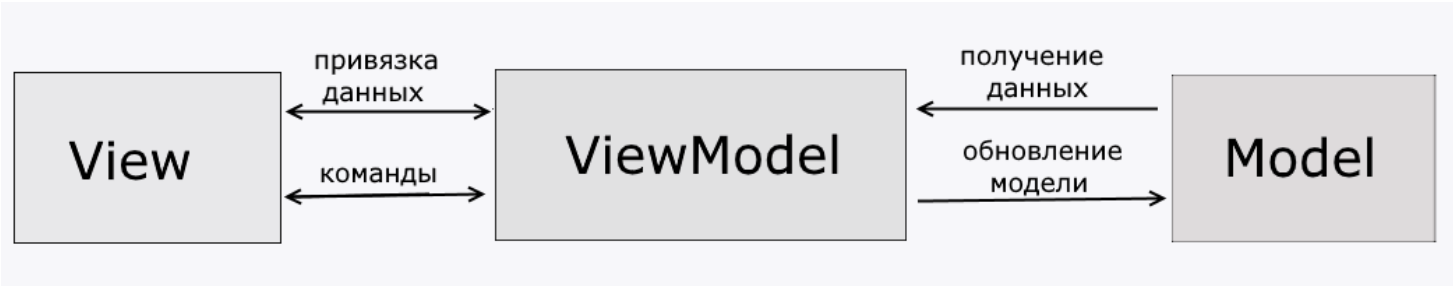
Паттерн MVVM

Определение паттерна MVVM

Паттерн **MVVM (Model-View-ViewModel)** позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения.

Данный паттерн был представлен Джоном Госсманом (John Gossman) в 2005 году как модификация шаблона Presentation Model и был первоначально нацелен на разработку приложений в WPF. И хотя сейчас данный паттерн вышел за пределы WPF и применяется в самых различных технологиях, в том числе при разработке под Android, iOS, тем не менее WPF является довольно показательной технологией, которая раскрывает возможности данного паттерна.

MVVM состоит из трех компонентов: модели (Model), модели представления (ViewModel) и представления (View).



Model

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную с этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.

Нередко модель реализует интерфейсы INotifyPropertyChanged или INotifyCollectionChanged, которые позволяют уведомлять систему об изменениях свойств модели. Благодаря этому облегчается привязка к представлению, хотя опять же прямое взаимодействие между моделью и представлением отсутствует.

View

View или представление определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Применительно к WPF представление - это код в xaml, который определяет интерфейс в виде кнопок, текстовых полей и прочих визуальных элементов.

Хотя окно (класс Window) в WPF может содержать как интерфейс в xaml, так и привязанный к нему код C#, однако в идеале код C# не должен содержать какой-то логики, кроме разве что конструктора, который вызывает метод InitializeComponent и выполняет начальную инициализацию окна. Вся же основная логика приложения выносится в компонент ViewModel.

Однако иногда в файле связанного кода все может находиться некоторая логика, которую трудно реализовать в рамках паттерна MVVM во ViewModel.

Представление не обрабатывает события за редким исключением, а выполняет действия в основном посредством команд.

ViewModel

ViewModel или модель представления связывает модель и представление через механизм привязки данных. Если в модели изменяются значения свойств, при реализации моделью интерфейса INotifyPropertyChanged автоматически идет изменение отображаемых данных в представлении, хотя напрямую модель и представление не связаны.

ViewModel также содержит логику по получению данных из модели, которые потом передаются в представление. И также VewModel определяет логику по обновлению данных в модели.

Поскольку элементы представления, то есть визуальные компоненты типа кнопок, не используют события, то представление взаимодействует с ViewModel посредством команд.

Например, пользователь хочет сохранить введенные в текстовое поле данные. Он нажимает на кнопку и тем самым отправляет команду во ViewModel. А ViewModel уже получает переданные данные и в соответствии с ними обновляет модель.

Итогом применения паттерна MVVM является функциональное разделение приложения на три компонента, которые проще разрабатывать и тестировать, а также в дальнейшем модифицировать и поддерживать.

Реализация MVVM. ViewModel

Для работы с паттерном MVVM создадим новый проект. По умолчанию в проект добавляется стартовое окно MainWindow - это и будет представление. И теперь нам нужна модель и ViewModel.

Добавим в проект новый класс `Phone`, который и будет представлять модель приложения:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace MVVM
{
    public class Phone : INotifyPropertyChanged
    {
        private string title;
        private string company;
        private int price;

        public string Title
        {
            get { return title; }
            set
            {
                title = value;
                OnPropertyChanged("Title");
            }
        }
        public string Company
        {
            get { return company; }
            set
            {
                company = value;
                OnPropertyChanged("Company");
            }
        }
        public int Price
        {
            get { return price; }
            set
            {
                price = value;
                OnPropertyChanged("Price");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged([CallerMemberName]string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

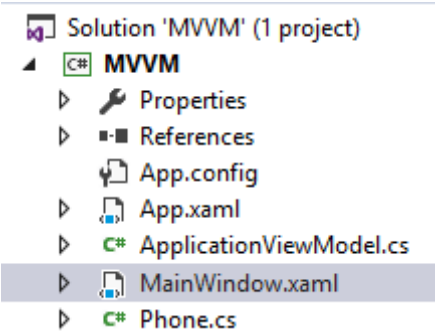
Для уведомления системы об изменениях свойств модель Phone реализует интерфейс INotifyPropertyChanged. Хотя в рамках паттерна MVVM это необязательно. В других конструкциях и ситуациях все может быть определено иначе.

Также добавим в проект новый класс `ApplicationViewModel`, который будет представлять модель представления:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Collections.ObjectModel;

namespace MVVM
{
    public class ApplicationViewModel : INotifyPropertyChanged
    {
        private Phone selectedPhone;

        public ObservableCollection<Phone> Phones { get; set; }
        public Phone SelectedPhone
        {
            get { return selectedPhone; }
            set
            {
                selectedPhone = value;
                OnPropertyChanged("SelectedPhone");
            }
        }

        public ApplicationViewModel()
        {
            Phones = new ObservableCollection<Phone>
            {
                new Phone { Title="iPhone 7", Company="Apple", Price=56000 },
                new Phone {Title="Galaxy S7 Edge", Company="Samsung", Price =60000 },
                new Phone {Title="Elite x3", Company="HP", Price=56000 },
                new Phone {Title="Mi5S", Company="Xiaomi", Price=35000 }
            };
        }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged([CallerMemberName]string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

Это класс модели представления, через который будут связаны модель `Phone` и представление `MainWindow.xaml` . В этом классе определен список объектов `Phone` и свойство, которое указывает на выделенный элемент в этом списке.

