

# Упражнение №3 по ПС

## Софтуерен шаблон Model-View-ViewModel (MVVM).

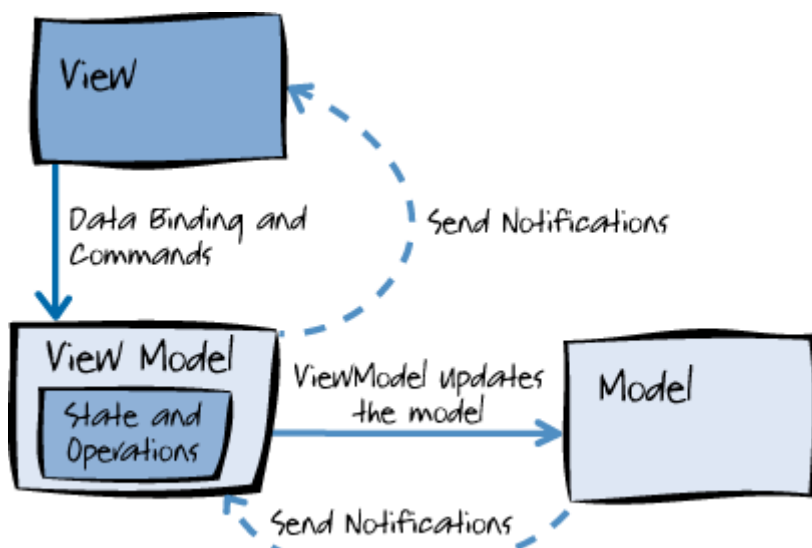
Технологии като Windows Forms, WPF, Silverlight и Windows Phone предоставят по подразбиране среда за разработка, която подтиква потребителя да провлачва контроли от тентата с инструменти върху прозореца за дизайн и след това да пише код в code-behind файла. Такива приложения често се разрастват, което води до проблеми при тяхното модифициране и поддръжка. Това включва тясната връзка (tight coupling) между UI контролите и бизнес логиката. Това от своя страна увеличава разходите за модификации по UI и усложнява тестването.

Основната мотивация за имплементирането на приложение, използвайки MVVM шаблона, е:

1. Предвижда чисто разделение между логиката на приложението и на потребителския интерфейс. Това прави приложението по-лесно за тестване, поддръжка и развитие. Той подобрява възможностите за преизползване на код.
2. Това е естествения модел за XAML платформи. Благодарение на широките възможности за data binding и dependency свойствата, се осигуряват средства за свързване на UI към view model.
3. Позволява разделението на работата между разработчика и дизайнер.
4. Отделянето на логиката в отделни класове, позволява по-лесното ѝ тестване.

## MVVM Шаблона

Шаблонът Model-View-ViewModel (MVVM) може да се използва при всички XAML платформи. Неговата цел е да се осигури чисто разделяне на връзките между контролите от потребителския интерфейс и тяхната логика. В MVVM шаблона има 3 основни компонента: model (модел), view (изглед) и view model (изглед-модел). Всеки от тях изпълнява различна и отделна роля. Фигура 1 показва връзките между трите компонента.



Фиг. 1. Шаблон MVVM

Компонентите са отделени един от друг, което от своя страна позволява:

- Размяна/подмяна на компоненти
- Вътрешната имплементация на компонент да бъде променена без това да засегне останалите
- Работата по компонентите да се извършва независимо
- Индивидуално тестване на компонентите

За да се разбере по-добре какви са отговорностите на трите компонента е важно да се схване и как те си взаимодействат помежду си.

Слоят **View**, бидейки най-високо ниво в архитектурата (отварям една скоба за многослойните архитектури... история, но от нея и трислойната архитектура произлизат съвременните шаблони), знае всичко за **ViewModel**, а **ViewModel** от своя страна знае за **Model**. Но **Model** не е наясно за **ViewModel** и **ViewModel** не знае за **View**. **ViewModel** изолира класовете на **View** от тези на **Model** и позволява на **Model** да се развива независимо от **View**.

## View

Слоят **View** е отговорен за създаването на структурата, подредбата и външния вид на това, което вижда потребителя на екрана. **View** се дефинира изцяло с XAML, с ограничен code-behind, който НЕ съдържа бизнес логика.

