MVVM

Why use MVVM?

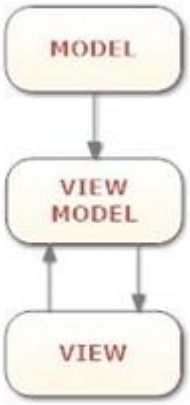
In traditional UI development - developer used to create a View using window or user control or page and then write all logical code (Event handling, initialization and data model, etc.) in the code behind and hence they were basically making code as a part of view definition class itself. There is a very tight coupling between the following items.

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

What is MVVM?

The MVVM pattern includes three key parts:

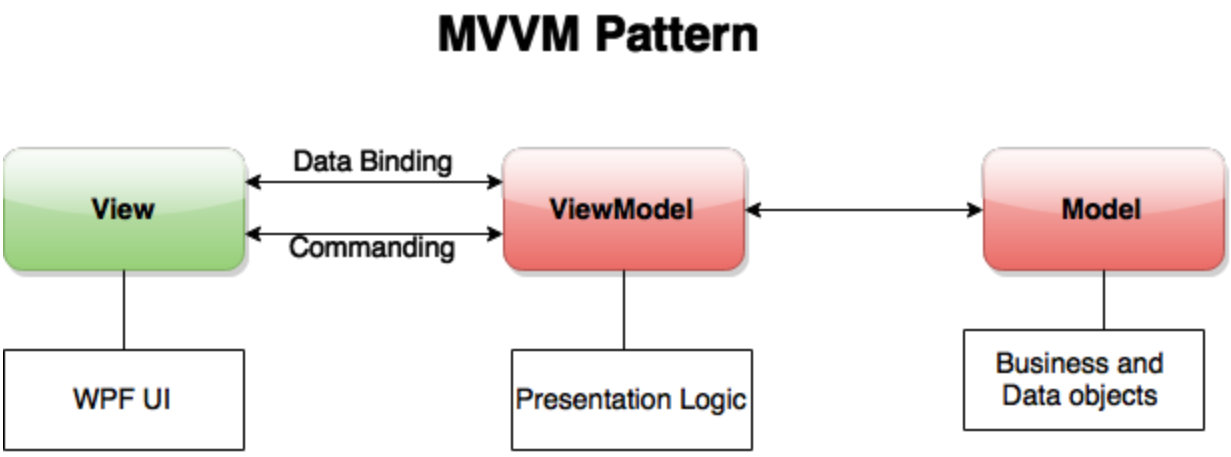
1. `Model` (Business rule, data access, model classes)
2. `View` (User interface (XAML))
3. `ViewModel` (Agent or middle man between view and model)



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

The `View` binds its control value to properties on a `ViewModel`, which, in turn, exposes data contained in `Model` objects.

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.

