MVVM

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Every good developer wants and tries to create the most sophisticated applications to delight their users. Most of the times, developers achieve this on the first release of the application. However, with new feature addition, fixing the bug without putting a lot of consideration into the structure of the application code becomes difficult due to code complexity. For this, there is a need for good clean structure of code.

In this tutorial, you will learn how to reduce code complexity and how to maintain a clean and reusable structure of your code by using MVVM pattern.

Audience

This tutorial is designed for software developers who want to learn how to develop quality applications with clean structure of code.

Prerequisites

MVVM is a pattern that is used while dealing with views created primarily using WPF technology. Therefore, it would help a great deal if you have prior exposure to WPF and its bindings.

Disclaimer & Copyright

Table of Contents

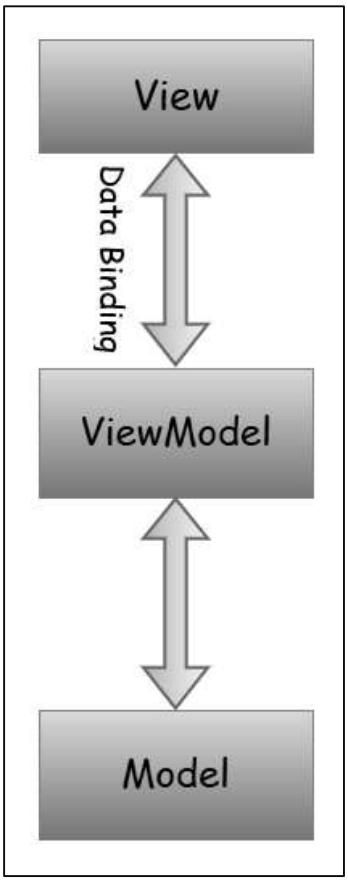
1. MVVM – Introduction

The well-ordered and perhaps the most reusable way to organize your code is to use the 'MVVM' pattern. The **Model, View, ViewModel (MVVM pattern)** is all about guiding you in how to organize and structure your code to write maintainable, testable and extensible applications.

Model: It simply holds the data and has nothing to do with any of the business logic.

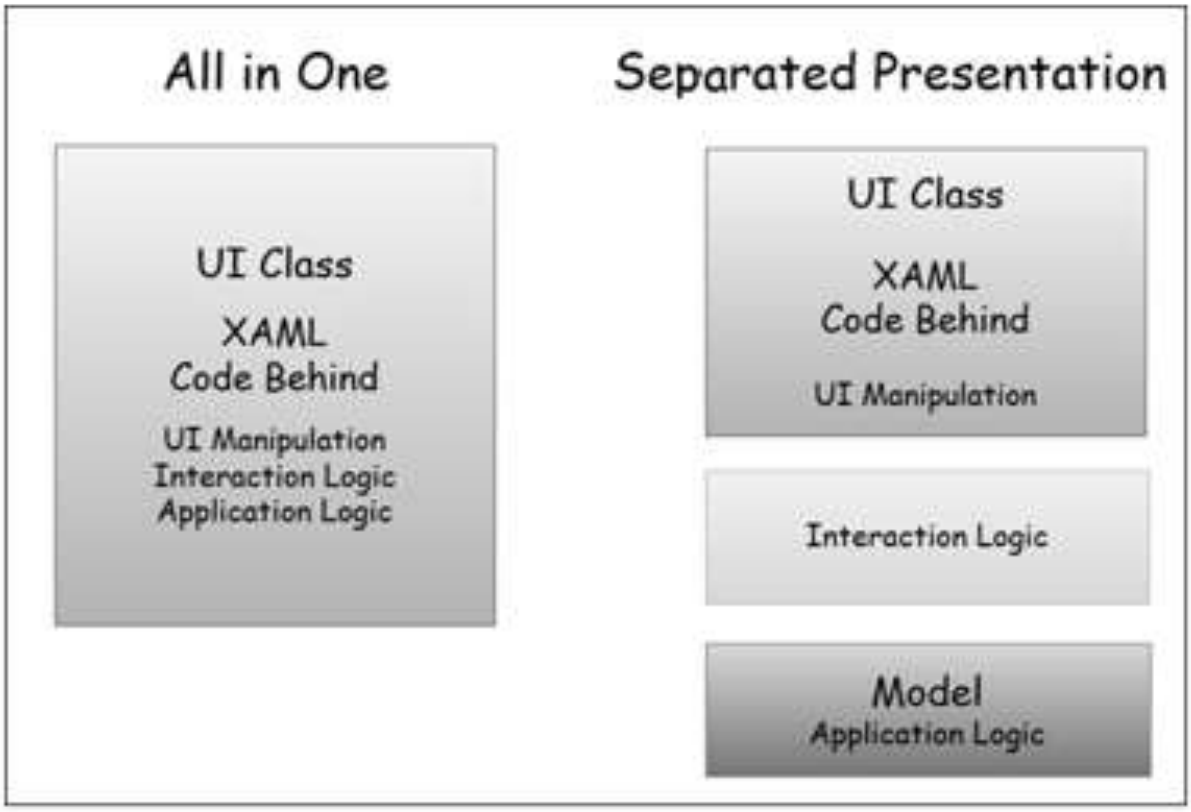
ViewModel: It acts as the link/connection between the Model and ViewModel and makes stuff look pretty.