В итоге весь проект будет выглядеть следующим образом:



Далее изменим код нашего представления - файла MainWindow.xaml:

```
<Window x:Class="MVVM.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:MVVM"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="FontSize" Value="14" />
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="0.8*" />
        </Grid.ColumnDefinitions>

        <ListBox Grid.Column="0" ItemsSource="{Binding Phones}"
                 SelectedItem="{Binding SelectedPhone}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Margin="5">
                        <TextBlock FontSize="18" Text="{Binding Path=Title}" />
                        <TextBlock Text="{Binding Path=Company}" />
                        <TextBlock Text="{Binding Path=Price}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>

        <StackPanel Grid.Column="1" DataContext="{Binding SelectedPhone}">
            <TextBlock Text="Выбранный элемент"  />
            <TextBlock Text="Модель" />
            <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Производитель" />
            <TextBox Text="{Binding Company, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Цена" />
            <TextBox Text="{Binding Price, UpdateSourceTrigger=PropertyChanged}" />
        </StackPanel>
    </Grid>
</Window>
```

Здесь определен элемент ListBox, который привязан к свойству `Phones` объекта `ApplicationViewModel` , а также определен набор элементов, которые привязаны к свойствам

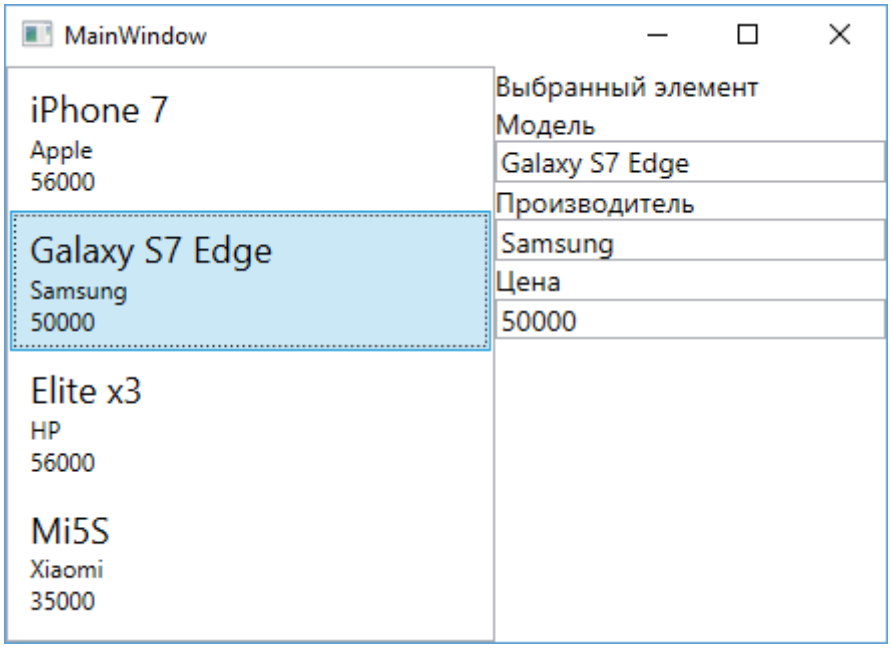
объекта `Phone` , выделенного в `ListBox` .

И изменим файл кода MainWindow.xaml.cs:

```
using System.Windows;

namespace MVVM
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            DataContext = new ApplicationViewModel();
        }
    }
}
```

Здесь достаточно установить контекст данных для данного окна в виде объекта `ApplicationViewModel` , который свяжет представление и модели `Phone` .

И если мы запустим приложение, то увидим список объектов. Мы можем выбрать один из них, и его данные появятся в полях справа:



При этом не надо определять код загрузки объектов в ListBox, определять обработчики выбора объек

Определение модели

В данном случае мы сами определяем модель Phone. Однако не всегда мы имеем возможность реализовать в используемой модели интерфейс INotifyPropertyChanged. Также, возможно, мы захотим предусмотреть отдельное представление (отдельное окно) для манипуляций над одной моделью (добавление, изменение, удаление). Подобное представление может иметь в качестве ViewModel объект модели Phone. И в подобных случаях мы можем создать отдельную ViewModel для работы с одним объектом Phone, наподобие:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace MVVM
{
    public class PhoneViewModel : INotifyPropertyChanged
    {
        private Phone phone;

        public PhoneViewModel(Phone p)
        {
            phone = p;
        }

        public string Title
        {
            get { return phone.Title; }
            set
            {
                phone.Title = value;
                OnPropertyChanged("Title");
            }
        }
        public string Company
        {
            get { return phone.Company; }
            set
            {
                phone.Company = value;
                OnPropertyChanged("Company");
            }
        }
        public int Price
        {
            get { return phone.Price; }
            set
            {
                phone.Price = value;
                OnPropertyChanged("Price");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged([CallerMemberName]string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

Команды в MVVM

Для взаимодействия пользователя и приложения в MVVM используются команды. Это не значит, что вовсе не можем использовать события и событийную модель, однако везде, где возможно, **вместо событий следует использовать команды**.

В WPF команды представлены интерфейсом ICommand:

```
public interface ICommand
{
    event EventHandler CanExecuteChanged;
    void Execute (object parameter);
    bool CanExecute (object parameter);
}
```

Однако WPF имеет в качестве реализации этого интерфейса имеет класс **System.Windows.Input.RoutedCommand**, который ограничен по функциональности. Поэтому, как правило, придется реализовывать свои собственные команды с помощью реализации ICommand.

Для использования команд продолжим работу с проектом из прошлой темы и добавим в него новый класс, который назовем `RelayCommand`:

```
using System;
using System.Windows.Input;

namespace MVVM
{
    public class RelayCommand : ICommand
    {
        private Action<object> execute;
        private Func<object, bool> canExecute;

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        public RelayCommand(Action<object> execute, Func<object, bool> canExecute = null)
        {
            this.execute = execute;
            this.canExecute = canExecute;
        }

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

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

Класс реализует два метода:

- **CanExecute**: определяет, может ли команда выполняться
- **Execute**: собственно выполняет логику команды

Событие **CanExecuteChanged** вызывается при изменении условий, указывающий, может ли команда выполняться. Для этого используется событие CommandManager.RequerySuggested.

Ключевым является метод Execute. Для его выполнения в конструкторе команды передается делегат типа Action. При этом класс команды не знает какое именно действие будет выполняться. Например, мы можем написать так:

