**MVVM Pattern in XAML**

Right let’s get started.

My name is Mark Pearl; I am a developer coach for Driven Software. I often get asked what exactly a developer coach is. I get to work with developers from a number of organizations with the focus to help them improve practices that will make them develop better software.

Many of the core underlying concepts of the MVVM pattern extend much further than as just a UI design pattern. Many of these concepts are fundamental to good object oriented programming.

So, a quick chat about whom this session is targeted towards.

This session is predominantly going to cover content relevant to people writing XAML based applications. At a conceptual level the pattern extends further than just XAML – but we are going to stay in the XAML realm.

The second thing is this is a level 200 session. Level 200 is a tricky place to be. What it means is that for some of you this session is going to be way to basic – and for those of you that fall into this category I ask you to be nice in your reviews – I did warn you upfront. For others there are going to be sections where I move quickly and don’t explain everything – all of the material I will cover in this session is available on my github account so don’t despair.

So let’s look at the agenda. Originally I wanted to cover 6 sections in this session. Unfortunately after going over the content for this session I don’t think I am going to be able to cover all of them, so I will probably make brief mention of Maintainability and MVVM Toolkits at the end, and will spend most of my time covering the first four sections.

**[Pause]**

**Why the MVVM pattern?**

So why the MVVM pattern?

There are really two overarching forces for the MVVM pattern. They are maintainability and having a good separation of concerns.

With maintainability I have seen so many that look like the left hand picture. Absolute spaghetti code. You get a feature request from a client and before you have even written a line of code you can already feel your stomach going into a spasm and your headache beginning.

In line with maintainability is keeping a good separation of concerns. We don’t want to write systems that are a mess. We want to keep our plumbing code away from our core domain. At the UI level where there is typically a lot of plumbing code this is extremely challenging – without UI design patterns like the MVVM pattern we would be in a lot of trouble very quickly.

So when you apply the MVVM pattern what do you get? Well, it gets you an application that has an easily maintainable UI level with components that are easily interchangeable. For me this is a good UI design pattern. If you are able to swap things in and out easily you are in the pound seat.

Going along with the question why the MVVM pattern is the question who is this pattern for? The MVVM pattern is very definitely targeted towards business applications. Anything that has forms and textboxes is probably a good candidate. This is its area where it shines. That doesn’t mean it is a pattern you use everywhere - I would not for instance try and implement this pattern if I was writing a game or something along those lines.

**High Level Overview of the Pattern**

So with that let’s move on to a high level overview of the actual pattern.

Simply put we have three things, a model, a view and a view model – thus MVVM. So, in my opinion they messed up with the naming of this pattern - if you look at how these three components interact between each other it should really have been called the MVMV pattern – but MVMV pattern doesn’t have as nice a ring to it as MVVM so we ended up with MVVM instead.

So with that said, let’s drill into the roles and responsibility of each of these three components. Let’s start with the view.

**View**

The view is really only concerned with visual elements of your system. This means the view is concerned with Colours, Layout, Spacing, Animations, etc. The view doesn’t care about the rest of the system – it is just going to ask for things and expect someone to do it for it without caring who.

In our instance the view is going to be XAML based and one goal I like to have when writing views in XAML is to have little if any code behind. We will see how we can achieve this later.

**View Model**

Now moving on to the view model – think of the view model as the gofer – it is in the middle between the View and the Model and it really tries to keep the View happy by supplying it what it asks for from the Model.

From a code point the view model is a non-visual class; it should be unit testable and totally independent of the view.

A core responsibility of the view model is to implement change notification events when state changes occur in the system. The view model may also define additional properties to specifically support the view in representing data from the model.

**Model**

Finally we have the concept of the model. The model is also a non-visual class. Your model does not directly reference the view or the view model classes – in fact it should have no knowledge of them.

In the client server paradigm, if I were to describe a model I would generally be describing the client side domain of the system as the model that we are talking about.

**Portability**

So having briefly covered these three components there is just one thing I need to mention. If we look at the MVVM pattern from a perspective of reusability, the view is going to be very tightly coupled to the platform you are targeting and your view model and model should be extremely portable.

One common mistake I often see is specifically with view model implementations where they are not really portable. We are going to see an example of this later on and we will see what we can do to avoid this.

**XAML Core Concepts**

Ok, so that is a really high level over view of the MVVM pattern, let’s now look at some XAML core concepts that we need to understand to effectively apply this pattern.

There are four core concepts that I am going to cover today. As we go through these core concepts you are slowly going to see the MVVM pattern take place.

The first and most fundamental of them is data binding. Now, we can talk about data binding until the cows come home – but I find the best way to understand how data binding works is to look at some code.

