Упражнение №6.2 (№6 част 2) по ПС WPF

Работа с INotifyPropertyChanged, ObservableCollection, ICommand, ViewModel.

Целта на това упражнение

Запознаване с Data Binding при промяна на данни.
Запознаване техниката за команди в WFP и интерфейсът ICommand.
Работа с ViewModel в отделен клас.

Задачите в упражнението изграждат:

Четири отделни примерни проекта:
Приложение за следене на разходи
Второ примерно приложение, работещо с команди.
Още две примерни приложения, работещи по шаблона MVVM

В това упражнение:

- *Разглежда се примерно приложение за раходи:*

- добавяме свойство и контрола свързана към него, което описва последна интеракци с приложението. Промяна на свойството опреснява и контролата.

- *Разглежда се примерно приложение, работещо с команди.*
- *Разглеждат се две примерни приложения, работещи по шаблона MVVM.*

В края на упражнението:

- *Потребителят вижда прозорец със списък на всички хора с разходи и бутон за преминаване към друга страница, в която се извеждат на разходите на избрания човек.*
- *Реализирано е примерно приложение, работещо с команди.*
- *Реализирани са още две примерни приложения, работещи по шаблона MVVM*

За домашно:

Да се промени кода така, че:

1. Да се приложи демонстрираното в проекта UI за да стане MVVM.

Важни знания от упражнението: **INotifyPropertyChanged, ObservableCollection, ICommand, ViewModel**

Зареждане на проект

1. Отворете **Visual Studio**
2. Заредете **Solution**-а създаден в предишните упражнения.
3. Създаваме нов проект, който кръщаваме **WpfExample**. Типа на проекта трябва да бъде **WPF**, задължително **C#**

Създаване на InfoCommand

1. . Създайте нов клас и го кръстете "InfoCommand".

2. Направете го публичен и наследете интерфейса ICommand:

```
public class InfoCommand : ICommand
```

3. Имплементираме интефейса. Започваме с Execute метода. Параметъра не го използваме. Директно изкрваме “Hello world” съобщение.

```
public void Execute(object parameter)
{
MessageBox.Show("Hello, world!");
}
```

4. CanExecute метода. Правим командата винаги налична.

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

5. Събитието CanExecuteChanged. Макар че няма да го възникваме ръчно, е необходимо за имплементацията на интерфейса.

```
public event EventHandler CanExecuteChanged;
```

Така вече класа InfoCommand представлява преизползвема команда, която знае кога може да бъде изпълнявана.

Свързване (Binding) на контоли към команди

Така създадената команда следва да се изпълни в отговор на потребителско събитие. За целта трябва да се даде начин, чрез който ползвателите й да я извикват. Ще демонстрираме следния начин: достъп до командата като свойство чрез Data context на прозореца и свързаване към нея на бутон чрез binding.

1. Създаваме частно поле с инстанция на `InfoCommand` в `MainWindow`.

```
private InfoCommand _infoCommand = new InfoCommand();
```

2. Създаваме свойство за публичен достъп до командата. Към него ще се свържем чрез binding.

```
public InfoCommand InformationCommand
{
get { return _infoCommand; }
}
```

3. Задате DataBinding на MainWindow.

```
this.DataContext = this;
```

4. Създайте бутон.

```
<Button >Info</Button>
```

5. Добавете свързане на *Command* свойстовото на бутона към *InformationCommand* свойството на посочения data context:

```
Command="{Binding InformationCommand}"
```

Вижте резултата от изпълнението.

Задача: Създайте следната функционалност:

- Създайте нов клас на прозорец NamesWindow.
- Променете командата `InfoCommand` след като изведе съобщението да отваря нов прозорец от тип NamesWindow.

Команди с параметри

Параметрите са от тип object в ICommand за да могат да се подават параметри от всякакъв тип.

