Introduction to “Model-View-View Model” Pattern using WPF in C#

Step-by-step tutorial to creating your first MVVM application

Introduction

For as long as user interfaces have existed, the goal of separating the interface from the business logic has existed. The Service Oriented Architecture (SOA), the Client/Server architecture and most web architectures, as well as others, have all pressed this into practice by necessity. Those who have ignored the practice of separation of layers, who tightly integrate the interface into the business logic, have paid the price in maintenance costs, complexity of debugging and inability to scale.

It is a simple matter within some paradigms, such as WinForms, to drive the business logic directly into the interface code. In fact it is even a simple matter to store data inside interface object such as lists, as they control sorting automatically, eliminating the need for the developer to perform this function. While simplicity and ability drive this possible condition, this is not a valid pattern, even if it does work.

With the advent of ASP.NET the separation of markup and logic became common place, but the code-behind model still allowed, and even through ease, drove many developers to include business logic at the interface layer. It can be argued successfully that the code behind is a middle layer. If careful separation of interface event handling and business logic routines is maintained, this can be a valid point as the code-behind does only execute at the server, generating the client side output that may include JavaScript for the pure interface logic.

As the Microsoft technologies move forward, and with the advent of the Windows Presentation Foundation (WPF) and Silverlight, the concept of separation of interface become almost mandatory. While it is possible to combine layers and place business logic in the interface logic, this has serious drawbacks and limitations. Moving forward, adoption of the “Model-View-View Model” pattern becomes almost mandatory. The language paradigms are wrapped around this pattern, so ignoring it can greatly complicate designs.

Overview of the Model-View-View Model (M-V-VM) Pattern

There are five primary parts to this powerful pattern. These parts are the “Model”, the “View”, the “View Model”, the “Commands” and the “Data Handler”. Yep they left out the “C” for commands and the “DH” for the data handler, but we’ll get to this later. In truth these are simply extensions to the pattern, but because they are very common, I will include them in my description. I’d hate to go as far as creating an “M-V-VM-C-DH” pattern… it’s already confusing enough.

Also, understand that the proper definition for the M-V-VM pattern is still debatable, with multiple descriptions and positions as to the best way to define each part of the pattern. This assures that my description will align, to some degree, with some interpretations, while disagreeing more or less with others. I have tried to find common ground with most interpretations where possible, but you can't please all the people all the time.

For instance, when I describe the M-V-VM pattern, I always try to put the dashes in the name, but this is not always done. "MVVM" is also a valid form for the name of the pattern; it just makes visually separating the components more difficult for the novice. I also describe the components by skipping the first "M" for model until I have described the other components. I just find the understanding flows easier this way.

The View

This first part of the pattern to discuss is the "View". The view is the user interface itself. This is the part of the system which interacts with the user, displays information and retrieves information from the user. It is important to point out that the view never reads or writes, to or from, any data source directly. It simply focuses on display and gathering of information.

Every interface needs to display data and/or gather user input, so if the view can't talk to any data source directly, then how can it perform its primary function of displaying data and collecting input? This must be a conflict, right?

The View Model

This is where the real beauty of M-V-VM comes into play. For every view, there is a fully separate layer knows as the "View Model". This layer can be thought of as the data layer for the view. Every view has a view model where it gets its data and where it sends its user input. The view is focused on user interactions, flow of the users focus and interaction with the user to display and collect the required data.

The view model exposes the data for the view as public properties and methods. It is fully unaware of the view and this is a key feature of the view model. Being fully unaware of the interface means that the view model can be tested without the need for any user interface at all, allowing automated testing to be used with ease. (Accept that this concept will sink in firmly as you move forward with this tutorial)

This also means that interface designers need only know what properties and methods are exposed by the view model to design an interface. They can pull one interface (view) out and replace it with a completely different interface (view) without any need to change the view model.

Pause and contemplate this for a moment as it is extremely important. Never again will an interface designer break your code! Sorry interface designers, the programmers can still break your design by eliminating or changing the available properties and methods in the view model.

Think about teams using HTML and the massive challenge of programmers working with designers and how each one regularly breaks the work of the other. CSS helped by pulling the styles out to separate files. ASP.NET helped more by pulling the code into a separate file. Still the tightly coupled code and design caused a situation where programmers and designers were at odds and had to be highly territorial and enforce ownership through procedure and agreement vs. having it enforced by the technology itself.

The View-View Model Relationship

Because these first two components are so basic to understanding the M-V-VM pattern, let me state the relationship between the view and the view model another way, just to assure clarity. The view model exposes data in the form of public properties and methods.

The view model unaware of what is accessing the properties and methods it exposes. This allows the views, which are independent from the view model, to be as simple as a set of calls to test the functionality for the view model's exposed parts, or as complex as a fully functional, beautiful, highly interactive user interface.

Many different views can be used to access, and if needed display and update the properties of the view model. These views can be swapped in and out without the need to update any part of the view model. This separation is powerful, beautiful and worth adopting.

A Binding Relationship

I hope you understand this relationship and the power it offers. The developers at Microsoft surely do and they have done a great job of supporting the M-V-VM pattern in WPF. To help simplify things, WPF uses "bindings" to hook the view to the view model. The view model can expose a property as simple as a Boolean value or as complex as a collection. The view can then bind a control to that properly.

To accomplish the binding, the view sets the DataContext to the view model, and binds a property of a control to a specific property of the view model. Once you master the syntax, this is powerful and simple.

You can bind a view input object, such as a TextBox, and have the users input automatically used to update the bound view model property, or you can populate a GridView by binding it to a view model property that is a collection. As items are added or deleted from the collection, the GridView can be made to update automatically in real-time.

The binding from the view to the view model assures that the view is highly aware of the view model, but that the view model is only aware that something is accessing its properties and methods, but does not care what that something is. This is why it is so simple to swap in and out different views of a single view model.

The Data Handler

Not everyone agrees on this as a separate part of the pattern, but you will find that by separating the data handling from the view model gives you yet another layer of modularity and allows for greater scalability and options for shifts of data storage types. Basically the only requirement here is that you keep your data handling routines in a separate file as classes used by the view model.

