Understanding Basics of UI Design Pattern MVC, MVP and MVVM

Avtar Sohi, 20 Jul 2011 CPOL

Rate this:

4.67 (73 votes)

An article on the basics of UI design pattern MVC, MVP and MVVM

Introduction

This is my first article and I hope you will like it. After reading this article, you will have a good understanding about "Why we need UI design pattern for our application?" and "What are basic differences between different UI patterns (MVC, MVP, MVVP)?".

In traditional UI development - developer used to create a `View` using window or usercontrol or page and then write all logical code (Event handling, initialization and data model, etc.) in code behind and hence they were basically making code as a part of view definition class itself. This approach increased the size of my `view`class and created a very strong dependency between my UI and data binding logic and business operations. In this situation, no two developers can work simultaneously on the same view and also one developer's changes might break the other code. So everything is in one place is always a bad idea for maintainability, extendibility and testability prospective. So if you look at the big picture, you can feel that all these problems exist because there is a very tight coupling between the following items.

1. View (UI)
2. Model (Data displayed in UI)
3. Glue code (Event handling, binding, business logic)

Definition of Glue code is different in each pattern. Although view and model is used with the same definition in all patterns.

In case of **MVC** it is controller. In case of **MVP** it is presenter. In case of **MVVM** it is view model.

If you look at the first two characters in all the above patterns, it remain same i.e. stands for model and view. All these patterns are different but have a common objective that is **“Separation of Duties"**

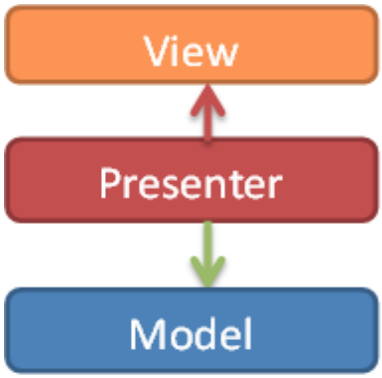
In order to understand the entire article, I request readers to first understand the above entity. A fair idea about these will help you to understand this article. If you ever worked on UI module, you can easily relate these entities with your application.

MVC (model view controller), **MVP** (model view presenter) and **MVVM** (model view view model) patterns allow us to develop applications with loss coupling and separation of concern which in turn improve testability, maintainability and extendibility with minimum effort.

MVVM pattern is a one of the best solutions to handle such problems for WPF and Silverlight application. During this article, I will compare MVC, MVP and MVVM at the definition level.

MVP & MVC

Before we dig into MVVM, let’s start with some history: There were already many popular design patterns available to make UI development easy and fast. For example, MVP (model view presenter) pattern is one of the very popular patterns among other design patterns available in the market. MVP is a variation of MVC pattern which is being used for so many decades. Simple definition of MVP is that it contains three components: Model, View and presenter. So view is nothing but a UI which displays on the screen for user, the data it displays is the model, and the Presenter hooks the two together (View and model).



The view relies on a Presenter to populate it with model data, react to user input, and provide input validation. For example, if user clicks on save button, corresponding handling is not in code behind, it's now in presenter. If you wanted to study it in detail, here is the MSDN link.

In the **MVC**, the Controller is responsible for determining which View is displayed in response to any action including when the application loads. This differs from MVP where actions route through the View to the Presenter. In MVC, every action in the View basically calls to a Controller along with an action. In web application, each action is a call to a URL and for each such call there is a controller available in the application who respond to such call. Once that Controller has completed its processing, it will return the correct View.

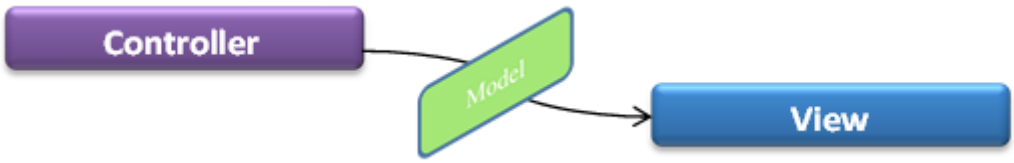
In case of MVP, view binds to the Model directly through data binding. In this case, it's the Presenter's job to pass off the Model to the View so that it can bind to it. The Presenter will also contain logic for gestures like pressing a button, navigation. It means while implementing this pattern, we have to write some code in code behind of view in order delegate (register) to the presenter. However, in case of MVC, a view does not directly bind to the Model. The view simply renders, and is completely stateless. In implementations of MVC, the View usually will not have any logic in the code behind. Since controller itself returns view while responding to URL action, there is no need to write any code in view code behind file.

MVC Steps

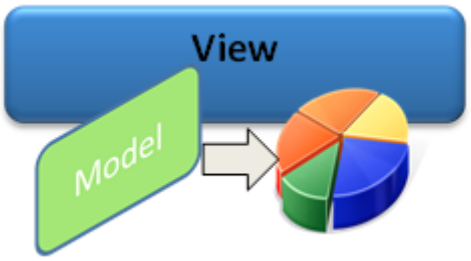
Step 1: Incoming request directed to **Controller.**

Step 2: **Controller** processes request and forms a data **Model.**

Step 3: **Model** is passed to **View.**



Step 4: **View** transforms **Model** into appropriate output format.



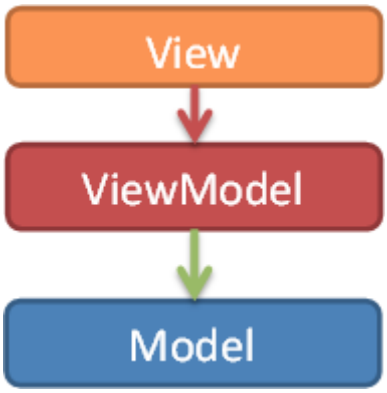
Step 5: Response is rendered.

So now you have basic understanding of MVC and MVP. Let's move to MVVM.

MVVM (Model View ViewModel)

The MVVM pattern includes three key parts:

1. `Model` (Business rule, data access, model classes)
2. `View` (User interface (XAML))
3. `ViewModel` (Agent or middle man between view and model)



`Model` and `View` work just like MVC and "`ViewModel`" is the model of the `View`.

- `ViewModel` acts as an interface between `model` and `View`.
- `ViewModel` provides data binding between `View` and `model` data.
- `ViewModel` handles all UI actions by using command.

In MVVM, `ViewModel` does not need a reference to a view. The view binds its control value to properties on a `ViewModel`, which, in turn, exposes data contained in model objects. In simple words, `TextBox` text property is bound with `name`property in `ViewModel`.

In `View`:

Hide Copy Code

```
<TextBlock Text="{Binding Name}"/>
```

In `ViewModel`:

Hide Copy Code

```
public string Name
        {
            get
            {
                return this.name;
            }
            set
            {
                this.name = value;
                this.OnPropertyChanged("Name");
            }
        }
```

`ViewModel` reference is set to a `DataContext` of `View` in order to set `view`data binding (glue between `view` and `ViewModel` model).

Code behind code of `View`:

Hide Copy Code

```
public IViewModel Model
        {
            get
            {
                return this.DataContext as IViewModel;
            }
            set
            {
                this.DataContext = value;
            }
        }
```

If property values in the `ViewModel` change, those new values automatically propagate to the view via data binding and via notification. When the user performs some action in the view for example clicking on save button, a command on the`ViewModel` executes to perform the requested action. In this process, it's the`ViewModel` which modifies `model` data, `View` never modifies it. The `view`classes have no idea that the `model` classes exist, while the `ViewModel` and model are unaware of the `view`. In fact, the `model` doesn't have any idea about`ViewModel` and `view` exists.