project...** (**Проект...**). Перед нами откроется диалоговое окно создания проекта, в котором мы выберем шаблон **WPF Application**.

По умолчанию студия открывает создает и открывает нам два файла:

- файл декларативной разметки интерфейса *MainWindow.xaml*
- файл связанного с разметкой кода *MainWindow.xaml.cs*.

Файл *MainWindow.xaml* имеет два представления: визуальное - в режиме **WYSIWIG** отображает весь графический интерфейс данного окна приложения, и под ним декларативное объявление интерфейса в **XAML**.

XAML (eXtensible Application Markup Language)

- язык разметки, используемый для инициализации объектов в технологиях на платформе .NET. Применительно к WPF данный язык используется прежде всего для создания пользовательского интерфейса декларативным путем.

Использование XAML все-таки несет некоторые преимущества:

- Возможность отделить графический интерфейс от логики приложения, благодаря чему над разными частями приложения могут относительно автономно работать разные специалисты: над интерфейсом - дизайнеры, над кодом логики - программисты.
- Компактность, понятность, код на XAML относительно легко поддерживать.

При компиляции приложения в Visual Studio код в хaml-файлах также компилируется в бинарное представление кода хaml, которое называется BAML (Binary Application Markup Language).

Затем код baml встраивается в финальную сборку приложения - exe или dll-файл.

Структура и пространства имен XAML

Файл *MainWindow.xaml* будет иметь следующую разметку:

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

        xmlns:local="clr-namespace:XamlApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Пространства имен XAML

- пространство имен **<http://schemas.microsoft.com/winfx/2006/xaml/presentation>** базовое пространство имен, которое охватывает все классы WPF, включая элементы управления, которые применяются при построении пользовательского интерфейса. Так как данное пространство имен объявлено без префикса, то оно распространяется на весь XAML-документ
- **<http://schemas.microsoft.com/winfx/2006/xaml>** - это пространство имен XAML. Оно включает различные свойства утилит XAML, которые позволяют влиять на то, как XAML-документ следует интерпретировать.

Используемый префикс **x** в определении `xmlns:x` означает, что те свойства элементов, которые заключены в этом пространстве имен, будут использоваться с префиксом **x** - `x:Name` или `x:Key`. Это же пространство имен используется уже в первой строчке `x:Class="XamlApp.MainWindow"` - здесь создается новый класс `MainWindow` и соответствующий ему файл кода, куда будет прописываться логика для данного окна приложения.

Эти пространства имен не эквивалентны тем пространствам имен, которые подключаются при помощи директивы `using` в `c#`.

Второе *пространство* имен используется для включения специфичных для *XAML* лексем – «ключевых слов»

Таблица. Ключевые слова XAML

Ключевое слово	Назначение
x:Array	Представляет <i>тип массива .NET</i> на <i>XAML</i>
x:ClassModifier	Позволяет определять видимость типа класс (<i>internal</i> или <i>public</i>), обозначенного ключевым словом <i>Class</i>
x:FieldModifier	Позволяет определять видимость члена типа (<i>internal</i> , <i>public</i> , <i>private</i> или <i>protected</i>) для любого именованного элемента корня. Именованный элемент определяется с использованием ключевого слова <i>Name</i>
x:Key	Позволяет установить значение ключа для элемента <i>XAML</i> , которое должно быть помещено в элемент словаря
x>Name	Позволяет указывать сгенерированное <i>C#</i> имя заданного элемента <i>XAML</i>
x:Null	Представляет <i>null-ссылку</i>
x:Static	Позволяет ссылаться на статический член типа
x:Type	<i>XAML</i> -эквивалент операции <i>C# typeof</i> (вызывает <i>System.Type</i> на основе указанного имени)
x:TypeArgument	Позволяет устанавливать элемент как обобщенный тип с определенными параметрами

Элементы и их атрибуты

XAML предлагает очень простую и ясную схему определения различных элементов и их свойств. Каждый элемент, как и любой элемент XML, должен иметь открытый и закрытый тег, как в случае с элементом Window:

`<Window></Window>` или сокращенную форму `<Window />`.

Каждый элемент в XAML соответствует определенному классу C#.

Например, элемент `Button` соответствует классу `System.Windows.Controls.Button`.

А свойства этого класса соответствуют атрибутам элемента `Button`.

Создание кнопки

```
<Window x:Class="XamlApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:XamlApp"
  mc:Ignorable="d"
  Title="MainWindow" Height="350" Width="525">
  <Grid x:Name="grid1">
    <Button x:Name="button1" Width="100" Height="30" Content="Кнопка" />
  </Grid>
</Window>
```

Специальные символы

При определении интерфейса в XAML мы можем столкнуться с некоторыми ограничениями. В частности, мы не можем использовать специальные символы, такие как знак амперсанда &, кавычки " и угловые скобки < и >.

