Diary of a Windows developer
Windows, mobile and much more

The MVVM pattern – Introduction

Posted on December 28, 2015 by qmatteoq

Model-View-ViewModel (from now on, just MVVM) is a "love and hate" topic when it comes to Universal Windows app development. If you have never used it and you try it for the first time, you'll probably find yourself a little bit confused, since it's a completely different approach than the standard one based on the code behind. On the other side, if you're a long time MVVM user, probably you won't be able anymore to create a new project using any other approach. This is the reason why I've decided to write the following series of posts: what's MVVM? Why is it so widely adpoted when it comes to Universal Windows apps development and, generally speaking, by any XAML based technology? I hope that, at the end of the journey, you'll find an answer to all your questions and you'll be able to start using the MVVM pattern in your apps without being scared anymore

The MVVM pattern

The first thing you have to understand is that MVVM isn't a framework or a library, but a **pattern**: it isn't a set of APIs or methods, but a way to define the architecture of an application. Probably you've already heard about MVVM LIght or Caliburn Micro, but you don't have to confuse them with MVVM: they're tools that helps developers to adopt the MVVM pattern, they don't represent the pattern itself.

The purpose of a pattern is to help developers to define the architecture of an application. Why is so important to do it? Why we cant't simply continue to develop an application in the way we are used to, which is writing all the code in the code-behind class? The standard approach is very quick and simple to understand but it has many limitations when it comes to more complex projects, that needs to be maintained over time. The reason is that the code-behind class has a very tight dependency with the XAML page. Consequently, most of the code can't be isolated and we end up to mix business logic and the presentation layer.

In the long run, the code behind approach introduces many problems:

1. **It's more complicated to maintain the code and evolve the project**. Every time we need to add a new feature or solve a bug, it's hard to understand where precisely we need to it, since there isn't a clear distinction between the various components of the app. This becomes even more true if we need to resume working on a project which has been "on hold" for a long time.
2. **It's complex to perform unit testing.** When it comes to complex projects, many developers and companies are adopting the unit test approach, which is a way to perform automatic tests that validate small pieces of code. This way it becomes easier to evolve the project: every time we add a new feature or we change some existing code, we can easily verify if the work we've done has broken the already existing features of the app. However, having a tight dependency between the logic and the user interface makes nearly impossible to write unit tests, since the code isn't isolated.
3. **It's complex to design the user interface:** since there's a tight relationship between the user interface and the business logic, it isn't possible for a designer to focus on the user interface without knowing all the implementation details behind it. Questions like "where the data is coming from? A database? A cloud service?" shouldn't be asked by a designer.

The goal of the MVVM pattern is to "break" this strong connection between the code behind and the user interface, making easier for a developer to understand which are the different components of the application.

More precisely, it's fundamental to distinguish the components which take care of the business logic and the ones that handle the data presentation.

The name of the pattern comes from the fact that the project is split into three different components, which now we're going to explore in details.

The model

The model is the component of the application that defines and handles all the basic entities of the application. The goal of this layer is to remove any dependency from the way the data is represented. Ideally, you should be able to take the classes that belong to this component and use them in another application without applying any change. For example, if you're working on an application to handle orders and customers of a company, the model could be defined by all the classes which define the base entities, like a customer, an order, a product, etc.

The view

The view is at the opposite side of the model and it's represented by the user inteface. In the Universal Windows apps world, views are made by the XAML pages, which contain all the controls and animations that define the visual layout of the application. Recyicling the already mentioned sample of an app to handle orders and customers, we can have multiple views to display the list of customers, the available products in the warehouse, the orders made by a customer, etc.

The ViewModel

The ViewModel is the connection point between the view and model: it takes care of retrieving the raw data from the model and to manipulate it so that it can be properly displayed by the view. The huge difference with a code behind class is that the ViewModel is just a plain simple class, without any dependency from the View. In an application based on the MVVM pattern, typically you create a ViewModel for every View.

Why the MVVM pattern?

After this brief introduction, it should be easier to understand why the MVVM pattern is so important and how, by adopting it, we can solve all the problems mentioned at the beginning of the post.

1. By splitting the code in three different layers it becomes easier, especially if you're working in a team, to maintain and evolve the application. If you need to add a feature or to solve a bug, it's easier to identify which layer has to be manipulated. Moreover, since there is no dependency between each layer, the work can be also done in parallel (for example, a designer can start working on the user interface while another developer can create the services which will be used by the page to retrieve the data).
2. To properly perform unit testing, the code to test has to be as simple and isolated as possible. When you work with the code-behind approach, this is simply not possible: often the logic is connected to an event handler (for example, because the code has to be executed when you press a button) and you would need to find a way to simulate the event in order to trigger the code to test. By adopting the MVVM pattern we break this tight dependency: the code included in a ViewModel can be easily isolated and tested.
3. Since we have broken the tight connection between the user interface and the business logic, for a designer it's easy to define the interface without having to know all the implementation details of the application. For example, if the designer has to work on a new page which displays a list of orders, we can easily swap the real ViewModel (which retrieves the data from a real data source, like a cloud service or a database) with a fake one, which can generate fake data that allows to the designer to easily understand which kind of information the page should display.

