

# WPF Windows Presentation Foundation

2 часть

# DependencyProperty

## Свойства зависимости

- ▶ Есть у ЭУ унаследованных от `DependencyObject`
- ▶ Могут наследовать свои значения от родительского элемента
- ▶ Позволяют вычислять значение на основе нескольких внешних значений
- ▶ Используются при анимации, привязке данных и стилей


# ► 1) Определение свойства зависимостей у TextBlock

Класс унаследован от DependencyProperty


A white arrow points from the text box to the `FrameworkElement` part of the code snippet below.

```
public class TextBlock : FrameworkElement
{
    // свойство зависимостей
    public static readonly DependencyProperty TextProperty;
```

статическое поле →  
свойство должно быть  
доступно другим  
классам


A white arrow points from the text box to the `public static readonly` part of the code snippet above.

Соглашение по именованию → имя  
обычного свойства + Property в конце

A white arrow points from the text box to the `TextProperty` part of the code snippet above.

## ► 2) Регистрация свойства зависимостей

```
static TextBlock()  
{  
    // Регистрация свойства  
    TextProperty = DependencyProperty.Register( ...
```



Определение в статическом конструкторе  
связанного класса → до использования свойства

### ► 3) Упаковка свойства зависимостей

```
public string Text
{
    get { return (string) GetValue(TextProperty); }
    set { SetValue(TextProperty, value); }
}
```

оболочка для свойства зависимостей  
SetValue() и GetValue() → определены в классе  
DependencyObject

Предоставление способа вычисления значения свойства на основе значений других источников

# Пример задания свойства зависимости

```
public class TextBlock : FrameworkElement
{
```

1) Должен наследоваться от DependencyObject

```
// СВОЙСТВО ЗАВИСИМОСТЕЙ
```

```
public static readonly DependencyProperty TextProperty;
```

```
static TextBlock()
{
```

2) общедоступное, статическое, только для чтения поле в классе типа DependencyProperty

```
// Регистрация свойства
```

```
TextProperty = DependencyProperty.Register(
```

```
"Text",
```

```
typeof (string),
```

```
typeof (TextBlock),
```

```
new FrameworkPropertyMetadata(
```

```
string.Empty,
```

```
FrameworkPropertyMetadataOptions.AffectsMeasure |
```

```
FrameworkPropertyMetadataOptions.AffectsRender));
```

3) зарегистрировано в static construct

```
// ...
```

```
}
// Обычное свойство .NET - обертка над свойством зависимостей
```

```
public string Text
```

```
{
```

```
get { return (string) GetValue(TextProperty); }
```

```
set { SetValue(TextProperty, value); }
```

4) обертка - обычное свойство .NET

```
}
```

```
}
```

ИМЯ СВОЙСТВА

ТИП СВОЙСТВА

тип, который владеет свойством

доп. свойства

# Провайдеры свойств – для вычисления базового значения

Получение локального значения свойства (то есть то, которое установлено разработчиком через XAML или через код C#)

Вычисление значения из родительского элемента

Вычисление значения из применяемых стилей

Вычисление значения из шаблона родительского элемента

Вычисление значения из применяемых тем

Получение унаследованного значения (если свойство `FrameworkPropertyMetadata.Inherits` имеет значение `true`)

Извлечение значения по умолчанию, которое устанавливается через объект `FrameworkPropertyMetadata`



приоритет

# определение значения свойства:

- ▶ определяется базовое значение (как описано выше)
- ▶ Если свойство задается выражением, производится вычисление этого выражения - привязка данных и ресурсы
- ▶ Если данное свойство предназначено для анимации, применяется эта анимация.
- ▶ Выполняется метод `CoerceValueCallback` для "корректировки" значения.



# Создание собственного свойства зависимости

```
public class Pasport : DependencyObject
```

надо унаследовать

```
{
```

```
    public static readonly DependencyProperty NumberProperty;
```

определяем свойство зависимости

```
    static Pasport()
    {
```

```
        NumberProperty = DependencyProperty.Register(
            "Number",
            typeof(string),
            typeof(Pasport));
    }
```

регистраруем в  
статическом  
конструкторе

```
    public string Number
```

```
{
```

```
    get { return (string)GetValue(NumberProperty); }
    set { SetValue(NumberProperty, value); }
```

```
}
```

```
}
```

получаем доступ к значению  
свойств

# Использование

Ресурс окна,  
имеет ключ, по которому можем к  
нему обратиться

```
<Window.Resources>  
    < local:Pasport  Number="MP3467234" x:Key="BelPasport" />  
</Window.Resources>
```

```
<Grid x:Name="grid1" DataContext="{StaticResource BelPasport}">  
    <Grid.RowDefinitions>  
        <RowDefinition />  
        <RowDefinition />  
    </Grid.RowDefinitions>  
    <Grid.ColumnDefinitions>  
        <ColumnDefinition />  
        <ColumnDefinition />  
    </Grid.ColumnDefinitions>  
    <TextBlock Text="Номер паспорта" Grid.Row="0" />  
    <TextBlock Text="{Binding Number, Mode=TwoWay}"  
Grid.Column="0" Grid.Row="1" />  
  
</Grid>
```

Устанавливаем ресурс как  
контекст данных

привязываем Text к свойству ресурса  
Для обычного свойств привязку не сможем сделать

# Добавление валидации-

Свойства можно проверять на valid

**1) ValidateValueCallback:** делегат - true и false – прошло или нет проверку – срабатывает первым

**2) CoerceValueCallback:** делегат, который может подкорректировать уже существующее значение свойства, если оно вдруг не попадает в диапазон допустимых значений  
срабатывает вторым

Могут использоваться вместе или по-отдельности

# Пример валидации

```
static Passport()
{
    FrameworkPropertyMetadata metadata =
        new FrameworkPropertyMetadata();
    NumberProperty = DependencyProperty.Register("Number",
        typeof(string),
        typeof(Passport), metadata,
        new ValidateValueCallback(ValidateValue));
}

public string Number
{
    get { return (string)GetValue(NumberProperty); }
    set { SetValue(NumberProperty, value); }
}

private static bool ValidateValue(object value)
{
    string currentValue = (string)value;
    if (currentValue.Contains("MP")) // если
        return true;
    return false;
}
```

применим делегат, который указывает на метод

принимает значение свойства

валидация пройдена

# Прикрепляемые свойства

- Attached properties - являются свойствами зависимостей - определяются в одном классе, а применяются в другом

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Button x:Name="button1" Content="Hello"
    Grid.Column="1" Grid.Row="0" />
</Grid>
```

# Регистрация прикрепляемого свойства

```
Grid.ColumnProperty = DependencyProperty.RegisterAttached(  
    "Column",  
    typeof(int),  
    typeof(Grid),  
    new FrameworkPropertyMetadata(0,  
        new PropertyChangedCallback(Grid.OnCellAttachedPropertyChanged)  
        new ValidateValueCallback(Grid.IsIntValueNotNegative))
```

не создается обертка в виде стандартного свойства C#

установка и получение значения для прикрепленных свойств