Why is this helpful? Imagine you wrote an application and decided to use an Access database for your data store, then the client insisted that you use XML files only to later insist that you use an SQL Database stored locally, only then to change to using a remote SQL database and then later to insist that you use a fully service oriented architecture and access all your data from web services. With so many options for data storage, it only makes sense to separate the data handling logic from any interface components.

Our example does not access any remote data sources, but if it did you'd add a DataHandlers folder to the project and create a file, that following the convention of this example might be called MainDatahandler.c. In that file you'd create handlers that passed data elements for each associated property in the view model. You'd build the logic to retrieve and store the data in whatever data store was selected inside the data handler module. This would separate the need for data from the methods to access it, allowing changes to the data formats and technologies without requiring changes to the view model.

Commands

This can be a confusing concept for some developers as the meaning of the term is somewhat counter intuitive. Some initially think that any commands issued by the view should reference the commands modules, but this is incorrect. Those commands are actually methods of the view model. So while the term command is used in several places, care must be taken so that it is not confused with the Commands modules.

The commands modules are low level data marshaling and framework connection commands. The commands can be thought of as the glue or framework that pulls the various parts together. In our example this will be built for us automatically and we will not spend any time reviewing this part. Until you progress deeper into M-V-VM just understanding that this is required and that the project template creates it for us is enough.

The Model

The first letter of the pattern name is the last part I like to cover when talking about the M-V-VM pattern. The model contains the class definitions for the data classes. In our example this will be the UserDataCollection. (More on this later) We use the model to define this data type and then use that in the view model to maintain the data collection.

The model is typically broken into one or two sections, the parent and child modules. The main file will contain parent types, but as with all data descriptions, there is often a hierarchy to the data that demands a parent/child relationship. Many children may be required to fully make up the full set of data describing the parent.

Imagine an invoicing application where the parent object may contain the customer data. There may be child objects to contain the line item data. The child cannot stand alone and be complete and the parent requires one or more instances of the child to complete the full data set required for the application.

Other Perspectives

Another nice, quick overview of the M-V-VM pattern can be found at http://johnpapa.net/silverlight/5-minute-overview-of-mvvm-in-silverlight/ and is suggested reading before advancing any further with this tutorial if all of the above concepts are not yet clear. It is short, sweet and to the point. It is very well written.

The M-V-VM Toolkit 0.1

There has only been a small amount of guidance from Microsoft when it comes to using the M-V-VM pattern with WPF. They did however release a simple toolkit that includes a new project type for Visual Studio that works as a foundation for a simple M-V-VM application using WPF. The version for this toolkit is 0.1, which should work as the first clue as to its robustness. As a side note, as you grow to using M-V-VM, you should look into PRISM for enterprise level development. I will not cover any PRISM features in this tutorial.

You can download the M-V-VM Toolkit 0.1 from the following URL:

http://www.codeplex.com/wpf/Release/ProjectReleases.aspx?ReleaseId=14962

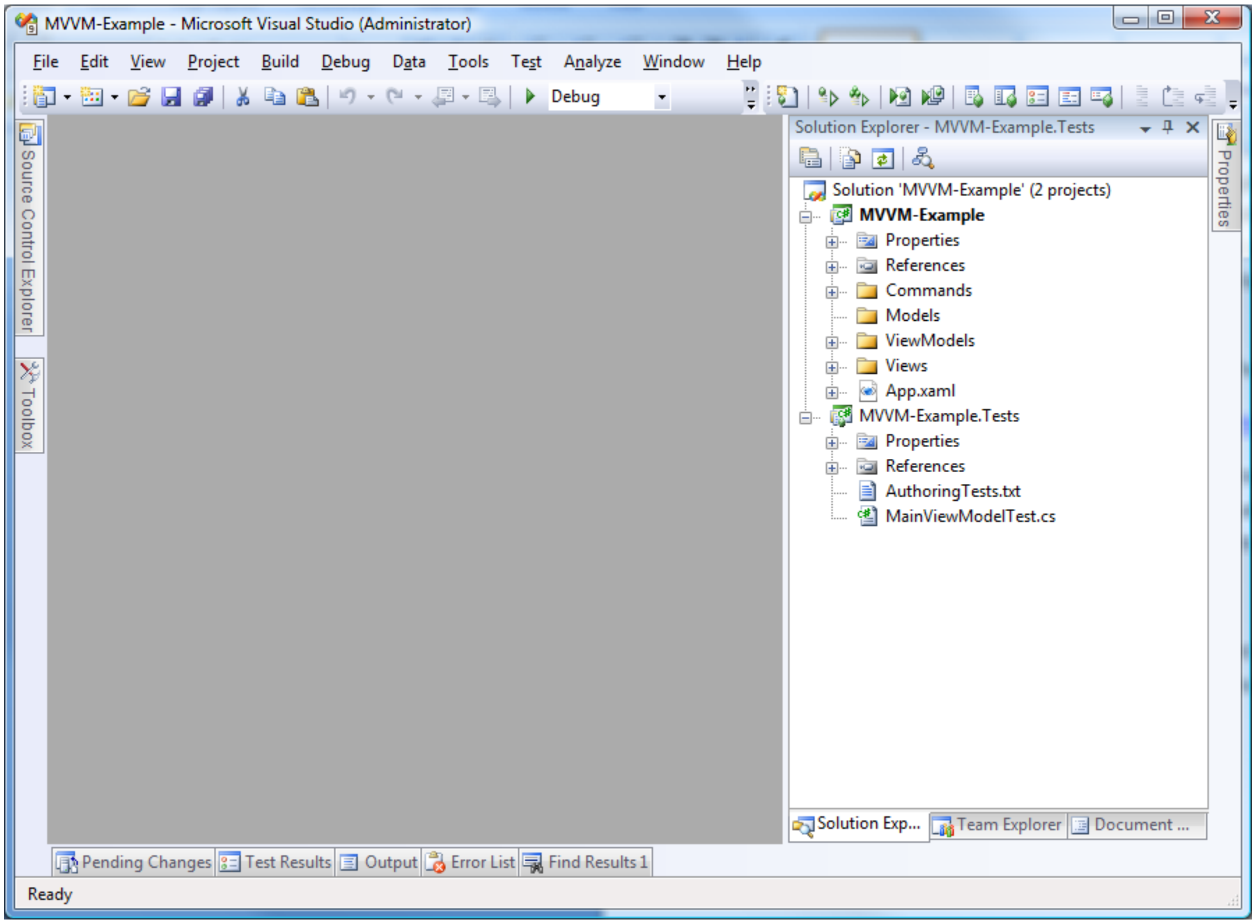
Once you download the toolkit, extract the contents. With Visual Studio closed, run the WPFModelViewTemplate.msi file located in the Visual Studio Template folder. This will add the new project type to Visual Studio. Included with the toolkit are samples, but if you finish this tutorial first, you'll have a much better chance of getting those examples working.