Параметри могат да се посочват:

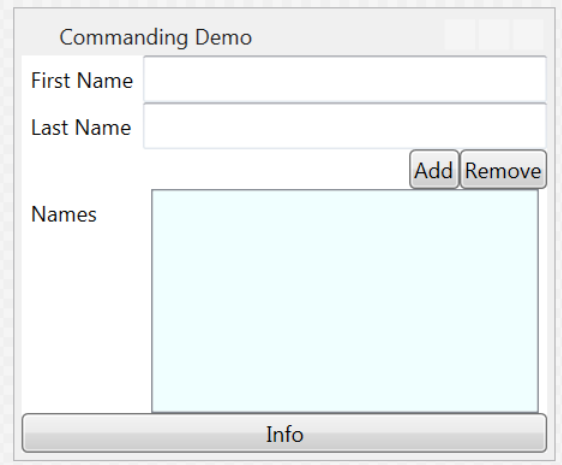
- директно в xaml (константи)
- чрез binding
- в C# кода

Ще демонстрираме с команда, която приема обект съдържащ две низови свойства и списък от низове, и го обработва, като конкатенира двата низа и ги добавя към списъка.

1. Добавете нов прозорец **NamesWindow**
2. Задайте на прозореца **NamesWindow** следното съдържание:

```
<Window x:Class="WpfExample.NamesWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Commanding Demo" Height="250" Width="300">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
```

```
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <Label>First Name</Label>
        <TextBox Grid.Column="1"
                 Text="{Binding FirstName,UpdateSourceTrigger=PropertyChanged}" />

        <Label Grid.Row="1">Last Name</Label>
        <TextBox Grid.Column="1" Grid.Row="1"
                 Text="{Binding LastName,UpdateSourceTrigger=PropertyChanged}" />

        <StackPanel Grid.ColumnSpan="2" Grid.Row="2" Orientation="Horizontal"
                    HorizontalAlignment="Right">
            <Button>Add</Button>
            <Button>Remove</Button>
        </StackPanel>

        <Label Grid.Row="3">Names</Label>
        <ListBox Grid.Column="1" Grid.Row="3" ItemsSource="{Binding Names}"
                 SelectedItem="{Binding SelectedName}" Margin="5 0"
Background="Azure"/>

        <Button Grid.ColumnSpan="2" Grid.Row="4">Info</Button>
    </Grid>
</Window>
```



3. Създайте нов клас NamesList, чиито обект ще служи за data context, със следното съдържание:

```
public class NamesList : INotifyPropertyChanged
{
    string _firstName = "";
    string _lastName = "";
    string _selectedName;

    public NamesList()
    {
        Names = new ObservableCollection<string>();
    }

    public string FirstName
    {
        get { return _firstName; }
        set
        {
            if (_firstName != value)
```

```
            {
                _firstName = value;
                OnPropertyChanged("FirstName");
            }
        }
    }

    public string LastName
    {
        get { return _lastName; }
        set
        {
            if (_lastName != value)
            {
                _lastName = value;
                OnPropertyChanged("LastName");
            }
        }
    }

    public string SelectedName
    {
        get { return _selectedName; }
        set
        {
            if (_selectedName != value)
            {
                _selectedName = value;
                OnPropertyChanged("SelectedName");
            }
        }
    }

    public ObservableCollection<string> Names { get; private set; }

    private void OnPropertyChanged(string property)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(property));
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

4. Задайте data context в контруктора на новия прозорец:

```
public NamesWindow ()
{
    InitializeComponent();
    DataContext = new NamesList();
}
```

5. Добавете нов клас на команда AddCommand със следното тяло:

```
public class AddCommand : ICommand
{
    public void Execute(object parameter)
    {
        var nameList = parameter as NamesList;
        var newName = string.Format("{0} {1}", nameList.FirstName,
nameList.LastName);
```

```
            nameList.Names.Add(newName);
            nameList.FirstName = nameList.LastName = "";
        }

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

        public event EventHandler CanExecuteChanged;
    }
```

Свойството CanExecute винаги връща true, т.е. командата винаги е изпълнима. Двата низа с имената се лсиват с помощта на `string.Format` и резултатът се добавя към списъка *Names* . След това в двете низови свойсва се поставя празен низ, което ще изичст исвързаните към тях контроли.

6. Добавяме ново поле за команда към класа NamesList. Създаваме и съответното свойство:

```
AddCommand _addNameCommand = new AddCommand();

public AddCommand AddNameCommand
{
    get { return _addNameCommand; }
}
```

7. Добавяме свързване на командата към бутона Add. Добавяме и свързване на параметър за командата. За параметър подаваме целия data context обект на прозореза (чрез binding без посочен път):

