ПРАКТИЧЕСКАЯ РАБОТА № 4

Тема: Модель событий в WPF, обработка команд в MVVM. Привязка данных в WPF.

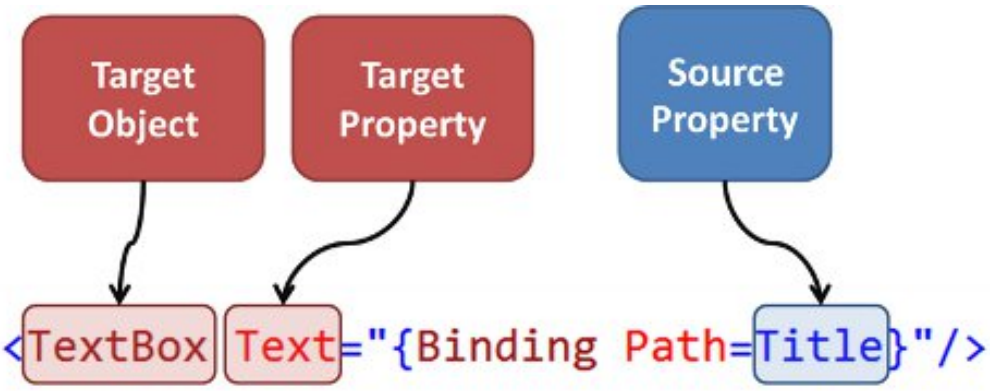
Цель: освоить методики обработки событий и команд в WPF. Научиться задавать корректную привязку данных к элементам формы.

Перечень оснащения и оборудования, источников: ПК, раздаточный материал, система управления базами данных, среда разработки Microsoft Visual Studio.

Задание: написать обработчик команд RelayCommand, а также привязать поля ввода логина и пароля

№ 1. В MVVM реализован очень важный механизм, обеспечивающий связь между данными в окне и моделью предсталвения: **Data Binding** (привязка данных).

Data Binding (привязка данных) – это механизм в WPF, который автоматически синхронизирует данные между источником (обычно объектом в коде) и элементом пользовательского интерфейса (UI), без необходимости писать ручной код для обновления значений.



Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. В случае модификации объекта-источника, объект-приемник также будет модифицирован. Таким образом, в привязке данных задействованы 2 основных компонента:

1. **Источник данных**, который может быть представлен как свойство другого объекта или класса, коллекция элементов, элемент управления и др.
2. **Цель**, представляющая собой свойство элемента, которое будет обновляться.

WPF **сам следит** за изменениями и автоматически обновляет UI.

У Binding есть ряд свойств, которые могут быть определены вместе с привязкой:

1. Path – указатель на данные, которые нужно привязать.

2. Mode – режим привязки (один из предложенных в таблице на пред. слайде).

3. UpdateSourceTrigger – когда нужно обновить данные.

4. StringFormat – форматирование строки.

5. FallbackValue – сообщение, которое будет выводиться при ошибке.

6. TargetNullValue – что показывать, если значение null.

7. Source, RelativeSource, ElementName – откуда брать данные.

Стандартный вариант привязки предполагает подстановку значения переменной из модели представления в соответствующий элемент в окне (или наоборот: считывание данных с окна с последующей обработкой в модели представления). Выглядит пример такой привязки следующим образом:

<TextBox **Text="{Binding login}"**/>

Для того, чтобы привязка корректно работала, необходимо определить контекст данных, то есть связать модель представления с самим представлением. Для этого в Code-Behind окна авторизации (Auth.xaml.cs) задайте контекст данных следующим образом (предварительно создайте модель представления для окна авторизации AuthViewModel):

DataContext = new AuthViewModel();

Теперь, в модели представления создайте 2 новых переменных: _login (private) и login (public). Не забудьте указать get и set для публичного атрибута. Но на данный момент времени модель представления не будет реагировать на изменения в самом представлении (если попробуем ввести данные в поле для логина, то сама переменная login не будет обновлена). Дело в том, что динамическое обновление нужно настроить вручную.

ViewModel служит посредником между View и Model, предоставляя данные в форме, удобной для отображения в UI, и обрабатывая пользовательские действия.

Для динамического обновления содержимого интерфейса пользователя необходимо уведомлять элементы об изменении тех или иных свойств (например, поменялось содержимое таблицы в БД).

В MVVM модели представления (ViewModels) наследуют один общий интерфейс – **INotifyPropertyChanged**.

