1. Introduction

MVVM seems, to me, to be the best way to develop applications that are both flexible - allowing changes to the GUI without complex rewrites, and allowing testing of the client logic without needing to resort to complex macros.

In this series of articles, I will present a small application in WPF, using an enhanced MVV pattern that I'm calling MVVM#.

So, what's different about my implementation? The main things are:

* Message Tracking: When a message is sent, we now know if it has been handled.
* Cancellable messages: A ViewModel can stop the message being sent further down the chain.
* Easy Peasy Modal Dialog windows
* No 'Main window': Everything is a View.
* Use of a Controller: The Controller controls the application.
* Easy to use Design Time Data for 'Blendability'
* Use of ViewData for binding data to a View, as well as a ViewModel to bind behaviour to a view.

In this article, I introduce the topics, and ideas, and explain some of the differences.

In the [second article](https://www.codeproject.com/KB/WPF/MVVMEpisode2.aspx), I'll show the way I set up any project using MVVM#, creating the base classes ready to begin the application specific development.

In the [third article](https://www.codeproject.com/KB/WPF/MVVMEpisode3.aspx), I'll add sufficient meat to the bones created in article 2 to give us a running application, albeit one that doesn't do too much.

In the [fourth article](https://www.codeproject.com/KB/WPF/MVVMEpisode4.aspx), I'll finish off the application to show a (small) but functioning application demonstrating some of the functions available.

1. Background

I've been looking at MVVM for a while now, without the opportunity to develop real applications using the pattern. I've downloaded and looked at most of the various frameworks, and even understood a few! But I don't like using a framework when I am learning something new - I want to understand the subject from the ground up. So I started playing, developing my own application using MVVM, and re-engineering to surmount the many obstacles I encountered.

In this article, I will describe some of the thought processes that drove me to extend the MVVM pattern, and introduce that pattern.

In the next article in this series, I will walk through developing an application from scratch, in VS2010 using C#, to implement this pattern.

1. Points of Interest

Man! It's a minefield out there!

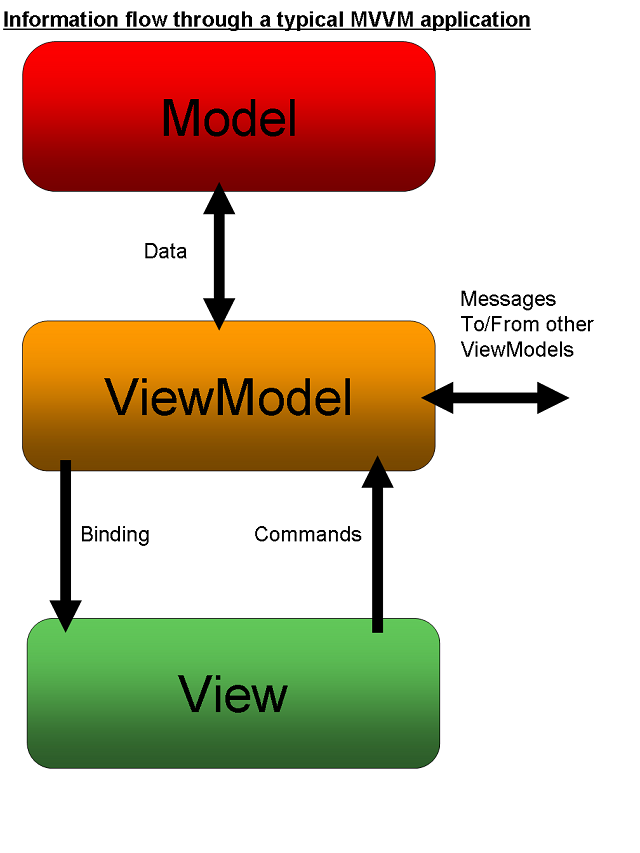
There are so many articles, written by the extremely knowledgeable through to the novices, and it is really hard to sort the wheat from the chaff. Debts are due to the WPF Disciples in general, Josh and Marlon in particular, and especially Pete Hanlon who has responded to my often cryptic questions on CodeProject with never a groan! Thanks also go to any of the many people out there in etherland who have helped me, knowingly or not, to develop this series. Isn't it *amazing* how much free help there is out there? Hopefully, this series will help me give a little back.