View: It simply holds the formatted date and essentially delegates everything to the Model.



Separated Presentation

To avoid the problems caused by putting application logic in code-behind or XAML, it's best to use a technique known as separated presentation. We're trying to avoid this, where we will have XAML and code-behind with the minimum required for working with user interface objects directly. User interface classes also contain code for complex interaction behaviors, application logic, and everything else as shown in the following figure on the left side.



- With separated presentation, the user interface class is much simpler. It has the XAML of course, but the code behind does as little as is practical.
- The application logic belongs in a separate class, which is often referred to as the model.
- However, this is not the whole story. If you stop here, you're likely to repeat a very common mistake that will lead you down the path of data binding insanity.
- A lot of developers attempt to use data binding to connect elements in the XAML directly to properties in the model.

- Now sometimes this can be okay, but often it's not. The problem is the model is entirely concerned with matters of what the application does, and not with how the user interacts with the application.

- The way in which you present data is often somewhat different from how it's structured internally.

- Moreover, most user interfaces have some state that does not belong in the application model.

- For example, if your user interface uses a drag and drop, something needs to keep track of things like where the item being dragged is right now, how its appearance should change as it moves over possible drop targets, and how those drop targets might also change as the item is dragged over them.

- This sort of state can get surprisingly complex, and needs to be thoroughly tested.

- In practice, you normally want some other class sitting between the user interface and the model. This has two important roles.

    - First, it adapts your application model for a particular user interface view.

    - Second, it's where any nontrivial interaction logic lives, and by that, I mean code required to get your user interface to behave in the way you want.

2. MVVM – Advantages

MVVM pattern is ultimately the modern structure of the MVC pattern, so the main goal is still the same to provide a clear separation between domain logic and presentation layer. Here are some of the advantages and disadvantages of MVVM pattern.

The key benefit is allowing true separation between the View and Model beyond achieving separation and the efficiency that you gain from having that. What that means in real terms is that when your model needs to change, it can be changed easily without the view needing to and vice-versa.

There are three important key things that flow out of applying MVVM which are as follows.

Maintainability

- A clean separation of different kinds of code should make it easier to go into one or several of those more granular and focused parts and make changes without worrying.
- That means you can remain agile and keep moving out to new releases quickly.

Testability

- With MVVM each piece of code is more granular and if it is implemented right your external and internal dependences are in separate pieces of code from the parts with the core logic that you would like to test.
- That makes it a lot easier to write unit tests against a core logic.
- Make sure it works right when written and keeps working even when things change in maintenance.

Extensibility

- It sometimes overlaps with maintainability, because of the clean separation boundaries and more granular pieces of code.
- You have a better chance of making any of those parts more reusable.
- It has also the ability to replace or add new pieces of code that do similar things into the right places in the architecture.