To illustrate data binding I have made a sample project - in this case it is a WPF based application but the concept of binding is universal across all XAML based technologies – so what we will see will work in Silverlight, WinRT, etc.

In my WPF application I have a textbox, a list box and a button declared in XAML. I also have a few plain C# classes including a person class that I am going to use these classes to illustrate how binding works.

My goal is to get the data in my person class to bind to my textbox and the listbox to display information in stored in these classes.

As you can see the XAML part is very simple stuff. If we look go and at the person class we can see it’s got a name and a nickname property. One more thing I need to point out is that if I go to my app.xaml.cs file we can see I am creating my main window and it is here that I am showing the MainWindow.

So this is simple enough - if I run this application you can see my textbox and my listbox are not displaying anything right now. So to get these UI objects to bind to my person class I need to do the following.

First and most importantly I need to set the data context of my window to equal my person class. Now a quick not about how the DataContext works. All my UI controls inherit from the FrameworkElement class which is where the DataContext property is implemented. The DataContext is implemented in such a way that all descendant elements in the logical tree will share this DataContext. So in this instance if we go and look at the xaml. If we set the DataContext of the Window all its descendants including the Textbox and Listbox will have the same DataContext unless I explicitly set them otherwise.

So I have set the DataContext - the next thing I need to do is set the binding path for each control so that it knows what to display. In this case I am going to set the binding path of text property in my text box to the “Name” and the binding to my ItemsSource property of my Listbox to “Nicknames”. If I rerun the application, you will see binding has kicked in and the data from the class is being displayed on the Window.

**[Show Application Running]**

Now a quick but important side note on data binding - if I were to change the DataContext of my Window to get it to point to another object – So I have Frodo the dog and Kit the Car. And if I were to change the DataContext to Frodo – the binding handles this really elegantly. So rerunning my application we can see it has bound appropriately. If we look at the dog class it doesn’t share an interface or anything – so with binding, it binds to what it can, and what it can’t it ignores.

With that let’s handle the other part of data binding - we have got the information coming through once, but what happens if we change the data in the underlying object – our binding is not aware of this change.

To demonstrate this I am going to put a click event on my button that when fired will change the underlying data that my window has bound to. To demonstrate this we need to do a bit of refactoring.

**[Do Refactoring]**

If I rerun the application and I put a breakpoint on the button click, we can see it is setting the underlying data however if I look at my UI it hasn’t updated yet. To get the data binding to update I need to implement notifications.

To do this I go to my person class and implement the INotifyPropertyChanged interface . Now that we have implemented the interface we need to add a call to the OnNotifyPropertyChanged method in the setter of our properties we are binding to.

**[Implement Notification Interfaces]**

Before we rerun out application we need to make one more change – our Nicknames collection is currently a List of string, to get binding to detect changes in our collection we need to create an collection that implements the INotifyCollectionChanged interface. Out of the box we get an ObservableCollection which implements this interface, so if we change our collection to this and rerun our application we will see DataBinding is now picking up changes in the underlying objects.

**[Change to Observable Collection]**

**[Run program and show View Updating]**

So let’s move back to the slides and do a quick summary on DataBinding – the key points are

1. Set the DataContext of the View to the object you are binding to
2. Set the Binding Path of each UI element in XAML
3. Implement the INotifyPropertyChanged and INotifyCollectionChanged interfaces on your view models

**Commands**

And with that covered we are ready to move onto the second core concept. Right now we are able display and edit data in the view but we are not able to trigger actions or operations from the view to the view model.

So before we look at how to implement commands let’s look at what each component in the MVVM pattern can see. First we have a model – it is not aware of the View or ViewModel. We also have a ViewModel – it can see the Model, but is not aware of the View. And finally we have this View that can’t talk directly to the Model or the ViewModel. This gives us a loose coupling.

If we look at the previous example with the button click, currently the action performed behind the click is tightly coupled to the view – it’s in the code behind. We also have no way in the code behind to reference our ViewModel – nor do we want to. So, to remove this tight coupling from our view we use commands and to illustrate how commands work let’s move on to our second example project.

If we look at the second project you can see I have done a bit of refactoring. I have renamed my MainWindow to a MainView and have now made this class called the MainViewModel. I have also extracted the notification methods into a ViewModelBase class which my MainViewModel inherits from.

In my MainViewModel all I am exposing right now is the name and on my view I have a Textbox and a Button. If we run this project we can see if I click on the button we have a message box display a hello message from the code behind.

To remove this tight coupling the first thing I need to do is create a property in my ViewModel that implements the ICommand interface. Looking briefly at the interface we will see we have three things, the Execute, CanExecute methods and the CanExecuteChanged event. I then need to create a command class that implements this interface – in this case it is going to do a simple hello world message popup.