```
var cmd = new RelayCommand(o => { MessageBox.Show("Команда" + o.ToString()); });
cmd.Execute("1");
```

В результате вызова команды будет выведено окно с надписью "Команда1". Но мы могли также передать любое другое действие, которое бы соответствовало делегату Action.

Применение команд

Для ряда визуальных элементов WPF, например, для кнопок, определена поддержка команд. Однако сами команды определяются в ViewModel и затем через механизм привязки устанавливаются для элементов управления. Например, изменим код ApplicationViewModel следующим образом:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Collections.ObjectModel;

namespace MVVM
{
    public class ApplicationViewModel : INotifyPropertyChanged
    {
        private Phone selectedPhone;
        public ObservableCollection<Phone> Phones { get; set; }

        // команда добавления нового объекта
        private RelayCommand addCommand;
        public RelayCommand AddCommand
        {
            get
            {
                return addCommand ??
                  (addCommand = new RelayCommand(obj =>
                  {
                      Phone phone = new Phone();
                      Phones.Insert(0, phone);
                      SelectedPhone = phone;
                  }));
            }
        }

        public Phone SelectedPhone
        {
            get { return selectedPhone; }
            set
            {
                selectedPhone = value;
                OnPropertyChanged("SelectedPhone");
            }
        }

        public ApplicationViewModel()
        {
            Phones = new ObservableCollection<Phone>
            {
                new Phone { Title="iPhone 7", Company="Apple", Price=56000 },
                new Phone {Title="Galaxy S7 Edge", Company="Samsung", Price =60000 },
                new Phone {Title="Elite x3", Company="HP", Price=56000 },
                new Phone {Title="Mi5S", Company="Xiaomi", Price=35000 }
            };
        }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged([CallerMemberName]string prop = "")
```

```
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

Здесь добавлена команда на добавление объекта:

```
private RelayCommand addCommand;
public RelayCommand AddCommand
{
    get
    {
        return addCommand ??
            (addCommand = new RelayCommand(obj =>
            {
                Phone phone = new Phone();
                Phones.Insert(0, phone);
                SelectedPhone = phone;
            }));
    }
}
```

Команда хранится в свойстве AddCommand и представляет собой объект выше определенного класса RelayCommand. Этот объект в конструкторе принимает действие - делегат Action. Здесь действие представлено в виде лямбда-выражения, которое добавляет в коллекцию Phones новый объект Phone и устанавливает его в качестве выбранного.

Используем эту команду. Для этого изменим код представления в MainWindow.xaml:

```
<Window x:Class="MVVM.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:MVVM"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="Button">
            <Setter Property="Width" Value="40" />
            <Setter Property="Margin" Value="5" />
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="0.8*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="0.2*" />
        </Grid.RowDefinitions>
        <ListBox Grid.Column="0" ItemsSource="{Binding Phones}"
                 SelectedItem="{Binding SelectedPhone}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Margin="5">
                        <TextBlock FontSize="18" Text="{Binding Path=Title}" />
                        <TextBlock Text="{Binding Path=Company}" />
                        <TextBlock Text="{Binding Path=Price}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
        <StackPanel Grid.Row="1" Orientation="Horizontal">
            <Button Command="{Binding AddCommand}">+</Button>
        </StackPanel>

        <StackPanel Grid.Column="1" DataContext="{Binding SelectedPhone}">
            <TextBlock Text="Выбранный элемент"  />
            <TextBlock Text="Модель" />
            <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Производитель" />
```

```
            <TextBox Text="{Binding Company, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Цена" />
            <TextBox Text="{Binding Price, UpdateSourceTrigger=PropertyChanged}" />
        </StackPanel>
    </Grid>
</Window>
```

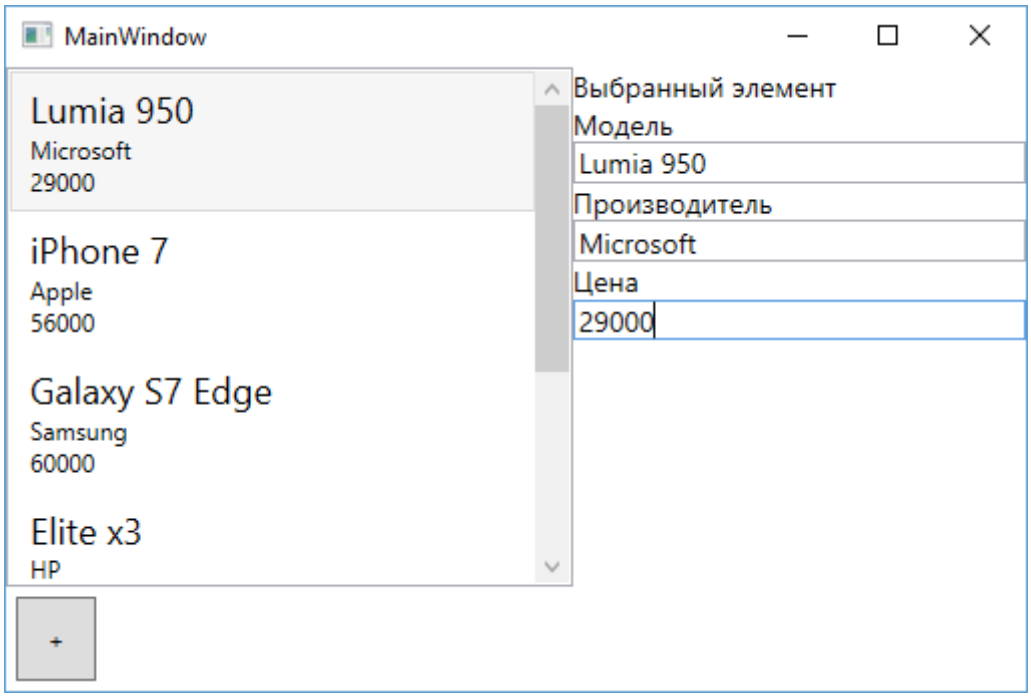
Здесь добавлена кнопка, свойство Command которой приязано к свойству AddCommand объекта ApplicationViewModel:

```
<Button Command="{Binding AddCommand}">+</Button>
```

И нам не надо писать никаких обработчиков нажатия. Автоматически при нажатии на кнопку сработает команда, которая добавит в список еще один объект. А код в файле MainWindow.xaml.cs остается прежним:

```
using System.Windows;

namespace MVVM
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            DataContext = new ApplicationViewModel();
        }
    }
}
```

И при нажатии на кнопку в список будет добавлен новый объект, который мы сразу сможем отредактировать в текстовых полях справа:



Передача параметров команде

Команда может принимать один параметр типа object, вместо которого мы можем передать любой объект или даже коллекцию объектов. Например, продолжим работу с проектом из прошлой темы и добавим в него удаление объекта из списка. Для этого изменим код ApplicationViewModel следующим образом:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Collections.ObjectModel;

namespace MVVM
{
    public class ApplicationViewModel : INotifyPropertyChanged
    {
        private Phone selectedPhone;
        public ObservableCollection<Phone> Phones { get; set; }

        // команда добавления нового объекта
        private RelayCommand addCommand;
        public RelayCommand AddCommand
        {
            get
            {
                return addCommand ??
                  (addCommand = new RelayCommand(obj =>
                  {
                      Phone phone = new Phone();
                      Phones.Insert(0, phone);
                      SelectedPhone = phone;
                  }));
            }
        }

        // команда удаления
        private RelayCommand removeCommand;
        public RelayCommand RemoveCommand
        {
            get
            {
                return removeCommand ??
                  (removeCommand = new RelayCommand(obj =>
                  {
                      Phone phone = obj as Phone;
                      if (phone != null)
                      {
                          Phones.Remove(phone);
                      }
                  },
                 (obj) => Phones.Count > 0));
            }
        }

        public Phone SelectedPhone
        {
            get { return selectedPhone; }
            set
```

```
            {
                selectedPhone = value;
                OnPropertyChanged("SelectedPhone");
            }
        }

        public ApplicationViewModel()
        {
            Phones = new ObservableCollection<Phone>
            {
                new Phone { Title="iPhone 7", Company="Apple", Price=56000 },
                new Phone {Title="Galaxy S7 Edge", Company="Samsung", Price =60000 },
                new Phone {Title="Elite x3", Company="HP", Price=56000 },
                new Phone {Title="Mi5S", Company="Xiaomi", Price=35000 }
            };
        }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged([CallerMemberName]string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

Здесь добавлена команда удаления объекта из списка:

```
private RelayCommand removeCommand;
public RelayCommand RemoveCommand
{
    get
    {
        return removeCommand ??
            (removeCommand = new RelayCommand(obj =>
            {
                Phone phone = obj as Phone;
                if (phone != null)
                {
                    Phones.Remove(phone);
                }
            },
            (obj) => Phones.Count > 0));
    }
}
```

Здесь предполагается, что в качестве параметра в команду будет передаваться удаляемый объект Phone. Ну а поскольку в реальности параметр имеет тип object, то его еще надо

привести к типу Phone.

Стоит отметить, что в качестве второго параметра в конструктор RelayCommand передается делегат Func<obj, bool>, который позволяет указать условие, при котором будет доступна команда. В нашем случае нет смысла удалять элементы из списка, если в списке нет элементов.

И также изменим код MainWindow.xaml:

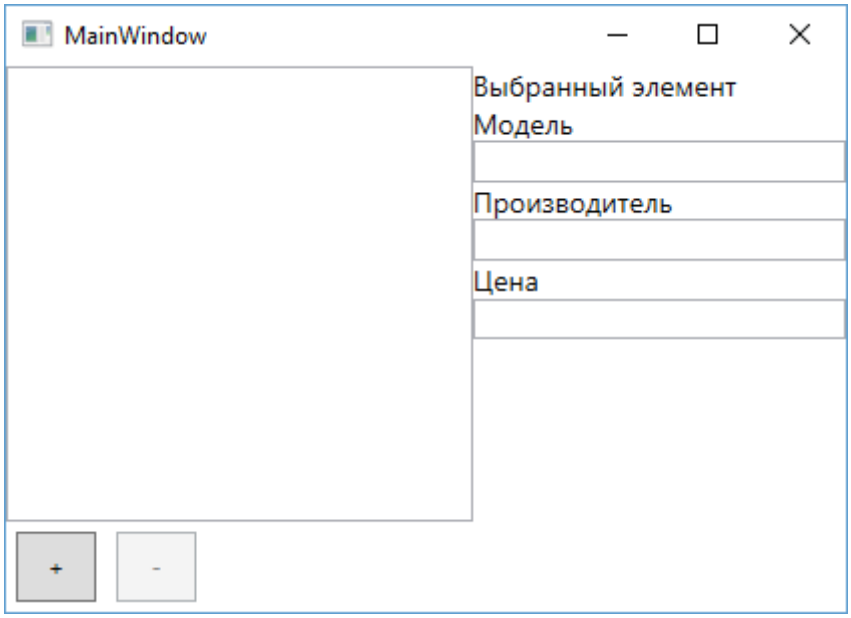
```
<Window x:Class="MVVM.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:MVVM"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="Button">
            <Setter Property="Width" Value="40" />
            <Setter Property="Margin" Value="5" />
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="0.8*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="0.2*" />
        </Grid.RowDefinitions>
        <ListBox Grid.Column="0" ItemsSource="{Binding Phones}"
                 SelectedItem="{Binding SelectedPhone}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Margin="5">
                        <TextBlock FontSize="18" Text="{Binding Path=Title}" />
                        <TextBlock Text="{Binding Path=Company}" />
                        <TextBlock Text="{Binding Path=Price}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
        <StackPanel Grid.Row="1" Orientation="Horizontal">
            <Button Command="{Binding AddCommand}">+</Button>
            <Button Command="{Binding RemoveCommand}"
                    CommandParameter="{Binding SelectedPhone}">-</Button>
        </StackPanel>

        <StackPanel Grid.Column="1" DataContext="{Binding SelectedPhone}">
            <TextBlock Text="Выбранный элемент"  />
            <TextBlock Text="Модель" />
```

```
            <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Производитель" />
            <TextBox Text="{Binding Company, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Цена" />
            <TextBox Text="{Binding Price, UpdateSourceTrigger=PropertyChanged}" />
        </StackPanel>
    </Grid>
</Window>
```

Здесь добавлена кнопка удаления. Для нее установлена привязка к команде RemoveCommand. Кроме того, с помощью атрибута CommandParameter кнопка устанавливает объект, который передается команде. В данном случае это объект из свойства SelectedPhone:

```
<Button Command="{Binding RemoveCommand}"
        CommandParameter="{Binding SelectedPhone}">-</Button>
```

Теперь мы сможем удалять элементы из списка. Причем, если в списке не будет элементов, то кнопка будет недоступна, благодаря тому, что в конструктор RelayCommand выше было передано выражение (obj) => Phones.Count > 0, которое устанавливает условие выполнения команды:



Взаимодействие команд и событий

Кнопка поддерживает команды, и вместо обработки события Click мы можем прикрепить к ней команду. Но что делать, если элемент не поддерживает команду, а нам надо обработать какое-то его действие.

Например, у нас определен элемент ListBox, и мы хотим отслеживать выбор объекта в списке. Если бы мы использовали событийную модель, то мы бы обрабатывали событие SelectionChanged. Но мы знаем, что выделение объекта в списке ведет к изменению свойства SelectedItem элемента ListBox. Поэтому вместо применения события мы можем просто обрабатывать изменения свойства SelectedPhone в ApplicationViewModel, которое привязано к свойству SelectedItem у ListBox.

Аналогично у элемента TextBox есть событие TextChanged, которое вызывается при изменении вводимого текста. Но опять же изменение текста приводит к изменению свойства Text у элемента TextBox. Поэтому мы могли бы установить привязку для этого элемента к свойству Text:

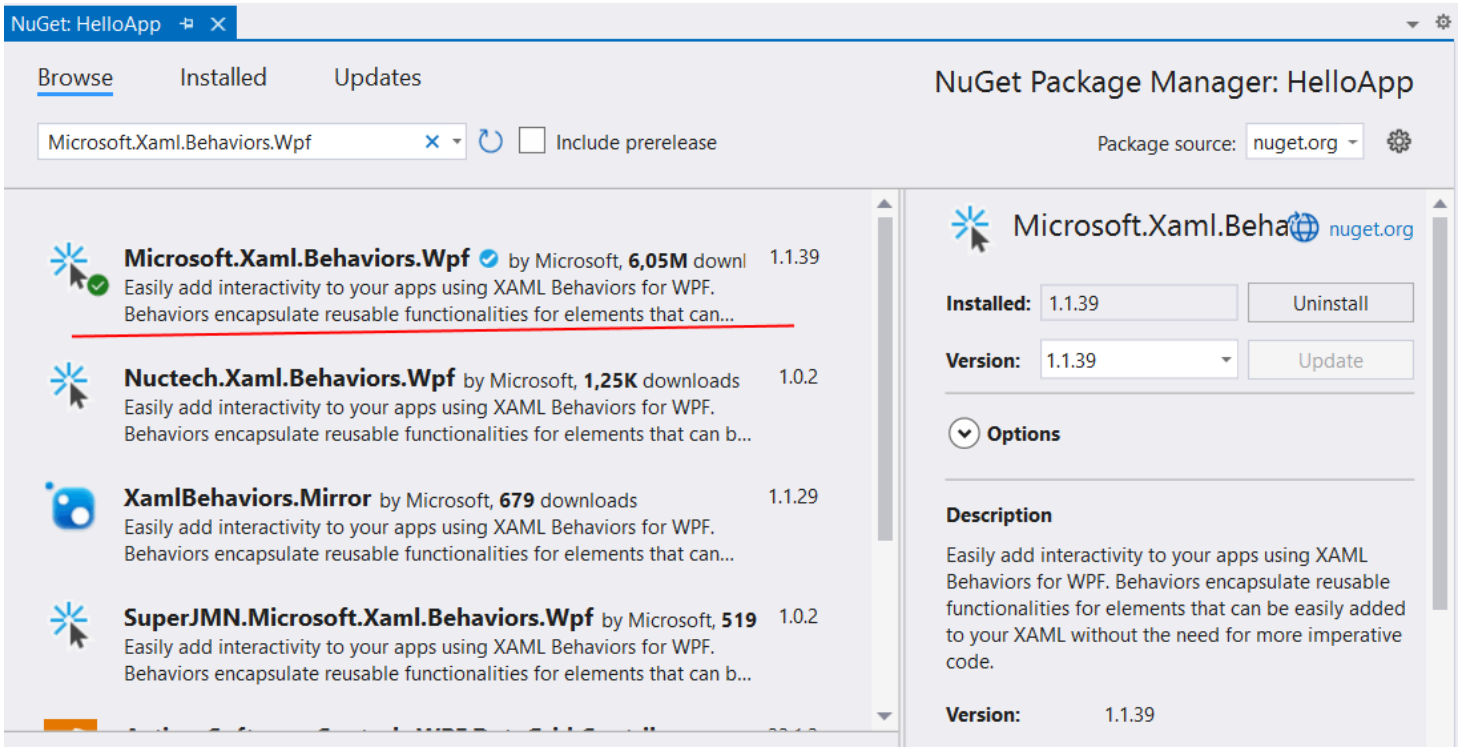
```
<TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" />
```

А в классе Phone (либо в специальной ViewModel, которая инкапсулирует объект Phone) обрабатывать изменение свойства Title, которое бы синхронно менялось при вводе новых символов в текстовое поле:

```
public string Title
{
    get { return phone.Title; }
    set
    {
        // обработка изменения свойства
        phone.Title = value;
        OnPropertyChanged("Title");
    }
}
```

В тоже время подобные действия можно установить не для всех событий. И если мы хотим связать события с командами, необходимо использовать дополнительный функционал.

В частности, нам надо добавить в проект через пакетный менеджер Nuget специальный пакет **Microsoft.Xaml.Behaviors.Wpf**:



Пусть у нас в проекте есть следующая модель **Phone**:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace HelloApp
{
    public class Phone : INotifyPropertyChanged
    {
        private string title;
        private string company;
        private int price;

        public Phone(string title, string company, int price)
        {
            this.title = title;
            this.company = company;
            this.price = price;
        }

        public string Title
        {
            get { return title; }
            set
            {
                title = value;
                OnPropertyChanged("Title");
            }
        }
        public string Company
        {
            get { return company; }
            set
            {
                company = value;
                OnPropertyChanged("Company");
            }
        }
        public int Price
        {
            get { return price; }
            set
            {
                price = value;
                OnPropertyChanged("Price");
            }
        }

        public event PropertyChangedEventHandler? PropertyChanged;
        public void OnPropertyChanged([CallerMemberName] string prop = "")
        {
            if (PropertyChanged != null)
```

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

Затем определим следующий класс **ApplicationViewModel**, который будет представлять ViewModel:

```
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

namespace HelloApp
{
    public class ApplicationViewModel : INotifyPropertyChanged
    {
        Phone? selectedPhone;
        public ObservableCollection<Phone> Phones { get; set; }

        // команда добавления нового объекта
        RelayCommand? addCommand;
        public RelayCommand AddCommand
        {
            get
            {
                return addCommand ??
                  (addCommand = new RelayCommand(obj =>
                  {
                      Phone phone = new Phone("", "", 0);
                      Phones.Insert(0, phone);
                      SelectedPhone = phone;
                  }));
            }
        }

        // команда удаления
        RelayCommand? removeCommand;
        public RelayCommand RemoveCommand
        {
            get
            {
                return removeCommand ??
                  (removeCommand = new RelayCommand(obj =>
                  {
                      Phone? phone = obj as Phone;
                      if (phone != null)
                      {
                          Phones.Remove(phone);
                      }
                  },
                 (obj) => Phones.Count > 0));
            }
        }

        RelayCommand? doubleCommand;
        public RelayCommand DoubleCommand
```

```
    {
        get
        {
            return doubleCommand ??
                (doubleCommand = new RelayCommand(obj =>
                {
                    Phone? phone = obj as Phone;
                    if (phone != null)
                    {
                        Phone phoneCopy = new Phone(phone.Company,  phone.Title, phone.Price
                        Phones.Insert(0, phoneCopy);
                    }
                }));
        }
    }

    public Phone? SelectedPhone
    {
        get { return selectedPhone; }
        set
        {
            selectedPhone = value;
            OnPropertyChanged("SelectedPhone");
        }
    }

    public ApplicationViewModel()
    {
        Phones = new ObservableCollection<Phone>
        {
            new Phone("iPhone 7", "Apple", 56000),
            new Phone("Galaxy S7 Edge", "Samsung", 60000),
            new Phone("Elite x3", "HP", 56000),
            new Phone("Mi5S", "Xiaomi", 35000)
        };
    }

    public event PropertyChangedEventHandler? PropertyChanged;
    public void OnPropertyChanged([CallerMemberName] string prop = "")
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(prop));
    }
}


public class RelayCommand : ICommand
{
    Action<object?> execute;
    Func<object?, bool>? canExecute;
```

```
        public event EventHandler? CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }
        public RelayCommand(Action<object?> execute, Func<object?, bool>? canExecute = null)
        {
            this.execute = execute;
            this.canExecute = canExecute;
        }
        public bool CanExecute(object? parameter)
        {
            return canExecute == null || canExecute(parameter);
        }
        public void Execute(object? parameter)
        {
            execute(parameter);
        }
    }
}
```

Среди прочего здесь определена команда DoubleCommand, которая добавляет копию объекта в список.