The obvious purpose of MVVM pattern is abstraction of the View which reduces the amount of business logic in code-behind. However, following are some other solid advantages:

- The ViewModel is easier to unit test than code-behind or event-driven code.
- You can test it without awkward UI automation and interaction.
- The presentation layer and the logic is loosely coupled.

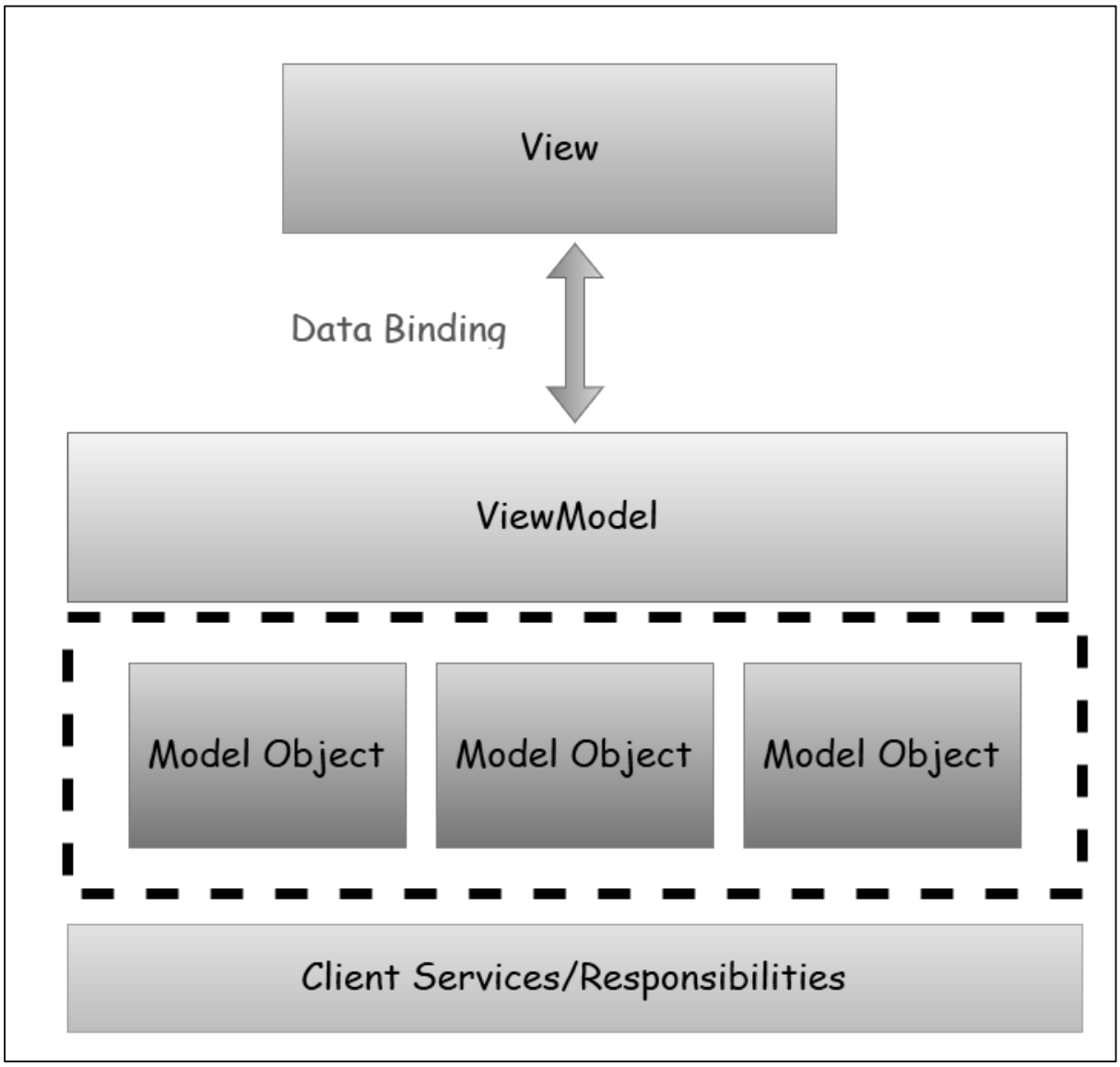
Disadvantages

- Some people think that for simple UIs, MVVM can be overkill.
- Similarly in bigger cases, it can be hard to design the ViewModel.
- Debugging would be bit difficult when we have complex data bindings.

