*Данная работа является продолжение предыдущей. Программу нужно будет модифицировать согласно требованиям подхода MVVM, а так же внести некоторый новый функционал. Файлы VMBase.cs и Command.cs лежат в той же папке, что и этот документ.*

## Code-behind и ViewModel

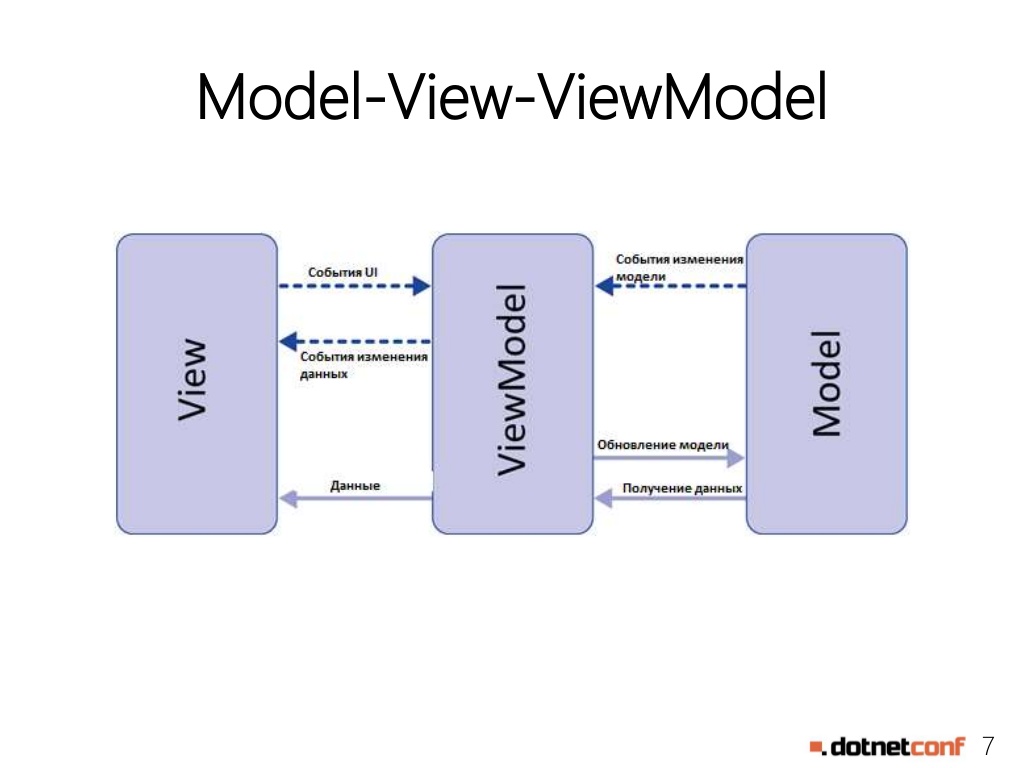
При создании UI-элемента (окно, страница и др.) создаётся два файла, один с расширением .xaml, другой .cs (напимер MainWindow.xaml и MainWindow.cs). Cs-файл содержит обработчики событий данного элемента и называется Code-behind (выделенный код).

Вообще, весь код может быть написан внутри Code-behind (начинающие программисты обычно так и делают). Недостаток такого подхода в том, что работа с компоненты UI, логика их состояния поведения, а также бизнес-логика – всё оказывается «в одной куче». Предположим, потребовалось поменять элементы пользовательского интерфейса (например, поменять выпадающий список на radiobutton) – скорее всего, придётся вносить изменения и в логику поведения элементов, и во взаимодействие с бизнес-логикой. Такие изменения могут быть весьма трудоёмкими, так же есть риск сделать ошибки.

Альтернативный подход – добавить разделение на блоки. В паттерне MVVM блоки следующие:

* *View (представление)*. XAML-файл;
* *Model (модель данных)*. Бизнес-логика (данные предметной области, операции преобразования и правила для манипулирования этими данными);
* *ViewModel (модель представления)*. Преобразование данных бизнес-логики в форму, удобную для отображения (при этом ViewModel ничего не знает о конкретных UI-контролах, находящихся во View).

Каким образом достигается разделение? Создаётся ViewModel – класс, объект которого привязывается к представлению (например окну). Взаимодействие происходит через привязку (Binding) к свойствам этого класса.



В программу из предыдущей работы было добавлено окно, содержащее таблицу GridView. Добавим в проект класс ViewModel. Имя класса может быть любым, но обычно берут имя XAML-файла, и добавляют к нему «ViewModel» или «VM» (чтобы избежать путаницы). Например, файл UI-компонента назывался *GridWindow.xaml*, назовём новый класс *GridWindowVM*.

Установим объект нового класса в качестве контекста данных для окна. Для этого нужно добавить в атрибуты окна пространство имён (namespace) класса:

xmlns:local="clr-namespace:Tpu.Aics.BindingSample"

В этом примере пространству имён присвоено название *local*, но можно использовать любое название.

<Window.DataContext>

<local:GridWindowVM/>

</Window.DataContext>

Данный код означает, что при создании окна будет автоматически создан объект класса *DataGridVM* и этот объек будет установлен в качестве контекста данных окна (т.е. можно выполнять привязку к свойствам данного класса).

Перенесём работу с бизнес-логикой из Code-behind в ViewModel.

public class GridWindowVM

{

public ObservableCollection<MarketItem> Items { get; set; }

public GridWindowVM()

{

// Загрузка содержимого файла в строку

string json = File.ReadAllText(@"..\..\items.json");

// Десериализация JSON-данных в объект MarketItem

Items = JsonConvert.DeserializeObject<ObservableCollection<MarketItem>>(json);

}

}

Заметим, что в code-behind остаётся только автоматически сгенерированный код:

public GridWindow()

{

InitializeComponent();

}

Остальное перешло во ViewModel.

