



ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЛЕКЦИЯ № 6

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.



ПАТТЕРН MVVM

- Паттерн MVVM (Model-View-ViewModel) позволяет разделить 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)

- Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную 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 (ПРЕДСТАВЛЕНИЕ)

- 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 (МОДЕЛЬ ПРЕДСТАВЛЕНИЯ)

- 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 ViewModel 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

- 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:

- 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.

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

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

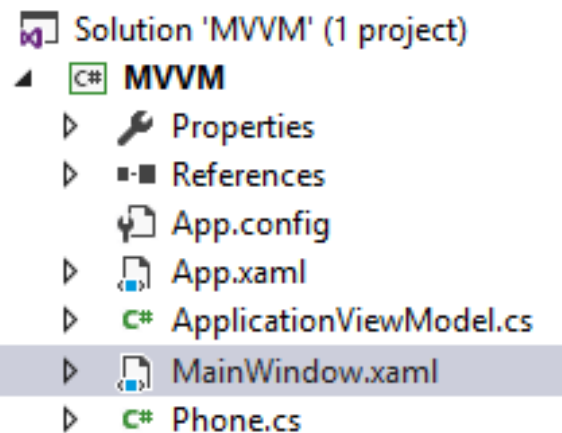
        public string Title
        {
            get { return title; }
            set
            {
                title = value;
                OnPropertyChanged("Title");
            }
        }

        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));
            }
        }
    }
}
```

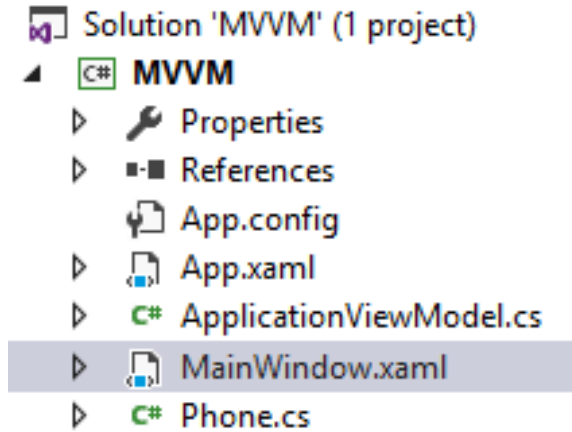

- Также добавим в проект новый класс ApplicationViewModel, который будет представлять модель представления.
- Это класс модели представления, через который будут связаны модель Phone и представление MainWindow.xaml. В этом классе определен список объектов Phone и свойство, которое указывает на выделенный элемент в этом списке.
- В итоге весь проект будет выглядеть следующим образом:



```
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));
    ....    ....}
    ....}
}
```

- В итоге весь проект будет выглядеть следующим образом:



- Далее изменим код нашего представления - файла MainWindow.xaml.
- 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:

The screenshot shows a window titled 'MainWindow' with a list of smartphones on the left and a details panel on the right. The 'Galaxy S7 Edge' is selected, and its details are displayed in the right panel.

Выбранный элемент
Модель
Galaxy S7 Edge
Производитель
Samsung
Цена
50000

Left Panel Data:

Model	Manufacturer	Price
iPhone 7	Apple	56000
Galaxy S7 Edge	Samsung	50000
Elite x3	HP	56000
Mi5S	Xiaomi	35000

- При этом не надо определять код загрузки объектов в ListBox, определять обработчики выбора объекта в списке или сохранения его данных. За нас все делает механизм привязки данных.


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 той, что объявлено в файле `PhoneViewModel.cs`

КОМАНДЫ 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` (файл `RelayCommand.cs`)

- 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<object>. 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<object>.

ПРИМЕНЕНИЕ КОМАНД

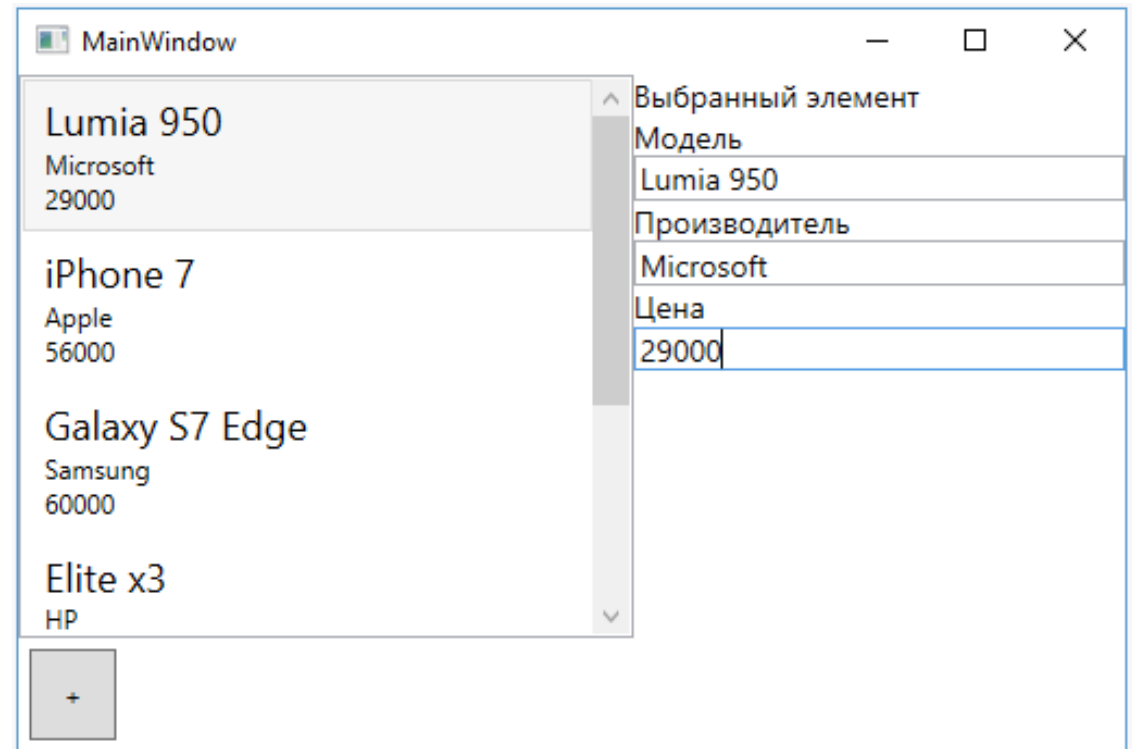
- Для ряда визуальных элементов WPF, например, для кнопок, определена поддержка команд.
- Однако сами команды определяются в ViewModel и затем через механизм привязки устанавливаются для элементов управления.
- Например, изменим код ApplicationViewModel следующим образом, добавим туда (файл ApplicationViewModel1.cs):
- `private RelayCommand addCommand;`
- `public RelayCommand AddCommand;`

-
- Команда хранится в свойстве `AddCommand` и представляет собой объект выше определенного класса `RelayCommand`. Этот объект в конструкторе принимает действие - делегат `Action<object>`.
 - 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` (`MainWindow1.xaml`).

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



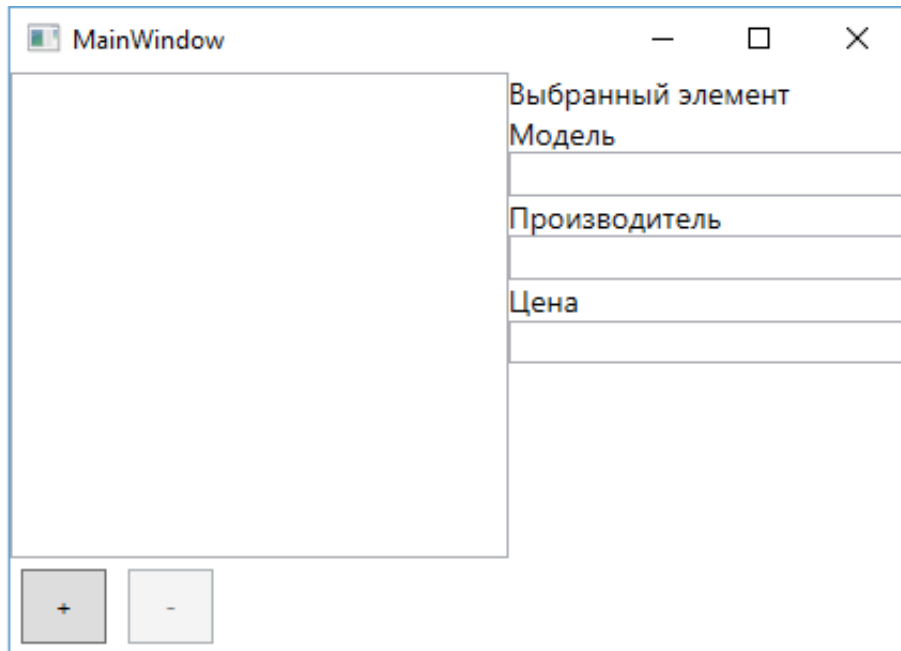
ПЕРЕДАЧА ПАРАМЕТРОВ КОМАНДЕ

- 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 (файл `ApplicationViewModel2.cs`)
- Здесь добавлена команда удаления объекта из списка.
- 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` (файл `MainWindow2.xaml`)

- 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:



MVVM. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА.

Продолжение лекции:

- <https://metanit.com/sharp/wpf/22.5.php>
- <https://metanit.com/sharp/wpf/22.6.php>

Дополнительная литература:

- <https://habr.com/ru/post/338518/>
- <https://habr.com/ru/post/339538/>