MVVM

Why use MVVM?

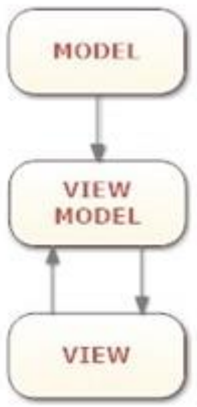
In traditional UI development - developer used to create a View using window or user control or page and then write all logical code (Event handling, initialization and data model, etc.) in the code behind and hence they were basically making code as a part of view definition class itself. There is a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

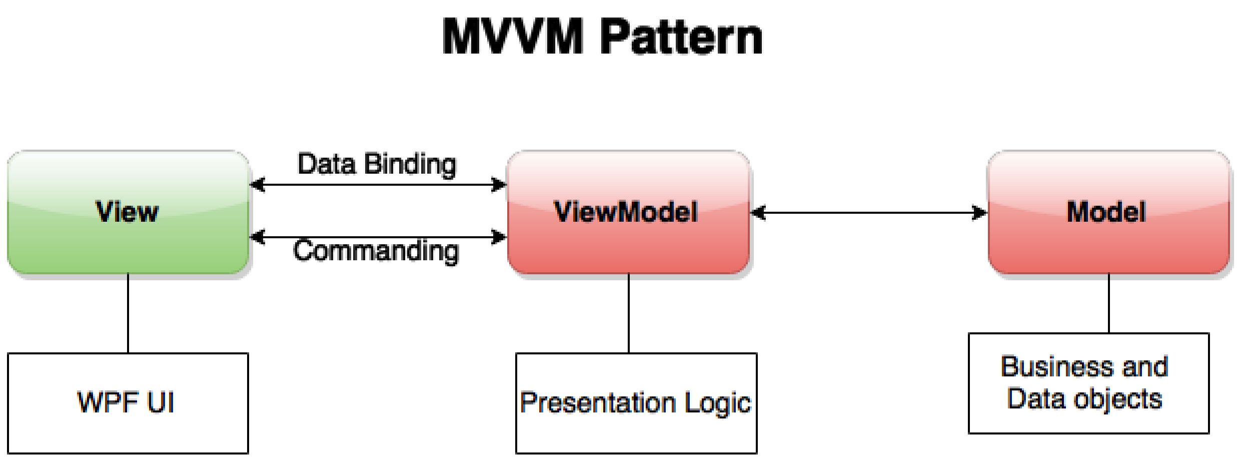
1. `Model` (Business rule, data access, model classes)
2. `View` (User interface (XAML))
3. `ViewModel` (Agent or middle man between view and model)



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.