3. MVVM – Responsibilities

MVVM pattern consists of three parts: Model, View, and ViewModel. Most of the developers at the start are little confused as to what a Model, View and ViewModel should or shouldn't contain and what are the responsibilities of each part.

In this chapter we will learn the responsibilities of each part of the MVVM pattern so that you can clearly understand what kind of code goes where. MVVM is really a layered architecture for the client side as shown in the following figure.



- The presentation layer is composed of the views.
- The logical layer are the view models.
- The presentation layer is the combination of the model objects.
- The client services that produce and persist them either directed access in a two-tier application or via service calls in and then to your application.
- The client services are not officially part of the MVVM pattern but it is often used with MVVM to achieve further separations and avoid duplicate code.

Model Responsibilities

In general, model is the simplest one to understand. It is the client side data model that supports the views in the application.

- It is composed of objects with properties and some variables to contain data in memory.
- Some of those properties may reference other model objects and create the object graph which as a whole is the model objects.
- Model objects should raise property change notifications which in WPF means data binding.
- The last responsibility is validation which is optional, but you can embed the validation information on the model objects by using the WPF data binding validation features via interfaces like INotifyDataErrorInfo/IDataErrorInfo

View Responsibilities

The main purpose and responsibilities of views is to define the structure of what the user sees on the screen. The structure can contain static and dynamic parts.

- Static parts are the XAML hierarchy that defines the controls and layout of controls that a view is composed of.
- Dynamic part is like animations or state changes that are defined as part of the View.
- The primary goal of MVVM is that there should be no code behind in the view.
- It's impossible that there is no code behind in view. In view you at least need the constructor and a call to initialize component.

- The idea is that the event handling, action and data manipulation logic code shouldn't be in the code behind in View.

- There are also other kinds of code that have to go in the code behind any code that's required to have a reference to UI element is inherently view code.

ViewModel Responsibilities

- ViewModel is the main point of MVVM application. The primary responsibility of the ViewModel is to provide data to the view, so that view can put that data on the screen.

- It also allows the user to interact with data and change the data.

- The other key responsibility of a ViewModel is to encapsulate the interaction logic for a view, but it does not mean that all of the logic of the application should go into ViewModel.

- It should be able to handle the appropriate sequencing of calls to make the right thing happen based on user or any changes on the view.