Едно **View** може да има собствен **ViewModel** или да наследява този на своя родител. Всяко **View** получава данните си от собствения **ViewModel** чрез връзки (bindings) или чрез извикване на методи от **ViewModel**. По време на изпълнение на програмата **View** се променя, когато UI отговорят на **ViewModel** свойства, повдигащия събития за известяване за промяна (change notification events).

Тук има различни възможности за изпълнение на код в **ViewModel** в отговор на взаимодействията във **View**, като например натискане на бутон или избор на елемент. Ако контролата е източник на команда (Command Source), свойството ѝ **Command** може да бъде свързано с **ICommand** свойство на **ViewModel**. Когато се извика (invoke) командата на контролата се изпълнява кода във **ViewModel**. Освен командите, към обект на **View** може да бъде прикачено и поведение (behavior) и може да се „слуша“ за извикване на команда или възникване на събитие. В отговор събитието може да извика **ICommand** на **ViewModel** или негов метод.

## Model

Модела в MVVM е имплементация на основния модел на приложението, който включва модел на данните, заедно с бизнес и валидационната логика. Примери за обекти на модела са хранилища (repositories), бизнес обекти, data transfer objects (DTOs), Plain Old CLR Objects (POCOs) и генерирани entity и proxy обекти.

## ViewModel

Слоят **ViewModel** работи като посредник между **View** и **Model** и е отговорен за обработката на логиката на изгледа. Обикновено **ViewModel** взаимодейства с **Model** като извиква методи от неговите класове. След това **ViewModel** предоставя данни от модела в подходящ за изгледа формат. При това **ViewModel** може да преформатира извлечените от модела данни, като

например ги опрости и ги направи по-лесни за работа в изгледа. Той също така предоставя и имплементация на команди, които потребителя на приложението иницира в изгледа. Например, когато потребителя натисне върху бутон в UI, това действие може да иницира команда във **ViewModel**. **ViewModel** може да бъде отговорен също за дефинирането на промени в логически състояние, което засяга някои аспекти на визуализацията във **View**, като например като например индикация, че някоя операция е предстояща.

За да може **ViewModel** да участва в двупосочна връзка на данните (two-way data binding) с **View**, неговите свойства трябва да предизвикват събитието **PropertyChanged**.

Класовете на **ViewModel** изпълняват изискването за имплементация на интерфейса [INotifyPropertyChanged](#) и предизвикването на събитието **PropertyChanged** при промяна на свойство. Така слушателите могат да отговорят адекватно на тези промени, когато настъпят.

Когато е необходима работа с колекции, е предоставен удобния за **View** [System.Collections.ObjectModel.ObservableCollection<T>](#). Тази колекция имплементира нотификация за промяна на колекцията, облекчавайки разработчика от необходимостта да имплементира интерфейса [INotifyCollectionChanged](#) за колекциите си.

## Свързване на ViewModels към Views

Съществуват много подходи за свързването на един **ViewModel** с **View**, като например директна връзка или подход базиран на контейнер. Всички обаче споделят една и съща цел, която е на свойството **DataContext** на **View** да се присвои конкретен **ViewModel**. Тази връзка може да се направи както в code-behind файла, така и във самото **View**.

### Code-Behind

Едно **View** може да има в своя code-behind файл код, който служи за задаване на **ViewModel** като стойност на **DataContext** свойството. Това може да е просто като инстанцирането на нов **ViewModel** и присвояването му на **DataContext** свойството на **View**, или с инжектиране на **ViewModel** във **View**, използвайки inversion-of-control контейнер. При всички случаи осъществяването на тази връзка в code-behind файла не е препоръчителна, тъй като може да доведе до проблеми на дизайнерите в Visual Studio и Microsoft Expression Blend®.

### View

Ако **ViewModel** няма параметри в конструктора си, то той може да бъде инстанциран във **View** от неговия **DataContext**. Често използван подход за постигането на това е използването на **ViewModel** locator. Това е ресурс, който показва **ViewModel**-ите на приложението като свойства, към които отделните изгледи могат да се свържат. Този подход предполага, че приложението има един клас, отговарящ за връзката между **ViewModel**-и и **View**-та.