Далее добавим в представление кнопку, которая будет вызывать данную команду. Но, допустим, мы хотим выполнять эту команду, если по кнопке был совершен двойной щелчок, то есть если произошло событие MouseDoubleClick:

```
<Window x:Class="HelloApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:i="http://schemas.microsoft.com/xaml/behaviors"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="Button">
            <Setter Property="Width" Value="40" />
            <Setter Property="Margin" Value="5" />
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="0.8*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="0.2*" />
        </Grid.RowDefinitions>
        <ListBox Grid.Column="0" ItemsSource="{Binding Phones}"
                 SelectedItem="{Binding SelectedPhone}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Margin="5">
                        <TextBlock FontSize="18" Text="{Binding Path=Title}" />
                        <TextBlock Text="{Binding Path=Company}" />
                        <TextBlock Text="{Binding Path=Price}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
        <StackPanel Grid.Row="1" Orientation="Horizontal">
            <Button Command="{Binding AddCommand}">+</Button>
            <Button Command="{Binding RemoveCommand}"
                    CommandParameter="{Binding SelectedPhone}">-</Button>
            <Button Content="2x">
                <i:Interaction.Triggers>
                    <i:EventTrigger EventName="MouseDoubleClick">
                        <i:InvokeCommandAction
                            Command="{Binding DoubleCommand}"
```

```
                        CommandParameter="{Binding SelectedPhone}" />
                </i:EventTrigger>
            </i:Interaction.Triggers>
        </Button>
    </StackPanel>

    <StackPanel Grid.Column="1" DataContext="{Binding SelectedPhone}">
        <TextBlock Text="Выбранный элемент"  />
        <TextBlock Text="Модель" />
        <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" />
        <TextBlock Text="Производитель" />
        <TextBox Text="{Binding Company, UpdateSourceTrigger=PropertyChanged}" />
        <TextBlock Text="Цена" />
        <TextBox Text="{Binding Price, UpdateSourceTrigger=PropertyChanged}" />
    </StackPanel>
</Grid>
</Window>
```

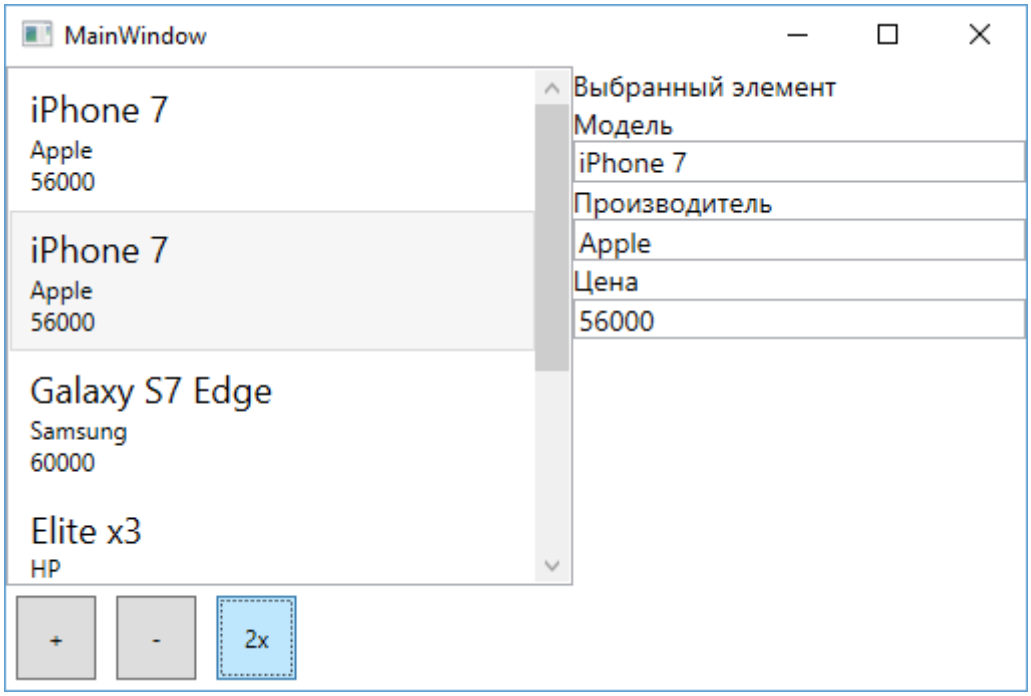
Для добавленной кнопки устанавливается свойство Interaction.Triggers, которое позволяет связать триггеры событий с командами и передать этим командам параметры:

```
<Button Content="2x">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="MouseDoubleClick">
            <i:InvokeCommandAction
                Command="{Binding DoubleCommand}"
                CommandParameter="{Binding SelectedPhone}" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</Button>
```

В файле связанного кода **MainPage.xaml.cs** определим привязку ApplicationViewModel к контексту страницы MainPage

```
using System.Windows;

namespace HelloApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            DataContext = new ApplicationViewModel();
        }
    }
}
```

Теперь по двойному нажатию на эту кнопку произойдет дублирование в списке выделенного элемента:



Работа с диалоговыми окнами

Работа с диалоговыми окнами при использовании паттерна MVVM может вызывать некоторые трудности. Например, если мы хотим по клику на кнопку или на пункт меню сохранять или открывать объекты с помощью диалоговых окон, не всегда может быть ясно, как вписать взаимодействие с диалоговыми окнами в MVVM.

