Chapter 2 – Brief Overview of WPF and MVVM

This chapter is intended to help someone who is fairly new to WPF and/or MVVM get an idea of the big picture. It might also be interesting for a seasoned veteran looking to get a different perspective on familiar topics. All subsequent chapters are for people who are already up-to-speed with WPF and MVVM. If you don't want or need to read this chapter, feel free to skip ahead to the next chapter now.

WPF

Microsoft's Windows Presentation Foundation (WPF) is a user interface programming platform that operates on top of the .NET Framework. It has a mile long feature list, so we won't cover everything it can do. Instead, let's focus on the things that are most important for understanding how BubbleBurst works.

The most salient feature of the platform to know about while reading this book is its data binding system. I have met quite a few developers who switched from Windows Forms to WPF just because of its incredible data binding capabilities. Data binding is an automated way to move data objects from one place to another. You can specify when, how, and why a binding will transfer values between its source and target. It's possible to bind a property on one UI element to a property on another UI element, or even to bind an element to itself. Additionally, value converters make it possible to bind two properties of different types. Bindings can be created in code or markup.

User interfaces in WPF are typically declared in a markup language known as XAML, which stands for eXtensible Application Markup Language. It is an XML-based format useful for declaring a graph of .NET objects, and configuring those objects with property values and event handling methods. XAML files are often associated with a code-behind file, which allows you to have code, such as event handling methods, which responds to events of the controls declared in XAML. A XAML file and its code-behind both contain partials of the same class, which means that you can access from code the objects declared in XAML, and vice versa.

Several controls, such as Button and MenuItem, allow you to bypass the code-behind altogether via their Command property. If you set a button's Command property to an object that implements the ICommand interface, when the user clicks the button it will automatically execute the command. As we will see soon, declaring a binding to a command in XAML can help simplify your application's design.

Arranging UI elements into various layouts is the bread and butter of user interface development. In WPF you can achieve an endless variety of layouts by leveraging its support for panels. The term 'panel' refers to an element container that knows where to place its child elements relative to each other, based on their size and the layout logic coded into the panel. Some panels will resize their child elements in order to achieve the desired layout. WPF includes the abstract Panel class, from which the common panels, such as Grid and StackPanel, derive. If you need to use coordinate-based positioning (i.e. use X and Y offsets), the Canvas panel allows you to gain that level of layout precision.

All but the simplest of demo applications must display a list of objects. WPF treats a list of objects as a first-class citizen by providing you with ItemsControl. Many common controls derive from ItemsControl, such as ListBox and ListView. Its ItemsSource property can be set to any collection of objects. ItemsControl generates UI elements that render those objects. You can tell the ItemsControl what UI elements to render each item with by setting the ItemTemplate property to a custom DataTemplate. In complex scenarios, you can use an ItemTemplateSelector to programmatically select a DataTemplate based on values of the data object. Another great feature of ItemsControl is that it allows you to specify the layout panel with which it arranges its child elements. You can use any layout panel, including a custom panel, as the ItemsPanel to achieve your desired layout strategy for the ItemsControl's child elements.

One other thing to know about about WPF that will help BubbleBurst make sense is that it uses a "retained" rendering system. Unlike Windows Forms, and other HWND-based UI platforms, in WPF you very rarely ever need to write code that paints the screen. As opposed to a "destructive" rendering system, WPF's retained rendering system caches vector drawing instructions and intelligently manages the job of handling things like region invalidations for you. You provide WPF with a description of what you want rendered, via high-level objects like Ellipse and TextBlock, and it works out what needs to be drawn to the screen when and where.

Learn More about WPF

If you would like to learn more about WPF before reading further, consider visiting these links:

Introduction to Windows Presentation Foundation

http://msdn.microsoft.com/en-us/library/aa970268.aspx

WPF Architecture

http://msdn.microsoft.com/en-us/library/ms750441.aspx

A Guided Tour of WPF

http://joshsmithonwpf.wordpress.com/a-guided-tour-of-wpf/

Customize Data Display with Data Binding and WPF

http://msdn.microsoft.com/en-us/magazine/cc700358.aspx

ItemsControl: 'I' is for Item Container via Dr. WPF

http://drwpf.com/blog/2008/03/25/itemscontrol-i-is-for-item-container/

MVVM

It was once commonly said that all roads lead to Rome. Today, all WPF and Silverlight best practices lead to the Model-View-ViewModel design pattern. MVVM has become a common way of discussing, designing, and implementing WPF and Silverlight programs. Like all other design patterns, it is a set of practical guidelines and ideas that can help you create structurally sound software that is maintainable and understandable. Like all other design patterns, it provides you with a common vocabulary with which you can have meaningful discussions with your technical colleagues. It takes root in the Model-View-Presenter design pattern, but diverges in ways that enable you to leverage capabilities of the UI platform to simplify your life as a developer.