## Пример 1.

Следвайки следващите стъпки ще създадем приложение, ползващо MVVM шаблона. То ще включва най-важните компоненти (без командите), така че да деменстрира особеностите и предимствата на този архитектурен шаблон.

1. Създайте нов WPF проект с име **EasyMVVM**.

## Създаване на модела (Model)

Първо ще разработим модела на данните на приложението.

2. Към проекта (десен бутон върху името на проекта в Solution Explorer) добавете нов файл от тип Code (Add → New Item... → Code) и го кръстете **Model.cs**

Класът на модела съдържа колекция от стрингове. Ще използваме спеманатия по-горе [ObservableCollection](#) клас, с който не е нужно да пишем сами имплементацията на интерфейса [INotifyCollectionChanged](#). За целта ни е необходима библиотеката [System.Collections.ObjectModel](#). Данните в колекцията ще се достъпват посредством публичен метод [GetData\(\)](#), който за момента ще реализираме така, е да пълни колекцията ни с някакви dummy данни и да ги връща. При реална реализация той най-често ще изпълнява заявки към база данни и ще черпи колекцията от там.

3. Ето и предлаганата реализация на нашия модел. Сложете във новосъздадения файл следния код.

```
using System;
using System.Collections.ObjectModel;

namespace EasyMVVM
{
    //The model is a class which the ViewModel knows and uses to get data...
    public class Model
    {
        // Using a private data store is a good idea
        private ObservableCollection<string> _data = new ObservableCollection<string>();
        public ObservableCollection<string> GetData()
        {
            // these steps represent the same data to be returned each time GetData is called.
            // typically you'd query a database or put other buisness logic here
            _data.Add("First Entry");
            _data.Add("Second Entry");
            _data.Add("Third Entry");
            return _data;
        }
    }
}
```

## Създаване на ViewModel

Изготвяния ViewModel ще поставим също в отделен клас файл.

4. Към проекта (десен бутон върху името на проекта в Solution Explorer) добавете нов файл от тип Class (Add → New Item... → Class) и го кръстете **MainWindowVM.cs**

5. Добавете наследяване на интерфейсите [DependencyObject](#), [INotifyPropertyChanged](#). Уверете се че модификатора за достъп до класа е public и сте добавили необходимите using клаузи.

6. Добавете private поле, което да съхранява данните получени от модела.

```
//set up a private class varialbe that holds the value of the _Backing Property
private ObservableCollection<string> _BackingProperty;
```

7. Направете публично свойство за достъп до полето от точка 6.

```
//This is the publically viewable Property for this class
public ObservableCollection<string> BoundProperty
{
    get { return _BackingProperty; }
    set { _BackingProperty = value;
        PropertyChanged("BoundProperty");
    }
}
```

\* Вече сте говорили за MVVM и сте разгледали свойствата в упражнението 2 по ПМУ (<http://fksu1409-5.tu-sofia.bg/mobile-devices/exercises/wikis/exercise2>, ако не помните прегледайте отново).

8. Ще реализираме заложения в предната точка метод **PropertyChanged()**. Той приема като параметър името на промененото свойство, а в имплементацията си посредством специализирания **PropertyChangedEventHandler** се предизвиква събитие за уведомление за промяната. Ето и кода, добавете го в класа:

```
//This event will be fired to notify any listeners of a change in a property value.
public event PropertyChangedEventHandler PropertyChanged;

//Tell WPF Binding that this property value has changed
public void PropertyChanged(String propertyName)
{
    //Did WPF registerd to listen to this event
    if (PropertyChanged != null)
    {
        //Tell WPF that this property changed
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

9. Добавете конструктор на класа и в него създайте обект от Model и на свойството BoundProperty присвоете резултата от изпълнението на метода GetData() за обекта.