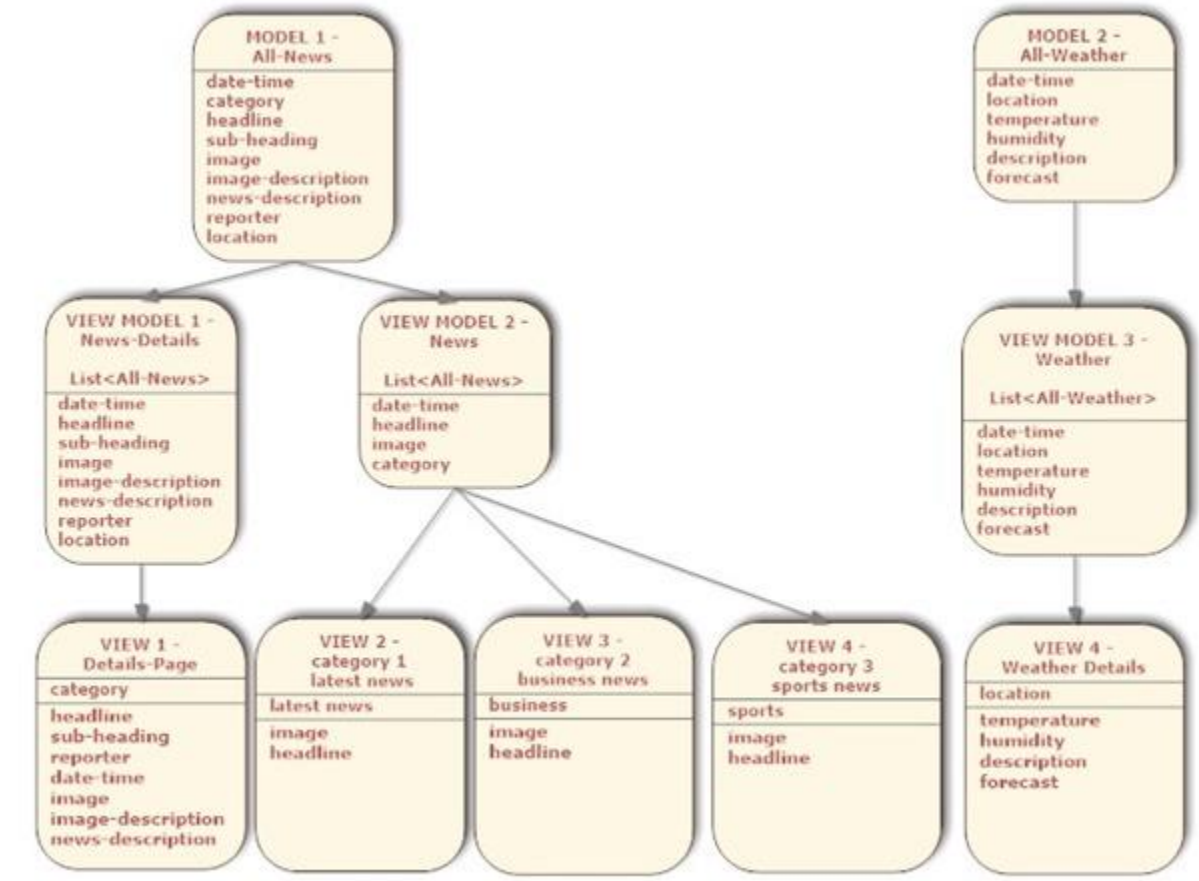


Advantages of MVVM

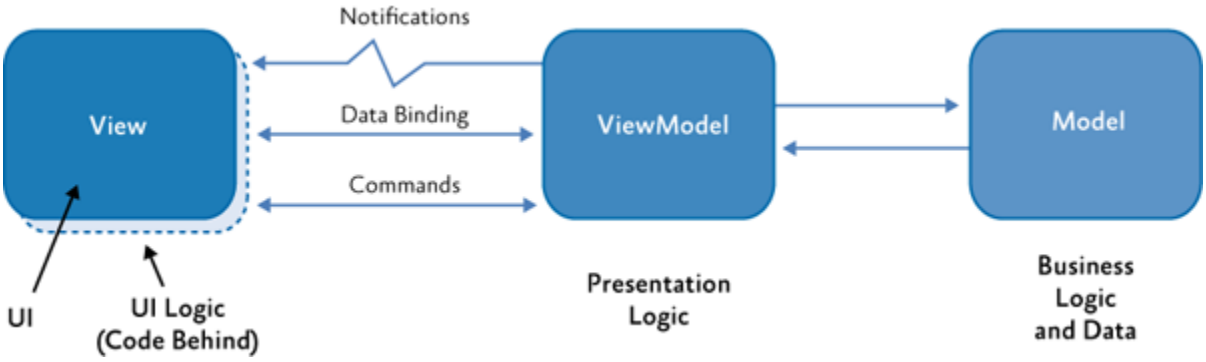
1. Lossley coupled architecture : MVVM makes your application architecture as loosley coupled. You can change one layer without affecting the other layers.
2. Extensible code : You can extends View, ViewModel and the Model layer separately without affecting the other layers.
3. Testable code : You can write unit test cases for both ViewModel and Model layer without referencing the View. This makes the unit test cases easy to write.

More about. http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx

A simple representation of MVVM design pattern for a news reader application



Let’s have a look at the MVVM architecture.



