

# 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

*уведомление об изменении и динамическое разрешение значений*

# ► 1) Определение свойства зависимостей у TextBlock


Класс унаследован от DependencyObject



```
graph TD;
    A[Класс унаследован от DependencyObject] --> B[FrameworkElement];
```


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

статическое поле →  
свойство должно быть  
доступно другим  
классам



```
graph TD;
    C[статическое поле → свойство должно быть доступно другим классам] --> D[public static readonly DependencyProperty TextProperty;];
```


Соглашение по именованию → имя  
обычного свойства + Property в конце



```
graph TD;
    E[Соглашение по именованию → имя обычного свойства + Property в конце] --> F[TextProperty];
```

## ► 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
{
```

```
    // СВОЙСТВО ЗАВИСИМОСТЕЙ
```

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

```
    static TextBlock()
    {
```

```
        // Регистрация свойства
```

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

```
            "Text",
```

```
            typeof (string),
```

```
            typeof (TextBlock),
```

```
            new FrameworkPropertyMetadata(
```

```
                string.Empty,
```

```
                FrameworkPropertyMetadataOptions.AffectsMeasure |
```

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

```
        // ...
```

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

```
    public string Text
```

```
    {
```

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

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

```
    }
```

```
}
```

1) Должен наследоваться от DependencyObject

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

3) зарегистрировано в static construct

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 свойств 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 Pasport()
{
    FrameworkPropertyMetadata metadata =
        new FrameworkPropertyMetadata();
    NumberProperty = DependencyProperty.Register("Number",
        typeof(string),
        typeof(Pasport), 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);
```

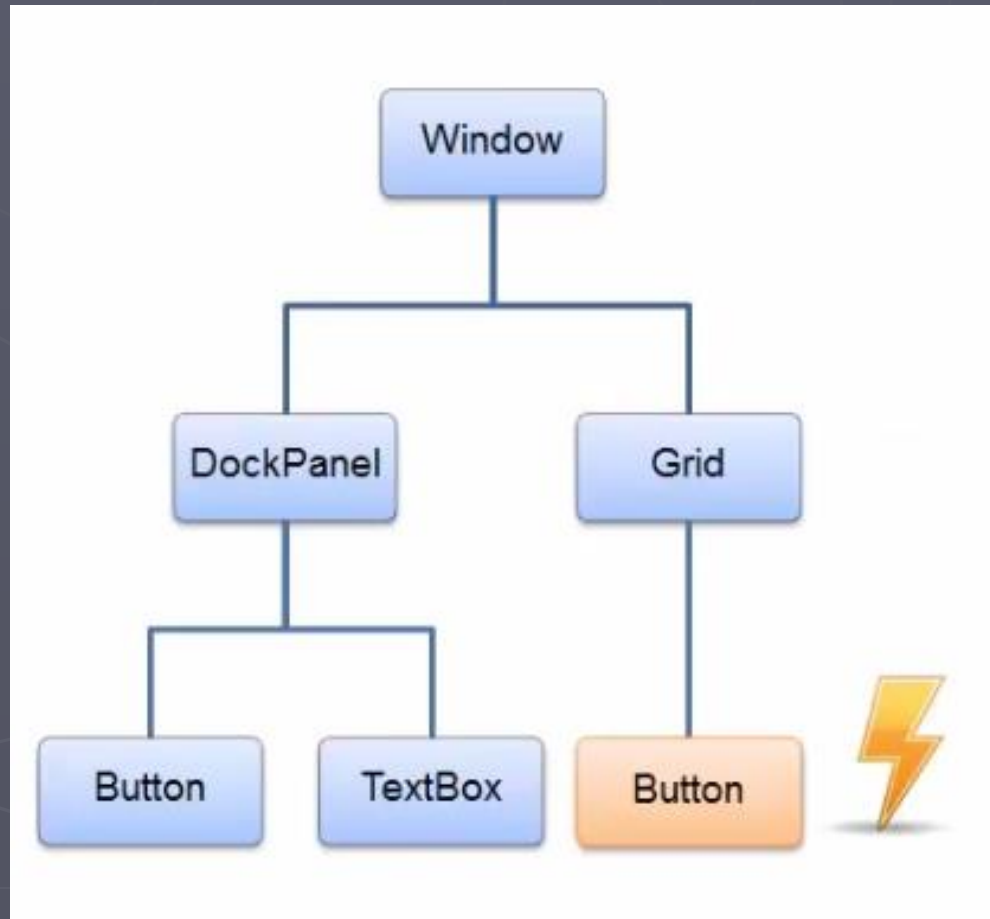
# Отработка событий

- 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).