Например, мы хотим, чтобы текст кнопки был следующим: <"Hello">.

У кнопки есть свойство Content, которое задает содержимое кнопки. И можно предположить, что нам надо написать так:

```
<Button Content="<"Hello">" />
```

Но такой вариант ошибочен и даже не скомпилируется. В этом случае нам надо использовать специальные коды символов:

Символ	Код
--------	-----

<	<
---	------

>	>
---	------

&	&
---	-------

"	"
---	--------

Например: <Button Content="<"Hello">" />

Еще одна проблема, с которой мы можем столкнуться в XAML - добавление пробелов. Возьмем, к примеру, следующее определение кнопки:

```
<Button>
```

```
    Hello    World
```

```
</Button>
```

Здесь свойство **Content** задается неявно в виде содержимого между тегами `<Button>...</Button>`. Но несмотря на то, что у нас несколько пробелов между словами "Hello" и "World", XAML по умолчанию будет убирать все эти пробелы. И чтобы сохранить пробелы, нам надо использовать атрибут **`xml:space="preserve"`**:

```
<Button xml:space="preserve">
```

```
    Hello    World
```

```
</Button>
```

Файлы отделенного кода

При создании нового проекта WPF в дополнение к создаваемому файлу *MainWindow.xaml* создается также файл отделенного кода *MainWindow.xaml.cs*, где, как предполагается, должна находиться логика приложения связанная с разметкой из *MainWindow.xaml*.

Файлы XAML позволяют нам определить интерфейс окна, но для создания логики приложения, например, для определения обработчиков событий элементов управления, нам все равно придется воспользоваться кодом C#.

По умолчанию в разметке окна используется атрибут **x:Class**:

```
<Window x:Class="XamlApp.MainWindow"
```

Атрибут **x:Class** указывает на класс, который будет представлять данное окно и в который будет компилироваться код в XAML при компиляции, т.е. во время компиляции будет генерироваться класс **XamlApp.MainWindow**, унаследованный от класса **System.Windows.Window**.

Кроме того в файле отделенного кода MainWindow.xaml.cs, который Visual Studio создает автоматически, мы также можем найти класс с тем же именем - в данном случае класс XamlApp.MainWindow. По умолчанию он имеет некоторый код:

```
1using System;
2using System.Collections.Generic;
3using System.Linq;
4using System.Text;
5using System.Threading.Tasks;
6using System.Windows;
7using System.Windows.Controls;
8using System.Windows.Data;
9using System.Windows.Documents;
10using System.Windows.Input;
11using System.Windows.Media;
12using System.Windows.Media.Imaging;
13using System.Windows.Navigation;
14using System.Windows.Shapes;
15 namespace XamlApp
16{
17     public partial class MainWindow : Window
18     {
19         public MainWindow()
20         {
21             InitializeComponent();
22         }
23     }
24 }
25}
```

Во время компиляции этот класс объединяется с классом, сгенерированном из кода XAML. Чтобы такое слияние классов во время компиляции произошло, класс `XamlApp.MainWindow` определяется как частичный с модификатором **partial**.

А через метод `InitializeComponent()` класс `MainWindow` вызывает скомпилированный ранее код XAML, разбирает его и по нему строит графический интерфейс окна.

Взаимодействие кода C# и XAML

В приложении часто требуется обратиться к какому-нибудь элементу управления. Для этого надо установить у элемента в XAML свойство **Name**.

Еще одной точкой взаимодействия между `xaml` и C# являются **события**. С помощью атрибутов в XAML мы можем задать события, которые будут связаны с обработчиками в коде C#.

Итак, создадим новый проект WPF, который назовем `XamlApp`. В разметке главного окна определим два элемента: кнопку и текстовое поле.

В файле.XamlApp

```
<Grid x:Name="grid1">
```

```
    <TextBox x:Name="textBox1" Width="150" Height="30"  
VerticalAlignment="Top" Margin="20" />
```

```
    <Button x:Name="button1" Width="100" Height="30" Content="Кнопка"  
Click="Button_Click" />
```

```
</Grid>
```

