A study on Model-View-View Model (MVVM) Design Pattern

Varusai Mohamed¹, Abubucker Shamsudeen Shaffi²

¹Lecturer in Faculty of Computing Sciences, Gulf College, Muscat, Sultanate of Oman
²Acting Head of Faculty of Computing Sciences

Abstract - The software design needs a solution which transforms directly into code. The design pattern provides a template which is a description to solve a commonly occurring problem in software design that can be used in many different situations. Design pattern can speed up the software development process and some of them are reusable and it reduces the issues that cause major problems and it provides easiness to architects to implement and it improves code readability.

The people feel difficulty to apply certain software design techniques to certain problems. The developers interact using well-known design patterns by using their names. But the developers feel difficulty in the traditional model or design patterns to separate view from the model completely. This issue is addressed in MVVM (Model-View-ViewModel). The View of MVVM consists of behaviors, events and data binding information.

The view model of MVVM manipulates the model and it manages the state of a view which is synchronized with the view model. The MVVM design pattern is used in applications which supports for bi-directional data binding. It brings number of benefits to WPF and Silverlight development. It provides the flexibility for design work and development work to happen near-simultaneously. No need to refactor other logic in the code base if there is changes in the user interface. It insists thorough unit testing for your solutions.

Keyword : Model –View-View Model, MVVM Design Pattern, MVVM design technique, MVVM

MVVM is essentially a dressed-up version of the MVC architecture. It reduces the complexity of one’s view controllers and makes one’s presentation logic easier to test. It is compatible with existing MVC architecture. It makes application more testable. It works best with a binding mechanism.

I. DIFFERENCE BETWEEN TRADITIONAL DESIGN PATTERN AND MVVM

In traditional UI development, there is more binding between view layer and data binding logic. The view is created by the developer using a window or a user control or page and then the developer writes his code at the background of the page. Some time they are making code as a part of view definition also.

This approach increases the size of the View class and creates a very strong dependency between UI and data binding logic. Sometime more than one developer wants to work on a same view without disturbing other developer. The traditional approach does not support for this due to strong dependency.[1] It is the bad idea. The following diagram describes a very tight coupling between the View (UI), Model (Data displayed in UI) and Glue code (Event handling, binding, business logic).

MVC is a pattern and as such is a guideline and the implementations of the pattern vary. Variations of MVC exist like “Model-View-Presenter” (MVP) and “Model-View-ViewModel” (MVVM). In a very brief explanation:

MVC – The view sits at the top of the architecture, the controller sits below the view. The model sits below the controller. So the view knows about the controller, the controller knows the model. The view is notified when the model changes.

MVP – The controller is replaced with a presenter. The presenter sits at the same level as the view. The presenter listens to the events of both the view and the model and mediates the interactions between both the view and model.[2]

MVVM – The controller is replaced with a view model. The view model sits below the UI layer. The view model exposes the data and command objects that the view needs. You could think of this as a container object that view goes to get its data and actions from. The view model pulls its data from the model(s) below.

In MVVM, the Glue code is View Model. So it is easy to maintain and it provides simplicity in the application structure. If there are any changes in the ViewModel properties, the View also automatically changes. The notification and data binding sends values to the View. In MVVM, the View classes have no idea that the Model classes exist and the View never modifies it but he ViewModel modifies data. The ViewModel and Model are unaware of the View. And also the model does not have any idea about ViewModel and View exists.[3]

I want to discuss MVVM concept with an example of a windows application.
II. MVVM IN WPF APPLICATION

MVVM is the most preferred architectural pattern for WPF and Silver light projects. MVVM has 3 sections as the following:

All the software has an evaluation history. At first, we created two tier architecture. In two tier architecture, the business layers and data layers are connected together.

These are connected with UI layer as the following:

After that the business layer is separated from the data layer. It contains all the business logics as the following:

The following diagram represents the 3-tier architecture.

Each one of these layer has certain responsibility. The user interface layer handles presentation responsibility and business layer has validation responsibility and data access layer has SQL interaction responsibility. The three tier architecture has two many advantages.

One of the main advantage is contain changes reusability and the another advantage is increased reusability. We can connect the business logic with any type applications such as windows, mobile…etc.
We can divide each layer and each layer has its own responsibility and if we change in one layer it does not affect another layer.

It follows single responsibility principle. A class should only do one responsibility.

In MVVM, there is a glue code which connects UI layer with Middle Layer and Middle Layer with Data Access layer. This glue code works for mapping between two layers and some time it transforms the code to match with user interface. So the code behind is associated with the UI. This code has no resuability. The following example shows the mapping code and transformation code of the windows application.

```
textCustomerName.text = custobj.CustomerName; // mapping code
if (obj.Gender == "M") // transformation code
    {chkMale.IsChecked = true;}
else
    {chkMale.IsChecked = false;}
```

Some time in a project, lot of codes have no reusability. The MVVM avoids this problem. It separates the class files entirely with UI which can reusable with many UI layers of same application or with many application. [4]

The following example explains the importance of MVVM design pattern in the windows application.