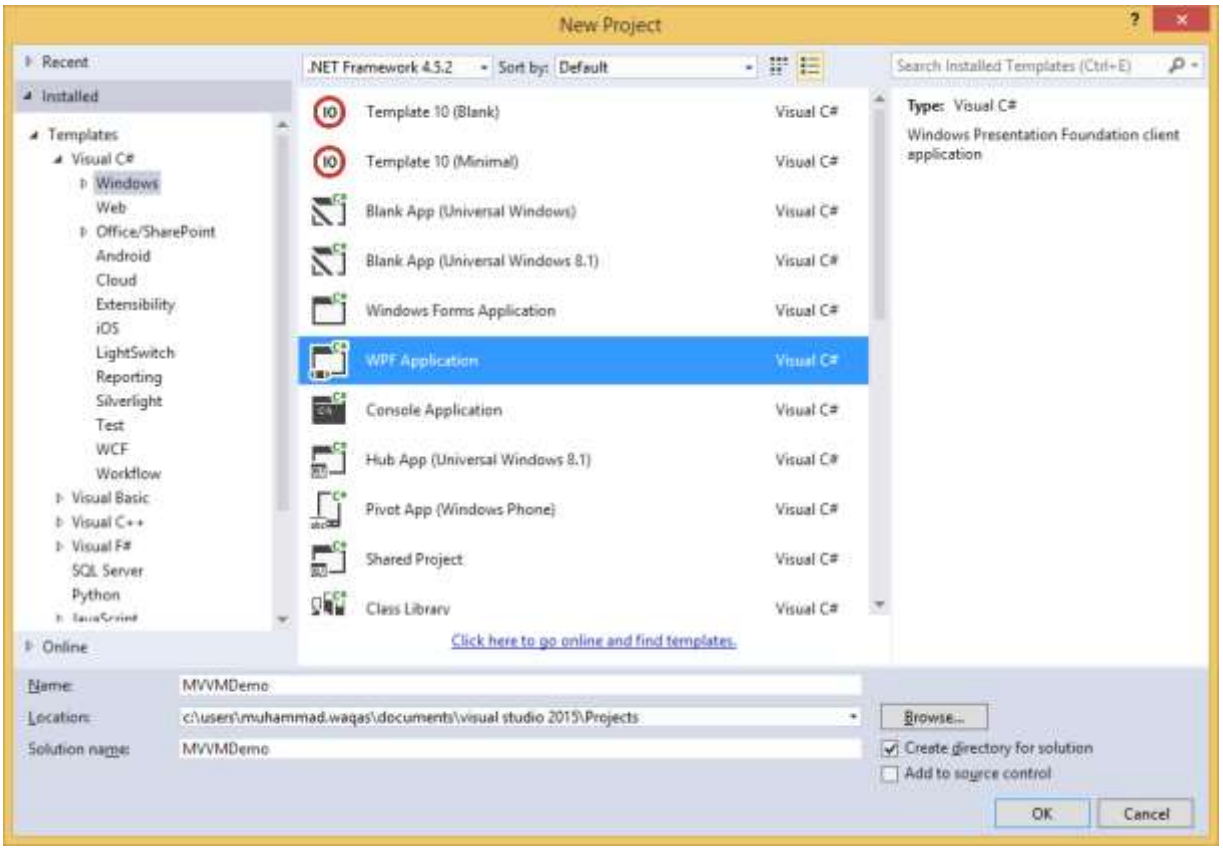
- ViewModel should also manage any navigation logic like deciding when it is time to navigate to a different view.

4. MVVM – First Application