INotifyPropertyChanged – это системный интерфейс из пространства имён “System.ComponentModel”, который позволяет объекту уведомлять интерфейс пользователя о том, что оно из его свойств изменилось.

Интерфейс содержит **только одно событие**: PropertyChanged. Никаких методов в нём нет.

Таким образом, структура интерфейса выглядит следующим образом:

```
namespace System.ComponentModel
{
    public interface INotifyPropertyChanged
    {
        event PropertyChangedEventHandler PropertyChanged;
    }
}
```

PropertyChanged - это событие, которое уведомляет подписчиков об изменении свойства, содержит информацию о том, какое именно свойство изменилось и автоматически вызывает обновление UI в WPF. Ключевое слово event представляет собой механизм обновления. Он позволяет объекту (модели или ViewModel) отправлять сигнал другим объектам о том, что произошло событие.

PropertyChangedEventHandler представляет собой тип делегата, на основе которого создаётся событие PropertyChanged.

То есть – это шаблон функции, которая должна обрабатывать событие.

Создайте новую модель представления BaseViewModel, которую будут наследовать все остальные модели представления. Это нужно для того, чтобы не писать для каждой модели представления дополнительный обработчик событий, что будет усложнять программный код.

Модель представления BaseViewModel должна наследовать интерфейс INotifyPropertyChanged. В BaseViewModel объявите событие PropertyChanged, как это было показано выше.

Данное событие нужно вызвать в случае возникновения изменений в данных. Создайте публичный метод OnPropertyChanged для удобного вызова уведомлений об изменении свойств и задайте ему параметр propertyName, который по умолчанию будет равен null. Параметр propertyName = null означает, что можно вызывать метод без указания имени свойства.

Должна получиться следующая структура модели представления:

```
public class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void OnPropertyChanged(string propertyName = null)
    {

    }
}
```

Теперь, в методе OnPropertyChanged нужно вызвать событие изменения состояния объекта. Для этого реализован метод **Invoke**, который в качестве входных данных принимает 2 параметра:

1. Ссылку на объект, который вызвал изменение. В большинстве случаев указывается значение **this**.

2. Объект с информацией о том, какое свойство изменилось. Для упрощения структуры класса можно прямо на позиции параметра создавать объект класса PropertyChangedEventArgs, который будет хранить соответствующую информацию об объекте (нужно указать только название свойства propertyName).

Таким образом, получаем следующую реализацию метода OnPropertyChanged:

```
public void OnPropertyChanged(string propertyName = null)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Получаем следующую базовую модель представления:

```
public class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void OnPropertyChanged(string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Внимание: все остальные модели представления должны наследовать BaseViewModel.

№ 2. Теперь нужно изменить атрибуты _login и login в модели представления AuthModelView:

1. Присвойте любое значение атрибуту _login.

2. В login методу get передавайте значение атрибута _login, а в методе set установите значение value для атрибута _login и вызовите метод OnPropertyChanged.

Проверьте работу программы.

№ 3. С привязками неразрывно связаны команды.

Команда в WPF – это объект, реализующий интерфейс ICommand, который:

1. **Определяет, что нужно сделать** (Execute).
2. **Когда это можно сделать** (CanExecute).

Кнопка, меню, сочетание клавиш и даже элементы контекстного меню могут привязываться к команде.

Основная идея команд следующая:

WPF-команды позволяют написать:

```
<Button Command="ApplicationCommands.Copy" Content="Копировать" />
```

и система сама:

- знает, когда команду можно выполнить (например, если есть выделенный текст);
- автоматически вызывает метод Execute при нажатии;
- отключает кнопку, если команду нельзя выполнить (CanExecute == false).

В WPF также предусмотрена возможность создания собственных команд через ICommand.

Интерфейс ICommand – это один из ключевых элементов архитектуры MVVM (Model-View-ViewModel) в WPF и других XAML-фреймворках. Он используется для связывания действий пользователя (например, нажатий кнопок) с логикой приложения (командами во ViewModel).

Без него, обработчики событий (например, Button.Click) создают сильную связь между представлением (View) и логикой (ViewModel), нарушая принципы MVVM.ICommand решает эту проблему, позволяя:

1. Вызывать методы ViewModel напрямую из View.
2. Автоматически управлять доступностью кнопок (например, "Сохранить" становится неактивной, если данные невалидны).
3. Обеспечивать чистое разделение логики и интерфейса.