```
Model m = new Model();
BoundProperty = m.GetData();
```

С това връзката между **ViewModel** и **Model** е направена.

### Създаване на изгледа (View)

Последната ни задача, за да завършим MVVM модела, е да изградим и изгледа **View** и неговата връзка към **ViewModel**. За целта ще добавим някои неща във вече съществуващия прозорец **MainWindow.xaml**.

10. Добавете връзка към namespace-а на приложението в xaml файла, като добавите следния атрибут в отварящия таг на Window:

```
xmlns:vm="clr-namespace:EasyMVVM"
```

Това ще направи видими публичните класове в EasyMVVM namespace и ще може да ги ползваме в xaml кода с префикса **vm**.

11. В Grid елемента добавете следния ресурс, правещ връзката с **ViewModel**.

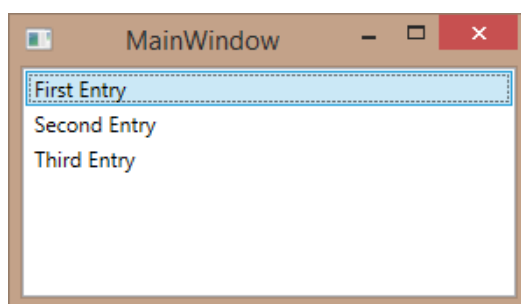
```
<Grid.Resources>
  <vm:MainWindowVM x:Key="vm"></vm:MainWindowVM>
</Grid.Resources>
```

12. Добавете един **ListBox**, който ще визуализира данните получени от модела, през ViewModel.

```
<ListBox ItemsSource="{Binding Source={StaticResource vm}, Path=BoundProperty}"/>
```

Атрибута Source задава обекта, който се използва като източник за получаване на данните. В кода горе е използван и най-простия вариант за стойност на Path атрибута. Тази стойност е името на свойство на обекта-източник, който се използва за binding, и се задава като Path=PropertyName. В конкретния случай като източник имаме **обект vm** от класа MainWindowVM и Path задава свойството BoundProperty.

13. Вече може да стартирате и да тествате приложението. То трябва да изглежда по следния начин.



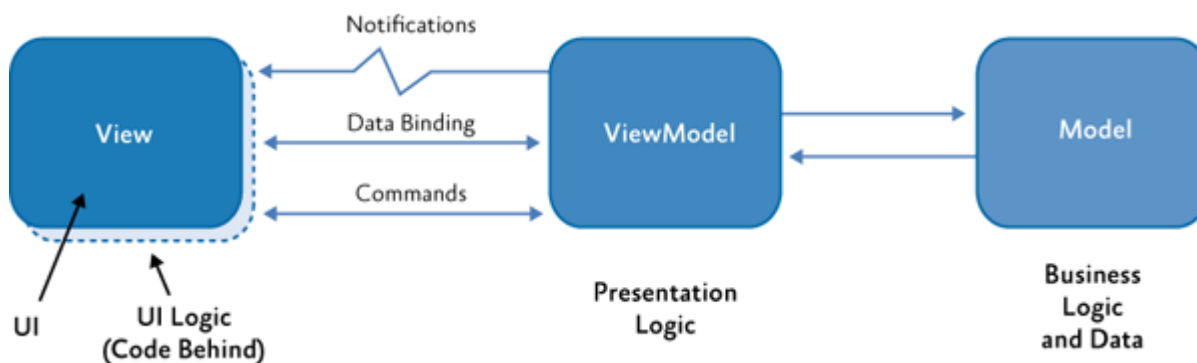
Какво не е наред в горния пример?!

- Класовете от различните „слоеве“, трябва да се отделят в отделни namespace пространства, за да нямат директен достъп един до друг, освен ако изрично не им разрешим.
- Добре е да ползваме команди, за да „развържем“ действията от eventhandlers.

## Пример 2. Команди

В този пример ще се запознаем с WPF Commands, които са много удобни при реализацията на MVVM шаблона. Ще разгледаме как изглежда „знае“ за неговия ViewModel и как изглежда извиква операциите на ViewModel, което се случва с използването на командите в WPF.

Нека да разгледаме отново архитектурата на MVVM.



Фиг. 2. MVVM архитектура

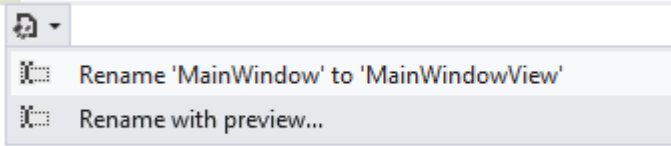
Ще следваме стандартните конвенции за именуване на класовете, както следва:

- **Views** имат надставката „View“ след името си (например: StudentListView)
- **ViewModels** имат надставката „ViewModel“ след името си (например: StudentListViewModel)
- **Models** имат надставката „Model“ след името си Model (например: StudentModel).

Следвайте следващите точки от задачата за да реализирате MVVM модел с команди:

1. Добавете **нов проект** към вашия Solution, кръстете го **WpfExample**.
2. Преименувайте **MainWindow** на **MainWindowView**, за да отговаря на нашата конвенция. За целта преименувайте файла (десен бутон върху него и → Rename), след това преименувайте и класа. Когато го правите се възползвайте от автоматичното преименуване, за да сте сигурни, че промяната ще бъде на всички места, където се среща.