We use the standard conventions for naming the classes as follows:

- `View`s are suffixed with `View` after the name of the `View` (e.g.: `StudentListView`)
- `ViewModel`s are suffixed with `ViewModel` after the name of the `ViewModel`. (e.g.: `StudentListViewModel`)
- `Model`s are suffixed with `Model` after the name of the `Model` (e.g.: `StudentModel`).

How to implement the MVVM pattern: toolkits and frameworks

As I've mentioned at the beginning of the post, MVVM is a pattern, it isn't a library or a framework. However, as we've learned up to now, when you create an application based on this pattern you need to leverage a set of standard procedures: implementing the INotifyPropertyChanged interface, handling commands, etc.

Consequently, many developers have started to work on libraries that can help the developer's job, allowing them to focus on the development of the app itself, rather than on how to implement the pattern. Let's see which are the most popular libraries.

MVVM Light

MVVM Light (http://www.mvvmlight.net) is a library created by Laurent Bugnion, a long time MVP and one of the most popular developers in the Microsoft world. This library is very popular thanks to its flexibility and simplicity. MVVM Light, in fact, offers just the basic tools to implement the pattern, like:

- A base class, which the ViewModel can inherit from, to get quick access to some basic features like notifications.
- A base class to handle commands.
- A basic messaging system, to handle the communication between different classes (like two ViewModels).
- A basic system to handle dependency injection, which is an alternative way to initialize ViewModels and handle their dependencies. We'll learn more about this concept in another post.

Since MVVM Light is very basic, it can be leveraged not just by Universal Windows apps, but also in WPF, Sivlerlight and even Android and iOS thanks to its compatibility with Xamarin. Since it's extremely flexible, it's also easy to adapt it to your requirements and as a starting point for the customization you may want to create. This simplicity, however, is also the weakness of MVVM Light. As we're going to see in the next posts, when you create a Universal Windows app using the MVVM pattern you will face many challenges, since many basic concepts and features of the platform (like the navigation between different pages) can be handled only in a code behind class. From this point of view, MVVM Light doesn't help the developer that much: since it offers just the basic tools to implement the pattern, every thing else is up to the developer. For this reasons, you'll find on the web many additional libraries (like the Cimbalino Toolkit) which extend MVVM Light and add a set of services and features that are useful when it comes to develop a Universal Windows app.

Caliburn Micro

Caliburn Micro (http://caliburnmicro.com) is a framework originally created by Rob Eisenberg and now maintained by Nigel Sampson and Thomas Ibel. If MVVM Light is a toolkit, Caliburn Micro is a complete framework, which offers a completely differnent approach. Compared to MVVM Light, in fact, Caliburn Micro offers a rich set of services and features which are specific to solve some of the challenges provided by the Universal Windows Platform, like navigation, storage, contracts, etc.
Caliburn Micro handles most of the basic features of the pattern with naming conventions: the implementation of binding, commands and others concepts are hidden by a set of rules, based on the names that we need to assign to the various components of the project. For example, if we want to connect a ViewModel's property with a XAML control, we don't have to manually define a binding: we can simply give to the control the same name of the property and Caliburn Micro will apply the binding for us. This is made possible by a **boostrapper**, which is a special class that replaces the standard App class and takes care of intializing, other than the app itself, the Caliburn infrastructure.
Caliburn Micro is, without any doubt, very powerful, since you'll have immediate access to all the tools required to properly develop a Universl Windows app using the MVVM pattern. However, in my opinion, isn't the best choice if you're new to the MVVM pattern: since it hides most of the basic concepts which are at the core of the pattern, it can be complex for a new developer to understand what's going on and how the different pieces of the app are connecte together.

Prism

Prism (http://github.com/PrismLibrary/Prism) is another popular framework which, in the beginning, was created and maintaned by the Pattern & Practises division by Microsoft. Now, instead, it has become a community project, maintained by a group of independent developers and Microsoft MVPs.
Prism is a framework and uses a similar approach to the one provided by Caliburn Micro: it offers naming convention, to connect the different pieces of the app together, and it includes a rich set of services to solve the challenges provded by the Universal Windows Platform.

We can say that it sits in the middle between MVVM Light and Caliburn Micro, when it comes to complexity: it isn't simple and flexible like MVVM Light but, at the same time, it doesn't use naming convention in an aggressive way like Caliburn Micro does.