Creating the MVVM-Example Project

Open Visual Studio 2008 and create a new project. You will notice that under "Visual C#" there is a group called "Windows" and if you select that group you should see a project type called "WPF Model-View Application".

Select this and then name the project "MVVM-Example" and check the "Create a directory for solution" checkbox if it is not already checked by default. Once this is done, click "OK". When prompted, keep the default to generate a unit test project and click "OK".

Your screen should look similar to this:



Defining the Goal of our Sample Project

Our sample project has the goal of teaching us to build an M-V-VM project using WPF in C# using Visual Studio. Because the goal is not related to program functionality, the functionality will be quite derived. The application we build will be useless, except as a sample of programming methods and to bring comprehension of the pattern to your understanding.

Because our sample is all about learning and not about solving a business challenge, we will focus on sample code that demonstrates common functionality. We will create a TextBox that allows the user to enter data, a button to allow the user to save the entered data and a GridView to display the data as we collect it.

We will study one-way and two-way binding of controls, allowing the interface to drive data into the view model, display data from the view model and to do both on a single control. We will explore when it is proper to add code to the XAML code behind and when it isn't. We will also learn when it's ok to name a control and why we rarely need to do this. Naming controls is a default for any WinForms developers, so initially it may seem strange for the default rule to be not to name them. As we progress you will see why we no longer want to name controls.

The final functional goal of our example will be to allow a user to type in any text data, hit an "Add" button and have that data stored in memory as part of a collection. To demonstrate that data can exist that is computed or derived, we also store a random number each time we store some user data. We then display the contents of the collection in a GridView. We also want to enable and disable the Add button depending on the availability of user data.

We will build our program in phases so that each concept has the best opportunity to become clear in your mind. This is not typically how you will code once you comprehend the whole picture, but we will leave those choices up to the reader as they develop new applications in the future.

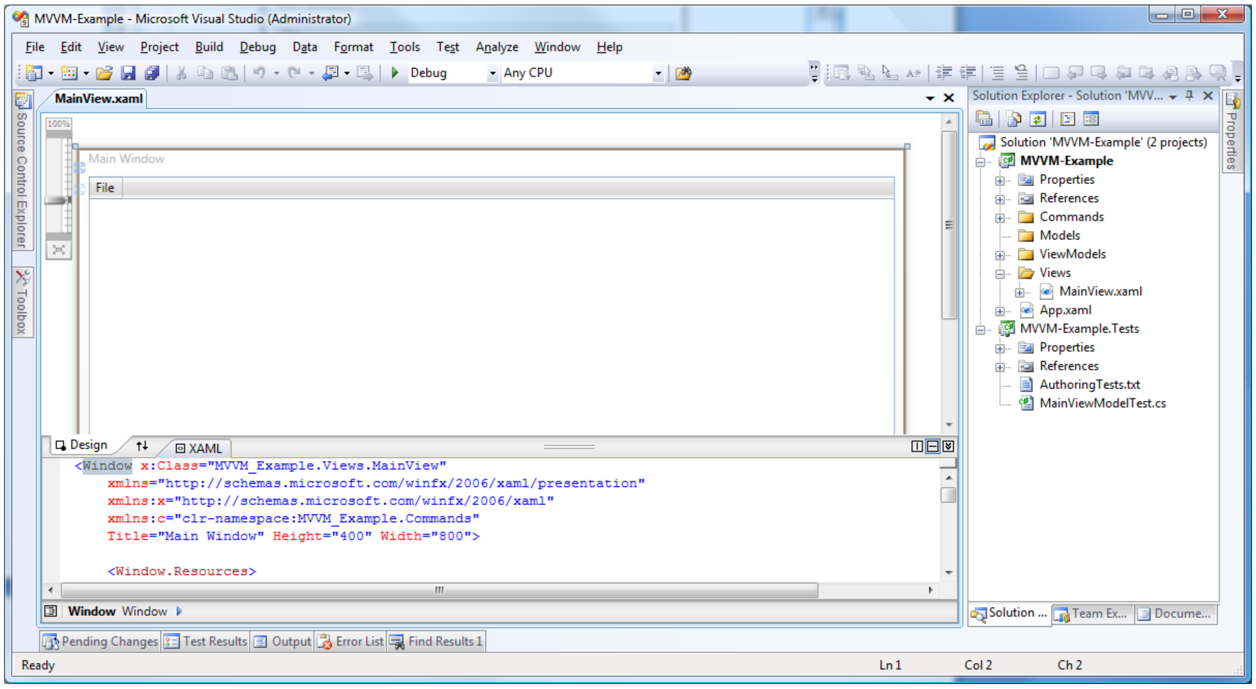
We also have two unused controls on our form; we will talk about their use, but will not go into details. We will leave that to you, as homework to show that you understand the new things you have learned. They are the load and save buttons.

You will need to bind these two buttons from the view to the view model and then create a Data Handler module that you call from the view model to do the actual saving and loading. This is of course optional, but it is a good exercise to complete the comprehension of the project and its teachings.

Adding the Controls to the Form

To construct the view, you will need to open the XAML for the view. It is found in the "Views" folder of the project in the solution explorer. The file is named "MainView.xaml". Double click this to open the view in its default state.