```
public static int GetColumn(UIElement element)  
{  
}  
public static void SetColumn(UIElement element, int value)  
{  
    Grid.SetRow(button1, 1);  
    Grid.SetColumn(button1, 1);
```

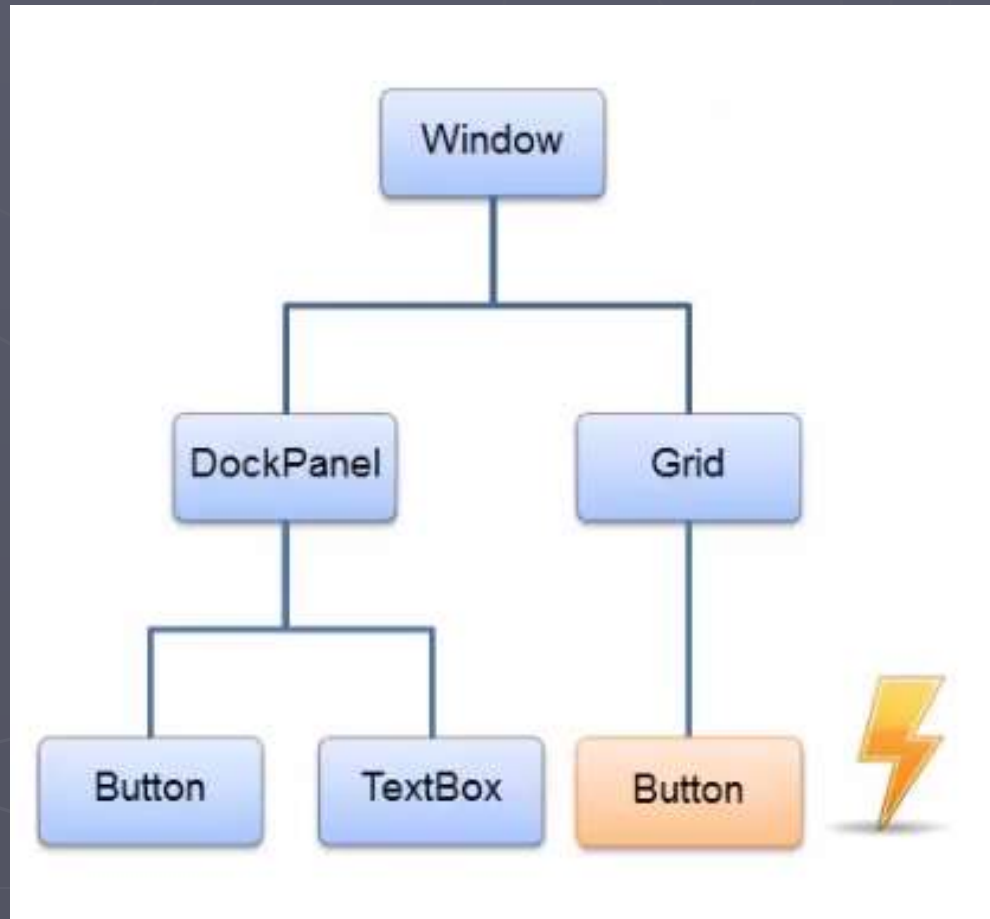
# Обработка событий

- Маршрутизация событий (**routed events**) –  
Маршрут по дереву элементов управления

Маршрутизируемые события позволяют обработать событие в одном элементе (например в panel), хотя оно возникло в другом (например в button).

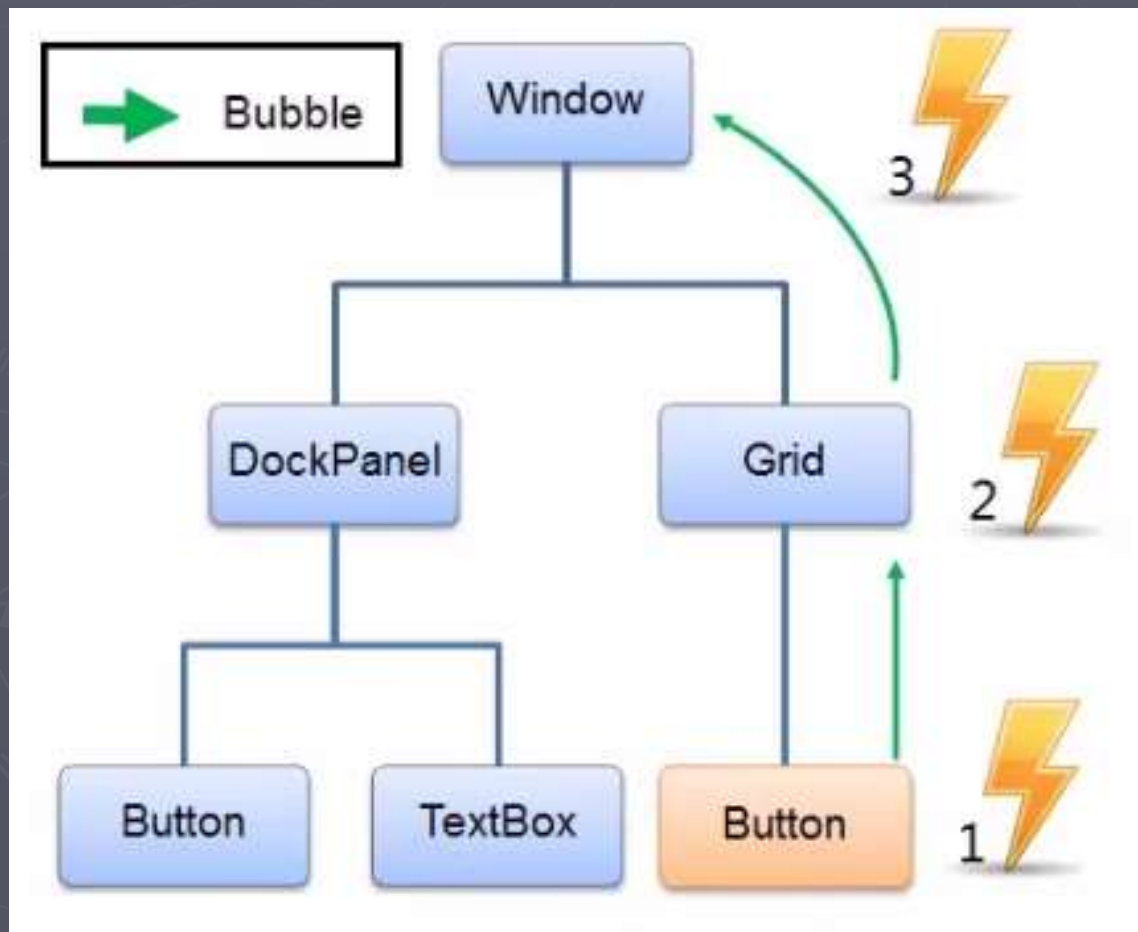
- Direct (=WinForms)
- Tunneling - туннельное
- Bubbling - поднимающееся

**Прямые (direct events)** - возникают и обрабатываются на одном элементе и нигде дальше не передаются. Действуют как обычные события.

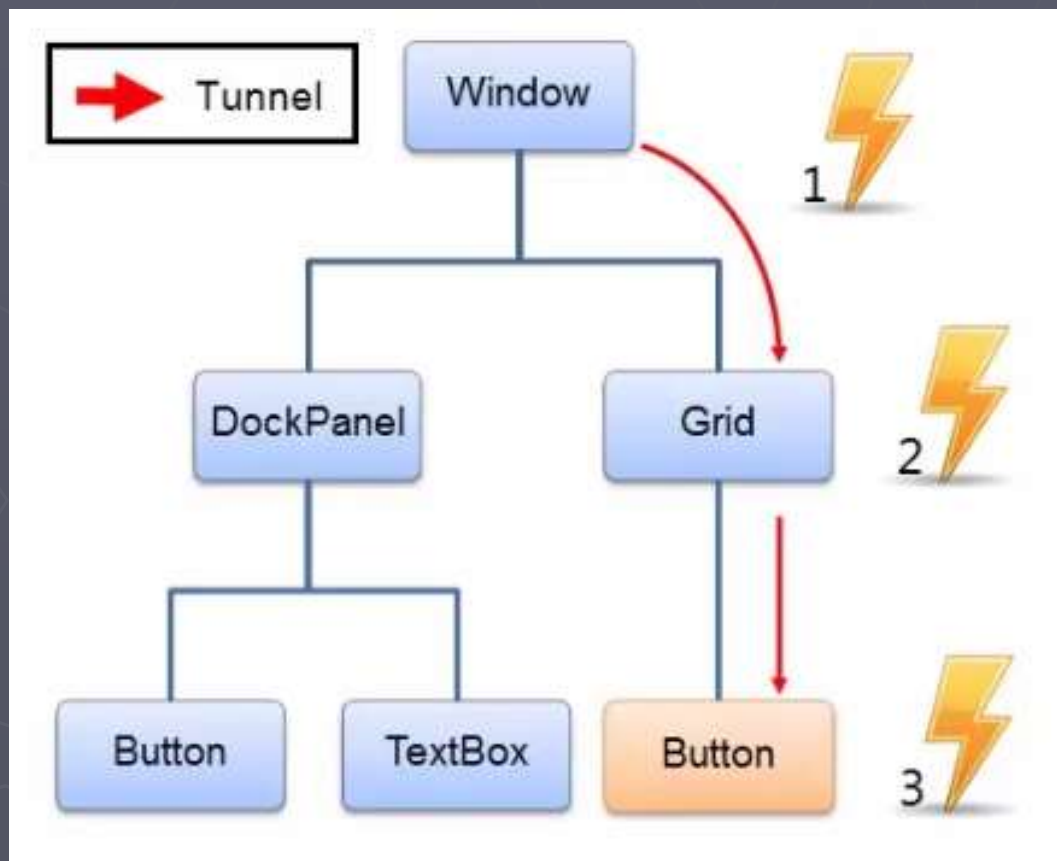




- **Поднимающиеся (bubbling events)** - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру.



**Опускающиеся, туннельные** (tunneling events) - начинает обрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.



# Подключение обработчиков событий

- декларативно в файле xaml-кода

```
<Button x:Name="Edit" Content="Click" Click="Edit_Click" />
```

- В КОДЕ

```
Edit.Click += Edit_Click;  
...  
  
private void Edit_Click(object sender, RoutedEventArgs e)  
{  
  
}  
}
```

<b>KeyDown</b>	Поднимающееся	Возникает при нажатии клавиши
<b>PreviewKeyDown</b>	Туннельное	Возникает при нажатии клавиши

<b>GotMouseCapture</b>	Поднимающееся
<b>LostMouseCapture</b>	Поднимающееся
<b>MouseEnter</b>	Прямое
<b>MouseLeave</b>	Прямое
<b>MouseDown</b>	Поднимающееся
<b>PreviewMouseDown</b>	Туннельное
<b>MouseUp</b>	Поднимающееся
<b>PreviewMouseUp</b>	Туннельное
<b>MouseRightButtonDown</b>	Поднимающееся

```
public abstract class CircleM : ContentControl
{
    // Определение события
    public static readonly RoutedEvent ClickEvent;
    // Регистрация события
    static CircleM()
    {
        CircleM.ClickEvent =EventManager.RegisterRoutedEvent(
            "Click", RoutingStrategy.Bubble,
            typeof(RoutedEventHandler), typeof(CircleM));
    }

    // Традиционная оболочка события
    public event RoutedEventHandler Click
    {
        add
        {base.AddHandler(CircleM.ClickEvent, value);
        }
        remove
        {base.RemoveHandler(CircleM.ClickEvent, value);
        }
    }
}
```

Определение маршрутизированных событий

правило именования – <Имя события>Event

указывается 1)тип маршрута события, 2) тип делегата события и 3) класс владеющий данным событием

AddHandler() и RemoveHandler() определенные в классе FrameworkElement

# Прикрепляемые события (Attached events)

- ▶ Несколько элементов одного и того же типа - привязать к одному событию

*Имя\_класса.Название\_события="Обработчик"*

```
<StackPanel x:Name="Selector" Grid.Column="0"
RadioButton.Checked="RadioButton_Click">
    <RadioButton GroupName="test" Content="A" />
    <RadioButton GroupName="test" Content="B" />
    <RadioButton GroupName="test" Content="C" />
    <RadioButton GroupName="test" Content="D" />
</StackPanel>
```

Или так