Интерфейс находится в пространстве имен System.Windows.Input и содержит три компонента:

```
public interface ICommand
{
    event EventHandler CanExecuteChanged;

    bool CanExecute(object parameter);
    void Execute(object parameter);
}
```

1. **Execute(object parameter)** – выполняет команду (например, сохраняет данные, удаляет запись и т.д.);
2. **CanExecute(object parameter)** – определяет, можно ли выполнить команду (например, кнопка “Сохранить” активна только при заполненных полях);
3. **CanExecuteChanged** – событие, которое уведомляет систему, что условия выполнения изменились (чтобы обновить состояние элементов управления, связанных с командой).

На практике не пишут под каждую команду реализацию ICommand, так как это занимает много времени. **Зачастую в рамках проекта пишут универсальную реализацию этого интерфейса**. Такая реализация позволяет не создавать реализации интерфейса на каждую команду в приложении, а обрабатывать их через один общий класс (например, **RelayCommand** или DelegateCommand).

Привязка таких команд к элементам выглядит следующим образом:

```
<Button Content="Сохранить" Command="{Binding SaveCommand}" />
```

Как написать класс RelayCommand в WPF:

1. Определить задачу создания такой реализации интерфейса ICommand.

Например, нам нужно создать класс, который:

- реализует ICommand;
- может выполнять любой метод (Action);
- может определять, активна команда или нет (Func<bool>).

2. Объявить класс RelayCommand и реализовать интерфейс (задать пустой шаблон).

Класс RelayCommand должен наследовать интерфейс ICommand и всё его содержимое:

```
public class RelayCommand : ICommand
{
    public event EventHandler? CanExecuteChanged;

    public bool CanExecute(object? parameter)
    {
        throw new NotImplementedException();
    }

    public void Execute(object? parameter)
    {
        throw new NotImplementedException();
    }
}
```

3. Добавить поля для делегатов интерфейса (метод _execute, выполняющий действие и метод _canExecute, проверяющий возможность выполнения действия).

_execute представляет собой действие в приложении, поэтому у него делегат Action.

_canExecute проверяет возможность выполнения действия и должен возвращать либо true, либо false, поэтому делегат Func<bool>.

```
public class RelayCommand : ICommand
{
    private readonly Action _execute;          // метод, выполняющий действие
    private readonly Func<bool>? _canExecute;  // метод, проверяющий возможность выполнения

    public event EventHandler? CanExecuteChanged;

    public bool CanExecute(object? parameter)
    {
        throw new NotImplementedException();
    }

    public void Execute(object? parameter)
    {
        throw new NotImplementedException();
    }
}
```

4. Создать конструктор класса, который в качестве параметров принимает делегаты execute и canExecute (canExecute необязательный, поэтому приравнивается к null).

В конструкторе значения из входных параметров передаются в _execute и _canExecute.

```
private readonly Action _execute;
private readonly Func<bool>? _canExecute;

public RelayCommand(Action execute, Func<bool>? canExecute = null)
{
    _execute = execute;
    _canExecute = canExecute;
}
```

5. Реализовать методы CanExecute и Execute.

CanExecute должен возвращать true или false – можно ли выполнять команду сейчас.

Если _canExecute не задан – команда всегда доступна.

Если задан – возвращается результат вызова делегата Func<bool>.

Execute – это метод, который выполняет основное действие команды.

Когда кнопка в XAML нажимается, WPF вызывает Execute(), а он передаёт управление методу. Основное действие хранится в _execute, оно и будет выполнено.

```
public bool CanExecute(object? parameter)
{
    return _canExecute == null || _canExecute();
}
```

6. Добавить уведомление об изменении состояния.

```
public event EventHandler? CanExecuteChanged;

public bool CanExecute(object? parameter)
{
    return _canExecute == null || _canExecute();
}

public void Execute(object? parameter)
{
    _execute();
}
```

Должен получиться следующий обработчик команд:

```
public class RelayCommand : ICommand
{
    private readonly Action _execute;
    private readonly Func<bool>? _canExecute;

    public RelayCommand(Action execute, Func<bool>? canExecute = null)
    {
        _execute = execute;
        _canExecute = canExecute;
    }

    public event EventHandler? CanExecuteChanged;
    public bool CanExecute(object? parameter)
    {
        return _canExecute == null || _canExecute();
    }

    public void Execute(object? parameter)
    {
        _execute();
    }
}
```

Создайте в AuthViewModel команду для авторизации в приложении и привяжите к команде метод авторизации (также реализуйте его в AuthViewModel): **LoginCommand = new RelayCommand(OnLogin)**. Команду привяжите к кнопке авторизации (свойство кнопки Command). Напишите логику авторизации в приложении и проверьте работу окна.