You should see:

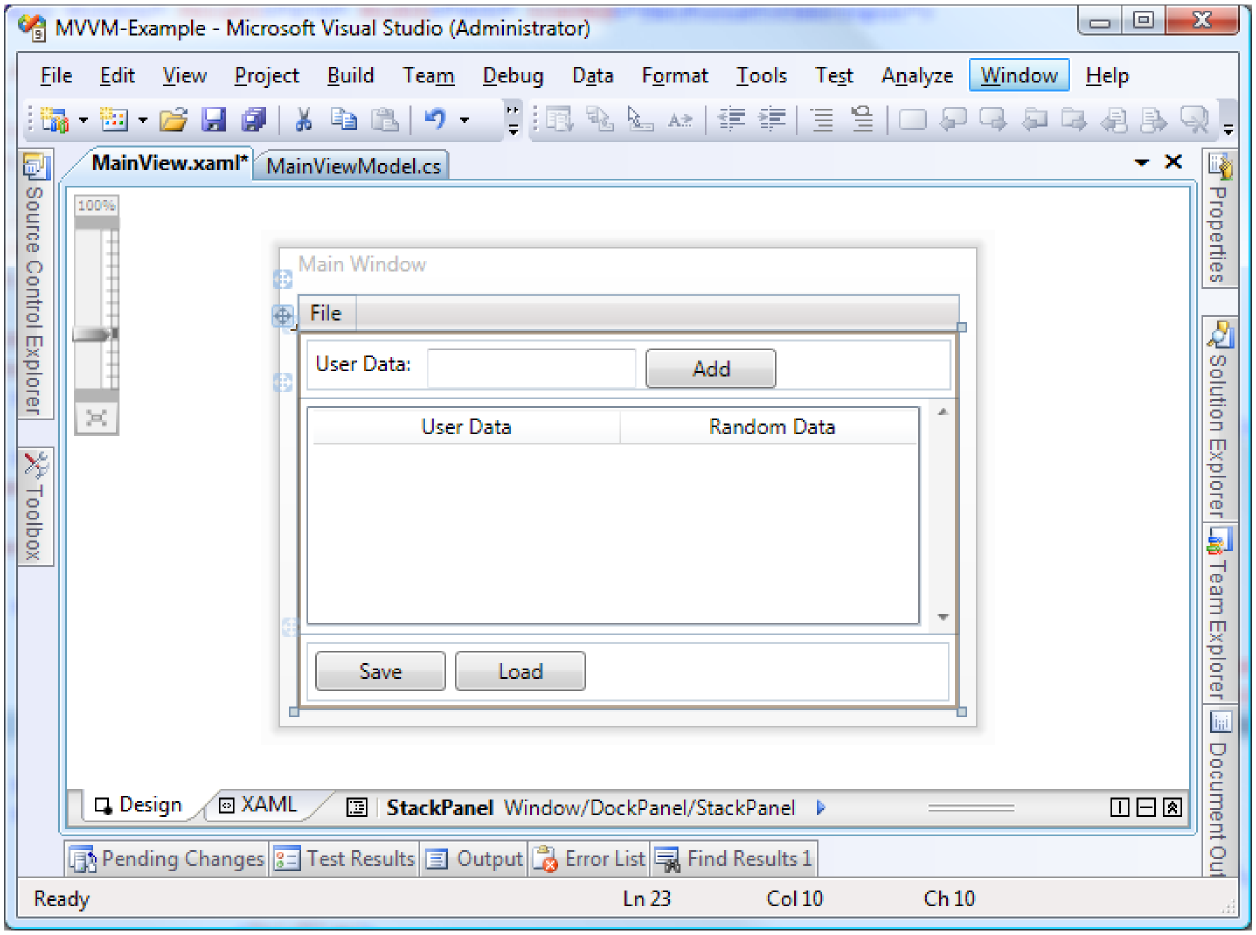


```
<Window x:Class="MVVM_Example.Views.MainView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c="clr-namespace:MVVM_Example.Commands"
    Title="Main Window" Height="400" Width="800">

    <Window.Resources>
```

To simplify building the interface, paste the XAML below to replace the existing XAML for the form.

```
<Window x:Class="MVVM_Example.Views.MainView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c="clr-namespace:MVVM_Example.Commands"
    Title="Main Window" Height="275" Width="400" >

    <Window.Resources>
        <!-- Allows a KeyBinding to be associated with a command defined in
the View Model  -->
        <c:CommandReference x:Key="ExitCommandReference" Command="{Binding
ExitCommand}" />
    </Window.Resources>

    <Window.InputBindings>
        <KeyBinding Key="X" Modifiers="Control" Command="{StaticResource
ExitCommandReference}" />
    </Window.InputBindings>

    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="_File">
                <MenuItem Command="{Binding ExitCommand}" Header="E_xit"
InputGestureText="Ctrl-X" />
            </MenuItem>
        </Menu>

        <StackPanel>

            <WrapPanel Margin="5" VerticalAlignment="Stretch">
                <Label>User Data:</Label>
                <TextBox Height="23" HorizontalAlignment="Left"
Margin="5,5,0,0" VerticalAlignment="Top" Width="120" />
                <Button Margin="5,5,0,0" Height="23"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
IsDefault="True">Add</Button>
            </WrapPanel>
            <Grid>
                <ScrollViewer Height="135"
VerticalScrollBarVisibility="Visible" Grid.Row="1">
                    <ListView Height="125" Margin="5,5,5,5">
                        <ListView.View>
                            <GridView>
                                <GridViewColumn Width="175" Header="User
Data" />
                                <GridViewColumn Width="175" Header="Random
Data" />
                            </GridView>
                        </ListView.View>
                    </ListView>
                </ScrollViewer>
            </Grid>
            <WrapPanel Height="33" Width="367" Margin="5"
HorizontalAlignment="Left">
                <Button Height="23" Width="75" Margin="5">Save</Button>
                <Button Height="23" Width="75">Load</Button>
            </WrapPanel>
        </StackPanel>
    </DockPanel>
</Window>
```

This tutorial is not a lesson on how to build a XAML form, so we will skip over talking about how to do this. When you paste the code above, you should get a form that matches what is shown below:


MVVM-Example - Microsoft Visual Studio (Administrator)
File Edit View Project Build Team Debug Data Format Tools Test Analyze Window Help
MainView.xaml* MainViewModel.cs
Main Window
File
User Data:
Add
User Data
Random Data
Save
Load
Design XAML StackPanel Window/DockPanel/StackPanel
Pending Changes Test Results Output Error List Find Results 1
Ready Ln 23 Col 10 Ch 10
Source Control Explorer
Toolbox
Properties
Solution Explorer
Team Explorer
Document Out

Is it pretty? No, but it will give us what we need for this example. If you run by pressing F5, you will see that we have an executable example, but the functionality is very limited. The menu with the File->Exit functionality was included for us by the project type. Once you have the form running, use the exit function to return to Visual Studio.