```
Selector.AddHandler(RadioButton.CheckedEvent,
    new RoutedEventHandler(RadioButton_Click));
```

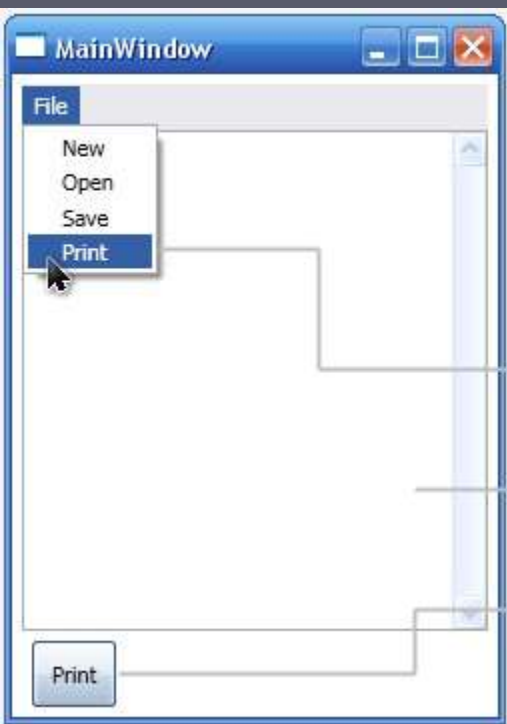
# Команда

- **Команды** - механизм выполнения задачи (паттерн "Команда" Command)

Назначение:

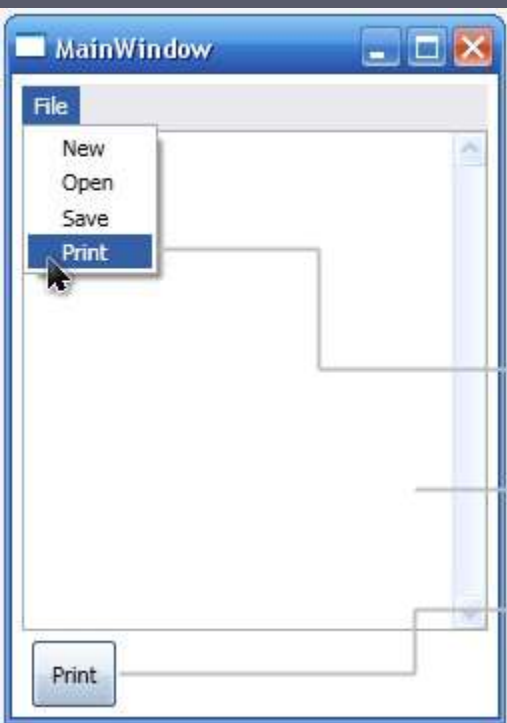
- использовать одну и ту же команду для нескольких ЭУ
- абстрагировать набор действий от конкретных событий конкретных элементов

# Модель обработки событий



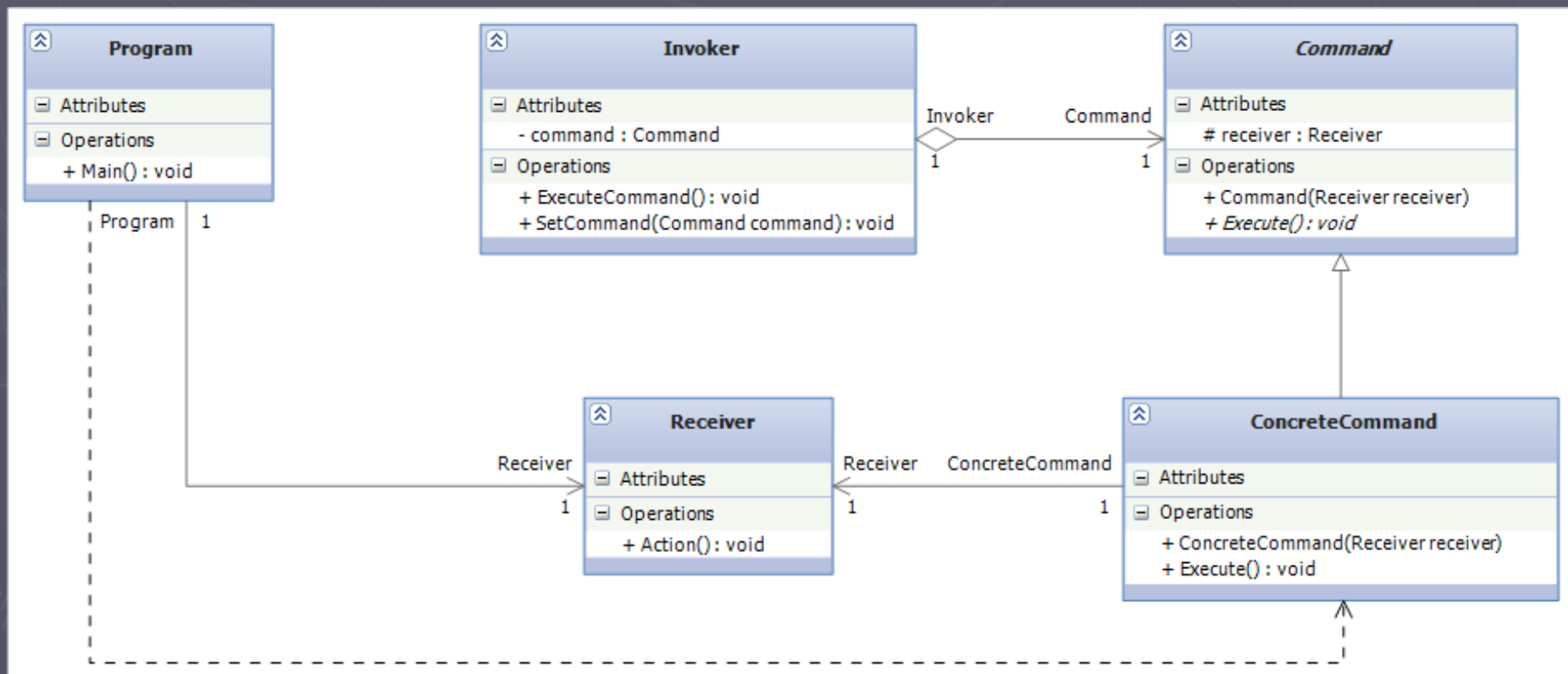


# Модель команд



# Паттерн Command

- Суть: Представить запрос как ООП объект (не метод)
  - Конфигурация команды
  - Для определения и выполнения в разное время: создания очереди, stop и ....
  - undo, redo
  - Протоколирования и структурирования системы



# Класс, который обсуживает команду

```
namespace PatternCommand
{
    class Invoker
    {
        Command command;

        public void StoreCommand(Command command)
        {
            this.command = command;
        }

        public void ExecuteCommand()
        {
            command.Execute();
        }
    }
}
```

## ► Абстракция команда

```
namespace PatternCommand
{
    abstract class Command
    {
        protected Receiver receiver;

        public Command(Receiver receiver)
        {
            this.receiver = receiver;
        }

        public abstract void Execute();
    }
}
```

## ► Конкретная команда

```
namespace PatternCommand
{
    class ConcreteCommand : Command
    {
        public ConcreteCommand(Receiver receiver)
            : base(receiver)
        {
        }

        public override void Execute()
        {
            receiver.Action();
        }
    }
}
```

## ► Выполняет команду

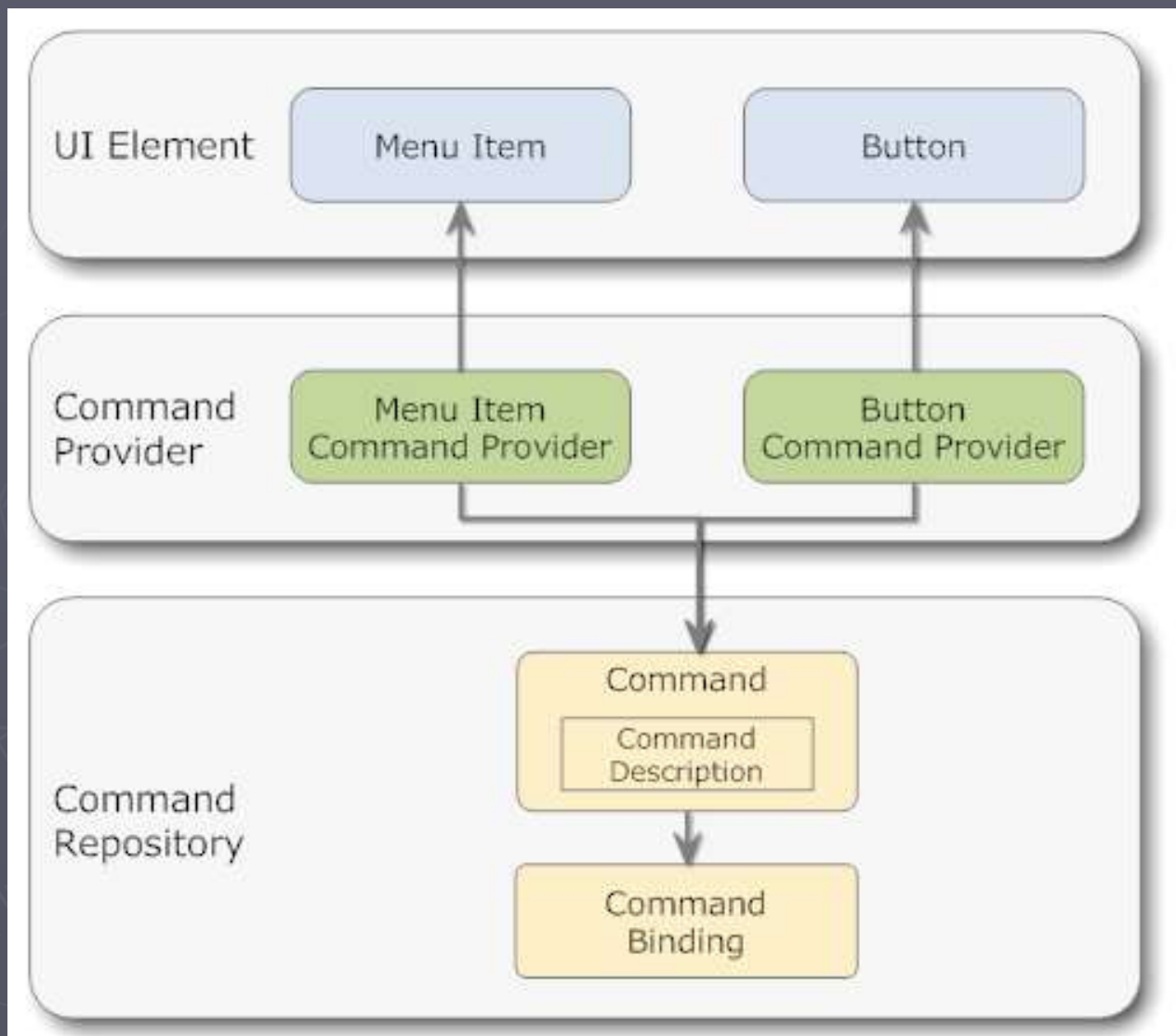
```
namespace PatternCommand
{
    class Receiver
    {
        public void Action()
        {
            Console.WriteLine("Receiver");
        }
    }
}
```

## ► Все вместе

