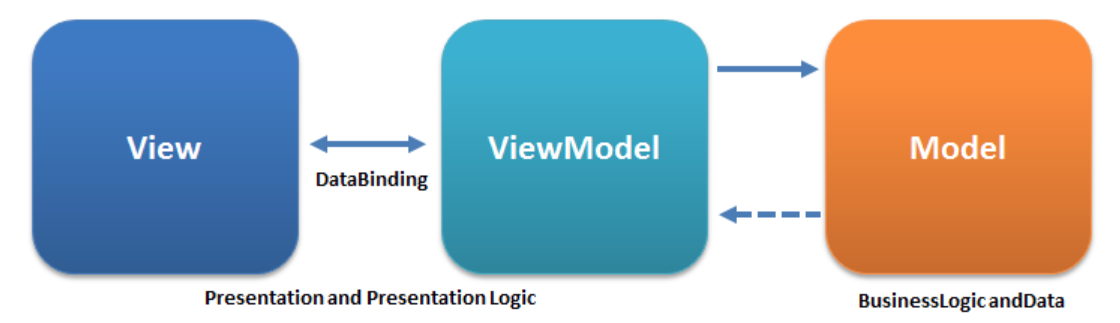
**Wzorzec MVVM**



Mamy trzy warstwy  
Warstwa View mamy wszystkie widoki, czyli pliki xaml.  
W warstwie Model mamy dane: nasze klasy domenowe, obiekty biznesowe i cała logika biznesowa.  
Warstwa ViewModel łączy warstwę widoku z warstwą modelu.   
Gwarantuje nam, że po wykonaniu jakieś akcji z widoku zostanie wywołana odpowiednia logika biznesowa. Warstwa ta komunikuje się z widokiem za pomocą wiązania danych.  
Model nie potrzebuje żadnych danych z warstwy View i ViewModel, równie dobrze mogła by być przeniesiona do innej aplikacji.  
Dzięki temu sam widok może projektować osoba, która nawet nie musi programować w C#.  
Mamy w aplikacji większy porządek. Możemy pisać testy jednostkowe.

**Zewnętrzne Frameworki Ułatwiające Pisanie Aplikacji w WPF:**

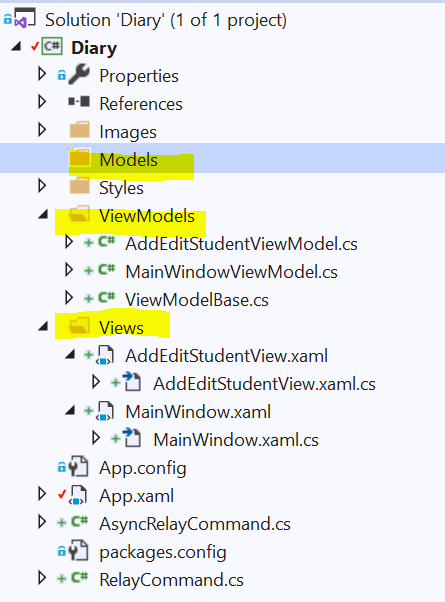
**Prism**Caliburn.Micro  
MVVM Light

Z powodzeniem możemy również pisać duże aplikacje, nie używając żadnego frameworka.

Frameworki te opakowują wiele mechanizmów, pozwalają pisać używając mniej kodu i poświęcając mniej czasu.