Adding Functionality to the Add Button

We assume that may readers have some WinForms or ASP.NET experience, so they are used to buttons firing an event in the code behind. This works in WFP as well, but this breaks the M-V-VM model. If we write the event in the code behind, then the function of the button, which is to take the user data and add it to the collection, will happen inside the view. This is not the desired result. So now is a good time to set old practices aside and trust that we will show you a better way and give yourself the freedom to accept a new paradigm.

In WPF using M-V-VM, any real functionality, especially when this will involve data, needs to be done in the view model. This makes sure that ANY interface that attaches to the view model automatically gains this same functionality. We can test the functionality at the view model level and know that any future errors that might pop up are the designer’s problem. If the view model is a proven, tested platform, then when a new view is added, any new errors that exist most likely reside in the new view. Regression testing against the view model can prove this.

Expose a Method

The first thing we need to do is expose a method in the view model that we can bind the click action of the button to. To do this, open the MainViewModel.cs file located in the ViewModels folder in the project, within the Solution Explorer.

By default the following code will exist:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Input;

using MVVM_Example.Commands;

namespace MVVM_Example.ViewModels
{
    public class MainViewModel : ViewModelBase
    {
        private DelegateCommand exitCommand;

        #region Constructor

        public MainViewModel()
        {
            // Blank
        }

        #endregion

        public ICommand ExitCommand
        {
            get
            {
                if (exitCommand == null)
                {
                    exitCommand = new DelegateCommand(Exit);
                }
                return exitCommand;
            }
        }

        private void Exit()
        {
            Application.Current.Shutdown();
        }
    }
}
```

This code creates a MainViewModel class that is derived from the ViewModelBase class. This gives us some basic wiring that we will look at later. For now I want you to notice one specific function, the last one. It is the Exit() function. Note that this is marked as private, but that it is exposed as a public method of the view model just above it through DelegateCommand(). The ICommand interface exposes the method and prepares it for later consumption. This is what is needed for all view model methods that will be accessed by the view controls.

To assure this is clear, let's add a new method to support our button. For now, we will have our button also exit the application. This lets us focus on the construction and not the functionality, for now.

Add the following to the code. Note that the existing code is in grey and the new code is properly color coded.

```
namespace MVVM_Example.ViewModels
{
    public class MainViewModel : ViewModelBase
    {
        private DelegateCommand exitCommand;
        private DelegateCommand addCommand;

        #region Constructor

        public MainViewModel()
        {
            // Blank
        }

        #endregion

        public ICommand ExitCommand
        {
            get
            {
                if (exitCommand == null)
                {
                    exitCommand = new DelegateCommand(Exit);
                }
                return exitCommand;
            }
        }

        private void Exit()
        {
            Application.Current.Shutdown();
        }

        public ICommand AddCommand
        {
            get
            {
                if (addCommand == null)
                {
                    addCommand = new DelegateCommand(Add);
                }
                return addCommand;
            }
        }

        private void Add()
        {
            Application.Current.Shutdown();
        }
    }
}
```

We are adding commands, but as mentioned before, these commands reside in the view model and should not be confused with the commands section of the project. While the term commands is indeed used don't be confused, think of these as methods of the view model and not global commands to the project.

Looking at the newly added code, you can see that we added the private Add() method and created a discovery tool to allow access to it using the "get" of the public AddCommand() method through DelegateCommand. At this point, our view model now has a second exposed method that can be called by any view.

To demonstrate this more clearly, let's add a binding to this method from the view and watch the execution. Set two breakpoints, the first on the following line:

```
addCommand = new DelegateCommand(Add);
```

And the second on the this line: (Inside the private Add() method)

```
Application.Current.Shutdown();
```

Once this is done, switch over to the XAML for the view and add the following:

```
<Button Command="{Binding AddCommand}" Margin="5,5,0,0" Height="23"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
IsDefault="True">Add</Button>
```

This tells the button control that we have added a command called "AddCommand" which it expects to find in our view model as an exposed public method. Because we have set our view model to properly expose this method, we can now expect the binding to flow execution to the breakpoints we set earlier.

As the form is created, we can expect the control to seek out and find the method. We will hit the first breakpoint as the discovery occurs and the method is exposed. Once we click the Add button, we can expect execution to hit the second breakpoint when the private function is called through the exposed reference.

Press F5 and you should see the program stop at the first breakpoint. Once you hit the breakpoint, hit F5 again to continue the application. The application should show on the screen. Click the "Add" button on the view and you should hit the second breakpoint.

It is surprisingly simple to bind the command for a button in the view, to a method in the view model. You should now be able to bind any button on any view to any method you create and expose in the view model. Nicely done! But wait, it gets much better.

Binding a TextBox

Exiting the application isn't really what we want the Add button to do. We really want it to get the text from the TextBox, entered by the user, and store it in a collection. To allow this, we first need to get the data out of the TextBox and into the view model, as a public property of the view model.

Open the MainViewModel.cs file and make the following changes:

```
.
.
.
namespace MVVM_Example.ViewModels
{
    public class MainViewModel : ViewModelBase
    {
        private DelegateCommand exitCommand;
        private DelegateCommand addCommand;

        private string _newUserData = "";