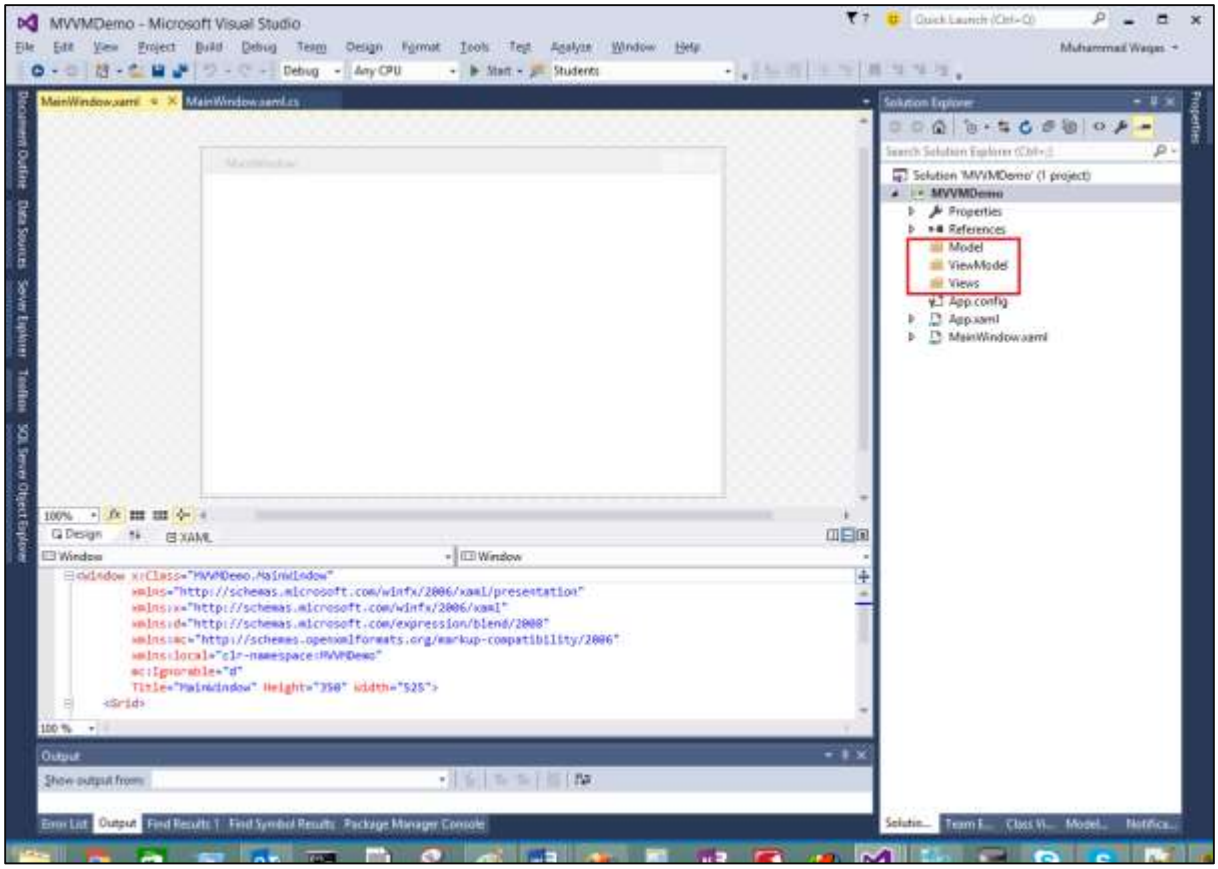
In this chapter, we will learn how to use MVVM patterns for simple input screen and the WPF application that you may already be used to.