MVVM implies three broad categories of objects. Model objects contain the data consumed and modified by the user. They can also include things like business rule processing, input validation, change tracking, and other things related to your system's data. Views, on the other hand, are entirely visual. A View is a UI control that displays data, allows the user to modify the state of the program via device input (i.e. keyboard and mouse), shows videos, displays a photo album, or whatever else your users want to see on the screen. So far, so good. Now let's talk about the middleman: the ViewModel.

A ViewModel is a model of a view.

In case that sentence didn't clarify the topic enough, let's dig a little deeper. First of all, a ViewModel is not merely the new code-behind for a View. That is a common misconception, which totally misses the point. If someone considers ViewModels to be a new form of code-behind, then they probably have too much of their program written in code-behind files. When using ViewModels, your Views can and, in many cases, *should* still have certain kinds of code in their code-behind files. The ViewModel is an abstraction of the user interface. It should have no knowledge of the UI elements on the screen. Logic that deals specifically with objects scoped to a particular View should exist in that View's code-behind.

"What is the point of having ViewModels?" you might be wondering. There are several reasons to create and use ViewModels. The most important reason is that it allows you to treat the user interface of an application as a logical system that can be designed with the same quality of engineering and object orientation that you apply to other parts of your application. This includes the ability to easily write unit and integration tests for the functionality of the user interface, without having to get into the messy world of writing tests for live UIs. It means that the Views that render your ViewModels can be modified or replaced whenever necessary, and little to no changes should be required in the ViewModel classes. It also means that you can use frameworks like MEF to dynamically compose your ViewModels, to easily support plug-in architectures in the user interface layer.

The fundamental mechanisms involved in creating applications based on MVVM are data binding and commands. ViewModel objects expose properties to which Views are bound, including properties that return command objects. When the properties of a ViewModel change, the Views bound to that ViewModel receive a notification when the ViewModel raises its PropertyChanged event (which is the sole member of the INotifyPropertyChanged interface). The data binding system will automatically get the new value from the modified property and update the bound properties in the View. Similarly, changes made to the data in the View are pushed back to the ViewModel via bindings. When the user clicks on a button whose Command property is bound to a command exposed by a ViewModel, that command executes and allows the ViewModel to act on the user's interaction. If the ViewModel needs to expose a modifiable collection of objects to a View, the ViewModel can use ObservableCollection<T> to get collection change notifications for free (which the binding system knows how to work with). It's all very simple to implement because data bindings do the grunt work of moving bound data values around when necessary.

The magic glue that ties Views and ViewModels together is, more often than not, the DataContext property inherited by all visual elements. When a View is created, its DataContext can be set to a ViewModel so that all elements in the View can easily bind to it. This is not a hard and fast rule, though. Some people prefer to put their ViewModels into resource dictionaries and bind to them via resource references. That technique can make it easier to work with Views in Microsoft's Expression Blend visual design tool.

Learn More about MVVM

If you would like to learn more about MVVM before reading further, consider visiting these links:

WPF Apps with the Model-View-ViewModel Design Pattern

http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Simplifying the WPF TreeView by Using the ViewModel Pattern

http://www.codeproject.com/KB/WPF/TreeViewWithViewModel.aspx

Hands-On Model-View-ViewModel (MVVM) for Silverlight and WPF

http://weblogs.asp.net/craigshoemaker/archive/2009/02/26/hands-on-model-view-viewmodel-mvvm-for-silverlight-and-wpf.aspx

M-V-VM via Karl Shifflett

http://karlshifflett.wordpress.com/mvvm/

MVVM Light Toolkit via Laurent Bugnion

http://www.galasoft.ch/mvvm/getstarted/

MVVM Foundation via Josh Smith

http://mvvmfoundation.codeplex.com/