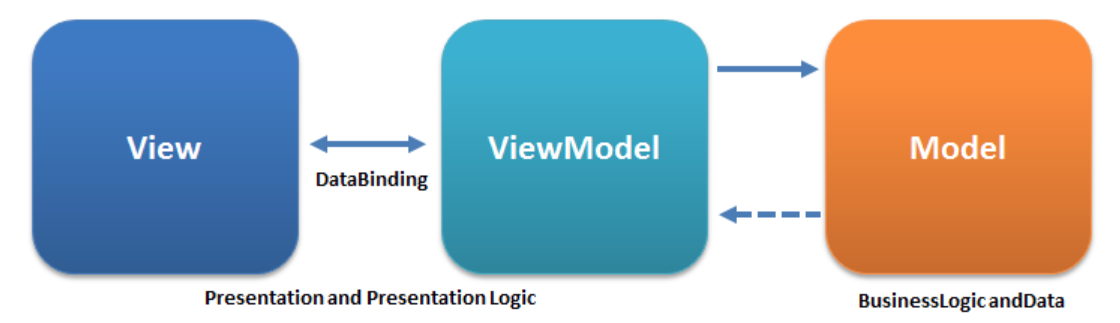
**Wzorzec MVVM**



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:**

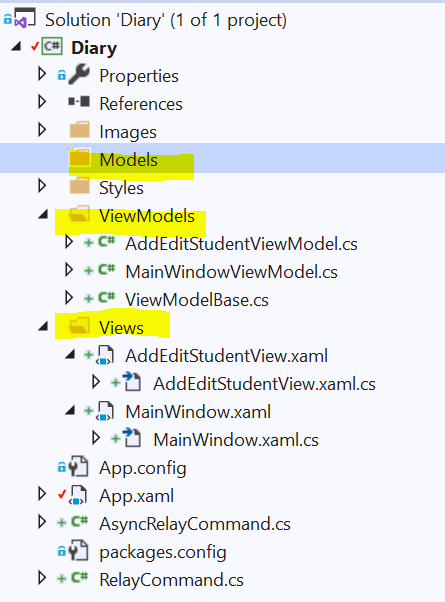
**Prism**Caliburn.Micro  
MVVM Light

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**

1 krok wskazanie w View jego ViewModelu

Możemy to zrobić w pliku xaml, ale wygodniej jest to zrobić w codebehind.  
Ten krok powtarzamy dla każdego widoku.

|  |
| --- |
| 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 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, 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 metodach i wywołujemy generate method  
generują się metody zgodne z sygnaturą konstruktora kalsy 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 ICommand RefreshStudentsCommand możemy podstawić obiekt klasy RelayCommand, ponieważ ten implementuje ten interfejs ICommand.

Bindowanie zwykłych właściwości

W View element Text={””}

|  |
| --- |
| <TextBlock Text="{Binding Test}" Height="20" Width="100" Margin="5" Background="AliceBlue"/> |

W ViewModel

|  |
| --- |
| private string \_test = "XXX";  public string Test  {  get { return \_test; }  set  {  \_test = value;  onPropertyChanged();  }  } |

Typ 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 Test, Mode=TwoWay}" Height="20" Width="100" Margin="5"/> |

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();  }  } |