**Szkielet MVVM**

Tworzymy katalogi dla każdej z warstw:



Widoki, które już mamy przenosimy do katalogu Views, po czym modyfikujemy namespace’y  
w xaml i w codebehind dla każdego z przeniesionych widoków.

|  |
| --- |
| <mah:MetroWindow x:Class="Diary.Views.MainWindow"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" |

|  |
| --- |
| namespace Diary.Views  {  /// <summary>  /// Interaction logic for MainWindow.xaml  /// </summary>  public partial class MainWindow : MetroWindow  {  public MainWindow() |

W App.xaml modyfikujemy ścieżkę do startowego widoku

|  |
| --- |
| <Application x:Class="Diary.App"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:local="clr-namespace:Diary"  StartupUri="Views/MainWindow.xaml">  <Application.Resources> |

W katalogu ViewModel tworzymy pliki z klasami po jednym pliku dla każdego z widoków  
Nazwę klasy tworzymy dodając do nazwy widoku słowo ViewModel.  
Tworzymy jeden bazowy ViewModel np. ViewModelBase, po którym będą dziedziczyć pozostałe ViewModel’e

|  |
| --- |
| namespace Diary.ViewModels  {  class ViewModelBase : INotifyPropertyChanged  {  // klasa implementująca intrfejs INotifyPropertyChanged  // musi mieć zadeklarowany event PropertyChanged  // dodatkowo dodajemy metodę pomocniczą onPropertyChanged  public event PropertyChangedEventHandler PropertyChanged;  // virtual - może zostać nadpisana w klasach pochodnych  // [CallerMemberName] dzięki temu nie musimy wpisywać nazwy "MyProperty"  protected virtual void onPropertyChanged([CallerMemberName] string propertyName = null)  {  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));  }  }  } |

|  |
| --- |
| // przykładowa właściwość zgłaszająca, że została zmieniona  // snipet propfull  private int \_myProperty;  public int MyProperty  {  get { return \_myProperty; }  set  {  \_myProperty = value;  // w set zgłaszamy zdażenie jako parametr przekazujemy nazwę właściwości  // onPropertyChanged("MyProperty");  // dzięki atrybutowi [CallerMemberName] nie musimy jawnie wpisywać  // a w zasadzie przepisywać nazwy metody  onPropertyChanged();  }  } |

Każdy nowy widok będzie dziedziczył po ViewModelsBase i będzie mógł wywoływać metodę onPropertyChanged.  
Dzięki tej metodzie będziemy mogli poprawnie **bindować właściwości**

|  |
| --- |
| namespace Diary.ViewModels  {  class MainWindowViewModel : ViewModelBase  { |

|  |
| --- |
| namespace Diary.ViewModels  {  class AddEditStudentViewModel : ViewModelBase  {  }  } |

**Bindowanie Zdarzeń**

Tworzymy katalog Commands**.** W tym katalogu umieszczamy RelayCommand.cs i wersję asynchroniczną AsyncRelayCommand.cs - klasy te pomogą nam w bindowaniu zdarzeń.  
Klasy te opakowują interfejs ICommand i upraszczają jej stosowanie.

|  |
| --- |
| namespace Diary.Commands  {  // może się ona dowolnie nazywać, przeważnie się nazywa RelayCommand lub DelegateCommand  // musi implementować interfejs ICommand  // delegat przekazany jako pierwszy to metoda jaka ma być wykonana  // delegat przekazany jako drugi - czy dana metoda może być wykonana    public class RelayCommand : ICommand  {  readonly Action<object> \_execute;  readonly Predicate<object> \_canExecute;  public RelayCommand(Action<object> execute)  : this(execute, null)  {  }  public RelayCommand(Action<object> execute, Predicate<object> canExecute)  {  \_execute = execute ?? throw new ArgumentNullException("execute");  \_canExecute = canExecute;  }  public bool CanExecute(object parameter)  {  return \_canExecute == null || \_canExecute(parameter);  }  public event EventHandler CanExecuteChanged  {  add  {  CommandManager.RequerySuggested += value;  }  remove  {  CommandManager.RequerySuggested -= value;  }  }  public void Execute(object parameter)  {  \_execute(parameter);  }  }  } |

**Wiązanie danych pomiędzy View a ViewModel**

**KROK 1**

Każdy z utworzonych widoków musimy powiązać z jego ViewModel’em.  
W widoku View wskazujemu jego ViewModel

Mamy dwie możliwości:  
1) w pliku **XAML**,  
2) w **codebehind** oknawidoku, co jest wygodniejsze.