Why in the Universal Windows app world most of the developers tend to use the MVVM pattern and not other popular patterns like MVC or MVP? Mainly, because the MVVM pattern is based on many features which are at the core of the XAML runtime, like binding, dependency properties, etc. In this series of post we're going to talk a bit more about these features. You can notice how I've just mentioned XAML runtime and not the Universal Windows Platform: the reason is that most of the things we're going to see in these posts aren't specific of the Universal Windows app world, but they can be applied to any XAML based technology, like WPF, Silverlight, Windows Phone, Xamarin, etc.

Let's see now, in details, which are the basic XAML features leveraged by the MVVM pattern.

The binding

Binding is one of the most important XAML features and allows to create a communication channel between two different properties. They can be properties that belong to different XAML controls, or a property declared in code with a control's property. The key feature leveraged by the MVVM pattern is the second one: 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 to store it into a specific property (like a collection of type **List<Order>**):

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

To display the collection in a traditional code behind app, at some point, you would manually assign this property to the **ItemsSource** property of a control like **ListView** or **GridView**, like in the following sample:

```
1 MyList.ItemsSource = Orders;
```

However, this code creates a tight connection between the logic and the UI: since we're accessing to the **ItemsSource** property using the name of the control, we can perform this operation only in the code behind class.

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

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

This way, we have broken the dependency between the user interface and the logic, since the **Orders** property can be defined also in a plain simple class like a ViewModel.

As already mentioned, binding can be also bidirectional: this approach is used when not just the ViewModel needs to display some data in the View, but also the View should be able to change the value of one of the ViewModel's properties. Let's say that your application has a page where he 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 **Mode** attribute and we 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:

```
1 <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.

The DataContext

In the previous section we've seen how, thanks to the binding, we are able to connect the ViewModel's properties to the controls in the XAML page. You may be wondering how the View model is able to understand which is the ViewModel that populates its data. To understand it, we need to introduce the **DataContext**'s concept, which 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 hierarachy: **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

If we try to create a simple application based on the MVVM pattern applying the concepts we've learned so far, we would quickly hit a big issue. Let's use the previous sample of the page to add a new order and 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 excecution of the app, the value of the **ProductName** proeprty 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). However, defining a new dependency property isn't very straightforward and in most of the cases, it

offers features which aren't needed for our MVVM scenario. Let's take the previous sample based on the **ProductName** property: we don't need to handle any special behavior or logic, we just need that, 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 propertyNa
    {
        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.

We're going to learn more about this approach in the next post.

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.

Coming soon

In the next posts we're going to turn what we've learned so far int oa realy project and we're going to leverage MVVM Light for this purpose: the reason is that, as I've already mentioned, I think MVVM Light is the easiest one to understand, especially if you're new to the pattern, since it will help us to learn all the basic concepts which are at the core of the pattern. If you want to start looking at a real project, you'll find many samples (which we're going to explain in a more detailed way) on my GitHub repository at https://github.com/qmatteoq/UWP-MVVMSamples. Stay tuned!

P.S.= this post has been written with OpenLiveWriter, the new open source version of Windows Live Writer, which has now become a community driven project. In my opinion (and not just mine) Open Live Writer is the best tool in the world to write blog posts, so thanks to Microsoft for making this happen and thanks to all the great developers that are contributing to the project and keeping it alive!

Introduction to MVVM – The series

1. Introduction
2. The practice
3. Dependency Injection
4. Advanced scenarios
5. Services, helpers and templates
6. Design time data

Related

First steps with Caliburn Micro in Windows Phone 8 – The theory
February 4, 2013
In "Windows Phone"

Xamarin Forms for Windows Phone devs – Using the MVVM pattern
January 20, 2015
In "wpdev"

Prism and Universal Windows app – The basic concepts
December 12, 2014
In "Universal Apps"

This entry was posted in Universal Apps, UWP, wpdev and tagged Universal Windows Platform, Windows 10. Bookmark the permalink.

One Response to *The MVVM pattern – Introduction*

jake.chen *says:*
April 5, 2016 at 11:17 am

Hi,qmatteoq ,thanks for your great work .I have read your blog since 2014~~

Reply