        #region Constructor

        public MainViewModel()
        {
            // Blank
        }

        #endregion

        public string NewUserData
        {
            get
            {
                return _newUserData;
            }

            set
            {
                if (value != _newUserData)
                {
                    _newUserData = value;
                    OnPropertyChanged("NewUserData");
                }
            }
        }

        public ICommand ExitCommand
.
.
.
```

We create a private field called _newUserData and we create a public property called NewUserData with a set and get to support marshalling the data in and out of our object. There is one line of code to make special note of. When a property is changed in the view model, we need a way to let the view know that it changed so that it can update any controls bound to that property. To do this we use the OnPropertyChanged function. This is defined in the ViewModelBase.cs file that was automatically created as part of the project type we used. We do not have to concern ourselves with the mechanics, but we do need to understand the importance of using this for notification to the view.

Now we just need to bind the TextBox control to this new property. Pretty simple to do, just open the XAML and make the following change:

```
<TextBox  Text="{Binding  NewUserData}"  Height="23"  HorizontalAlignment="Left"
Margin="5,5,0,0" VerticalAlignment="Top" Width="120" />
```

Notice that we bound the Text property of the view's TextBox's control to the NewUserData property of the view model. The actual binding is all done for us, automaticly and transparently. It's wonderful magic. We can focus on the logic and let the framework do all the heavy lifting.

It's time to run again, so press F5. You will hit the first breakpoint again, where the addCommand gets discovered. We can assume this will happen each time we run, so remove the breakpoint from this line so we don't stop here every time we run. Now press F5 again to contine execution.

One the view is displayed, type something in the User Input TextBox and click the Add button. This will bring us to the remaining breakpoint and allow us to examine the contents of the _newUserData field.

Scroll to the top of the page and hover over the _newuserData field and you will see that it contains whatever text you typed in the TextBox. Thus, we can see that we have successfully bound the TextBox control to the view model property.

At this point in the tutorial, you should know how to bind a button to a method and a TextBox to a property. This wasn't as hard as you expected was it? Now lets update the button so that it no longer exits the application, but instead stores the data in a collection.

Storing the User Data in a Collection

If you havn't done so already, stop the application and return to Visual Studio. Now add the following code to the MainViewModel.cs file:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Input;

using MVVM_Example.Commands;
using MVVM_Example.Models;
.
```

```
.
.
        private string _newUserData = "";
        private Random _rand = new Random();

        #region Constructor

        public MainViewModel()
        {
            // Blank
        }

        #endregion

        public ObservableCollection<UserData> _userDataCollection = new
ObservableCollection<UserData>();
        public ObservableCollection<UserData> UserDataCollection
        {
            get {return _userDataCollection;}
            set { _userDataCollection = value; }
        }

        public string NewUserData
        {
            get
            {
                return _newUserData;
            }
.
.
.
        private void Add()
        {
            UserData _newDataItem = new
UserData(_newUserData,Convert.ToString(_rand.Next(100)));
            UserDataCollection.Add(_newDataItem);
        }
    }
}
```

Please make sure when you paste the code into Add() that you remove the line that exits the application. We want to reserve this functionality only for Exit(). In fact, notice that you have deleted the line with your breakpoint. If the breakpoint floats up to a new line, remove it manually. At this point you should have no breakpoints left in your project.

To examine the results of our project so far, we do need a breakpoint. Please set a breakpoint in Exit() on the line that does the shutdown. It looks like this:

```
            Application.Current.Shutdown();
```

Yep, just like the line we had the breakpoint on in Add(). This is the last line in our code to execute, and runs when we choose File->Exit from the menu. Before we let the application shutdown, we can examine the contents of the collections to see that it contains the data we expect.

Here we have the opportunity to see an example of using the model. Remember, the model contains the type information for each of the data structures in our application. In this application we won't need any child data types and only a single parent data type that we will use as the foundation for our collection.

To create the model, right click on the Models folder in the Solution Explorer and select Add->New Item and then select Visual C#->Class and name the Class file MainModel.cs and click the Add button. If it didn't open by default, open this file by double clicking on it in the Solution Explorer.

By default the file will contain the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MVVM_Example.Models
{
    class MainModule
    {
    }
}
```

To specify our new data type, paste the following code into the file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MVVM_Example.Models
{
    class MainModule
    {
    }

    public class UserData
    {
        #region Declarations

        private string _theEnteredData;
        private string _theRandomData;

        #endregion

        #region Constructor

        public UserData(string theEnteredData, string theRandomData)
        {
            this._theEnteredData = theEnteredData;
            this._theRandomData = theRandomData;
        }

        #endregion
        #region Properties

        public string theEnteredData
        {
            get { return _theEnteredData; }
            set { _theEnteredData = value; }
        }

        public string theRandomData
        {
            get { return _theRandomData; }
            set { _theRandomData = value; }
        }

        #endregion
    }
}
```

This class contains two private fields called _theEnteredData and _theRandomData exposed as two public properties theEnteredData and theRandomData as well as a constructor for the class. This is then used by the previous code to create an ObservableCollection of this data type. An ObservableCollection is an extension of the classic Collection type but adds the hooks to allow data binding including notifications.

You should also note that when we updated the MainViewModel.cs file, we added a using statement to make the Models available. We can't reference a class in another file without this reference. We also added a reference to the System.Collections.ObjectModel so that we have reference to the ObservableCollection type.

Lastly we created a private field named _rand to hold a random number to be stored as part of the ObservableCollection each time we add a new entry. We use the _rand field in the Add method on the collection as we added a new element. This is simply to demonstrate that we can have data collected that did not originate in the view. This data might have come from an external source or might have been a calculated value.

This was the largest code change between executions of the code, so make sure each change makes sense and that you fully understand the syntax and needs. If not, re-read this section and study the code until it is clear.

To help assure understanding, let me summarize the changes. We created a UserData type in the MainModel.cs file. We used that type to create an ObservableCollection in the MainViewModel.cs file. We referenced the System.Collections.ObjectModel to support the ObservableCollection and referenced MVVM_Example.Models to allow us to use the UserData class. We created a _rand field to allow us to include a random number as we stored the user entered data to demonstrate that not all data must flow from the view. We also updated the Add() function to no longer exit the application but to store the user data and the random number in the collection. Lastly, we set a new breakpoint so that we will stop execution just before the application ends .

If you made all the changes above, you should have no errors and be ready to run the application.