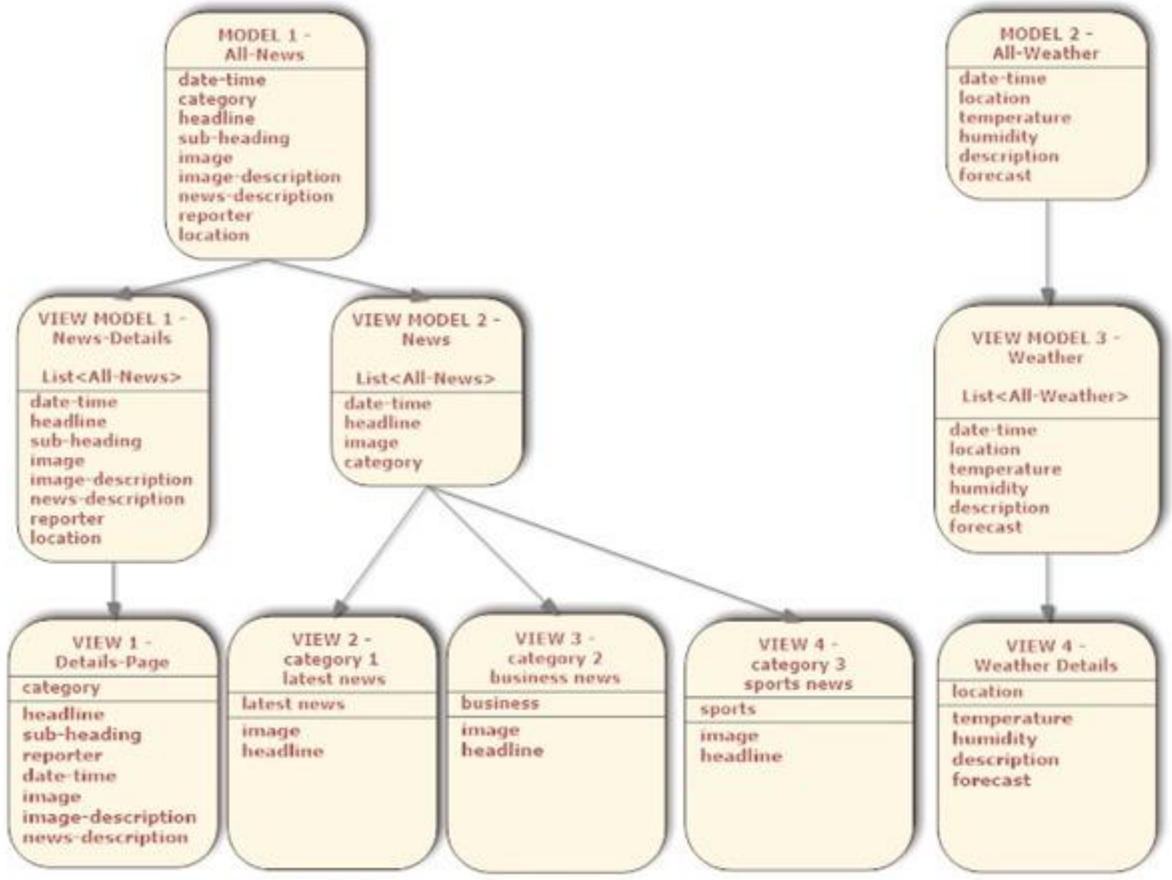


Advantages of MVVM

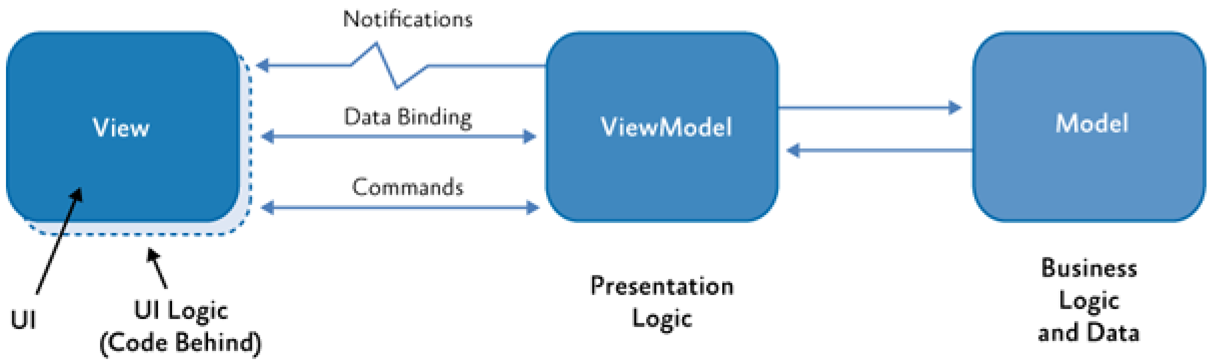
1. Lossley coupled architecture : MVVM makes your application architecture as loosley coupled. You can change one layer without affecting the other layers.
2. Extensible code : You can extends View, ViewModel and the Model layer separately without affecting the other layers.
3. Testable code : You can 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.