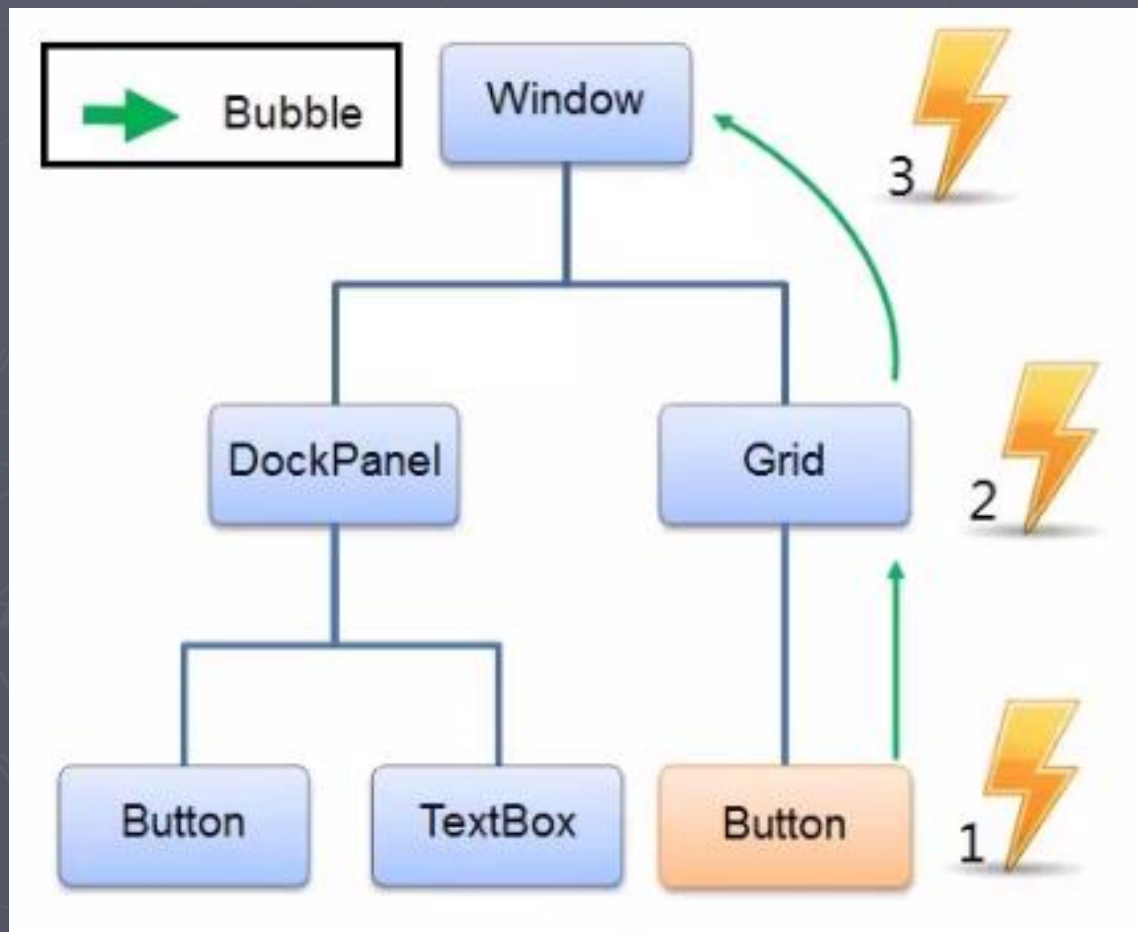
- Direct (=WinForms)
- Tunneling - туннельное
- Bubbling - поднимающееся