Observing the ObservableCollection in Action

When you have all the edits made and you have the new breakpoint in place, press F5. When the view appears, enter some data in the User Data TextBox and click Add. Enter some different data and click Add again. Note that the old data doesn't clear, for now, but we will fix that later on. For now you have to erase the old data before typing new data and you have to click the TextBox in order to get the cursor into it so that you can type. We will fix all that before we are done.

When you have entered a few data items and clicked Add a couple of times, use the menu and File->Exit the application. This should bring us to our breakpoint. If you scroll up near the top of the file, hover over the UserDataCollection variable and you should be able to inspect it to find the data you entered along with random numbers stored there.

Find and remove the one existing breakpoint and then press F5 to allow the application to exit normally.

Bind the Grid to the Collection

As with the other bindings, once you have the properties exposed in the view model, binding to them in the view is a simple matter. Unlike the other examples, this one is a bit more complex because it includes repeated data with an unknown number of entries, so we have to do a bit more work to get this to work. As you will see, while more steps are involved, the logic is clear. Once you see it, you will find it to be a simple procedure.

The key concept to learn here is that you have to bind both to the collection, and to each individual column. You bind the ListView to the public property of the view model (the collection) and the columns to the elements of that property. Make the following changes to the MainView.xaml file:

```
<ListView ItemsSource="{Binding Path=UserDataCollection}" Height="125"
Margin="5,5,5,5">
    <ListView.View>
        <GridView>
            <GridViewColumn DisplayMemberBinding="{Binding
Path=theEnteredData}" Width="175" Header="User Data" />
            <GridViewColumn DisplayMemberBinding="{Binding
Path=theRandomData}" Width="175" Header="Random Data" />
        </GridView>
    </ListView.View>
</ListView>
```

Notice that in XAML, while we talk about a Grid, we are actually working with a ListView. Don't let the semantics confuse you, for all reasonable purposes, this is a grid. Also note that at the ListView level we set the ItemSource and at the column level we set the DisplayMemberBinding. As was said previously, we bind the ItemSource to the collection as a whole, and we bind each column to the specific element we wish to display.

Binding Both Ways

We now have an example where data, entered by the user, flows from the view to the view model and then is manipulated and persisted by the view model, at least while the application is running, and then flows back to the view for display. We are now binding both ways, in and out of the view model, but we are not binding two-way. We will talk more about this soon.

For now, press F5 and let's run our application and see the results. Once the view is displayed, enter some user data and click the Add button. It should show the results almost immediately in the grid. Each time you Add user data, a new row should be added to the grid. Pretty cool isn't it? When you are done enjoying your handiwork, exit the application.

Two-Way binding

As was hinted to above, while we are binding both ways, we are not yet using two-way binding. Two-way binding is where a single binding pushes data into the view model and pulls it back out. We have also noticed that we have to manually delete the old user data before we can type new user data. This is not exactly a slick interface design.

To clear the value from the user input field, we will need to add some logic to the Add() function so that after it uses the user data, it clears the user input field. But how can we do that? The view model cannot have any awareness or direct control of the view. Perhaps if we clear the property, removing the user data, the interface will automatically update… what do you think?

Let's find out through experimentation. Add the following line to the Add() function:

```
        private void Add()
        {
            UserData _newDataItem = new UserData(_newUserData,
Convert.ToString(_rand.Next(100)));
            UserDataCollection.Add(_newDataItem);
            NewUserData = "";
        }
```

Note that we did not set the _newUserData field, instead we used the set for the property. This is important and this is how you must always set properties. When we do it this way, the set is called and the set includes notification back to the view in the way of the OnPropertyChanged event. This way the interface knows to update. That line has been in our code for quite awhile and now we finally see it needed.

So what do you think will happen if we press F5 and enter some data? Will the field clear when we empty the property? Well, let's find out. Press F5 and enter a couple values clicking the Add button between each. So, what happened? It worked didn't it… well it looked like it worked… but really, it didn't! Give me a chance to explain.

In this example, because it is simple, the binding worked because we did raise a notice, but if the example was more complex and we were creating different instances of the data for any reason, then the bindings would stay bound to the original instance, and once garbage collection cleared the old instance from memory, we'd be in trouble. To get real two-way binding that is reliable, thread-safe and ready for business application use, we need to do more than just notify of an update, and we need true two-way binding.

To add two-way binding, you must add "Mode=TwoWay" to the binding definition. So for our TextBox, make this small change in the XAML:

```
<TextBox Text="{Binding NewUserData, Mode=TwoWay}" Height="23"
HorizontalAlignment="Left" Margin="5,5,0,0" VerticalAlignment="Top"
Width="120" />
```

That's all there is to it. Now we are safe and we are officially binding two-way. Remember, you may get away with the appearance of two-way biding with simple notification, but if you want real two-way binding, then you need to specify it in the XAML.

Disable the Add Button

Unless there is something typed in the user input text box, the Add button really should be disabled. This is another one of those times when it seems a bit of code behind in the view would do the trick nicely, but that means every time we write a view we will need to include this logic, additionally with that method, the view model does not enforce its own rules. It would be better if the isEnabled property was controlled from the view model, so we get a consistent interface no matter which view we attach.

To add this feature, make the following changes to the MainViewModel.cs file:

```
.
.
.
namespace MVVM_Example.ViewModels
{
    public class MainViewModel : ViewModelBase
    {
        private DelegateCommand exitCommand;
        private DelegateCommand addCommand;

        private string _newUserData = "";
        private Random _rand = new Random();
        private bool _addButtonEnabled = false;

        #region Constructor
.
.
.
        public string NewUserData
        {
            get
            {
                return _newUserData;
            }

            set
            {
                if (value != _newUserData)
                {
                    _newUserData = value;
                    OnPropertyChanged("NewUserData");
                    if (_newUserData.Length > 0)
                    {
                        AddButtonEnabled = true;
                    }
                    else
                    {
                        AddButtonEnabled = false;
                    }
                }
            }
        }

        public bool AddButtonEnabled
        {
            get
            {
                return _addButtonEnabled;
            }
            set
            {
                if (value != _addButtonEnabled)
                {
                    _addButtonEnabled = value;
                    OnPropertyChanged("AddButtonEnabled");
                }
            }
        }
.
.
.
```

As we have done before, we added and exposed a public property by adding a private field and creating a public method to expose the set and get funtions. Note that as we have done in the past, we used the OnPropertyChanged function to assure that as we change this property the view gets notified.