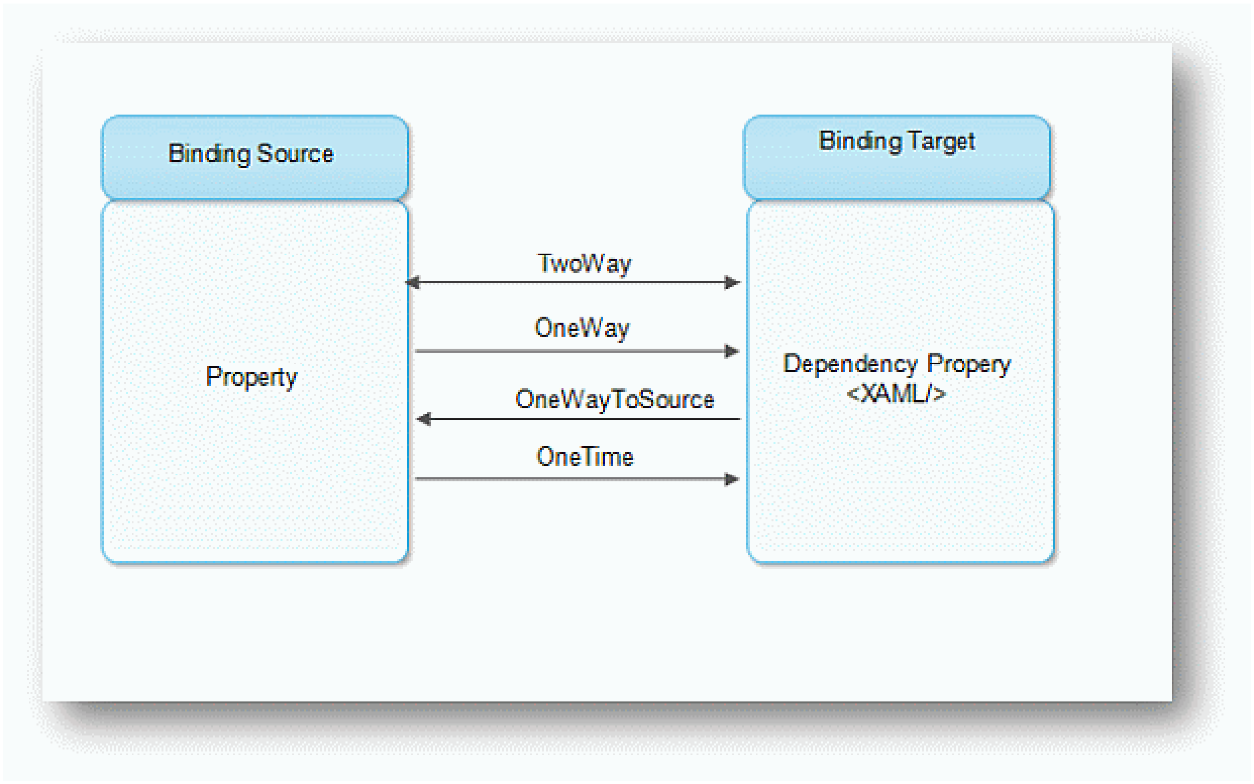
We use the standard conventions for naming the classes as follows:

- `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`).

Binding

Binding Modes in XAML(WPF,Silverlight,WP or Win8 App)

The Mode Property of Binding just changes the behaviors. The DataBinding mode defines the communication direction to the source or the direction of data flow from the source . In XAML (WPF, Silverlight, WP or Win8 App) there are five ways you can bind a data target object to a source.



The diagram above attempts to explain the databinding mode or communication way to communicate between the target to the source in XAML.

OneWay: Data moves only one direction, the source property automatically updates the target property but the source is not changed.

TwoWay: Data moves both directions, if you change it in the source or target it is automatically updated to the other.

OneWayToSource: Data moves from the target to the source changes to the target property to automatically update the source property but the target is not changed .

OneTime: Data is changed only one time and after that it is never set again, only the first time changes to the source property automatically update the target property but the source is not changed and subsequent changes do not affect the target property

View and ViewModels are connected thanks to binding. The ViewModel takes care of exposing the data to show in the View as properties, which will be connected to the controls that will display them using binding. Let's say, for example, that we have a page in the application that displays a list of products. The ViewModel will take care of retrieving this information (for example, from a local database) and store it into a specific property (like a collection of type **List<Order>**):

```
public List<Order> Orders { get; set; }
```

Let's say that your application has a page where it can create a new order and, consequently, it includes a TextBox control where to set the name of the product. This information needs to be handled by the ViewModel, since it will take care of interacting with the model and adding the order to the database. In this case, we apply to the binding the Modeattribute and set it to TwoWay, so that everytime the user adds some text to the TextBox control, the connected property in the ViewModel will get the inserted value.