Изменим файл отделенного кода, добавив в него обработчик нажатия

КНОПКИ:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string text = textBox1.Text;
    if (text != "")
    {
        MessageBox.Show(text);
    }
}
```

Определив имена элементов в XAML, затем мы можем к ним обращаться в коде c#: `string text = textBox1.Text`.

При определении имен в XAML надо учитывать, что оба пространства имен `"http://schemas.microsoft.com/winfx/2006/xaml/presentation"` и `"http://schemas.microsoft.com/winfx/2006/xaml"` определяют атрибут `Name`, который устанавливает имя элемента.

Во втором случае атрибут используется с префиксом `x`: `x>Name`. Какое именно пространство имен использовать в данном случае, не столь важно, а следующие определения имени `x>Name="button1"` и `Name="button1"` фактически будут равноценны.

В обработчике нажатия кнопки просто выводится сообщение, введенное в текстовое поле. После определения обработчика мы его можем связать с событием нажатия кнопки в `xaml` через атрибут `Click`: `Click="Button_Click"`. В результате после нажатия на кнопку мы увидим в окне введенное в текстовое поле сообщение.

MainWindow



HELLO

Кнопка



HELLO

OK

Создание элементов в коде C#

Рассмотрим еще одну форму взаимодействия C# и XAML представляет создание визуальных элементов в коде C#.

```
<Grid x:Name="layoutGrid">
```

```
</Grid>
```

```
namespace.XamlApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            Button myButton = new Button();
            myButton.Width = 100;
            myButton.Height = 30;
            myButton.Content = "Кнопка";
            layoutGrid.Children.Add(myButton);
        }
    }
}
```

Сложные свойства и конвертеры типов

```
Button x:Name="myButton"  
Width="120" Height="40" Content="Кнопка"  
HorizontalAlignment="Center"  
Background="Red" />
```

```
Button myButton = new Button();  
myButton.Content = "Кнопка";  
myButton.Width = 120;  
myButton.Height = 40;  
myButton.HorizontalAlignment =  
HorizontalAlignment.Center;  
myButton.Background = new  
System.Windows.Media.SolidColorBrush  
(System.Windows.Media.Colors  
.Red);
```

Чтобы выровнять кнопку по центру, применяется перечисление `HorizontalAlignment`, для установки фонового цвета - класс `SolidColorBrush`.

В WPF имеются встроенные конвертеры для большинства типов данных: `Brush`, `Color`, `FontWeight` и т.д. Все конвертеры типов являются производными от класса `TypeConverter`.

Например, конкретно для преобразования значения `Background="Red"` в объект `SolidColorBrush` используется производный класс `BrushConverter`. При необходимости можно создать свои конвертеры для каких-то собственных типов данных.

Пространства имен из C# в XAML

По умолчанию в WPF в XAML подключается предустановленный набор пространств имен `xml`.

Но мы можем задействовать любые другие пространства имен и их функциональность в том числе и стандартные пространства имен платформы .NET, например, `System` или `System.Collections`. Например, по умолчанию в определении элемента `Window` подключается локальное пространство имен:

```
xmlns:local="clr-namespace:XamlApp"
```

Локальное пространство имен, как правило, называется по имени проекта (в моем случае проект называется XamlApp) и позволяет подключить все классы, которые определены в коде С# в нашем проекте. Например, добавим в проект следующий класс:

```
public class Phone  
{  
    public string Name { get; set; }  
    public int Price { get; set; }  
  
    public override string ToString()  
    {  
        return $"Смартфон {this.Name}; цена: {this.Price}";  
    }  
}
```

```
<Window x:Class="XamlApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:XamlApp"
  mc:Ignorable="d"
  Title="MainWindow" Height="350" Width="525">
<Grid x:Name="layoutGrid">
  <Button x:Name="phoneButton" Width="250" Height="40" HorizontalAlignment="Center">
    <Button.Content>
      <local:Phone Name="Lumia 950" Price="700" />
    </Button.Content>
  </Button>
</Grid>
</Window>
```

Так как пространство имен проекта проецируется на префикс **local**, то все классы проекта используются в форме **local:Название_Класса**. Так в данном случае объект **Phone** устанавливается в качестве содержимого кнопки через свойство **Content**.

Мы можем подключить любые другие пространства имен, классы которых мы хотим использовать в приложении.

```
<Window x:Class="XamlApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d=http://schemas.microsoft.com/expression/blend/2008
  xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
  xmlns:local="clr-namespace:XamlApp"
  xmlns:col="clr-namespace:System.Collections;assembly=microsoft.collections"
  xmlns:sys="clr-namespace:System;assembly=microsoft.collections"
  mc:Ignorable="d"
  Title="MainWindow" Height="350" Width="525">
```

```
<Window.Resources>
  <col:ArrayList x:Key="days">
    <sys:String>Понедельник</sys:String>
    <sys:String>Вторник</sys:String>
    <sys:String>Среда</sys:String>
    <sys:String>Четверг</sys:String>
    <sys:String>Пятница</sys:String>
    <sys:String>Суббота</sys:String>
    <sys:String>Воскресенье</sys:String>
  </col:ArrayList>
</Window.Resources>
<Grid>

</Grid>
</Window>
```