Let's have a look at a simple example in which we will be using MVVM approach.

Step 1: Create a new WPF Application project MVVMDemo.



Step 2: Add the three folders (Model, ViewModel, and Views) into your project.



Step 3: Add a StudentModel class in the Model folder and paste the below code in that class

```
using System.ComponentModel;

namespace MVVMDemo.Model
{
    public class StudentModel
    {

    }
    public class Student : INotifyPropertyChanged
```

```
{
    private string firstName;
    private string lastName;

    public string FirstName
    {
        get { return firstName; }
        set
        {
            if (firstName != value)
            {
                firstName = value;
                RaisePropertyChanged("FirstName");
                RaisePropertyChanged("FullName");
            }
        }
    }

    public string LastName
    {
        get { return lastName; }
        set
        {
            if (lastName != value)
            {
                lastName = value;
                RaisePropertyChanged("LastName");
                RaisePropertyChanged("FullName");
            }
        }
    }

    public string FullName
    {
```

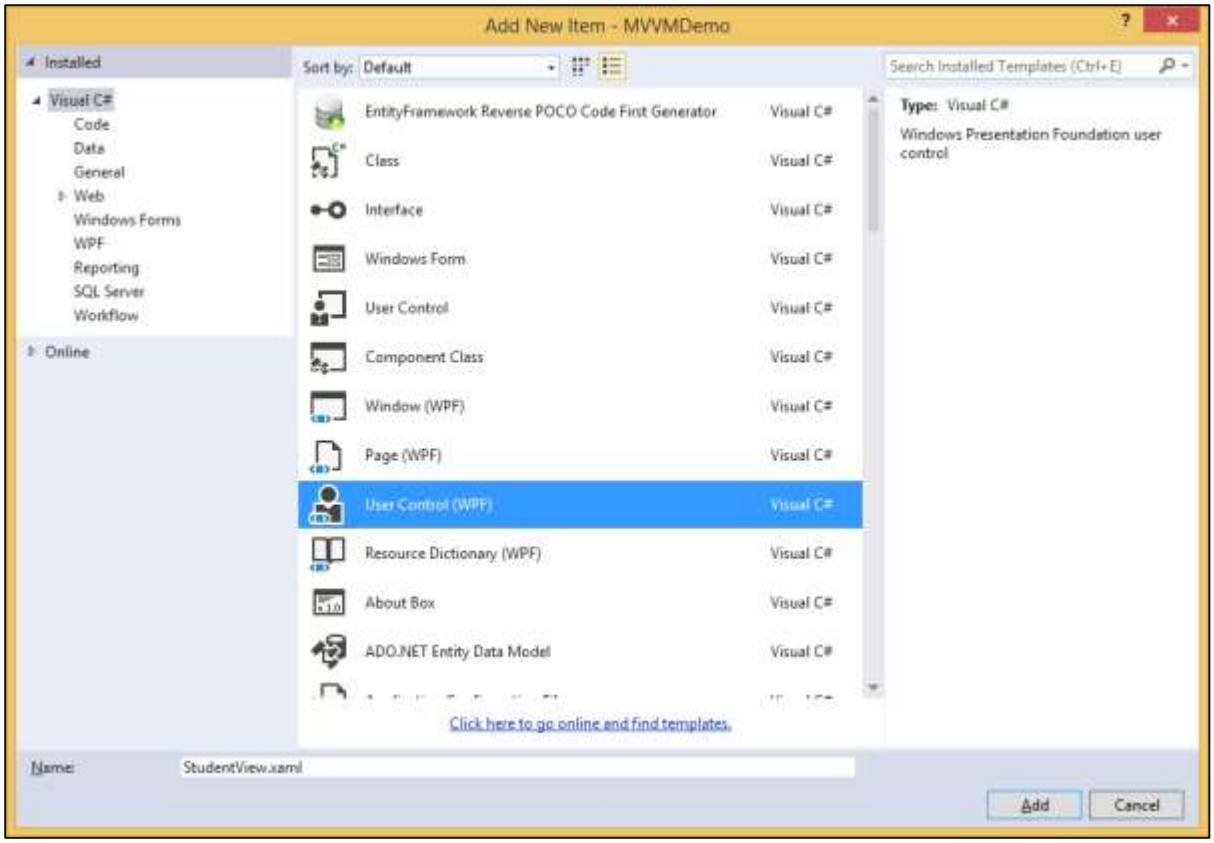
```
            get
            {
                return firstName + " " + lastName;
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void RaisePropertyChanged(string property)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(property));
            }
        }
    }
}
```

Step 4: Add another StudentViewModel class into ViewModel folder and paste the following code.

```
using MVVMDemo.Model;
using System.Collections.ObjectModel;

namespace MVVMDemo.ViewModel
{
    public class StudentViewModel
    {
        public ObservableCollection<Student> Students
        {
            get;
            set;
        }

        public void LoadStudents()
```

```
        {
            ObservableCollection<Student> students = new
ObservableCollection<Student>();

            students.Add(new Student { FirstName = "Mark", LastName = "Allain" });
            students.Add(new Student { FirstName = "Allen", LastName = "Brown" });
            students.Add(new Student { FirstName = "Linda", LastName = "Hamerski" });
            Students = students;
        }
    }
}
```

Step 5: Add a new User Control (WPF) by right click Views folder and Select Add > New Item…



Step 6: Click Add button. Now you will see the XAML file. Add the following code into StudentView.xaml file which contains different UI elements.

```
<UserControl x:Class="MVVMDemo.Views.StudentView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:MVVMDemo.Views"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <StackPanel HorizontalAlignment="Left">
            <ItemsControl ItemsSource="{Binding Path=Students}">
                <ItemsControl.ItemTemplate>
                    <DataTemplate>
```

```
                        <StackPanel Orientation="Horizontal">
                            <TextBox Text="{Binding Path=FirstName, Mode=TwoWay}"
Width="100" Margin="3 5 3 5"/>
                            <TextBox Text="{Binding Path=LastName, Mode=TwoWay}"
Width="100" Margin="0 5 3 5"/>
                            <TextBlock  Text="{Binding Path=FullName,
Mode=OneWay}" Margin="0 5 3 5"/>
                        </StackPanel>
                    </DataTemplate>
                </ItemsControl.ItemTemplate>
            </ItemsControl>
        </StackPanel>
    </Grid>
</UserControl>
```

Step 7: Now add the StudentView into your MainPage.xaml file using the following code.