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

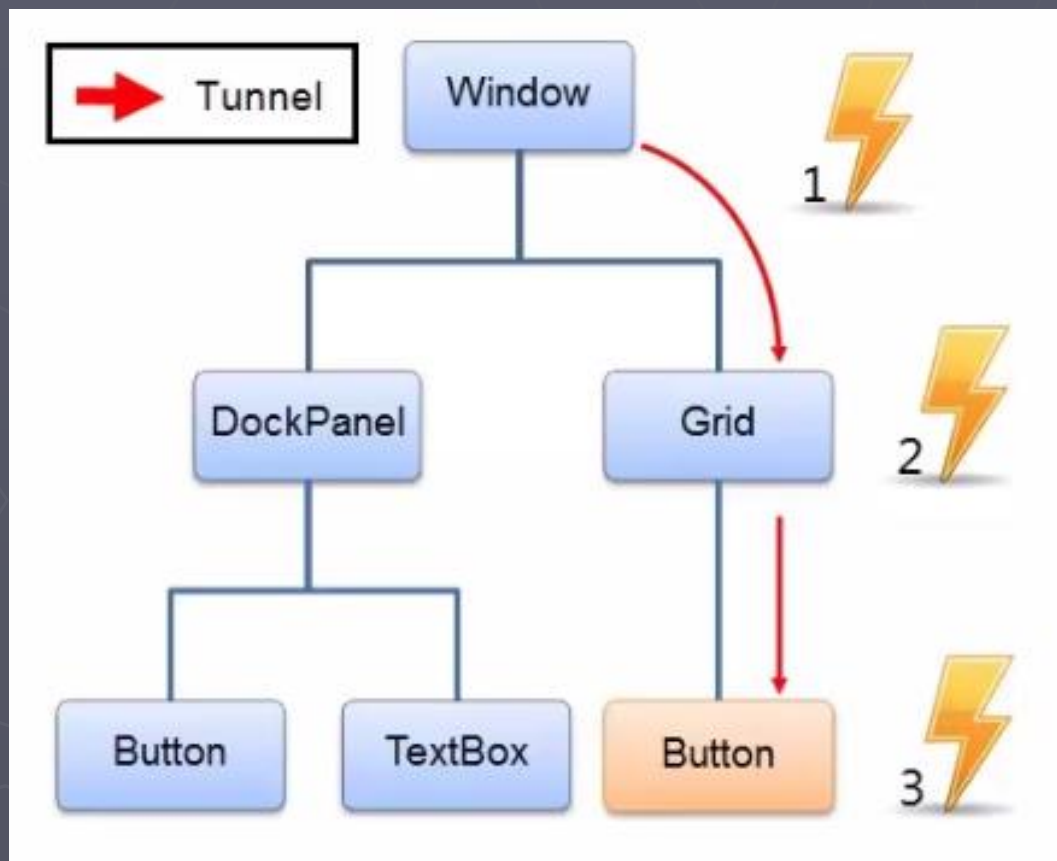




- **Поднимающиеся (bubbling events)** - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру.



**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)
{
}
}
```

**KeyDown**

Поднимающееся

Возникает при нажатии клавиши

**PreviewKeyDown**

Туннельное

Возникает при нажатии клавиши

**GotMouseCapture**

Поднимающееся

**LostMouseCapture**

Поднимающееся

**MouseEnter**

Прямое

**MouseLeave**

Прямое

**MouseLeftButtonDown**

Поднимающееся

**PreviewMouseLeftButtonDown**

Туннельное

**MouseLeftButtonUp**

Поднимающееся

**PreviewMouseLeftButtonUp**

Туннельное

**MouseRightButtonDown**

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));
```

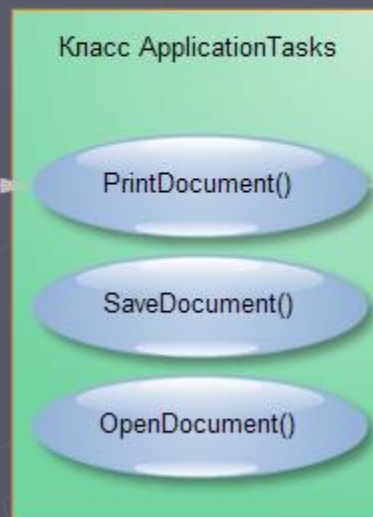
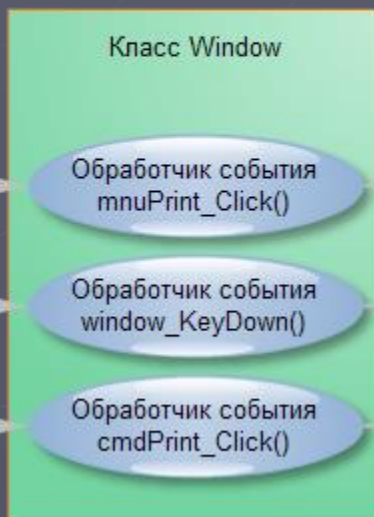
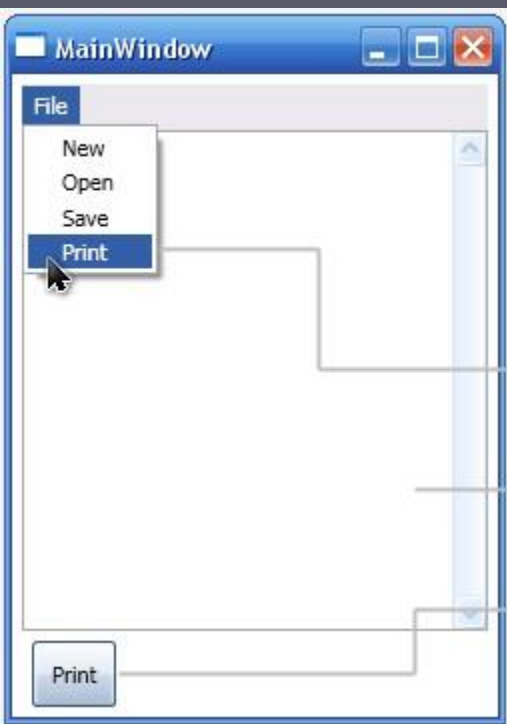
# Команда

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

Назначение:

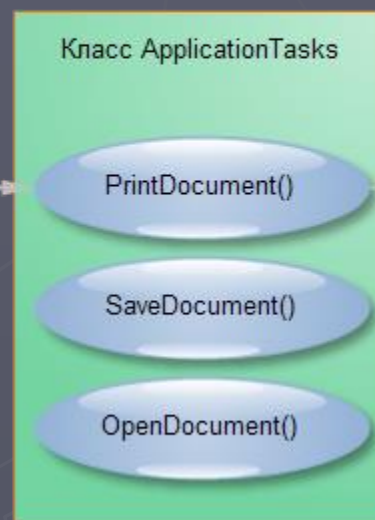
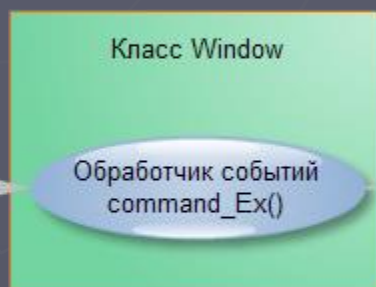
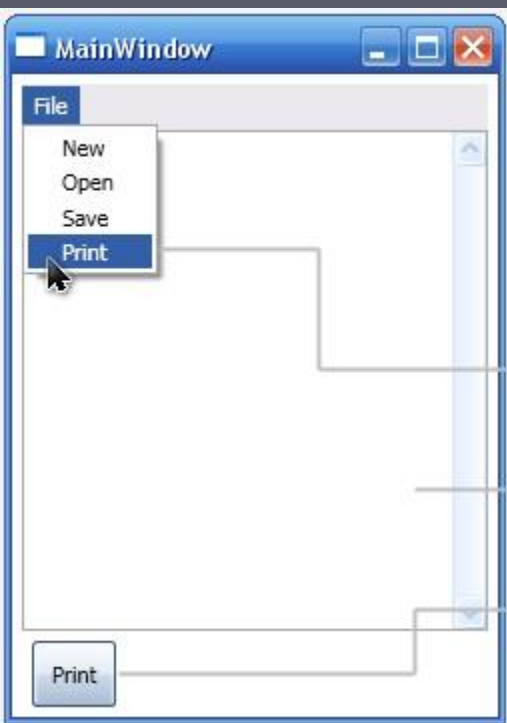
- использовать одну и ту же команду для нескольких ЭУ
- абстрагировать набор действий от конкретных событий конкретных элементов