```
/// </summary>
public partial class MainWindowView : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}
```



След като сме сменили името на началния прозорец, това трябва да се отрази и в App.xaml файла. Коригирайте атрибута на приложението **StartupUri="MainWindowView.xaml"**.

3. Създайте нов клас с име **MainWindowViewModel**, който ще играе ролята на ViewModel за нашия MainWindowView.

Това, което правим в MVVM е да „кажем“ на View за това какво ще представлява неговия ViewModel. Това оже да бъде направено посредством настройката на Data Context за този изглед. Моделите се използват във ViewModel и те нямат никаква връзка със специфичните изгледи.

Примерния код за настройка на DataContext е следния:

4. Отворете **MainWindowView.xaml** и настройте контекста по следния начин:

```
<Window x:Class="WpfExample.MainWindowView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"
        xmlns:local="clr-namespace:WpfExample">
    <Window.DataContext>
        <local:MainWindowViewModel/>
    </Window.DataContext>

    <Grid>
    </Grid>
</Window>
```

Тук използваме името **local** като псевдоним за нашия namespace WpfExample. Това е необходимо за може да се знае къде е достъпен MainWindowViewModel. С горния код сме настроили така че MainWindowView да знае, че неговия ViewModel е MainWindowViewModel.

За да проверим, че това работи ще използваме прост binding.

5. Добавете бутон към изгледа и настройте неговото съдържание (content) използвайки инстанция на ViewModel. Добавете следния код в Grid-a:

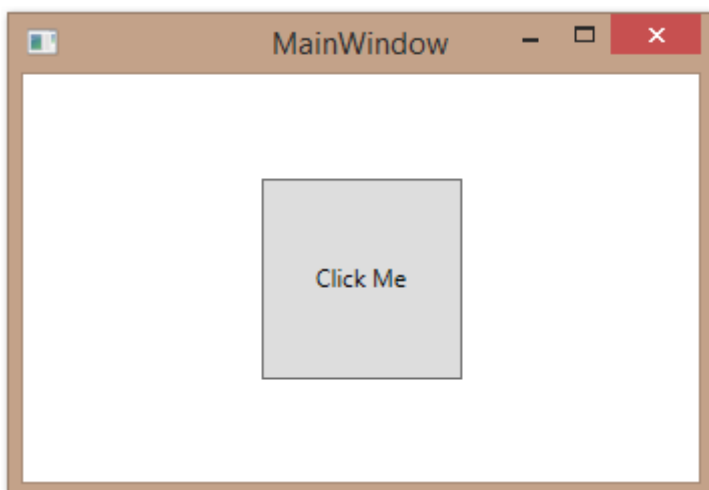
```
<Button Width="100" Height="100" Content="{Binding ButtonContent}"/>
```

6. В класа с име **MainWindowViewModel** добавете следното свойство:

```
class MainWindowViewModel
{
    public string ButtonContent
    {
        get
        {
            return "Click Me";
        }
    }
}
```

Благодарение на атрибута Content="{Binding ButtonContent}" на бутона, казваме на View-то да вземе съдържанието на бутона от **ButtonContent** свойството намиращо се във ViewModel-a.

7. Стартирайте приложението за да видите, че в бутона ще се изпише стринга "Click Me". Това означава, че началото на нашата MVVM структура е изградена.



### Въвеждане на ICommand Interface.

Сега ще добавим функционалността за натискане на бутона използвайки командите в WPF. Те предоставят механизъм, чрез който изгледа модифицира модела в MVVM архитектурата.

Ще създадем просто приложение, което показва съобщение с текст „Здрасти“, когато се натисне бутона и ще добавим още един бутон, който ще превключва възможността на първия бутон да бъде натискан.



8. Първо създаваме клас `RelayCommand`, който имплементира интерфейса `ICommand`. Добавете към проекта файл за клас с това име.

9. Добавете връзка към namespace-а на `ICommand` посредством `using System.Windows.Input;` и след това добавете в класа следващия код:

```
public class RelayCommand : ICommand
{
    private Action<object> execute;
    private Predicate<object> canExecute;
    private event EventHandler CanExecuteChangedInternal;

    public RelayCommand(Action<object> execute)
        : this(execute, DefaultCanExecute)
    {
    }

    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        if (execute == null)
        {
            throw new ArgumentNullException("execute");
        }

        if (canExecute == null)
        {
            throw new ArgumentNullException("canExecute");
        }

        this.execute = execute;
        this.canExecute = canExecute;
    }

    public event EventHandler CanExecuteChanged
    {
        add
        {
            CommandManager.RequerySuggested += value;
            this.CanExecuteChangedInternal += value;
        }

        remove
        {
            CommandManager.RequerySuggested -= value;
            this.CanExecuteChangedInternal -= value;
        }
    }

    public bool CanExecute(object parameter)
    {
        return this.canExecute != null && this.canExecute(parameter);
    }

    public void Execute(object parameter)
    {
        this.execute(parameter);
    }

    public void OnCanExecuteChanged()
    {
        EventHandler handler = this.CanExecuteChangedInternal;
        if (handler != null)
        {
            //DispatcherHelper.BeginInvokeOnUIThread(() => handler.Invoke(this, EventArgs.Empty));
            handler.Invoke(this, EventArgs.Empty);
        }
    }

    public void Destroy()
    {
        this.canExecute = _ => false;
        this.execute = _ => { return; };
    }
}
```

```

private static bool DefaultCanExecute(object parameter)
{
    return true;
}
}

```

Свойството `CommandManager.RequerySuggested` е отговорно за това да се активира и деактивира първия ни бутон.

Забележете също, че в кода горе са имплементирани следните методи и свойства на интерфейса `ICommand`:

```

bool CanExecute(object parameter);
void Execute(object parameter);
event EventHandler CanExecuteChanged;

```

10. В изгледа `MainWindowView` разделете прозореца на 2 части посредством колони в `Grid`-а.

```

<Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
</Grid.ColumnDefinitions>

```

11. Модифицирайте атрибутите на първия бутон и добавете втори, следвайки долния код.

```

<Button Width="100" Height="100" Content="{Binding HiButtonContent}" Grid.Column="0"
        Command="{Binding HiButtonCommand}" CommandParameter="Здрасти!" />
<Button Width="100" Height="100" Content="Toggle Click" Grid.Column="1"
        Command="{Binding ToggleExecuteCommand}"/>

```

12. От `MainWindowViewModel.cs` изтрийте свойството `ButtonContent`, с което в началото тествахме. Заменяме го с `HiButtonContent`. Добавете и тук `using System.Windows.Input;` за да е видим `ICommand` интерфейса.

13. Сложете следния код на класа `MainWindowViewModel`:

```

public class MainWindowViewModel
{
    private ICommand hiButtonCommand;
    private ICommand toggleExecuteCommand { get; set; }
    private bool canExecute = true;

    public string HiButtonContent
    {
        get {
            return "click to hi";
        }
    }

    public bool CanExecute
    {
        get { return this.canExecute; }
        set {
            if (this.canExecute == value) { return; }
            this.canExecute = value;
        }
    }

    public ICommand ToggleExecuteCommand
    {
        get
        {
            return toggleExecuteCommand;
        }
        set
    }
}

```

```

    {
        toggleExecuteCommand = value;
    }
}

public ICommand HiButtonCommand
{
    get
    {
        return hiButtonCommand;
    }
    set
    {
        hiButtonCommand = value;
    }
}

public MainWindowViewModel()
{
    HiButtonCommand = new RelayCommand(ShowMessage, param => this.canExecute);
    toggleExecuteCommand = new RelayCommand(ChangeCanExecute);
}