```
static void Main()
{
    Receiver receiver = new Receiver();
    Command command = new ConcreteCommand(receiver);
    Invoker invoker = new Invoker();

    invoker.StoreCommand(command);
    invoker.ExecuteCommand();
}
```

# Модель команд в WPF



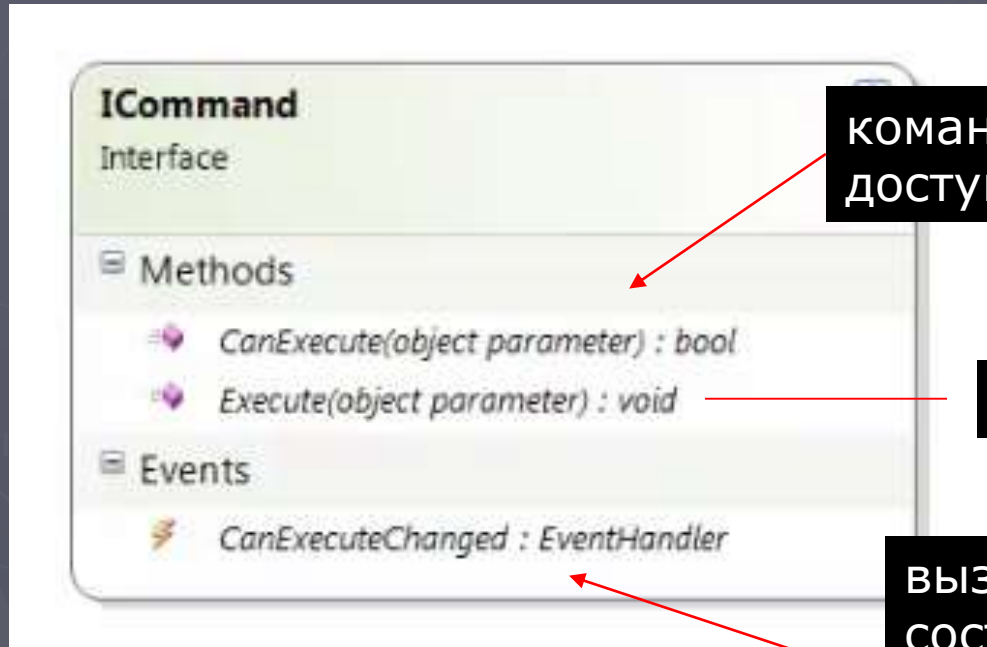


# Модель команд в WPF

- ▶ **команда** - представляет выполняемую задачу
- ▶ **Привязка команд** - связывает команду с определенной логикой приложения
- ▶ **Источник команды** - элемент UI, который запускает команду
- ▶ **Цель команды** - элемент интерфейса, на котором выполняется команда

Все команды реализуют интерфейс

`System.Windows.Input.ICommand:`



команда включена/выключена и доступна для использования

хранения логики команды

вызывается при изменении состояния команды

реализован встроенным классом, который является базовым для всех встроенных команд

**`System.Windows.Input.RoutedCommand`**

# Встроенные команды

**RoutedCommand**



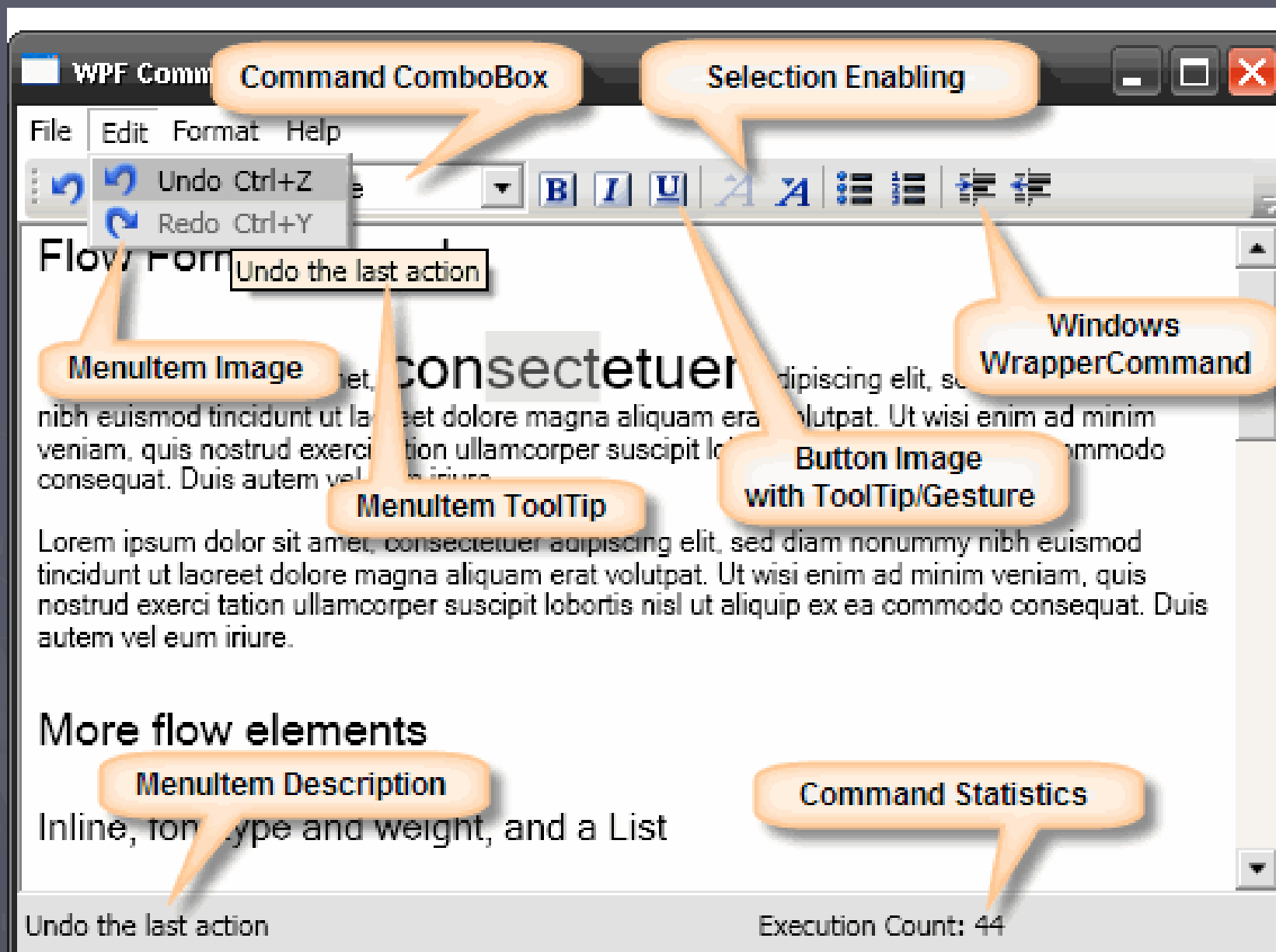
**RoutedUICommand**

- ▶ Общие **ApplicationCommands**: Это команды: *CancelPrint, Close, ContextMenu, Copy, CorrectionList, Cut, ....*
- ▶ Навигации  
**NavigationCommands**: *BrowseBack, BrowseForward, BrowseHome ....*
- ▶ Компонентов интерфейса  
**ComponentCommands**: *MoveDown, MoveLeft, MoveRight, MoveUp, SelectToEnd* и т.д

- ▶ Редактирования документов **EditingCommands:** *AlignCenter, DecreaseFontSize, MoveDownByLine* и т.д.
- ▶ Управления мультимедиа **MediaCommands:** *DecreaseVolume, Play, Rewind, Record*
- ▶ Системные команды **SystemCommands:** *CloseWindow, MaximizeWindow, MinimizeWindow, RestoreWindow* и т.д.
- ▶ Команды ленты панели инструментов **RibbonCommands:** *AddToQuickAccessToolBar, MaximizeRibbonCommand* и т.д.

# Пример дерева команд для RTF редактора





# Пример использования

## 1) Источник команд

```
<Button Name ="ButtonT"  
        Background="DarkGreen"  
        Content="New"  
        Height="20"  
        Command="New" />
```

ЭУ должен реализовывать  
интерфейс ICommandSource



```
public interface ICommandSource  
{  
    ICommand Command { get; }  
    object CommandParameter { get; }  
    IInputElement CommandTarget { get; }  
}
```

```
<Button x:Name="ButtonT"  
        Command="ApplicationCommands.New"  
        Content="New" />
```