# Модель обработки событий



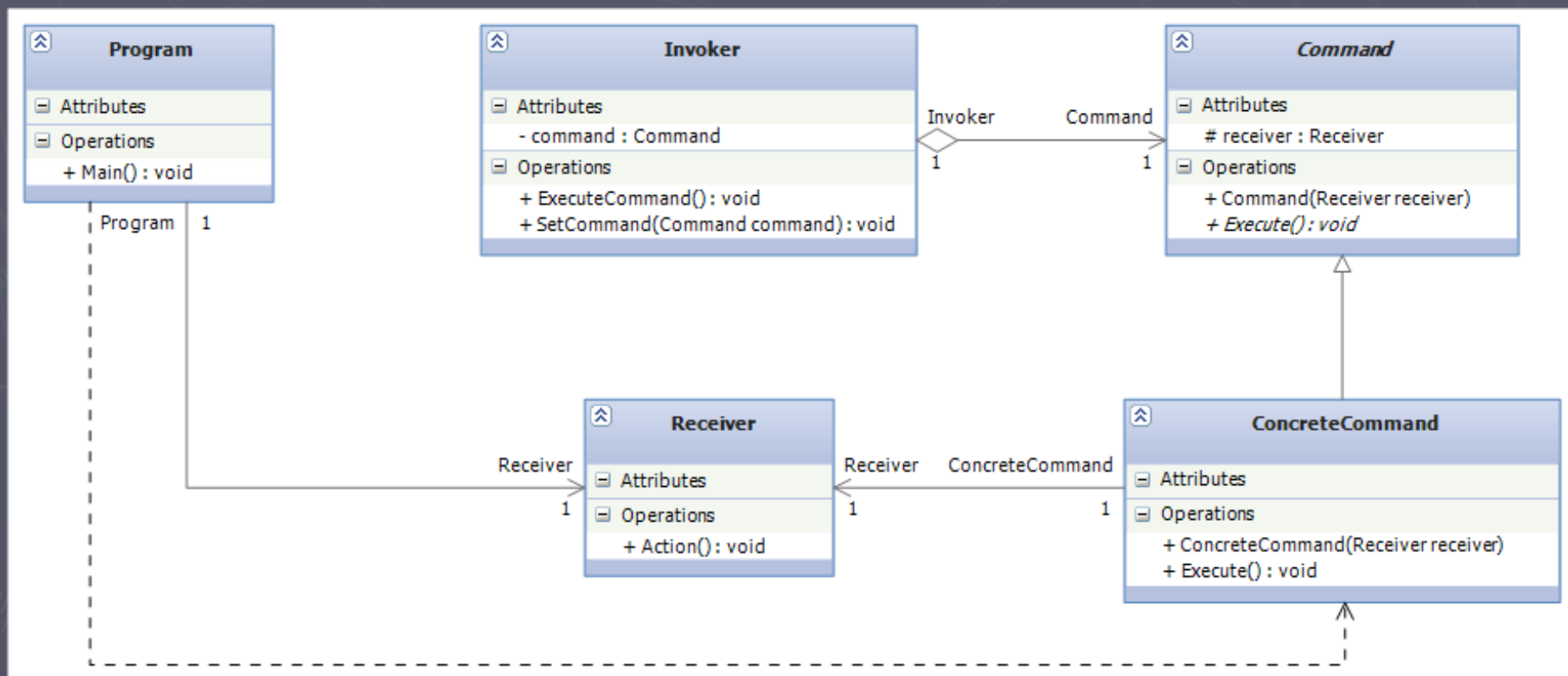


# Модель команд



# 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();

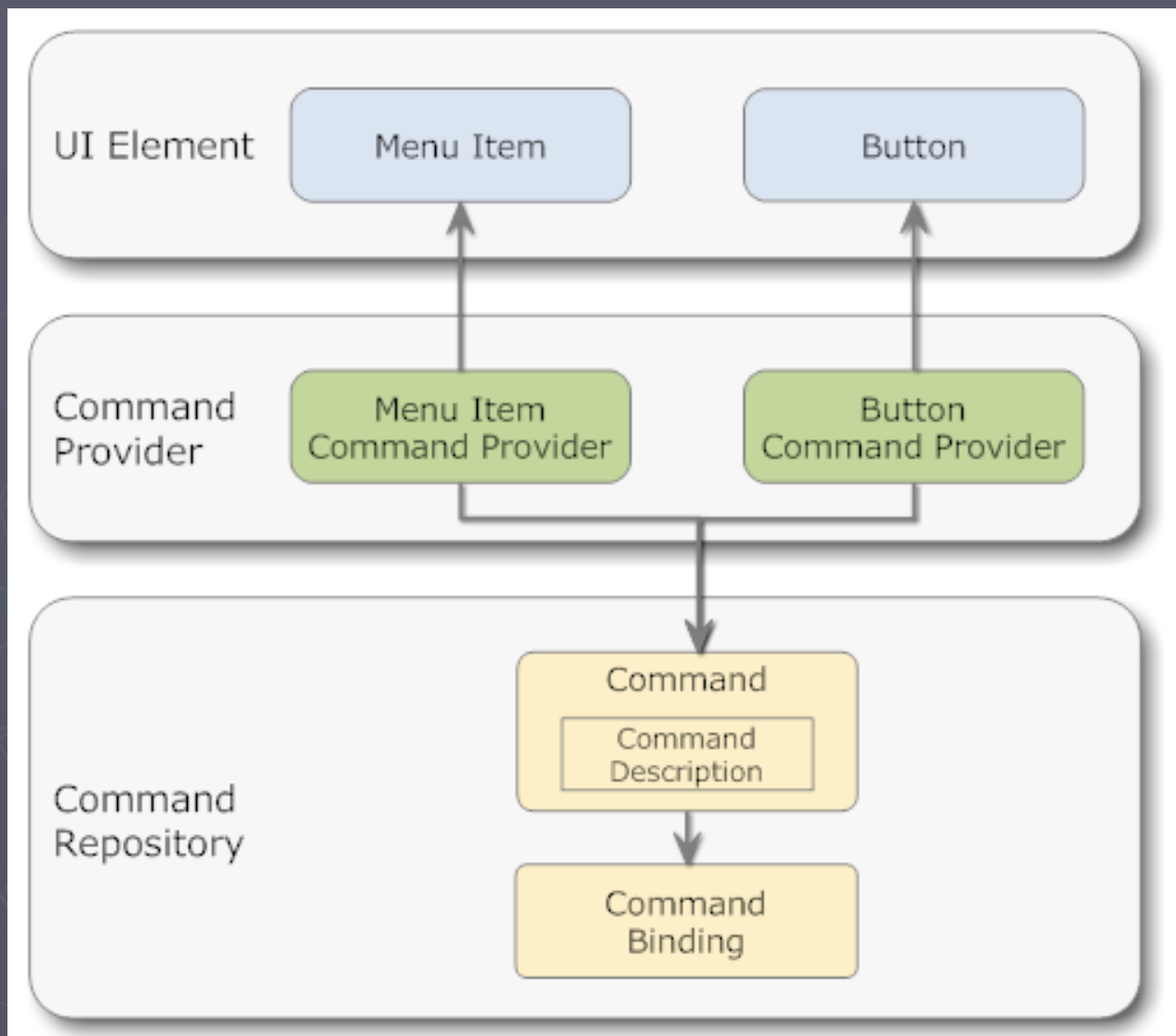
    invoker.StoreCommand(command);
    invoker.ExecuteCommand();
}
```

# Модель команд в WPF

- ▶ **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

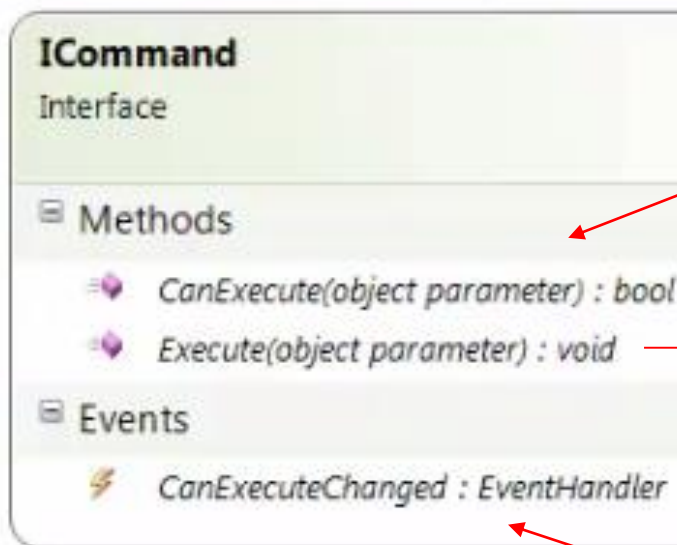


# Модель команд в WPF



Все команды реализуют интерфейс

`System.Windows.Input.ICommand:`



возвращает информацию о состоянии команды, а именно — значение `true`, если она включена, и `false`, если она отключена.

процесса, который, заканчивается возбуждением события

вызывается при изменении состояния команды

реализован встроенным классом, который является базовым для всех встроенных команд