W konstruktorze okna, do właściwości DataContext przypisujemy jego ViewModel.

|  |
| --- |
| namespace Diary.Views  {  /// <summary>  /// Interaction logic for MainWindow.xaml  /// </summary>  public partial class MainWindow : MetroWindow  {  public MainWindow()  {  InitializeComponent();  DataContext = new MainWindowViewModel();  }  }  } |

**KROK 2**

**DataBinding - Bindujemy zdarzenia.**

Przykładowo bindujemy zdarzenie Click() przycisku odśwież.  
W tym miejscu wskazujemy na właściwość **ICommand** czyli również **RelayCommand**, która musi się znaleźć w ViewModelu.

|  |
| --- |
| <Button Command ="{Binding RefreshStudentsCommand}" Content="Odśwież" Height="30" Width="100" Margin="5"/> |

W **ViewModel** tworzymy właściwość wskazaną pod przyciskiem jako Command, czyli: RefreshStudentsCommand.  
Następnie w konstruktorze inicjalizujemy tą właściwość, używamy do tego klasy RelayCommand  
i przekazujemy metodę, która ma się wykonać, czyli delegata wskazującego na tę metodę

|  |
| --- |
| Namespace Diary.ViewModels  {  class MainWindowViewModel : ViewModelBase  {    public ICommand RefreshStudentsCommand { get; set; }  public MainWindowViewModel()  {  RefreshStudentsCommand = new RelayCommand(RefreshStudents,CanRefreshStudents);  }    }  } |

Następnie klikamy na tych delegatach i wywołujemy **generate method.**  
Generują się metody zgodne z sygnaturą konstruktora klasy **RelayCommand**

|  |
| --- |
| Public MainWindowViewModel()  {  RefreshStudentsCommand = new RelayCommand(RefreshStudents,CanRefreshStudents);  }  private bool CanRefreshStudents(object obj)  {  throw new NotImplementedException();  }  private void RefreshStudents(object obj)  {  throw new NotImplementedException();  } |

Pod właściwość typu **ICommand** RefreshStudentsCommand możemy podstawić obiekt klasy **RelayCommand**, ponieważ ta klasa implementuje ten interfejs ICommand.

**KROK 3**

**Bindowanie zwykłych właściwości**

W View

|  |
| --- |
| <Label Grid.Row="1" Content="Adres Serwera" Margin="5"/>  <TextBox  Text ="{Binding ServerAddress}"  Grid.Row="1"  Grid.Column="1"  Margin="5"/> |

W ViewModel

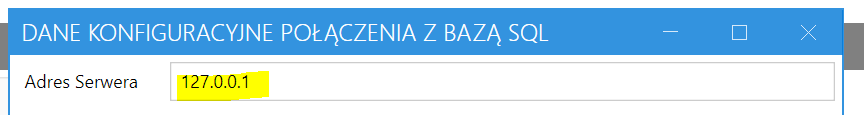
|  |
| --- |
| private string \_serverAddress;  public string ServerAddress  {  get  {  return \_serverAddress;  }  set  {  \_serverAddress = value;  OnPropertyChanged();  }  } |

|  |
| --- |
| **Typy Bindowania danych**  **TwoWay** – domyslny tryb wiązania dla kontrolki TextBox, kontrolki mogą mieć różne tryby domyślne zmiana wartości w kontrolce powoduje zmianę w ViewModel, zmiana wartości w ViewModelu aktualizuje również widok.  **OneWay** - Zmiana w widoku, nie aktualizuje w ViewModel’u  **OneTime** - tak sama jak OneWay ale tylko jeden raz  **OneWayToSource** – odwrotnie do OneWay |

|  |
| --- |
| <TextBox Text="{Binding ServerAddress, Mode=TwoWay}" Height="20" Width="100" Margin="5"/> |

Dla przykładu jeżeli w konstruktorze ViewModel do zmiennej ServerAddress przypiszemy wartość  
„127.0.0.1” to zostanie ona wyświetlona w odpowiednim polu tekstowym w View.