```
ButtonT.Command = ApplicationCommands.New;
```

## 2) Привязка команды

Команды (встроенные) не содержат конкретного кода по их выполнению. Чтобы связать эти команды с реальным кодом нужна привязка

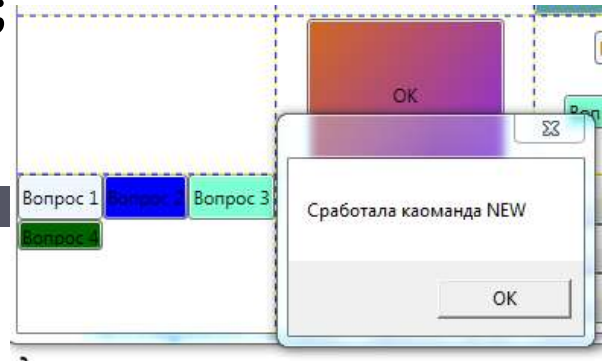
```
public MainWindow()
{
    InitializeComponent();

    CommandBinding binding =
        new CommandBinding(ApplicationCommands.New);
    binding.Executed+=
        new ExecutedRoutedEventHandler(binding_exec);

    // добавляем привязку к коллекции привязок элемента Button
    ButtonT.CommandBindings.Add(binding);

    // или
    this.CommandBindings.Add(binding);
}

private void binding_exec(object sender,
    ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Сработала команда NEW");
}
```





```
<Button x:Name="ButtonT" Command="ApplicationCommands.New"
Content="New">
```

```
<Button.CommandBindings>
```

```
<CommandBinding Command="New"
```

```
Executed="binding_exec" />
```

```
</Button.CommandBindings>
```

```
</Button>
```

## Привязка команды в XAML

```
xmlns:def="clr-namespace:System.Data.Sql;assembly=System.
```

```
mc:Ignorable="d"
```

```
Title="MainWindow" Height="350" Width="525">
```

```
ndow.CommandBindings>
```

```
<CommandBinding Command="" />
```

```
indow.CommandBindings>
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

```
<CommandBinding Command="New"
```

New

AlignCenter

AlignJustify

AlignLeft

AlignRight

Backspace

BoostBass

BrowseBack

Property System.Windows.Input.Route

Получает значение, представляюще

Маршрутизация  
команды к  
контейнеру

```
<Window.CommandBindings>
```

```
<CommandBinding Command="New"
```

```
CanExecute="CommandBinding_OnCanExecute"
```

```
Executed="binding_exec"/>
```

```
</Window.CommandBindings>
```

<Button

Параметры

Command="Cut"

CommandParameter="10"

CommandTarget="buf"></Button>

Цель

# Создание команды пользователя

```
public class NewCustomCommand
{
    private static RoutedUICommand pnvCommand;

    static NewCustomCommand()
    {
        InputGestureCollection inputs =
            new InputGestureCollection();
        inputs.Add
            (new KeyGesture(Key.P, ModifierKeys.Alt, "Alt+P"));

        pnvCommand =
            new RoutedUICommand("PNV", "PNV",
                                typeof(NewCustomCommand), inputs);
    }

    public static RoutedUICommand PnvCommand
    {
        get { return pnvCommand; }
    }
}
```

Текст

Имя команды

Горячие клавиши

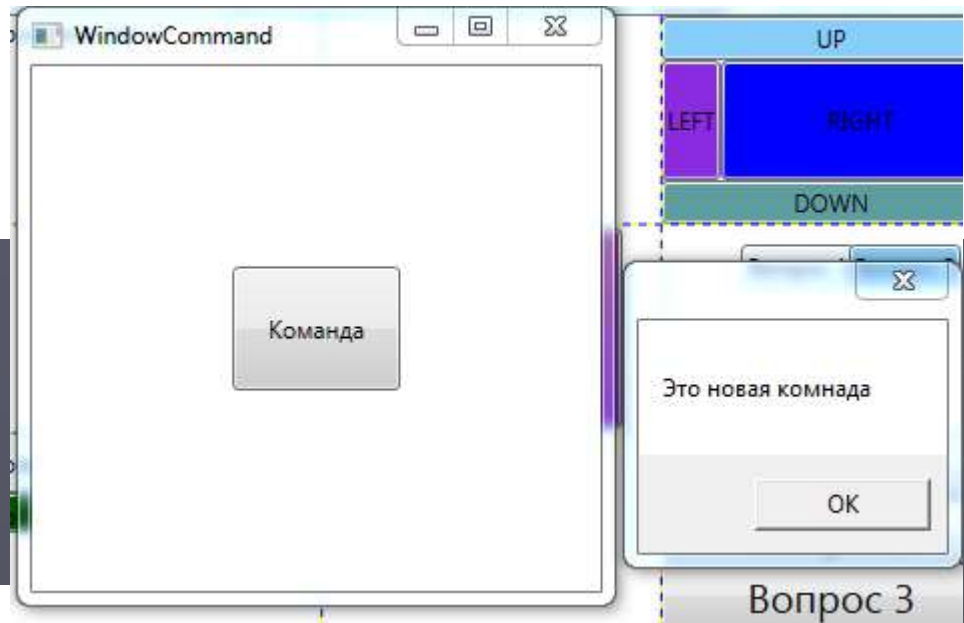
```

<Window.CommandBindings>
    <CommandBinding Command="local:NewCustomCommand.PrvCommand"
        Executed="CommandBinding_Executed"></CommandBinding>
</Window.CommandBindings>

<Grid>
    <Button Command="local:NewCustomCommand.PrvCommand"
        Margin="100">Команда</Button>

</Grid>
</Window>

```



```

private void CommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Это новая комната");
}

```

# Другой способ создания команды пользователя

```
public class CustomCommand : ICommand
{
    // изменения, которые могут повлиять на возможность запуска команды.
    public event EventHandler CanExecuteChanged
    {
        add
        { CommandManager.RequerySuggested += value; }
        remove
        { CommandManager.RequerySuggested -= value; }
    }

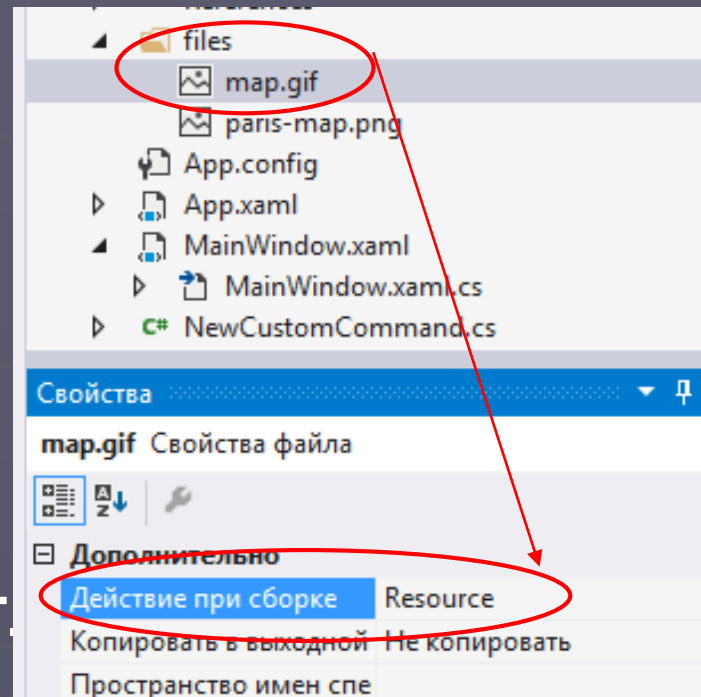
    public bool CanExecute(object parameter)
    {
        return true;
    }

    public void Execute(object parameter)
    {
        MessageBox.Show("Сработала");
    }
}
```

# Ресурсы

► **Ресурс сборки** – блок двоичных данных, встроенный в сборку

► **Ресурс объекта** – .NET-объект который объявляется в одном месте и используется в других (логический ресурс – кнопки, кисти и т.д.)



- 1) эффективность: определить один раз и многократно использовать его в различных местах приложения
- 2) поддержка : изменение ресурса в одном месте

```
<Window.Resources>
```

```
<ImageBrush x:Key="CommonImBrush"
```

```
ViewportUnits="RelativeToBoundingBox"
```

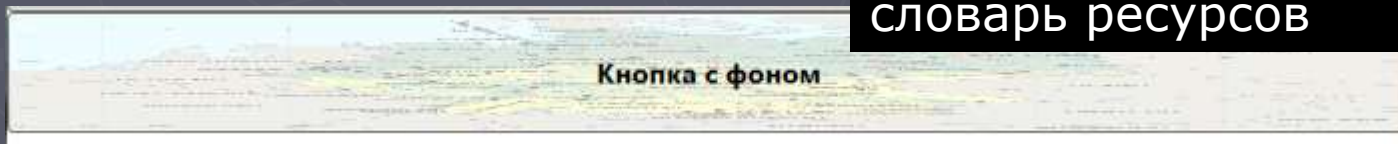
```
ImageSource="files/map.gif"
```

```
Opacity="0.3"></ImageBrush>
```

```
</Window.Resources>
```

определяет ключ в словаре

Свойство **Resources** представляет объект **ResourceDictionary** или словарь ресурсов



```
<Button Background="{StaticResource CommonImBrush}"
```

```
FontWeight="Bold"
```

```
FontSize="14"
```

```
Height="60"
```

```
>Кнопка с фоном</Button>
```

применить ресурс используя кл

## ► Добавление и установка ресурса

```
ImageBrush CommonimBrush = new ImageBrush();  
    //...  
  
// добавление ресурса в словарь ресурсов окна  
this.Resources.Add("CommonimBrush", CommonimBrush);  
  
// установка ресурса у кнопки  
button1.Background = (Brush)this.TryFindResource("CommonimBrush");
```

## ► ResourceDictionary:

- Метод **Add(string key, object resource)** добавляет объект с ключом key в словарь
- Метод **Remove(string key)** удаляет из словаря ресурс с ключом key
- Свойство **Uri** устанавливает источник словаря
- Свойство **Keys** возвращает все имеющиеся в словаре ключи
- Свойство **Values** возвращает все имеющиеся в словаре объекты



- ▶ **Статический** ресурс - свойство инициализируется один раз и не меняет свое значение, даже если ресурс был изменен
- ▶ **Динамический** ресурса - свойство элемента обновляется при обновлении ресурса
- ▶ Один и тот же ресурс может быть и стат. и динамич.