I now need to go to my ViewModel and create an instance of this class and point it to my DoSomething. Now using data binding I bind the command property on my button class to my DoSomething command in my ViewModel. If I now run the application and click on the button I see it is going to say hello.

But we haven’t really got us where we want us to be. Usually commands want to touch things in the ViewModel and right now my DoSomethingCommand can’t do this. So I am going to change this so my command can touch my ViewModel and to change it I am going to make use of dependency injection to inject my ViewModel into my command. So now my command can say hello and put the name of the ViewModel.

Now we don’t have to write these command classes every time – we can create something like a Delegate Command like this, and then in the constructor of our ViewModel we declare how we want the command to operate using a delegate expression.

So just to show you this working if I run the application and click the button we can see it is going to say “hello bob” and we can see now that it is doing this work in the ViewModel.

And with that we have had a brief look commands in XAML.

Just to recap with commands,

* We spoke about exposing the commands via properties in the ViewModel.
* We spoke about the ICommand interface and creating classes that implement the ICommand interface.
* We also spoke about the concept of delegate commands which makes declaring commands easier.

**Converters**

So now we have covered the first two core concepts – we get to move on to the third one which is converters.

Before we look at implementing a converter, let’s see where it fits in with the MVVM pattern.

If you look at the MVVM pattern and the roles of the Model and the ViewModel is to expose things in code – neither of them reference visual code. This is important to keep the classes portable.

Often in the ViewModel I see a common mistake made where a property exposes visibility or colours or something along those lines. The ViewModel should not do this. Keep in mind that the Model and ViewModel should not have any UI specific references - instead use converters.

To illustrate how converters work let’s move on to the next project – the structure is similar to the previous project but we now have two buttons – each button either increases or decreases a value we have exposed in our ViewModel which I have called rating.

So just to show this working – click up, click down and the value is changing.

**[Run Application]**

Now let’s say we wanted the background colour to change as the value changes – so for instance, if our rating is 3 it should display red, 2 should show orange and anything below 2 should be green - One approach would be to update this rating property which is an integer and expose some sort of brush property

**[Expose Brush Property]**

So if we do this we need to make a value change in our rating notify that our brush property is updated. If we run the application we can see it works.

**[Run Application]**

However if we do this we pollute our ViewModel and make it less portable. So I am going to delete this code and instead I am going to make a new converter class and use the converter to control the display.

Now, my converter is going to inherit form the MarkupExtension class and implements the IValueConveter interface.

The MarkupExtension abstract class is just going to make an instance of this converter easier. If we look at the IValueConverter interface we can see it exposes two things, the convert and the convertback method.

In this case we are only concerned with the convert method with our logic for the colours. In this case I am taking the value as a rating, checking it as an integer and returning the relevant colour.

So now if I go to my view I need to change the binding to make use of the converter – so if I pull that code snippet across, the syntax is very similar to the original binding except you can see now I am binding my background of the Textbox to the rating and I am setting a converter property on that binding that will point to the converter I have just written.

To see this is happening, let’s run the application and sure enough we can see straight away that the converter is firing and returning the appropriate colours.

**[Run Application & set appropriate breakpoint]**

So with that done we have manage to drive the view by colours, and I have managed to keep my view model clean so it is still portable.

Let’s just recap the info on converters, converters convert data from data to a visual representation. It is a nice way to keep your ViewModel UI agnostic.

So just a recap with converters. The main role of a converter is to convert data between the ViewModel and the Visual to a visual representation.

Converters implement the IValueConverter interface and an easy way to create them is to make them inherit from the MarkupExtension class.

**Behaviours**

Having covered converters there is one more core concept I want to cover – this is attached behaviours.

Now think of attached behaviours as the XAML equivalent to extension methods. Really what attached behaviours allow us to extend our UI controls to do things they were not originally designed to do.

So for instance – let’s say we have a project where we have a form and a button, similar to the one we used before. Our client says that they love the way everything is working in the application, but they want a small change made – they want it so that when they double click on the textbox it behaves the same way as if they clicked the button.

Well, the problem with this is the textbox does not have the command exposed on it that we can bind to the double click. So we need some way that I can trigger the DoSomething command from a textbox double click.

So how do I achieve this? I am going to create an attached behaviour to extend this functionality on my Textbox.

Now there is quite a bit of plumbing code to do this – but don’t despair, it really isn’t as hairy as it looks.

**[Paste Attached Behaviour]**

This is an attached behaviour, and when I run through it I have this command interface which I have made as a private static. I have tied my textbox double click event to fire off the command when the textbox is double clicked.

Now before we can use this attached behaviour we need to go to the view and apply this attached behaviour to my Textbox.