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

# Встроенные команды

**RoutedCommand**



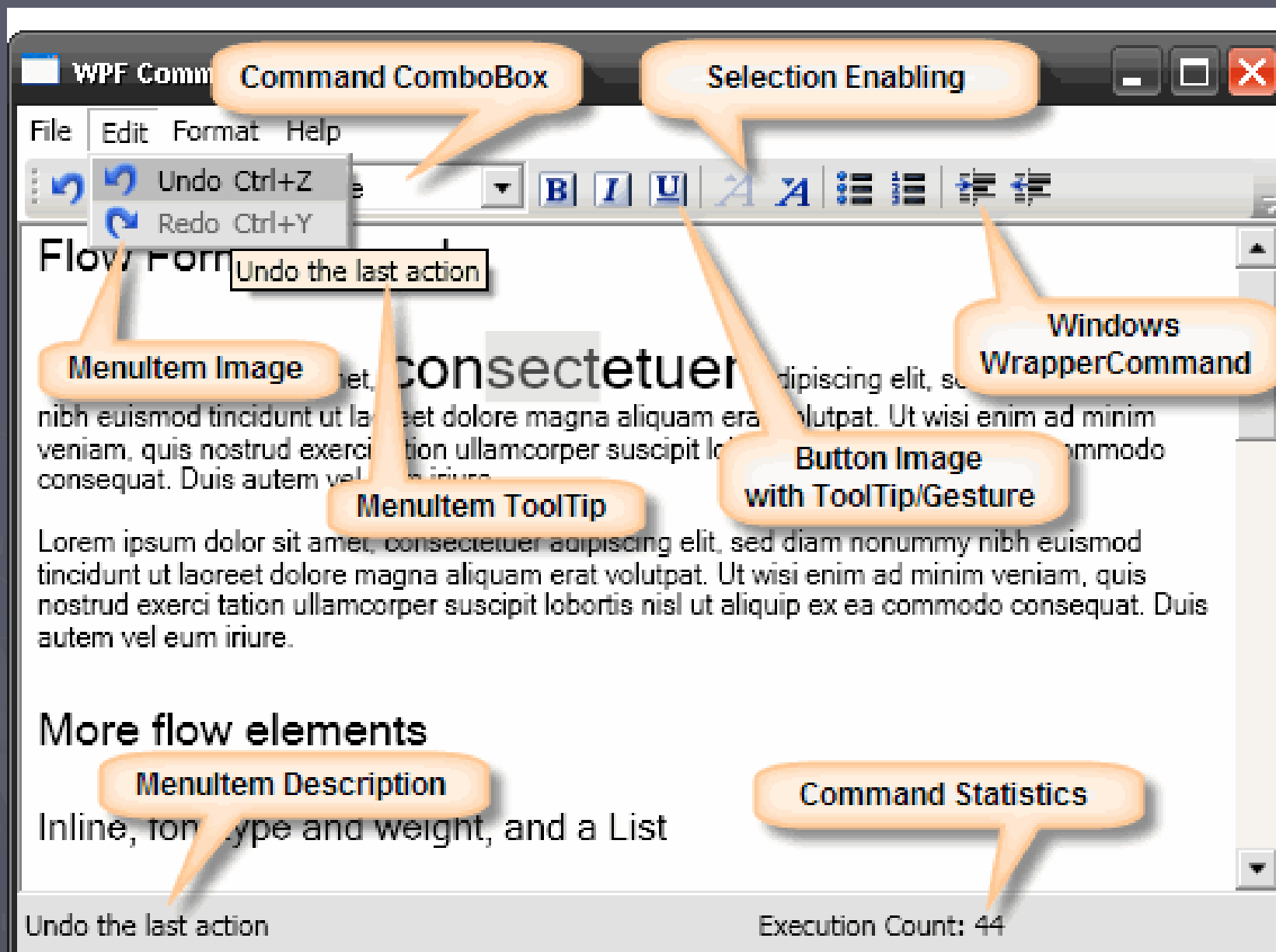
**RoutedUICommand**

- ▶ 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

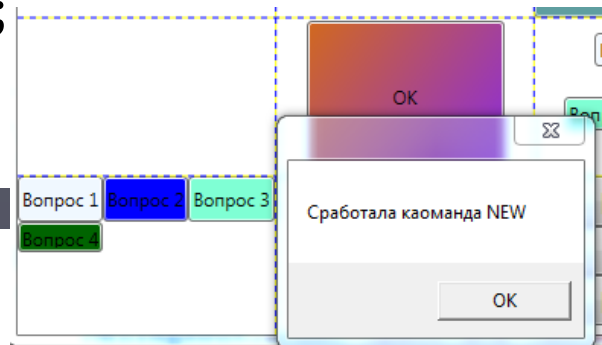
```
public MainWindow()
{
    InitializeComponent();
    CommandBinding binding =
        new CommandBinding(ApplicationCommands.New);
    binding.Executed+=
        new ExecutedRoutedEventHandler(binding_exec);
    // добавляем привязку к коллекции привязок элемента Button
    ButtonT.CommandBindings.Add(binding);
    // или
    this.CommandBindings.Add(binding);
}
```

Способ

Область, на которую распространяется действие команды

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

Действие