# Определение ресурса

```
<Window.Resources>  
  <ImageBrush x:Key="MunBrush"  
    TileMode="Tile"  
    ViewportUnits="Absolute"  
    Viewport="0 0 64 64"  
    ImageSource="files/munich.jpg"  
    Opacity="0.5"></ImageBrush>  
</Window.Resources>
```

<Grid>

<StackPanel Margin="5">

<Button Background="{DynamicResource MunBrush}"

Padding="5"

FontWeight="Bold"

FontSize="14"

Click="Change\_OnClick"

Margin="5">Динамический ресурс</Button>

<Button Name ="Change" Padding="5"

Margin="5"

Click="Change\_OnClick"

FontWeight="Bold"

FontSize="14">Изменить фон (ресурс)</Button>

<Button Background="{StaticResource MunBrush}"

Padding="5"

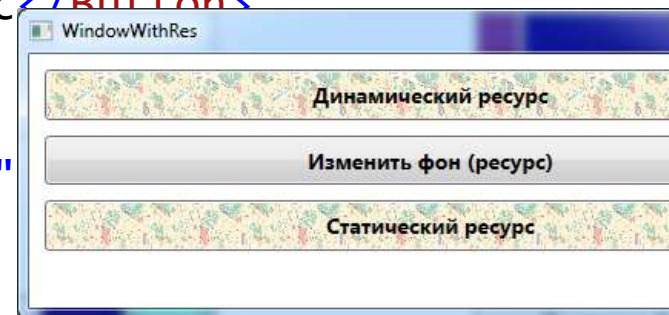
Margin="5"

FontWeight="Bold"

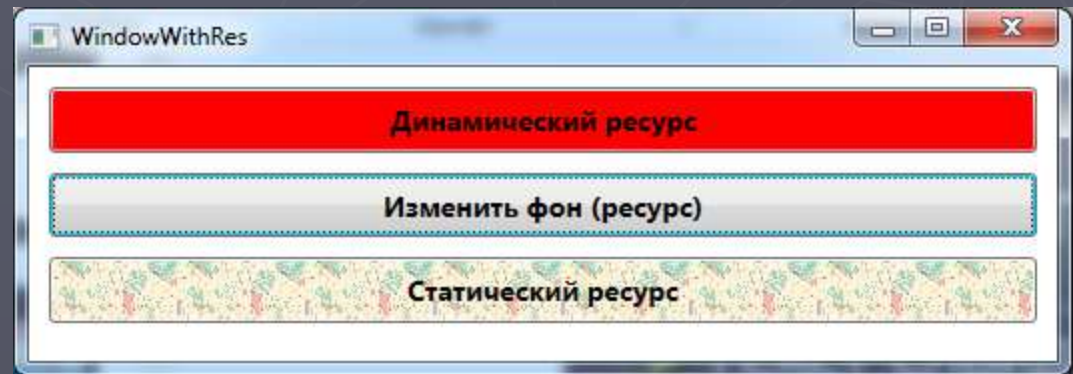
Click="Change\_OnClick"

FontSize="14">Статический ресурс</Button>

</StackPanel>



```
private void Change_OnClick(object sender, RoutedEventArgs e)
{
    this.Resources["MunBrush"] =
        new SolidColorBrush(Colors.Red);
}
```



## ► Установка динамического ресурса в коде

```
ImageBrush CommonimBrush = new ImageBrush();  
//...
```

```
// добавление ресурса в словарь ресурсов окна
```

```
this.Resources.Add("CommonimBrush", CommonimBrush);
```

```
// установка ресурса у кнопки
```

```
button1.SetResourceReference(Button.BackgroundProperty, "CommonimBrush")  
}
```

СВОЙСТВО ЗАВИСИМОСТИ ОБЪЕКТ

ключ ресурса

# Управление ресурсами

```
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace WpfAppDemo
{
    /// <summary>
    /// Логика взаимодействия
    /// </summary>
    /// ссылок: 4
    public partial class WindowWithResources
    {
        /// ссылок: 1
        public WindowWithResources()
        {
            InitializeComponent();
        }

        2
        /// ссылок: 1
        private void ChangeResource_Click(object sender, RoutedEventArgs e)
        {
            this.Resources["Red"] = new SolidColorBrush(Colors.Red);
            this.Resources["Blue"] = new SolidColorBrush(Colors.Blue);
        }
    }
}
```

Адресная панель (ContextMenu) для `Resources`:

- Add
- BeginInit
- Clear
- Contains
- CopyTo
- EndInit
- FindName
- GetEnumerator
- RegisterName
- Remove
- UnregisterName

Свойство `Red` в `Resources` реализует набор ресурсов.

## ► Элементы DynamicResource и StaticResource

```
<Button x:Name="button1" MaxWidth="80" MaxHeight="40" Content="Test">
    <Button.Background>
        <DynamicResource ResourceKey="CommonimBrush" />
    </Button.Background>
</Button>
```

```
<Window.Resources>
    <Button x:Key="button1" x:Shared="False" Content="Test" />
</Window.Resources>
    <StackPanel>
        <StaticResource ResourceKey="button1" />
        <StaticResource ResourceKey="button1" />
        <StaticResource ResourceKey="button1" />
    </StackPanel>
```

# Ресурсы приложения

## App.xaml

```
<Application.Resources>
```

```
    <ImageBrush x:Key="MunBrush" TileMode="Tile"  
        ViewportUnits="Absolute" Viewport="0 0 32 32"  
        ImageSource="files/munich.jpg"  
        Opacity="0.3"></ImageBrush>
```

```
</Application.Resources>
```



# Системные ресурсы

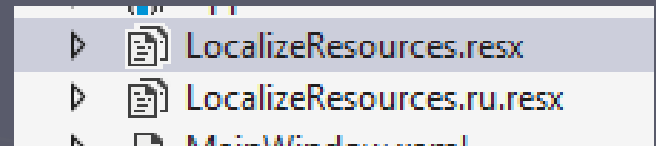
- ▶ SystemColors
- ▶ SystemFonts
- ▶ SystemParameters

```
<Label  
Foreground="{x:Static SystemColors.WindowTextBrush}">  
Статически</Label>
```

```
<Label  
Foreground="{DynamicResource  
    {x:Static SystemColors.WindowTextBrushKey}}">  
Динамически</Label>
```

# Локализация приложений

## ► Содержит



abc Строки		✱ Добавить ресурс	✕ Удалить ресурс	Модификатор доступа: Public
	Имя	Значение		
►	BtnLabel	Enter		
	FNLabel	First Name		
	LNLabel	Last Name		
*				

## ► Может



en\_GB  
ru\_RU

Язык культура

## ► Обращение к ресурсу в разметке

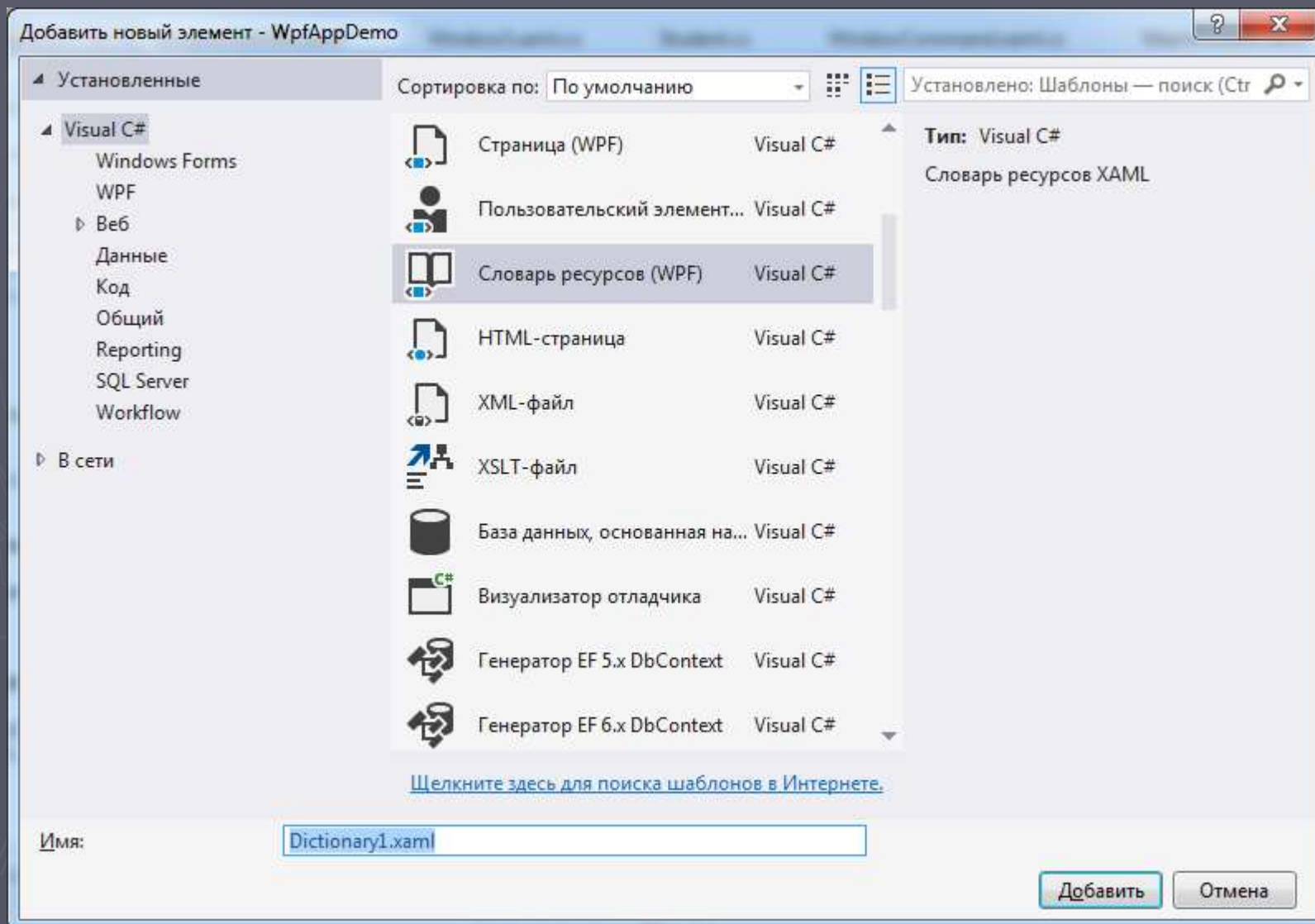
```
<TextBlock Text="{x:Static local:LocalizeResources.FNLabel}">  
</TextBlock>
```