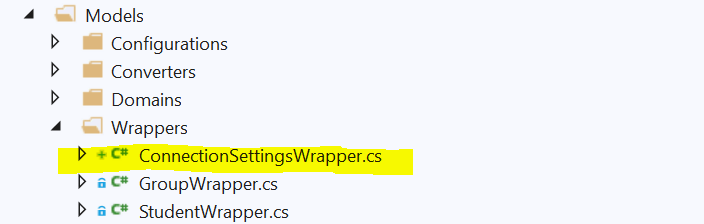
|  |
| --- |
| public ConnectionConfigurationViewModel()  {  CloseCommand = new RelayCommand(Close);  ConfirmCommand = new RelayCommand(Confirm);  ServerAddress = "127.0.0.1";  } |



Jeżeli ją w tym oknie zmienimy, to zmieni się również wartość **ServerAddress** w ViewModel.

Podobnie wygląda gdy wykorzystamy warstwę Model

W ModelsWrapper tworzymy klasę ConnectionSettingsWrapper



|  |
| --- |
| namespace Diary.Models.Wrappers  {  public class ConnectionSettingsWrapper  {  public string ServerAddress { get; set; }  public string ServerName { get; set; }  public string Database { get; set; }  public string User { get; set; }  public string Password { get; set; }  }  } |

W ViewModel tworzymy właściwość ConnectionSettings typu ConnectionSettingsWrapper

|  |
| --- |
| private ConnectionSettingsWrapper \_connectionSettings;  public ConnectionSettingsWrapper ConnectionSettings  {  get  {  return \_connectionSettings;  }  set  {  \_connectionSettings = value;  OnPropertyChanged();  }  } |

W Widoku XAML modyfikujemy wiązanie danych

|  |
| --- |
| <Label Grid.Row="2" Content="Nazwa Instancji" Margin="5"/>  <TextBox  Text ="{Binding ConnectionSettings.ServerName}"  Grid.Row="2"  Grid.Column="1"  Margin="5"/> |

W modelu widoku modyfikujemy zapis i odczyt

Co ważne w odczycie najpierw tworzymy obiekt, aby mieć dostęp do jego składowych

|  |
| --- |
| private void ReadSettings()  {  ConnectionSettings = new ConnectionSettingsWrapper();  ConnectionSettings.ServerAddress = Settings.Default.ServerAddress;  ConnectionSettings.ServerName = Settings.Default.ServerName;  ConnectionSettings.Database = Settings.Default.Database;  ConnectionSettings.User = Settings.Default.User;  ConnectionSettings.Password = Settings.Default.Password;  }  private void SaveSettings()  {  Settings.Default.ServerAddress = ConnectionSettings.ServerAddress;  Settings.Default.ServerName = ConnectionSettings.ServerName;  Settings.Default.Database = ConnectionSettings.Database;  Settings.Default.User = ConnectionSettings.User;  Settings.Default.Password = ConnectionSettings.Password;  Settings.Default.Save();  } |

Logika Biznesowa – wyświetlamy dane na sztywno wpisane z kodu

Tworzymy Obiekty Student, Group w Model

Ustawiamy ItemSource i SelectedItem w DataGrid  
SelectedItem – jeżeli jakiś element w Gridzie zostanie zaznaczony to będziemy mogli się do niego odwołać w ViewModel możemy wskazać we ViewModel element, który ma zostać zaznaczony

|  |
| --- |
| <DataGrid  Grid.Row="1"  Margin="5"  ItemsSource ="{Binding Students}"  SelectedItem="{Binding SelectedStudent}"  /> |