Итак, для работы с диалоговыми окнами продолжим работу с проектом из прошлой темы. Вначале добавим в него новый интерфейс IDialogService, который будет определять функционал для работы с диалоговыми окнами:

```
public interface IDialogService
{
    void ShowMessage(string message);   // показ сообщения
    string FilePath { get; set; }   // путь к выбранному файлу
    bool OpenFileDialog();  // открытие файла
    bool SaveFileDialog();  // сохранение файла
}
```

Далее добавим в проект реализацию этого интерфейса в виде класса DefaultDialogService:

```
using Microsoft.Win32;
using System.Windows;

namespace MVVM
{
    public class DefaultDialogService : IDialogService
    {
        public string FilePath { get; set; }

        public bool OpenFileDialog()
        {
            OpenFileDialog openFileDialog = new OpenFileDialog();
            if (openFileDialog.ShowDialog() == true)
            {
                FilePath = openFileDialog.FileName;
                return true;
            }
            return false;
        }

        public bool SaveFileDialog()
        {
            SaveFileDialog saveFileDialog = new SaveFileDialog();
            if (saveFileDialog.ShowDialog() == true)
            {
                FilePath = saveFileDialog.FileName;
                return true;
            }
            return false;
        }

        public void ShowMessage(string message)
        {
            MessageBox.Show(message);
        }
    }
}
```

Для получения пути файла для открытия/сохранения данный класс использует стандартные классы OpenFileDialog и SaveFileDialog, которые определены в пространстве имен Microsoft.Win32. Кроме того, для отображения сообщения здесь используется метод MessageBox.Show().

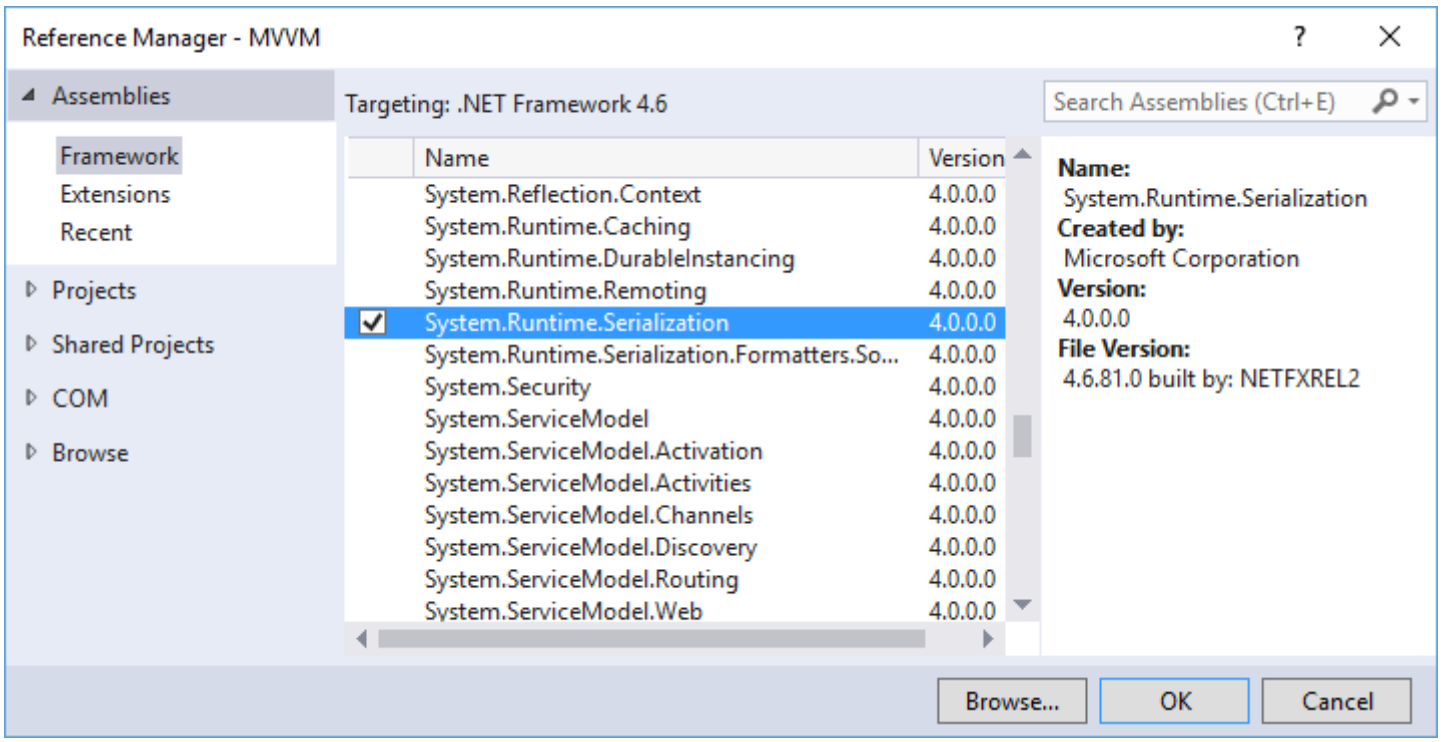
Для работы с файлами одного функционала по открытию/сохранению файла мало. Нам еще надо выполнять сами действия по считыванию информации из файла или ее сохранению в файл. Подобные действия, конечно, можно определить и в ViewModel. Однако для работы с информацией мы можем использовать различные типы файлов - бинарные файлы, xml, json и

т.д. Для json в .NET мы можем использовать один функционал, для xml - другой, для текстовых файлов - третий и так далее. Поэтому в этом случае для работы с файлами определим в проекте общий интерфейс IFileService:

```
using System.Collections.Generic;

namespace MVVM
{
    public interface IFileService
    {
        List<Phone> Open(string filename);
        void Save(string filename, List<Phone> phonesList);
    }
}
```

Первый метод предназначен для открытия файла. Он принимает путь к файлу и возвращает список объектов. Второй метод сохраняет данные из списка в файле по определенному пути.

В качестве типа файлов мы будем использовать файлы json. Поэтому для работы с ними добавим в проект библиотеку :



Затем добавим в проект следующий класс JsonFileService:

```
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Json;

namespace MVVM
{
    public class JsonFileService : IFileService
    {
        public List<Phone> Open(string filename)
        {
            List<Phone> phones = new List<Phone>();
            DataContractJsonSerializer jsonFormatter =
                new DataContractJsonSerializer(typeof(List<Phone>));
            using (FileStream fs = new FileStream(filename, FileMode.OpenOrCreate))
            {
                phones = jsonFormatter.ReadObject(fs) as List<Phone>;
            }

            return phones;
        }

        public void Save(string filename, List<Phone> phonesList)
        {
            DataContractJsonSerializer jsonFormatter =
                new DataContractJsonSerializer(typeof(List<Phone>));
            using (FileStream fs = new FileStream(filename, FileMode.Create))
            {
                jsonFormatter.WriteObject(fs, phonesList);
            }
        }
    }
}
```

С помощью класса DataContractJsonSerializer здесь производится сериализация/десериализация объектов в файл json в виде набора List.