## ► Переключение культуры (до инициализации компонент и требует перезапуска)

```
Thread.CurrentThread.CurrentCulture =  
    new CultureInfo(Settings.Default.Culture);
```

```
Thread.CurrentThread.CurrentCulture =  
    new CultureInfo("ru_RU");
```


# Словари ресурсов



коллекция объектов `ResourceDictionary`, которые добавляются к ресурсам

```
<ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="Dictionary1.xaml" />
    <ResourceDictionary Source="Dictionary2.xaml" />
    <ResourceDictionary Source="ButtonStyles.xaml" />
    <SolidColorBrush Color="Green" x:Key="GButton" />
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

файла ресурсов подключаем к  
ресурсам окна



```
<Window.Resources>
  <ResourceDictionary Source="Dictionary1.xaml" />
</Window.Resources>
```

# Стили

**Стиль** –коллекция значений свойств, которые могут быть применены к элементу (CSS)


Хранятся в ресурсах

# Работа с ресурсами

## 1) Объявление ресурса

```
<Window.Resources>
    <FontWeight x:Key="PNVWeigth">
        Bold
    </FontWeight>
    <system:Double x:Key="PNVSize">
        20
    </system:Double>
</Window.Resources>
```

Нет связи  
между  
ресурсами



## 2) Применение ресурса

```
<Button Command="local:NewCustomCommand.PnvCommand"
        Margin="100"
        FontWeight="{StaticResource PNVWeigth}"
        FontSize="{StaticResource PNVSize}"
        >
    Команда
</Button>
```

Объемный код



# Определение стиля окна

```
<Window.Resources>
```

```
  <Style x:Key="PNVStyle">
```

```
    <Setter Property="Control.FontSize" Value="20"></Setter>
```

```
    <Setter Property="Control.FontFamily" Value="Times New Roman"></Setter>
```

```
    <Setter Property="Control.FontWeight" Value="UltraLight"></Setter>
```

```
  </Style>
```

```
</Window.Resources>
```

Тип\_элемента.Свойство\_элемента

Группировка

представляет  
класс **System.Windows.Style**

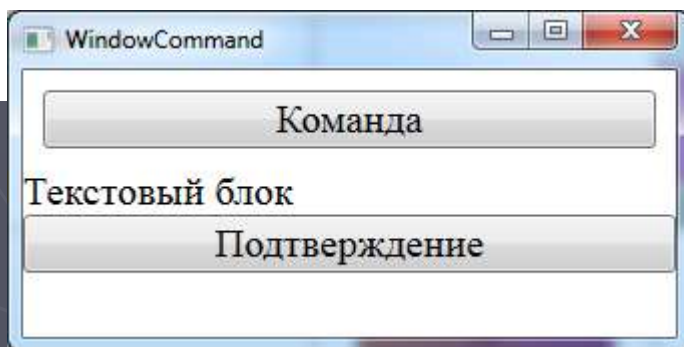
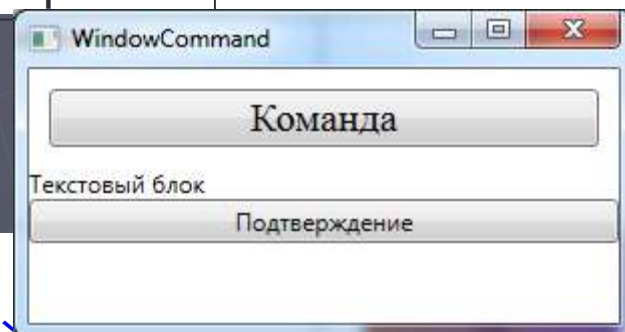
## Применение стиля

```
<Button Margin="100"
```

```
  Style="{DynamicResource PNVStyle}">
```

Команда

```
</Button>
```



- Значения параметров  
элементов приоритетнее  
стиля



# Ключевые свойства стиля

- ▶ **Setters** – коллекция объектов, которые автоматически устанавливают значение свойств элементов управления
- ▶ **Triggers** – коллекция объектов, которые позволяют автоматически изменять параметры стиля
- ▶ **BasedOn** – для создания стиля, который наследует другой стиль и переопределяет его значения
- ▶ **TargetType** – указывает тип элементов на которые действует стиль

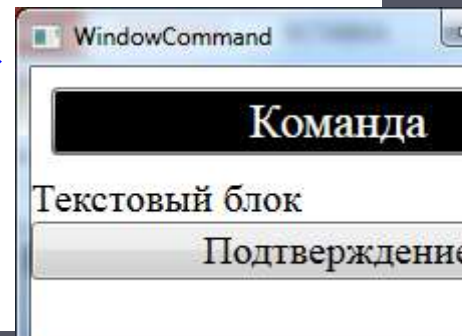
# Наследование стилей

```
<Style x:Key="PNVStyle">
    <Setter Property="Control.FontSize" Value="20"></Setter>
    <Setter Property="Control.FontFamily"
        Value="Times New Roman"></Setter>
    <Setter Property="Control.FontWeight"
        Value="UltraLight"></Setter>
</Style>
<Style x:Key="PNVStyleNext" BasedOn="{StaticResource PNVStyle}">
    <Setter Property="Control.Background" Value="Black"/>
    <Setter Property="Control.FontSize" Value="24"/>
    <Setter Property="Control.Foreground" Value="White"/>
</Style>
```

Наследование

переопределение

```
<StackPanel>
    <Button Margin="10" Style="{DynamicResource PNVStyleNext}">
        Команда </Button>
    <TextBlock Style="{DynamicResource PNVStyle}">
        Текстовый блок </TextBlock>
    <Button Style="{DynamicResource PNVStyle}">
        Подтверждение </Button>
</StackPanel>
```



# ► Автоматическое применение свойств к ЭУ

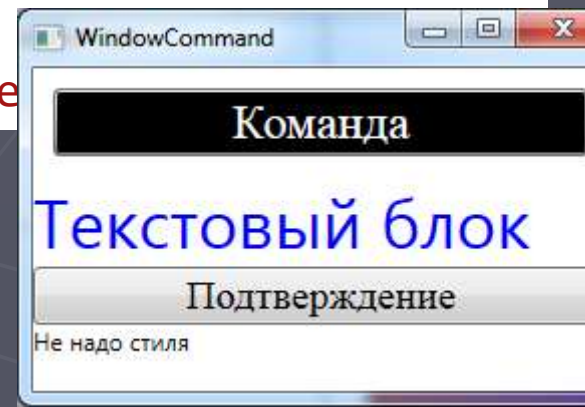
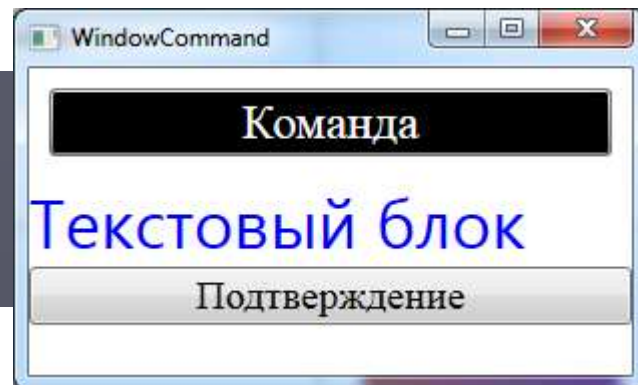
```
<Style TargetType="TextBlock">  
    <Setter Property="Control.FontSize" Value="34"></Setter>  
    <Setter Property="Control.BorderBrush" Value="Aqua"/>  
    <Setter Property="Control.Foreground" Value="Blue"/>  
</Style>
```

применяется

```
<TextBlock>    Текстовый блок    </TextBlock>
```

```
<Button Style="{DynamicResource PNVStyle}"> Подтверждение </Button>
```

```
<TextBlock Style="{x:Null}">    Не надо стилиа    </Te
```



# Задание событий и обработчиков (редко)

```
<Style TargetType="TextBlock">
    <Setter Property="Control.FontSize" Value="34"></Setter>
    <Setter Property="Control.BorderBrush" Value="Aqua"/>
    <Setter Property="Control.Foreground" Value="Blue"/>

    <EventSetter      Event="MouseDown"
                       Handler="control_MouseDown"/>
</Style>
```

# Триггеры

**Триггеры** – декларативное определение некоторых действий, которые выполняются при изменении свойств (свойств зависимостей) стиля

Определяется в XAML

# Основные типы триггеров

- ▶ **Trigger** – простой триггер. Следит за изменением значения свойства
- ▶ **MultiTrigger** – срабатывает при выполнении множества условий
- ▶ **DataTrigger** – срабатывает при изменении в связанных с ним данных
- ▶ **MultiDataTrigger** – множество триггеров данных
- ▶ **EventTrigger** – применяется при возникновении события

# Простой триггер (триггер свойств)

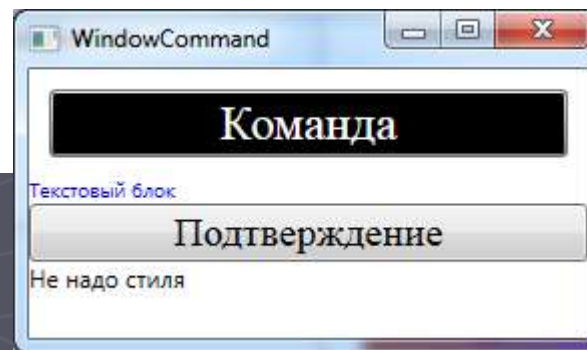
```
<Style TargetType="TextBlock">
    <Setter Property="Control.FontSize" Value="34"></Setter>
    <Setter Property="Control.BorderBrush" Value="Aqua"/>
    <Setter Property="Control.Foreground" Value="Blue"/>