```
<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"
```

```
CanExecute="CommandBinding_OnCanExecute"
```

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

```
</CommandBinding>
```

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

```
</Window>
```

New

AlignCenter

AlignJustify

AlignLeft

AlignRight

Backspace

BoostBass

BrowseBack

Property System.Windows.Input.Route

Получает значение, представляюще

ab

Demo\bin\Debug\WpfAppDemo.exe

данные из: сборки

начата: проект: WpfAppDemo, Конфигурация: De

C:\NATA\ITA\лекции\ООП\2\2016\лекции\Proj1

```
<Window.CommandBindings>
```

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

```
CanExecute="CommandBinding_OnCanExecute"
```

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

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

Маршрутизация  
команды к  
контейнеру

<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

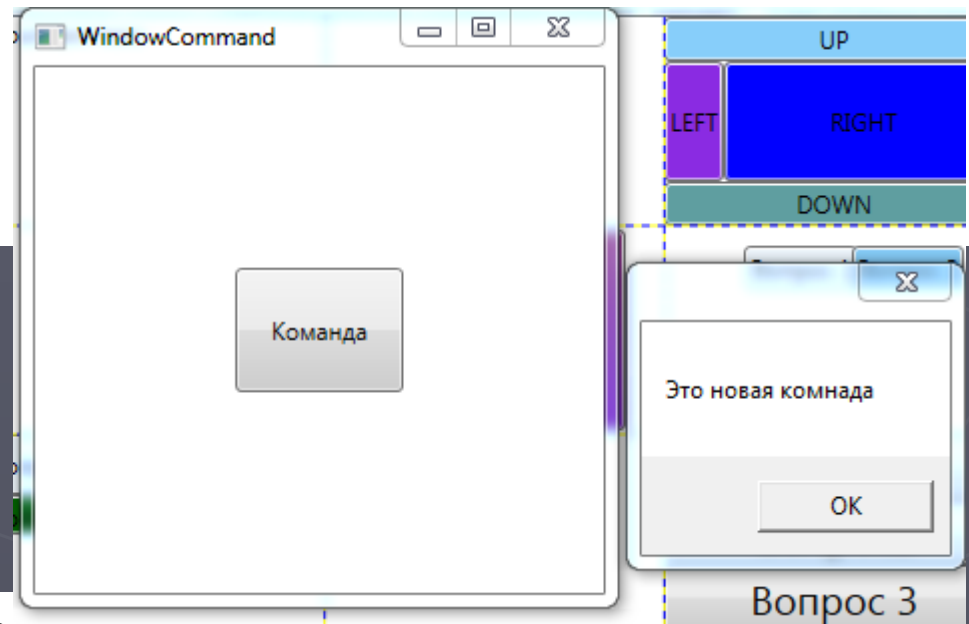
Горячие клавиши

```
<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 команда");
}
```

# 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");
    }
}
```

# Привязка Binding

Источник

Приемник

в случае модификации  
приемник также будет  
модифицирован

создает привязку к  
определенному свойству  
объекта-источника



```
<TextBlock x:Name="TextBlock1"
```

```
Text="{Binding ElementName=Button1,Path=Content}"
```

```
Height="30" />
```

Целевой объект привязки

Источник привязки

```
{Binding ElementName=Имя_объекта-источника, Path=Свойство_объекта-источника}
```

```
Binding binding = new Binding();
```





```
binding.ElementName = "TextBox1"; // источник  
binding.Path = new PropertyPath("Text"); // свойство  
TextBlock1.SetBinding(TextBlock.TextProperty, binding);  
// установка привязки
```