Далее изменим код ApplicationViewModel:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Collections.ObjectModel;
using System;
using System.Linq;

namespace MVVM
{
    public class ApplicationViewModel : INotifyPropertyChanged
    {
        Phone selectedPhone;

        IFileService fileService;
        IDialogService dialogService;

        public ObservableCollection<Phone> Phones { get; set; }

        // команда сохранения файла
        private RelayCommand saveCommand;
        public RelayCommand SaveCommand
        {
            get
            {
                return saveCommand ??
                  (saveCommand = new RelayCommand(obj =>
                  {
                      try
                      {
                          if (dialogService.SaveFileDialog() == true)
                          {
                              fileService.Save(dialogService.FilePath, Phones.ToList());
                              dialogService.ShowMessage("Файл сохранен");
                          }
                      }
                      catch (Exception ex)
                      {
                          dialogService.ShowMessage(ex.Message);
                      }
                  }));
            }
        }

        // команда открытия файла
        private RelayCommand openCommand;
        public RelayCommand OpenCommand
        {
            get
            {
                return openCommand ??
                  (openCommand = new RelayCommand(obj =>
```

```
            {
                try
                {
                    if (dialogService.OpenFileDialog() == true)
                    {
                        var phones = fileService.Open(dialogService.FilePath);
                        Phones.Clear();
                        foreach (var p in phones)
                            Phones.Add(p);
                        dialogService.ShowMessage("Файл открыт");
                    }
                }
                catch (Exception ex)
                {
                    dialogService.ShowMessage(ex.Message);
                }
            }));
    }
}

// команда добавления нового объекта
private RelayCommand addCommand;
public RelayCommand AddCommand
{
    get
    {
        return addCommand ??
          (addCommand = new RelayCommand(obj =>
          {
              Phone phone = new Phone();
              Phones.Insert(0, phone);
              SelectedPhone = phone;
          }));
    }
}

private RelayCommand removeCommand;
public RelayCommand RemoveCommand
{
    get
    {
        return removeCommand ??
          (removeCommand = new RelayCommand(obj =>
          {
              Phone phone = obj as Phone;
              if (phone != null)
              {
                  Phones.Remove(phone);
              }
          },
         (obj) => Phones.Count > 0));
```

```
    }
}
private RelayCommand doubleCommand;
public RelayCommand DoubleCommand
{
    get
    {
        return doubleCommand ??
          (doubleCommand = new RelayCommand(obj =>
          {
              Phone phone = obj as Phone;
              if (phone != null)
              {
                  Phone phoneCopy = new Phone
                  {
                      Company = phone.Company,
                      Price = phone.Price,
                      Title = phone.Title
                  };
                  Phones.Insert(0, phoneCopy);
              }
          }));
    }
}

public Phone SelectedPhone
{
    get { return selectedPhone; }
    set
    {
        selectedPhone = value;
        OnPropertyChanged("SelectedPhone");
    }
}

public ApplicationViewModel(IDialogService dialogService, IFileService fileService)
{
    this.dialogService = dialogService;
    this.fileService = fileService;

    // данные по умолчанию
    Phones = new ObservableCollection<Phone>
    {
        new Phone { Title="iPhone 7", Company="Apple", Price=56000 },
        new Phone {Title="Galaxy S7 Edge", Company="Samsung", Price =60000 },
        new Phone {Title="Elite x3", Company="HP", Price=56000 },
        new Phone {Title="Mi5S", Company="Xiaomi", Price=35000 }
    };
}

public event PropertyChangedEventHandler PropertyChanged;
```

```
        public void OnPropertyChanged([CallerMemberName]string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

Для работы с файлами в конструктор ApplicationViewModel передаются объекты IDialogService и IFileService:

```
public ApplicationViewModel(IDialogService dialogService, IFileService fileService)
{
    this.dialogService = dialogService;
    this.fileService = fileService;
    //............
}
```

Затем эти объекты используются в командах OpenCommand и SaveCommand.

Также изменим код MainWindow.xaml:

```
<Window x:Class="MVVM.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:MVVM"
        xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivit
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="FontSize" Value="14" />
        </Style>
        <Style TargetType="Button">
            <Setter Property="Width" Value="40" />
            <Setter Property="Margin" Value="5" />
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="0.8*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="25" />
            <RowDefinition Height="*" />
            <RowDefinition Height="0.2*" />
        </Grid.RowDefinitions>
        <Menu Grid.ColumnSpan="2" >
            <MenuItem Header="Файл">
                <MenuItem Header="Открыть" Command="{Binding OpenCommand}" />
                <MenuItem Header="Сохранить" Command="{Binding SaveCommand}" />
            </MenuItem>
        </Menu>
        <ListBox Grid.Row="1" ItemsSource="{Binding Phones}"
                 SelectedItem="{Binding SelectedPhone}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Margin="5">
                        <TextBlock FontSize="18" Text="{Binding Path=Title}" />
                        <TextBlock Text="{Binding Path=Company}" />
                        <TextBlock Text="{Binding Path=Price}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
        <StackPanel Grid.Row="2" Orientation="Horizontal">
```

```
            <Button Command="{Binding AddCommand}">+</Button>
            <Button Command="{Binding RemoveCommand}"
                    CommandParameter="{Binding SelectedPhone}">-</Button>
            <Button Content="2x">
                <i:Interaction.Triggers>
                    <i:EventTrigger EventName="MouseDoubleClick">
                        <i:InvokeCommandAction
                            Command="{Binding DoubleCommand}"
                            CommandParameter="{Binding SelectedPhone}" />
                    </i:EventTrigger>
                </i:Interaction.Triggers>
            </Button>
        </StackPanel>

        <StackPanel Grid.Row="1" Grid.Column="1" DataContext="{Binding SelectedPhone}">
            <TextBlock Text="Выбранный элемент"  />
            <TextBlock Text="Модель" />
            <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Производитель" />
            <TextBox Text="{Binding Company, UpdateSourceTrigger=PropertyChanged}" />
            <TextBlock Text="Цена" />
            <TextBox Text="{Binding Price, UpdateSourceTrigger=PropertyChanged}" />
        </StackPanel>
    </Grid>
</Window>
```

В отличие от предыдущей темы здесь добавлено меню с двумя пунктами, которые привязаны к командам SaveCommand и OpenCommand:

```
<Menu Grid.ColumnSpan="2" >
    <MenuItem Header="Файл">
        <MenuItem Header="Открыть" Command="{Binding OpenCommand}" />
        <MenuItem Header="Сохранить" Command="{Binding SaveCommand}" />
    </MenuItem>
</Menu>
```

И в конце изменим файл связанного кода MainWindow.xaml.cs:

```
using System.Windows;

namespace MVVM
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            DataContext =
                new ApplicationViewModel(new DefaultDialogService(), new JsonFileService());
        }
    }
}
```

В конструктор ApplicationViewModel для работы с файлами передаются объекты DefaultDialogService и JsonFileService.