    <!--<EventSetter Event="MouseDown"
Handler="control_MouseDown"/>-->
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="FontSize" Value="10"/>
        </Trigger>
        <Trigger Property="IsFocused" Value="True">
            <Setter Property="FontSize" Value="40"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

Если/отслеживаем  
свойство

Установить

по наведению на  
textblock высота  
шрифта  
устанавливается в 10



# MultiTrigger - содержит коллекцию элементов Condition

```
Style x:Key="PNVStyleNext" BasedOn="{StaticResource PNVStyle}">
    <Setter Property="Control.Background" Value="Black"/>
    <Setter Property="Control.FontSize" Value="24"/>
    <Setter Property="Control.Foreground" Value="White"/>
```

```
<Style.Triggers>
    <MultiTrigger>
```

```
        <!--Список условий-->
        <MultiTrigger.Conditions>
            <Condition Property="Control.IsMouseOver" Value="True"></Condition>
            <Condition Property="Control.IsPressed" Value="True"></Condition>
        </MultiTrigger.Conditions>
```

```
        <!--Список изменений, которые вступят в силу, если все условия выполнятся-->
        <MultiTrigger.Setters>
            <Setter Property="Control.Foreground" Value="DarkBlue"></Setter>
            <Setter Property="Control.FontSize" Value="20"></Setter>
        </MultiTrigger.Setters>
```

```
    </MultiTrigger>
</Style.Triggers>
</Style>
```



# DataTrigger

```
public class Student
{
    public String FName { get; set; }
    public String LName { get; set; }
    public int Number { get; set; }
    public override string ToString()
```

```
<Style TargetType="ListBoxItem">
    <Style.Triggers>
```

Задаёт значение отслеживаемого свойства, при котором сработает триггер

```
<!--Если значение свойства объекта будет равно 0 поменять свойства -->
```

```
<DataTrigger Binding="{Binding Path=Number}" Value="0">
    <Setter Property="Foreground" Value="Red" />
</DataTrigger>
```

Для соединения с отслеживаемыми свойствами триггеры данных используют выражения привязки

```
<MultiDataTrigger>
    <MultiDataTrigger.Conditions>
        <Condition Binding="{Binding Path=Number}"
                    Value="10" />
        <Condition Binding="{Binding Path=FName}"
                    Value="July" />
    </MultiDataTrigger.Conditions>
    <Setter Property="Background" Value="Green" />
</MultiDataTrigger>
</Style.Triggers>
</Style>
```

```
<ListBox Width="180"  
          HorizontalAlignment="Center"  
          Background="Honeydew"  
          ItemsSource="{Binding  
Source={StaticResource Na}}"/>
```



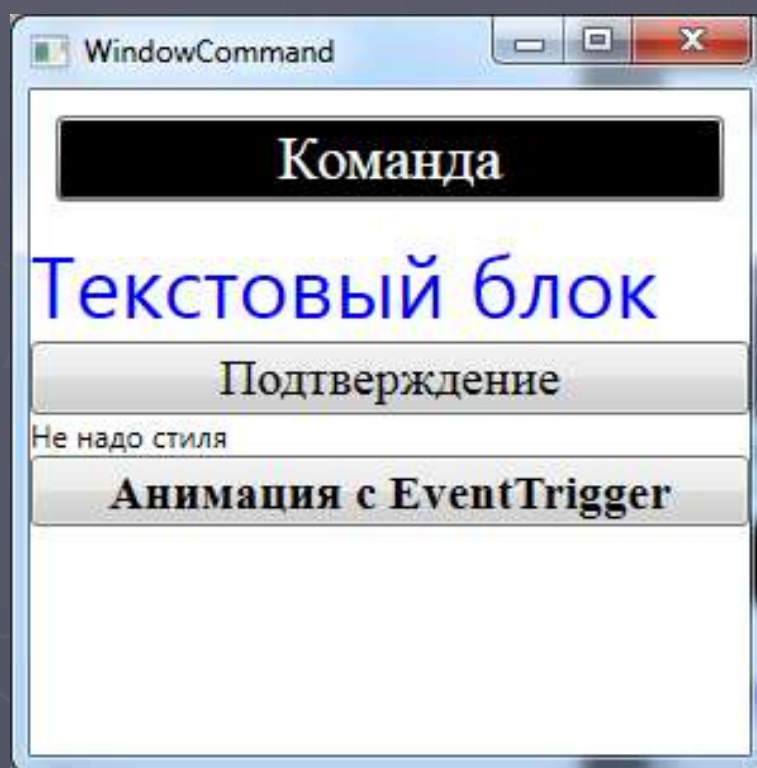
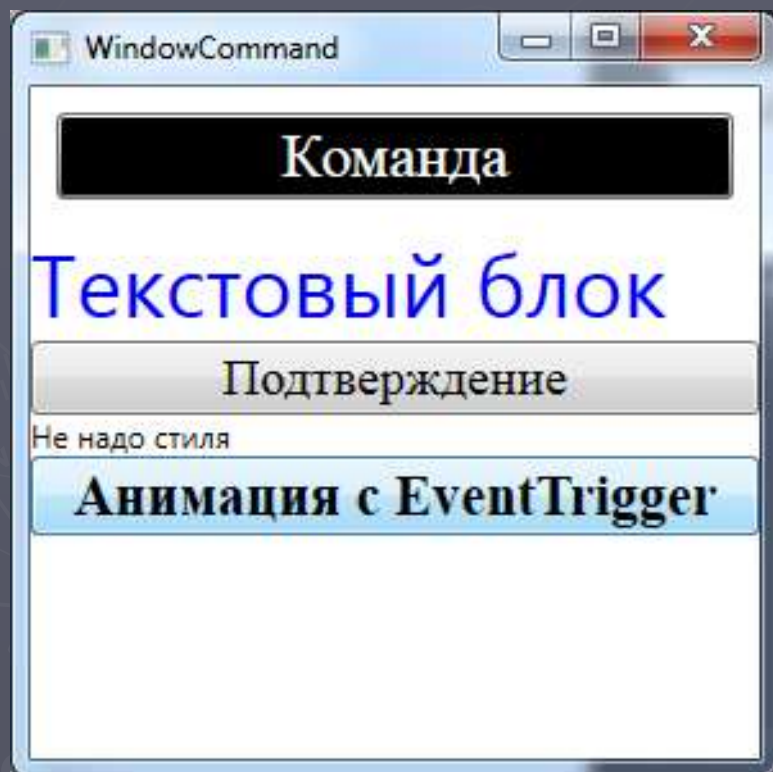
# EventTrigger

```
<Style x:Key="EventAnimation">
  <!--Стили-->
  <Style.Setters>
    <Setter Property="Control.FontFamily" Value="Times New Roman" />
    <Setter Property="Control.FontSize" Value="18" />
    <Setter Property="Control.FontWeight" Value="Bold" />
  </Style.Setters>
  <!--Триггеры
EventTrigger - ожидает определенного события-->
  <Style.Triggers>
    <!--Действие на событие MouseEnter-->
    <EventTrigger RoutedEvent="Mouse.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation
              Duration="0:0:0.3"
              Storyboard.TargetProperty="FontSize"
              To="22" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
    <!--Действие на событие MouseLeave-->
    <EventTrigger RoutedEvent="Mouse.MouseLeave">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation
              Duration="0:0:3"
              Storyboard.TargetProperty="FontSize" To="18" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Style.Triggers>
```

```
<Button Style="{StaticResource EventAnimation}">
```

Анимация с EventTrigger

```
</Button>
```



# Темы

## ► объединение стилей

1) Создается файл словаря ресурсов - *nigth.xaml*, и определяется некоторый набор ресурсов:

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfAppDemo">

    <Style x:Key="TextBlockNigth" TargetType = "TextBlock">
        <Setter Property="Background" Value="Black" />
        <Setter Property="Foreground" Value="Gray" />
    </Style>
    <Style x:Key="WindowNigth" TargetType="Window">
        <Setter Property="Background" Value="Black" />
    </Style>
    <Style x:Key="ButtonNigth" TargetType="Button">
        <Setter Property="Background" Value="Black" />
        <Setter Property="Foreground" Value="Gray" />
        <Setter Property="BorderBrush" Value="Gray" />
    </Style>
</ResourceDictionary>
```

day  
night

## ► 2) Выводим ЭУ для смены тем и меняем

```
public Window6()
{
    InitializeComponent();
    List<string> styles = new List<string> { "day", "night" };
    Theme.SelectionChanged += ThemeChange;
    Theme.ItemsSource = styles;
    Theme.SelectedItem = "nighth";
}

private void ThemeChange(object sender, SelectionChangedEventArgs e)
{
    string style = Theme.SelectedItem as string;
    // определяем путь к файлу ресурсов
    var uri = new Uri(style + ".xaml", UriKind.Relative);
    // загружаем словарь ресурсов
    ResourceDictionary resourceDict = Application.LoadComponent(uri) as
ResourceDictionary;
    // очищаем коллекцию ресурсов приложения
    Application.Current.Resources.Clear();
    // добавляем загруженный словарь ресурсов
    Application.Current.Resources.MergedDictionaries.Add(resourceDict);
}
}
```

# Тест