We know that to set the property we always set the property (AddButtonEnabled) and not the local field (_addButtonEnabled) to assure that we send the view notification. This also allows us to include other logic at this stage. In this case, we add simple logic to set the property based on the length of the data entered by the user.

Now all we just need to tie the isEnabled property of the views Add button to the AddButtonEnabled property of the view model. To do that, make the following addition to the XAML.

```
<Button Command="{Binding AddCommand}" IsEnabled="{Binding AddButtonEnabled,
UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}" Margin="5,5,0,0"
Height="23" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
IsDefault="True">Add</Button>
```

Note that we now have two separate bindings on this control. The first handles the command property of the control, executing the Add() method in the view model, and the second one controls the IsEnabled property of the same control.

You can have separate bindings for any number of separate control properties. This allows the view model to drive control for much of the view without actually knowing anythign about the view itself. It also means that the view designer only needs to know what properties exist and to bind to them to get all the functionality supported, no matter what view they are designing.

We also need the text box to fire the command method every time a key is pressed and not just when the control looses focus. To do this, make the following change to the XAML.

```
<TextBox Text="{Binding NewUserData, UpdateSourceTrigger=PropertyChanged,
Mode=TwoWay}" Height="23" HorizontalAlignment="Left" Margin="5,5,0,0"
VerticalAlignment="Top" Width="120" />
```

By setting an UpdateSourceTrigger, we tell the control to monitor for the Text property of the control to change, and when it does, pull the trigger, just like if the field had lost focus.

Ok, we once again made a number of changes, so let's summarize again. We added a private _addButtonEnabled field and exposed it via a public method like we have done several times before. We added code to the NewUserData property to check the length of the user data and set the AddButtonEnabled property true if the user has typed some data, else we set it false.

We then made two changes to the XAML. The first bound the isEnabled property of the button control to the view model and the second was to make sure the text box called the view model every time the text in the text box changed. This way we can update the status of the AddButtonEnabled property each time a key is pressed.

The Exception to the Rule

As with most sets of rules, there are exceptions. The M-V-VM pattern pushes most of the interface control to the view model, but there are exceptions, times when code behind and adding specific names to controls is OK. The key to a good M-V-VM design is keeping it minimal, moving what you can to the view model, but there are some things that just make no sense in the view model.

One of these exceptions is cursor control. Where should the cursor start, where should focus flow as the user inputs data. These are pure interface issues and are independent of the data, validation and processing. There is no fixed business logic to cursor flow, so putting this in the view model forces the view model to know about the view and this rule is far more important not to break than any rule we might break by adding a minimal amount of code in the views code behind.

Typically a view's code behind is pretty bland, it will initialize the component and that's about it. We need just a tiny bit more for our application to be finished so make the changes below to the code behind the XAML for the view:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace MVVM_Example.Views
{
    /// <summary>
    /// Interaction logic for MainView.xaml
    /// </summary>
    public partial class MainView : Window
    {
        public MainView()
        {
            InitializeComponent();
        }

        private void SetFocusToUserInput(object sender, RoutedEventArgs e)
        {
            this.UserInput.Focus();
        }
    }
}
```

Most of those using statements are unnecessary, the project type we used created this for us, and so we'll just leave it as-is for now and accept the overkill. What we did add was a tiny bit of code to set the focus to the text box on the view. Note we referenced the control by name, so we also have to go name it. To do that, make the following addition to our view's XAML:

```
<TextBox Name="UserInput" Text="{Binding NewUserData,
UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}" Height="23"
HorizontalAlignment="Left" Margin="5,5,0,0" VerticalAlignment="Top"
Width="120" />
```

Now we just need to tell the application when to set the focus to the test box. The code in our code behind can be executed from the button on the click event. We want the focus to return to the text box each time the user clicks the Add button, so add the following to the XAML:

```
<Button Click="SetFocusToUserInput" Command="{Binding AddCommand}"
IsEnabled="{Binding AddButtonEnabled, UpdateSourceTrigger=PropertyChanged,
Mode=TwoWay}" Margin="5,5,0,0" Height="23" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="75" IsDefault="True">Add</Button>
```

Now whenever the Add button is clicked, the view's SetFocusToUserInput function will get called.

Press F5 and check out your completed application. Play with it, love it and show it to the uninitiated, then pass this tutorial on to them.

Homework

One last task and I'm not going to help. You know what you need to know, or at least enough to hit Google and get the rest. You need to get the "Save" and "Load" buttons to work. When the user clicks the Save button, write the contents of the collection somewhere to persist them. Now clear the collection and read the data back into the collection when the user clicks the Load button. At that point, if the application had any value at all, you'd have a finished project.

Now you're ready to write a real WPF application using C# and the M-V-VM pattern. Best wishes friend!

That's All Folks

I hope you learned something from this tutorial and I hope it showed you how clean and simple the M-V-VM pattern can make working with XAML using C#. Try to avoid old habits learned from WinForms and ASP.NET and instead embrace the separation that M-V-VM can offer.

Basic M-V-VM rules:

- If you're naming a lot of your controls, you thinking WinForms or ASP.NET, stop that!
- If you find yourself adding any logic to the view's code behind that isn't a truly pure interface concern, then move the code to the view model.
- Never let the view know about anything but the view model. The view should be fully independent and it should see only itself and the view model, nothing else… well except the user of course.
- Never let the view model know anything about the view, just expose properties and methods and let the view do the rest. If you're referencing ANYTHING from the view, stop immediately and rework your design.
- If you can't get your bindings to work, there is a debugging aid. Add "PresentationTraceSources.TraceLevel=High" to the binding statement. Here is an example:
  ```
  <Button Command="{Binding AddCommand,
  PresentationTraceSources.TraceLevel=High}" >Add</Button>
  ```
- Never let a pattern dictate anything that doesn't make sense. A pattern is a guide, not a task master. Use logic, wisdom and experience to tell you when it makes sense to break the pattern. After all, you're the still the programmer!
- ENJOY!