Dodajemy we ViewModel nowe właściwości

|  |
| --- |
| private Student \_selectedStudent;  public Student SelectedStudent  {  get { return \_selectedStudent; }  set  {  \_selectedStudent = value;  OnPropertyChanged();  }  } |

|  |
| --- |
| // używamy zamiast List<> zachowuje się jak zwykła lista  // implementuje dodatkowo interfejsy INotifyCollectionChanged, INotifyPropertyChanged  // dzięki temu datagrid będzie informowany o tym  // czy jekiś element został dodany lub zmieniony  private ObservableCollection<Student> \_students;  public ObservableCollection<Student> Students  {  get { return \_students; }  set  {  \_students = value;  OnPropertyChanged();  }  } |

Ustawienie DataGrid

|  |
| --- |
| <DataGrid  Grid.Row="1"  Margin="5"  ItemsSource ="{Binding Students}"  SelectedItem="{Binding SelectedStudents}"  CanUserAddRows="False"  IsReadOnly="True"  SelectionMode="Single"  AutoGenerateColumns="False">  <DataGrid.Columns>  <DataGridTextColumn Binding="{Binding Id}" Visibility="Hidden"/>  <DataGridTextColumn Binding="{Binding FirstName}" Header="Imię"/>  <DataGridTextColumn Binding="{Binding LastName}" Header="Nazwisko"/>  <DataGridTextColumn Binding="{Binding Comments}" Header="Uwagi"/>  <DataGridTextColumn Binding="{Binding Math}" Header="Matematyka"/>  <DataGridTextColumn Binding="{Binding Technology}" Header="Technologia"/>  <DataGridTextColumn Binding="{Binding Physisc}" Header="Fizyka"/>  <DataGridTextColumn Binding="{Binding PolishLang}" Header="Język Polski" />  <DataGridTextColumn Binding="{Binding ForeignLang}" Header="Język Obcy" />  <DataGridCheckBoxColumn Binding="{Binding Activities}" Header="Act" />  <DataGridTextColumn Binding="{Binding Group.Name}" Header="Grupa"/>  </DataGrid.Columns>  </DataGrid> |

Ustawienia ComboBox

|  |
| --- |
| <ComboBox  Height="30"  Width="150"  Margin="5"  ItemsSource="{Binding Groups}"  SelectedValue="{Binding SelectedGroupId}"  DisplayMemberPath="Name"  SelectedValuePath="Id"  />  DisplayMemberPath – co ma być wyświetlane  SelectedValuePath – co ma być zwracane po wyborze |

Ustawienie Button

|  |
| --- |
| <Button  Command ="{Binding DeleteStudentsCommand}"  CommandParameter="{Binding SelectedStudent}"  Content="Usuń"  Height="30"  Width="100"  Margin="5"  />  CommandParameter – parameter przekazany do metody |

Zamykanie okna

W View

|  |
| --- |
| <mah:MetroWindow x:Class="Diary.Views.AddEditStudentView"  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:mah="clr-namespace:MahApps.Metro.Controls;assembly=MahApps.Metro"  xmlns:local="clr-namespace:Diary"  mc:Ignorable="d"  Name="AddEditStudentView"  WindowStartupLocation="CenterScreen"  Title="Dane ucznia" Height="520" Width="400"> |

|  |
| --- |
| <Button  Command="{Binding ConfirmCommand}"  CommandParameter="{Binding ElementName=AddEditStudentView}"  Content="Zatwierdź"  Height="30"  Width="100"  Margin="5"/> |

W modelView

|  |
| --- |
| public ICommand CloseCommand { get; set; }  public ICommand ConfirmCommand { get; set; }  (…)    CloseCommand = new RelayCommand(Close);  ConfirmCommand = new RelayCommand(Confirm);  (…)  private void Close(object obj)  {  // Obiekt rzutujemy na Window  CloseWindow(obj as Window);  }  private void Confirm(object obj)  {  // TODO: Logika zapisu  CloseWindow(obj as Window);  }  private void CloseWindow(Window window)  {  window.Close();  } |

Tworzymy View

Tworzymy ViewModel

W View  
jako DataContext ustawiamy ViewModel