If, in the XAML, we have the following code, for example:

<TextBox Text="{Binding Path=ProductName, Mode=TwoWay}" />

It means that in the ViewModel we will have a property called ProductName, which will hold the text inserted by the user in the box.

with the MVVM pattern we connect properties in the ViewModel with controls in the UI using binding, like in the following sample:

```
<ListView ItemsSource="{Binding Path=Orders}" />
```

The DataContext

DataContext is a property offered by any XAML Control. The **DataContext** property defines the binding context: every time we set a class as a control's DataContext, we are able to access to all its public properties. Moreover, the DataContext is hierarchical: properties can be accessed not only by the control itself, but also all the children controls will be able to access to them.
The core of the implementation of the MVVM pattern relies on this hieraracy: **the class that we create as ViewModel of a View is defined as DataContext of the entire page**. Consequently, every control we place in the XAML page will be able to access to the ViewModel's properties and show or handle the various information. In an

application developed with the MVVM pattern, usually, you end up to have a page declaration like the following one:

```
<Page x:Class="Sample.MainPage"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      DataContext="{Binding Source={StaticResource
MainViewModel}}"
      mc:Ignorable="d">

    <!-- page content goes here -->

</Page>
```

The **DataContext** property of the **Page** class has been connected to a new instance of the **MainViewModel** class.

The INotifyPropertyChanged interface

Let's say that we have, in the ViewModel, a property which we use to display the product's name, like the following one:

```
public string ProductName { get; set; }
```

According to what we have just learned, we expect to have a **TextBlock** control in the page to display the value of this property, like in the following sample:
?

```
<TextBlock Text="{Binding Path=ProductName}" />
```

Now let's say that, during the execution of the app, the value of the **ProductName** property changes (for example, because a data loading operation is terminated). We will notice how, despite the fact that the ViewModel will properly hold the new value of the property, the **TextBlock** control will continue to show the old one. The reason is that **binding isn't enough to handle the connection between the View and the ViewModel. Binding has created a channel between the** ProductName **property and the** TextBlock**, but no one notified both sides of the channel that the value of the property has changed.** For this purpose, XAML offers the concept of **dependency properties**, which are special properties that can define a complex behavior and, under the hood, are able to send a notification to both sides of the binding channel every time its value changes. Most of the basic XAML controls use dependency properties (for example, the **Text** property of the **TextBlock** control is a dependency property).

Every time the **ProductName** property changes, both sides of the binding channel receive a notification, so that the **TextBlock** control can update its visual layout to display the new value.

For these scenarios, **XAML offers a specific interface called** INotifyPropertyChanged**,** which we can implement in our ViewModels. This way, if we need to notify the UI when we change the value of a property, we don't need to create a complex dependency property, but we just need to implement this interface and invoke the related method every time the value of the property changes.
Here is how a ViewModel that implements this interface looks like:

```
public class MainViewModel: INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  [NotifyPropertyChangedInvocator]

  protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
  {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

You can notice how the implementation of this interface allows us to call a method called **OnPropertyChanged()**, that we can invoke every time the value of a property changes. However, to reach this goal, we need to change the way how we define the properties inside our ViewModel. When it comes to simple properties, usually we define them using the short syntax:
?

```
public string ProductName { get; set; }
```

Hower, with this syntax we can't change what happens when the value of the property is written or read. As such, we need to go back to use the old approach, based on a private variable which holds the value of the property. This way, when the value is written, we are able to invoke the **OnPropertyChanged()** method and dispatch the notification. Here is how a property in a ViewModel looks like:

```
private string _productName;

public string ProductName
{
   get { return _productName; }
   set
   {
      _productName = value;
      OnPropertyChanged();
   }
}
```

Now the property will work as expected: when we change its value, the **TextBlock** control in binding with it will change his appearance to display it.

Commands (or how to handle events in MVVM)

Another critical scenario when it comes to develop an application is to handle the interactions with the user: he could press a button, choose an item in a list, etc. In XAML, these scenarios are handled using events which are exposed by various controls. For example, if you want to handle that the button has been pressed, we need to subscribe to the **Click** event, like in the following sample:
?