1. History

* Initial draft: March 2011

1. MVVM

It's sensible, I guess, to start off with a brief introduction to what *I* understand to be MVVM.  
**M**odel **V**iew **V**iew**M**odel



1. Model

The word 'Model' (according to dictionary.com) means "*a representation, generally in miniature, to show the construction or appearance of something.*"

In our case, a Model is a Class that describes something In The Real World. In a business system, it may be something like Customer or Supplier, in a game it could be a spaceship or a monster (yes, I know, monsters aren't In Real Life - but you get the idea!). The important thing is that, as far as MVVM is concerned, there's nothing *special* about these classes. They may have all sorts of functionality, the ability to save themselves to a database, be decorated with attributes or be simple, vanilla classes with a couple of properties.

These are the objects which model the data that are being used by the system.

The Model classes may already exist, you may define them based upon an existing database, or you may be defining them from scratch to reflect the needs to the application you're developing.

1. ViewModel

A ViewModel is the class that defines some interactive visual element's data and functionality. It is the model of a View.

It is important to note that the ViewModel *does not* describe how the view *looks*. It describes how the view *functions,*and what information it provides to the user.

Arguments abound about just exactly how much a ViewModel should describe the visual aspects of a View; for example, is it up to the View to decide on the wording of a label, or should the ViewModel be involved? In my opinion, this is entirely up to you and the particular project you're working on. Sometimes your labels will come from the VM (because, for example, you're localising it), other times the designers may want more control over descriptive text. In my examples, I'm assuming that, essentially, if it's not on the database, then it's up to the designer to provide it - this is a single-language application.

So your project sponsor (aka The Boss) tells you "We want to be able to display a Customer's details, and allow the user to modify them."

That's the ViewModel specification right there! He hasn't said whether the name is in Tahoma Bold, or whether the State selection is a text box or a combo - he's just defined the functional and data requirements for the CustomerEditViewModel.

I should point out, here, that there are various schools of thoughts about this. Some proponents of MVVM have ViewModels that model the visual aspects of the View quite closely - for example, they may have properties like "System.WIndows.Visibility ShowTransactions;" in the ViewModel rather than using the Transactions.Count property with a converter to convert 0 to Visibility.Hidden and any other number to Visibility.Visible. If you're interested in how the WPF Disciples think about this (or did so, in 2008), then [see here](http://groups.google.com/group/wpf-disciples/browse_thread/thread/3fe270cd107f184f/2bddefb68dc7a283) (requires login to Google Groups).

The Data is a Customer. The requirements are to be able to view and modify the Customer's properties.

Importantly, however, we don't want our View to 'know about' our Model (in this case, the Customer class) directly.

Why not? Well, what if your customer class changes? You might rewrite the back end completely using Entity Framework or nHibernate, through whim or necessity. And your Customer class shouldn't need to be in any way aware of the GUI side of your application.

In this respect, you can think of the ViewModel as being the translator - it takes a Customer object and handles the mapping of this object in a View-Friendly way.

1. View

A view is the visual element that the user of this application will see. With WPF, this will be a UserControl. Importantly, though, while a View is a UserControl, a UserControl is not necessarily a View.

Remember that it confuses some people. If you create a UserControl, it *may* be a View, but it may just be a UserControl. All View classes will inherit (indirectly) from UserControl, but we can still use 'virginal' UserControls in our application.

*But how to I know?* I hear you ask. Easy. You should have started by defining your ViewModel - so if you have a ViewModel for the functionality required, then you need to create a View - if this is some visual functionality which is a part of the functionality of a ViewModel, but doesn't *have* a ViewModel, then it's a plain old vanilla UserControl.

For example, in the CustomerEditViewModel described above, the customer's address will be displayed and modified by the user. I may decide to encompass the display and modification into a UserControl, but this UserControl will use the CustomerEditViewModel as its source of data - NOT a CustomerAddressViewModel, or even an AddressViewModel.

There's a fallacy (IMHO) in MVVM circles that the developer should refrain from writing any code in the View's code behind file (the *view.xaml.cs* file). The truth of the matter is that the developer should refrain from writing any code in the View's code behind file***that doesn't pertain purely to the GUI***.

Let's face it, code is generated by the XAML - so why *shouldn't* we write our own? The rule that says "**Don't**" is a good *reminder* to developers not to put business logic in the View - but if you want to do something at the GUI side and it's convenient to write it in C#, then write it in C# for goodness sake!