Осталось привязать свойство Items класса ViewModel к представлению (View, XAML)

<ListView ItemsSource="{Binding Items}">

Таким образом в данном примере блок View – это файл *GridWindow.xaml*, блок Model – *MarketItem.cs*, ViewModel - *GridWindowVM*.

## Команды

Как происходит обмен значениями полей между View и ViewModel – понятно. Как быть с действиями, такими как нажатие кнопки? По канонам MVVM в code-behind не должно быть обращений к ViewModel. Есть решение: WPF позволяет осуществлять привязку не только значений, но и действий.

Например, требуется реализовать функционал удаления выбранного товара из таблицы. Для начала нужно привязать к ViewModel выбранный элемент ListView, это просто.

ViewModel:

public MarketItem SelectedMarketItem { get; set; }

View:

<ListView ItemsSource="{Binding Items}" SelectedItem="{Binding SelectedMarketItem}">

Для привязки действий во ViewModel нужно добавить свойства типа ICommand, пространство имён System.Windows.Input (В WPF не включена готовая реализация интерфейса ICommand, можно воспользоваться реализацией из файла Command.cs, нужно добавить этот файл в проект).

Добавляем во ViewModel свойство-команду:

public ICommand DeleteCmd { get; set; }

Добавляем метод для удаления элемента:

private void Delete()

{

Items.Remove(SelectedMarketItem);

}

В конструктор ViewModel добавим инициализацию команды и свяжем её с методом:

DeleteCmd = new Command(arg => Delete());

Осталось привязать команду к кнопке в XAML:

<Button Content="Удалить" Command="{Binding DeleteCmd}"/>

## Обновление значений

Допустим, нам требуется выводить в отдельном текстовом поле некое сообщение.

Добавим во ViewModel поле:

public string Message { get; set; }

В конструкторе зададим свойству Message значение «Данные загружены», а в методе Delete – значение «Запись удалена». Добавим в XAML текстовый элемент и выполним привязку. Запустив приложение, увидим, что сообщение «Данные загружены» отображается, а «Запись удалена» - нет. Так происходит, потому что при изменении значения свойства Message не происходит автоматического уведомления связанного графического элемента.

Для решения данной проблемы используются уведомления об изменении свойств. Класс ViewModel должен быть унаследован от интерфейса INotifyPropertyChanged (namespace System.ComponentModel). Так же, как и с ICommand, стандартной реализации данного интерфейса нет. Можно воспользоваться реализацией из файла VMBase.cs (нужно будет включить файл в проект). Файл содержит класс VMBase, достаточно будет унаследовать ViewModel от этого класса:

public class GridWindowVM : VMBase

Перепишем свойство Message так, чтобы при изменении его значения (set) вызывалось уведомление об изменении этого свойства:

private string \_message;

public string Message

{

get { return \_message; }

set

{

\_message = value;

NotifyPropertyChanged("Message");

}

}

Теперь всё в порядке, сообщение «Запись удалена» отображается.

## Зависимость View от ViewModel

Главное преимущество MVVVM – независимость ViewModel от View. Плюсы очевидны: можно сколько угодно менять UI в XAML, при этом изменять код ViewModel не требуется. Можно в одном приложении разрабатывать несколько вариантов окон и предоставлять пользователю выбор более удобного для него интерфейса, при этом разные окна используют один и тот же контекст данных – класс ViewModel.

Есть и другой принцип MVVM – независимость View от ViewModel. Его преимущества не столь очевидны, однако он порождает дополнительные проблемы. Допустим, по нажатию кнопки требуется открыть дополнительное окно. Если соблюдать принцип независимости View от ViewModel , эта задача сильно усложнится. Поэтому в данной работе такого требования не будет.

Добавим в code-behind окна ссылку на ViewModel:

public GridWindowVM ViewModel

{

get

{

return DataContext as GridWindowVM;

}

}

Теперь можно обращаться к ViewModel из code-behind. Далее будет добавлен функционал открытия другого окна для редактирования. XAML-код (блок View) для такого окна был написан в прошлой работе). В качестве модели данных остаётся класс MarketItem, остаётся создать класс-ViewModel для этого окна. Этот класс будет содержать свойство Model (имя свойства может быть другим)

public MarketItem Model { get; set; }

В XAML-коде нужно изменить привязку к свойствам класса MarketItem:

Было:

<Binding Path="Price.Value"/>

Меняем:

<Binding Path="Model.Price.Value"/>

Теперь можно перейти к созданию и открытию окна редактирования. Добавим в основное окно (с таблицей GridView) кнопку с обычным обработчиком события:

<Button Content="Редактировать" Click="Button\_Click"/>

private void Button\_Click(object sender, RoutedEventArgs e)

{

var wnd = new EditWindow();

wnd.DataContext = new EditWindowVM()

{

Model = ViewModel.SelectedMarketItem

};

wnd.ShowDialog();

}

## Задание

Модифицировать программу из предыдущей работы аналогично данному примеру.

* Создать классы ViewModel для обоих окон;
* Перенести весь функционал из code-behind во ViewModel (кроме функции открытия окна редактирования);
* Сделать функции создания нового элемента, редактирования и удаления;
* В главном окне сделать поля для сообщений о загрузке данных, создании, удалении и изменении элемента;
* Сделать функцию сохранения в исходный JSON-файл.