```
<Button Content="Click me" Click="OnButtonClicked" />
```

The event is managed by an **event handler**, which is a method that includes, among the various parameters, some information which are useful to understand the event context (for example, the control which triggered the event or which item of the list has been selected), like in the following sample:

```
private void OnButtonClicked(object sender, RoutedEventArgs e)
{
    //do something
}
```

The problem of this approach is that event handlers have a tight dependency with the View: they can be declared, in fact, only in the code behind class. When you create an application using the MVVM pattern, instead, all the data and logic is usually defined in the ViewModel, so we need to find a way to handle the user interaction there.

For this purpose, **the XAML has introduced** commands**, which is a way to express a user interaction with a property instead that with an event handler**. Since it's just a simple property, we can break the tight connection between the view and the event handler and we can define it also in an independent class, like a ViewModel.

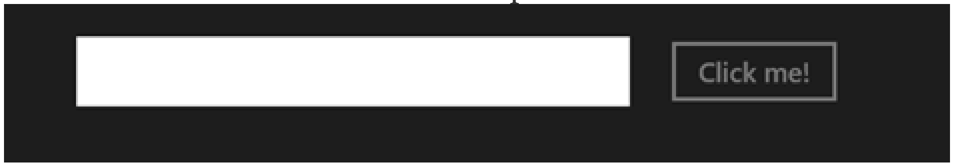
The framework offers the **ICommand** interface to implement commands: with the standard approach, you end up having a separated class for each command. The following example shows how a command looks like:
?

```
public class ClickCommand : ICommand
{
    public bool CanExecute(object parameter)
    {
    }

    public void Execute(object parameter)
    {
    }

    public event EventHandler CanExecuteChanged;
}
```

The core of the command is the **Execute()** method, which contains the code that is executed when the command is invoked (for example, because the user has pressed a button). It's the code that, in a traditional application, we would have written inside the event handler.
The **CanExecute()** method is one of the most interesting features provided by commands, since it can be used to handle the command's lifecycle when the app is running. For example, let's say that you have a page with a form to fill, with a button at the end of the page that the user has to press to send the form. Since all the fields are required, we want to disable the button until all the fields have been filled. If we handle the operation to send the form with a command, we are able to implement the **CanExecute()**method in a way that it will return **false** when there's at least one field still empty. This way, the **Button** control that we have linked to the command will automatically change his visual status: it will be disabled and the user will immediately understand that he won't be able to press it.



In the end, the command offers an event called **CanExecuteChanged**, which we can invoke inside the ViewModel every time the condition we want to monitor to handle the status of the command changes. For example, in the previous sample, we would call the **CanExecuteChanged** event every time the user fills one of the fields of the form.
Once we have define a command, we can link it to the XAML thanks to the **Command**property, which is exposed by every control that are able to handle the interaction with the user (like **Button, RadioButton,** etc.)
?

```
<Button Content="Click me" Command="{Binding Path=ClickCommand}" />
```

As we're going to see in the next post, however, most of the toolkits and frameworks to implement the MVVM pattern offers an easier way to define a command, without forcing the developer to create a new class for each command of the application. For example, the popular MVVM Light toolkit offers a class called **RelayCommand**, which can be used to define a command in the following way:

```
private RelayCommand _sayHello;
public RelayCommand SayHello
{
    get
    {
        if (_sayHello == null)
        {
            _sayHello = new RelayCommand(() =>
            {
                Message = string.Format("Hello {0}", Name);
            }, () => !string.IsNullOrEmpty(Name));
        }

        return _sayHello;
    }
}
```

As you can see, we don't need to define a new class for each command but, by using anonymous methods, we can simply create a new **RelayCommand** object and pass, as parameters:

1. The code that we want to excecute when the command is invoked.
2. The code that evaluates if the command is enabled or not.

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.