When an application is friendly to develop, test, install, maintain, and use, then it can be labelled as a well-designed application. Creating such an application generally includes some form of separation and encapsulation of responsibilities. A typical approach includes separation of the UI (the views) from the business logic (the model). This is so interesting when we use the model-view design patterns:

There are three components with which the MVVM pattern plays a wonderful role in application development.

**View, ViewModel, and Model.** The communication between these three is described in the following image.

---

1. **Model** - This is the layer that represents the application's data.
2. **View Model** - This is an intermediary between the view and the model.
3. **View** - This represents the presentation or the user interface layer.
I. Model

The main thing to remember with the model is that it holds the information, but not behaviors or services that manipulate the data. It is not responsible for formatting UI controls and contents to look beautiful on the monitor, or pulling a data from a remote server.

namespace MyMVVM
{
    public class StudentModel : INotifyPropertyChanged
    {
        private string _firstName;
        public string FirstName
        {
            get { return _firstName; }
            set
            {
                _firstName = value;
                RaisePropertyChanged("FirstName");
                RaisePropertyChanged("FullName");
            }
        }

        private string _lastName;
        public string LastName
        {
            get { return _lastName; }
            set
            {
                _lastName = value;
                RaisePropertyChanged("LastName");
                RaisePropertyChanged("FullName");
            }
        }

        private string _rollNo;
        public string RollNo
        {
            get { return _rollNo; }
            set
            {
                _rollNo = value;
                RaisePropertyChanged("RollNo");
            }
        }

        protected void RaisePropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        public override bool Equals(object obj)
        {
            return obj is ContactModel && ((ContactModel) obj).FullName.Equals(FullName);
        }

        public override int GetHashCode()
        {
            return FullName.GetHashCode();
        }
    }
}
2. ViewModel

The MVVM is a refinement of the popular MVC design and the ViewModel in MVVM is used to facilitate Presentation Separation.

namespace MyMVVM
{
    public class StudentVM : VMBase
    {
        public StudentVM()
        {
            Students = new ObservableCollection<StudentModel>();
            myData = new myData();
            myData.GetStudent(_selectStudent);
            Delete = new DeleteCommand(myClass,
                () => CanDelete,
                Students =>
                {
                    selStudent = null;
                    Service.GetStudent(_selectStudent);
                });
        }

        private void _selectStudent(IEnumerable<StudentModel> students)
        {
            Students.Clear();
            foreach (var student in students)
            {
                Students.Add(student);
            }
        }
    }

    public IStudent myData { get; set; } 
    public bool CanDelete
    {
        get { return _selectStudent != null; } 
    }
}

public ObservableCollection<StudentModel> Students
{
    get; set; }

public DeleteCommand Delete { get; set; }

private StudentModel _selectStudent;

public StudentModel SelectStudent
{
    get { return _selectStudent; }
    set
    {
        _selectStudent = value;
        RaisePropertyChanged("SelectStudent");
        RaisePropertyChanged("CanDelete");
        Delete.RaiseCanExecuteChanged();
    }
}

Viewport describes the properties like state information (IsBusy) and commands. The "Two-way binding" allows the model and properties of the viewmodel for being updated by the view.

I feel that a basic MVVM framework only requires two things:

1. A class that is either a DependencyObject or implements INotifyPropertyChanged to fully support data-binding, and
2. Commands to invoke the actions.

Here I wrote the VMBase class to handle INotifyPropertyChanged and I inherited this class in ViewModel.

namespace MyMVVM
{
    public abstract class VMBase : INotifyPropertyChanged
    {
        protected void RaisePropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
public event PropertyChangedEventHandler PropertyChanged;

3. View

This represents the presentation or the user interface layer. Below is a simple XAML program without any special formatting steps to concentrate only the Student data binding.

```xml
<UserControl x:Class="MyMVVM.StudentView" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="grdMain" Background="White"
        DataContext="{Binding SelectStudent }">
        <TextBlock Text="Name:" />
        <TextBlock Text="{Binding FullName}" />
        <TextBlock Text="Roll Number:" />
        <TextBlock Text="{Binding RollNo}" />
    </Grid>
</UserControl>
```

III. CONCLUSION

MVVM provides better solution for code reusability. It separates the class files (DLL) from the presentation (UI) layer which can be able to connect with many other applications easily. It allows to create a view-specific subsets of the model. The translation is required between the ViewModel and the View. It is done very simply as copying objects in MVVM. The key thing to remember is that you have one and only one View-Model per view. The view model communicates with the model. The view does not need to know what the view model is. Another important point is that the view do not need any code behind of it and the data-binding deals the communication between view and view-model. So MVVM supports for parallel development of UI and its surroundings. Then the unit testing is easily applied over the view model rather than even-driven code. It can be able to test separately without interaction of UI. It is not good to apply View Model in the small application where bindings heavier than the objects. The MVVM is so powerful for Silver light and WPF applications.

REFERENCES