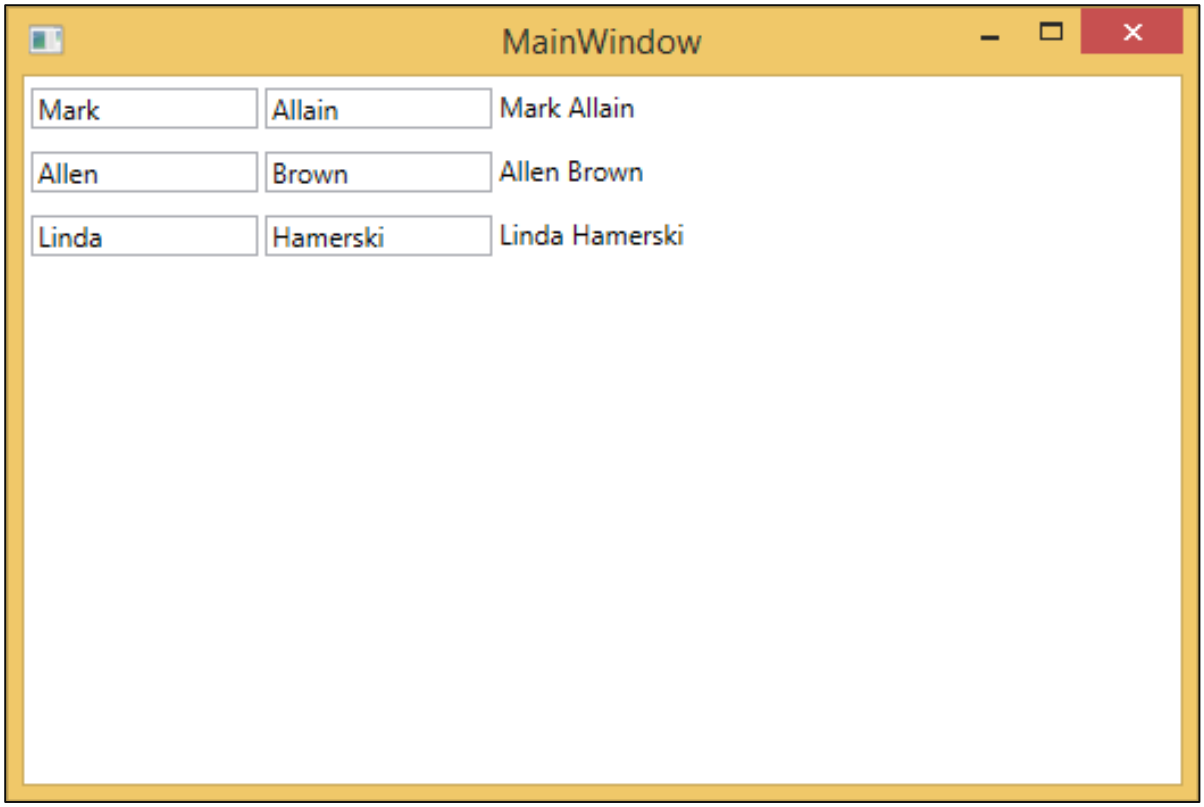
```
<Window x:Class="MVVMDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:MVVMDemo"
        xmlns:views="clr-namespace:MVVMDemo.Views"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <views:StudentView x:Name="StudentViewControl"
Loaded="StudentViewControl_Loaded"/>
    </Grid>
</Window>
```

Step 8: Here is the implementation for Loaded event in the MainPage.xaml.cs file, which will update the View from the ViewModel.

```
using System.Windows;
namespace MVVMDemo
{
```

```
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void StudentViewControl_Loaded(object sender, RoutedEventArgs e)
        {
            MVVMDemo.ViewModel.StudentViewModel studentViewModelObject = new
MVVMDemo.ViewModel.StudentViewModel();
            studentViewModelObject.LoadStudents();

            StudentViewControl.DataContext = studentViewModelObject;
        }
    }
}
```

Step 9: When the above code is compiled and executed, you will receive the following output on your main window.



We recommend you to execute the above example in a step-by-step manner for better understanding.