To do this I need to have a reference to the local namespace. Then I am going to say the attached property I want to bind to is the mapped command to the DoSomething command.

So here is my mapped command property – if I run this application I will see that my XAML will recognise this straight away and that the attached behaviour is being fired off. Now if I go and double click on this text box I can see sure enough it is firing off the DoSomethingCommand.

**[Run Program and Double Click on Textbox]**

So to recap - with attached behaviours we have managed to keep our ViewModel and our View clean yet extend the behaviour of an existing UI control.

I see attached behaviours as a powerful way to get your UI to work the way you want without having to go to code behind.

And with that we have covered the four core concepts in XAML that we need to understand to effectively implement the MVVM pattern – Binding, Commands, Converters & Attached Behaviours.

**Structuring Solutions**

Now that we have covered these core concepts let’s talk about structuring our solutions

The first thing I want to talk about is naming conventions. The convention in the MVVM pattern is to have a folder in your project called Views and a folder called ViewModels. Whenever you create a view you will normally have an associated ViewModel – Views typically end with the word “view” and then your view model will have the same name but end off with “view model”.

With that I want to touch briefly on what creates what.

There are two approaches of structuring your solution to implement the MVVM pattern. The first is the ViewModel first approach; this is the approach we have followed up to now.

If I go to the current project and look in the app.xaml.cs you can see I am making the ViewModel and then tying up the view. With the ViewModel first approach, the ViewModel creates the view and sets the binding. There are pros and cons to the ViewModel first approach.

* **ViewModel First** - Pros
  + Allows more complete testing of logic to open new Views and ViewModels
  + Tends to be DRYer as applications get larger
  + View and ViewModel are more independent and can be worked on separately more easily
* **ViewModel First** - Cons
  + More difficult to set up in Silverlight without DataTemplateSelector and typed DataTemplates.

If we move on to the second approach – I find this a little trickier; this is called the “View First Approach”. In the view first approach the view is responsible for creating an instance of the ViewModel.

How the view first approach generally works is we have this concept of a ViewModel locator – I have already created the ViewModel locator class. If I look here I can see that whenever I bind to a property called MainViewModel it returns an instance of the MainViewModel.

To get this to work we need to set up a static resource in the app.xaml.

Then we go to the view that we are want to bind and set the data context in XAML and bind it to my MainViewModel and set the path to the relevant property. So, having done this if we run the application you can see the view is binding to the view model.

**[Run Application]**

The main pro with the view first approach is that ..

* **View First** - Pros
  + Easy to track which ViewModel is used by a View
* **View First** - Cons
  + Doesn't allow a single View to be easily used with multiple ViewModels
  + Requires extra events to handle communication between Views and ViewModels

**MVVM Maintainability**

Now we are running out of time but I want to speak briefly on MVVM maintainability – mainly unit testability.

If you have applied the MVVM pattern correctly you should be able to generate unit test for your ViewModel and Model. I am a big fan of test driven development and writing unit tests, as far as I am concerned without unit tests you are writing code that will hurt you eventually.

If you cannot write unit tests for your Model and ViewModel you are doing something wrong and need to relook at how you are writing code.

**MVVM Toolkits**

Finally, there are 3 MVVM toolkits I want to mention. The first one is Prism developed by Microsoft patterns and practices. Prism goes beyond just the MVVM pattern – it is really a set of patterns and practices for implementing an enterprise level framework but it leverages the MVVM pattern extensively.

If you are writing WinRT applications look out for Kona, which is the equivalent to Prism but for client facing applications.

Another toolkit worth checking out is Caliburn Micro. Caliburn micro is a conventions based framework produced by Rob Eisenberg. It makes a whole bunch of inferences based on naming conventions. It’s a really cool toolkit and one worth investing time in.

And finally I need to make mention of the MVVM light toolkit with is implemented by Laurent Bugion. This is a popular toolkit that has implementations for Delegate Commands, Events Aggregators and more. I would really recommend looking into it also.

**Closing**

With that I believe we have come to the end of the session. Before we finish I want to make mention of the DeveloperUG – if you haven’t joined it yet, go to [www.DeveloperUG.org.za](http://www.DeveloperUG.org.za) and signup.

Also, before we go onto questions, please don’t forget to rate this session. Level 200 sessions are always tough to get right – so I would really appreciate any feedback that you feel a level 200 developer would need that I have missed out.

Now with that said I have time for a few questions…

**[Take Questions]**

Well, I am not going to take any more questions now – feel free to approach me after this session if you have any more questions.

Before we end just to recap,

We have spoken about why MVVM pattern – we then went through some core concepts including binding, commands, converters and attached behaviours.

I spoke briefly about structuring a solution and finally we touched on maintainability and toolkits. And with that we have come to the end of this session. Thank you for attending this session.