- ▶ **ElementName**: имя элемента, к которому создается привязка
- ▶ **IsAsync**: асинхронный режим (по умолчанию равно False)
- ▶ **Mode**: режим привязки
- ▶ **Path**: ссылка на свойство объекта, к которому идет привязка
- ▶ **TargetNullValue**: устанавливает значение по умолчанию, если привязанное свойство источника привязки имеет значение null

TargetNullValue="по умолчанию"

- ▶ **RelativeSource**: создает привязку относительно текущего объекта
- ▶ **Source**: указывает на объект-источник, если он не является элементом управления.
- ▶ **XPath**: используется вместо свойства path для указания пути к xml-данным

# Направление привязок Mode

- ▶ **OneWay**—целевое свойство обновляется при изменении значения источника.  

- ▶ **TwoWay**—при изменении источника меняется целевое свойство и наоборот.  

- ▶ **OneTime**—целевое свойство устанавливается изначально на основе свойства источника и с этого момента изменения значений в источнике игнорируются.  

- ▶ **OneWayToSource**—свойство источника обновляется при изменении целевого свойства.  

- ▶ **Default**—тип привязки зависит от целевого свойства. `TextBox.Text`—`TwoWay` для всех прочих `OneWay`.



```
<TextBlock x:Name="TextBlock"
           Text="{Binding
           ElementName=Button1,
           Path=Content,
           Mode=OneWay} "
           Height="30" />
```

# Обновление привязки

## Значения перечисления UpdateSourceTrigger

- ▶ **PropertyChanged**—обновление происходят сразу после изменения значения свойства.
- ▶ **LostFocus**—обновление происходит после изменения значения и потери фокуса.
- ▶ **Explicit**—обновления происходят после вызова метода `BindingExpression.UpdateSource()`;
- ▶ **Default**—Для большинства свойств значение `PropertyChanged` для `TextBox.Text`-`LostFocus`

обновлении источника привязки после изменения приемника в режимах `OneWayToSource` или `TwoWay`

```
Text="{Binding ElementName=textBox1,  
    Path=Text,  
    Mode=TwoWay,  
    UpdateSourceTrigger=PropertyChanged}"  
/>
```

# Привязка к объектам

- **Source**— позволяет установить привязку даже к тем объектам, которые не являются элементами управления WPF.

```
< TextBlock x:Name="nameTextBlock"  
Text="{Binding Source={StaticResource Student}, Path=FName}"  
Foreground="White"/>
```

- **DataContext**—указание источника для группы элементов управления.

```
<Grid DataContext="{StaticResource Student}" >  
    <TextBlock Text="Студент" />  
    <TextBlock Text="{Binding FName}" />  
    <TextBlock Text="{Binding Number}" />
```

вложенные элементы могут использовать объект `Binding` для привязки к конкретным свойствам этого контекста

# RelativeSource

**RelativeSource** – позволяет создать привязку относительно элемента-источника, который связан какими-нибудь отношениями с элементом-приемником или на другой элемент вверх по дереву.

- `Self`: привязка осуществляется к свойству этого же элемента. То есть элемент-источник привязки в то же время является и приемником привязки.
- `FindAncestor`: привязка осуществляется к свойству элемента-контейнера.

```
<TextBox Text="{Binding  
    RelativeSource={RelativeSource Mode=Self},  
    Path=Background,  
    Mode=TwoWay,  
    UpdateSourceTrigger=PropertyChanged}" />
```

# INotifyPropertyChanged

Для реализации механизма привязки, надо реализовать интерфейс

```
class Student : INotifyPropertyChanged
```

```
{
```

```
    private string name;
```

```
    public string Name
```

```
    {
```

```
        get { return name; }
```

```
        set
```

```
        {
```

```
            name = value;
```

```
            OnPropertyChanged("Name");
```

```
        }
```

```
    }
```

```
    public event PropertyChangedEventHandler PropertyChanged;
```

```
    public void OnPropertyChanged([CallerMemberName]string prop = "")
```

```
    {
```

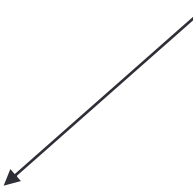
```
        if (PropertyChanged != null)
```

```
            PropertyChanged(this, new PropertyChangedEventArgs(prop));
```

```
    }
```

```
}
```

Когда объект класса изменяет значение свойства, то он через событие PropertyChanged извещает систему об изменении свойства. А система обновляет все привязанные объекты.



# Провайдеры данных. ObjectDataProvider

- ▶ ПОЗВОЛЯЮТ СВЯЗЫВАТЬ ИСТОЧНИКИ ДАННЫХ И ЭЛЕМЕНТЫ ИНТЕРФЕЙСА.
- ▶ ObjectDataProvider (для работы с объектами)
- ▶ XmlDataProvider (для работы с xml-файлами)