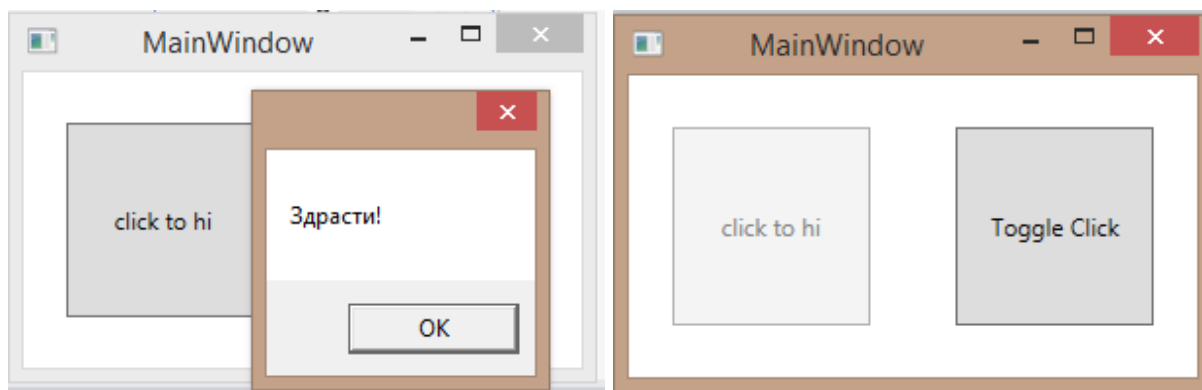
public void ShowMessage(object obj)
{
    //it is good we do this with binding to some control
    System.Windows.MessageBox.Show(obj.ToString());
}

public void ChangeCanExecute(object obj)
{
    canExecute = !canExecute;
}
}

```

14. Компилирайте и тествайте приложението (Да не забравите да настроите проекта като Startup Project на Solution-a).

Крайния изглед на приложението трябва да има този вид:



**Задача за самостоятелна работа:**

Добавете текстово поле, в което да се показва поздравлението, вместо в MessageBox. Добавете извеждане и на текущата дата и час.

За да получите времето ползвайте методите `DateTime.Now.ToLongDateString()`, `DateTime.Now.ToLongTimeString()` или техни сродни.

### Част 3.

Изтеглете следния файл (<http://tasheva.info/PS/PS-Lab3-example3.rar> ) и го разархивирайте. Отворете проекта и направете описаните промени по-долу, за да проработи.

1. Във View файла липсва връзката към namespace-а на ViewModel-а;

```
xmlns:ViewModel="clr-namespace:MinimalMVVM.ViewModel"
```

Забележете как, папката в нашия проект прави отделено подпространство.

2. ListBox-а трябва да бъде прикачен към свойството History;

```
ItemsSource="{Binding History}"
```

3. Бутона трябва да се върже към командата `ConvertTextCommand`.

```
Command="{Binding ConvertTextCommand}"
```

4. Стартирайте приложението и разгледайте неговата функционалност.

#### **Задача за самостоятелна работа:**

Добавете още един бутон, при натискането на който да се добавя в ListBox-а въведения низ, но преобразуван изцяло в малки букви. Използвайте команди и структурата на приложението, за да реализирате задачата аналогично.

#### **Източници:**

1. Microsoft MSDN, MVVM ultra easy example for beginners  
<https://social.msdn.microsoft.com/Forums/vstudio/en-US/62da4a56-4699-494e-a47f-33b89eb9e8dd/mvvm-ultra-easy-example-for-beginners?forum=wpf>
2. Microsoft MSDN, XAML Namespaces and Namespace Mapping for WPF XAML,  
<https://msdn.microsoft.com/library/ms747086%28v=vs.100%29.aspx>
3. 'Learn X' sites, Learn MVVM, <http://www.learnmvvm.com/tutorial.html#step1-1>
4. Nomes G, Basic MVVM and ICommand Usage Example, CodeProject, 31 Aug 2014,  
<http://www.codeproject.com/Tips/813345/Basic-MVVM-and-ICommand-Usage-Example>