```
<Button Command="{Binding AddNameCommand}" CommandParameter="{Binding .}">Add</Button>
```

Вижте резултата от изпълнението.

Задача:
Добавете и свържете команда за втория бутон Remove, реализирана в нов клас RemoveCommand . За реализацията на RemoveCommand ползвайте следните функции:

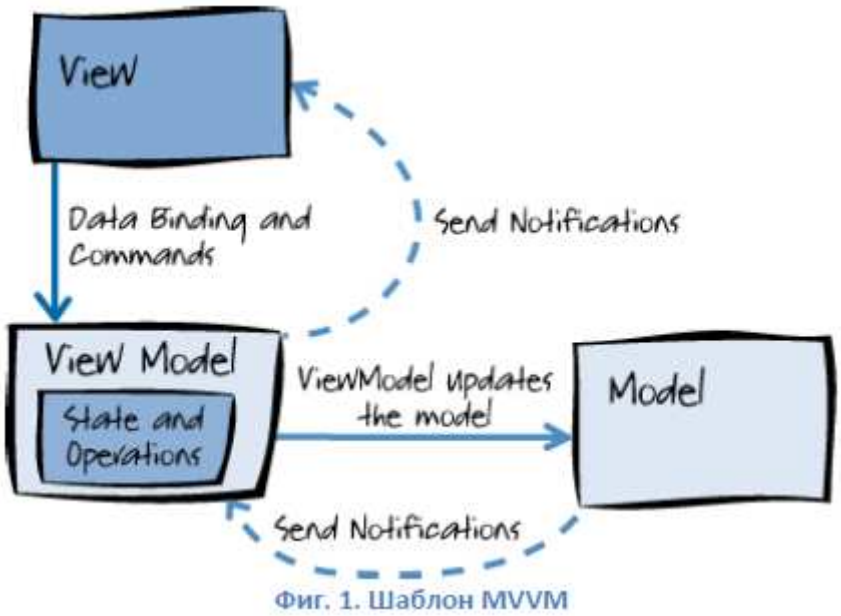
```
public bool CanExecute(object parameter)
{
    return true;
}

public void Execute(object parameter)
{
    var nameList = parameter as NameList;
    var oldName = nameList.SelectedName;
    nameList.Names.Remove(oldName);
}
```

Управляване изпълнимостта на команди

Не е разгледано. ☹

Ето пример:
https://www.c-sharpcorner.com/article/icommand-interface-in-mvvm/

Софтуерен шаблон Model-View-ViewModel



Фиг. 1. Шаблон MVVM

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

Създаване на модела (Model):
Първо ще разработим модела на данните на приложението.

В проекта **EasyMVVM:**

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 интерефейсите 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; PropChanged("BoundProperty"); }
}
```

8. Ще реализираме заложения в предната точка метод **PropChanged**(). Той приема като параметър името на променения 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 PropChanged(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 класовете 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**.

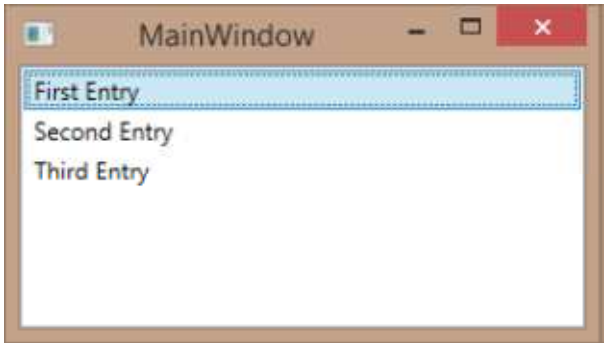
```
<Grid.Resources>
    <vm:MainWindowVM x:Key="vm"></vm:MainWindowVM>
</Grid.Resources>
```

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.

Свързване ViewModel

В проекта **MinimalMVVM:**

Изтеглете следния файл (http://tasheva.info/PS/PS-Lab3-example3.rar или https://drive.google.com/open?id=1KCOs1US2jrSfO7x3mcR2pdgevhVjuqo9) 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 проекти 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:
Добавете втори ViewModel клас,в който при натискането на бутона да се се добавя в ListBox-а въведения низ, но преобразуван изцяло в малки букви.
Добавете и възможност за превключване на режима (промяна към кой ViewModel е свързано View-то)

Използвайте команди и структурата на приложението, за да реализирате задачата аналогично.
DataContext на страницата може да се задава и в кода.

Реорганизация на UI

1. Прехвърлете наученото в проекта UI. Организирайте го съобразно MVVM с InotifyPropertyChanged и Icommand.