For example - you might bind a Button's Command property to an ICommand in the View's ViewModel - that makes sense, requires no code-behind and is neat and tidy. But what if I want to do something on MouseOver, for example? Sure, I can play around with behaviours, or something equally verbose. But why not write code in my code-behind to handle the MouseOver event by invoking the Command on my ViewModel? It's just writing it in C# rather than XAML.

Two or three lines of C# code in the code behind instead of forty lines of XAML or a couple of additional classes? Maintainability!

1. How It All Hangs Together

The idea of MVVM is that we model our views, and keep the actual View separated from the ViewModel. In principal, this allows us to change the GUI by solely changing the View - the functionality provided by the ViewModel will still be used, but the user gets a different experience. Obviously, the View needs to know about the ViewModel - as it will be binding to its properties, and sending it commands. The ViewModel, however, should know (almost) nothing about the View.

You can now give your View to a designer to play with. They can change the view as much as they want to, so long as its data source is the ViewModel and it uses the functionality of the ViewModel appropriately.

Because we're talking WPF here, what we do is to use the ViewModel as the DataContext of the View. Each of the elements of the View are bound to properties on the ViewModel. The ViewModel takes the Model data (our Customer) and maps each of the properties required to be modified from the Model to its own ObservableProperties.

Note that - they're **Observable** properties. That is, the properties of our ViewModel that are going to be bound-to by our View need to be Observable- by being part of an object that implements INotifyPropertyChanged. All this means is that, when the value of a property is changed, we raise an event using the name of the property - WPF's binding then handles updating anything that is bound to this property.

1. How It All Falls Apart

Well, perhaps "Falls Apart" is a bit strong!

There are a number of shortfalls in the way I've seen much MVVM done. In playing around, I've come up with my 'ideal' model of how I think it can work well. You may disagree with my thought processes - or you may have different solutions to the same problems. If nothing else, I hope this series of articles will provoke some thought.

1. Controlling the Application

When an application starts, it needs some piece of code, somewhere to start it up, show an initial window, etc. In most WPF applications, there's an initial WPF window that gets created - and most MVVM applications treat this as a View and create an appropriate ViewModel, possibly called MainWindowViewModel.

I disagree with this approach. I don't like the concept of one View being 'special' somehow. I might start of with the idea of showing a list of Customers from which to select - but the designers may decide that actually they want to see a Customer Edit form in Add mode first.

*My* initial class, therefore, is a singleton that I call Controller. It is a non-visual class which is responsible for Controlling the application. For large applications, there may be multiple Controllers, with some base functionality. Note, thought, that I don't hold with the idea of there being one Controller per ViewModel.

This Controller is responsible for:

* Instantiating Views, with their associated ViewModels
* Channelling requests for data
* Handling updates of data
* Handling any logic flow associated with the presentation (e.g. opening particular Views when certain events take place)

So, when the application starts, the singleton Controller is instantiated. It passes a reference to itself to every ViewModel, allowing the ViewModel to use the functionality provided by the Controller.

The Controller creates an initial View and its ViewModel, and shows that View in a Window.

The View it creates *may* have child Views, which are positioned at design time - in which case the ViewModelwill be responsible for creating the appropriate child ViewModel(s).

Now, the Controller merely handles requests from one or more of the ViewModels to "do stuff".

Incidentally, the controller can either be designed to 'push' data to the ViewModels, or the ViewModels can request data from the Controller. It is sometimes a matter of personal preference more than anything. In my case, I tend to use both options depending upon the circumstance. It just depends whether you think in terms of

**C**: "Hey! CustomerEditViewModel! Allow the user to edit this customer, will you?"   
or   
**C**: "Hey! CustomerEditViewModel, a user wants to use you to edit a Customer."  
**VM**: "OK! Which One, Controller?."  
**C**: "This one, please!."

Taking our Customer maintenance example a bit further. The specification now is; *"Present a list of Customers to the user, who can select one to modify. When they've selected one, they can change the details and save them."*

The Controller's job is to:

* Instantiate a CustomerSelectionViewModel
* Give the CustomerSelectionViewModel a collection of Customers to deal with
* Instantiate the CustomerSelectionView and sets its DataContext to the CustomerSelctionViewModel
* Show the CustomerSelectionView
* Wait...

Now, the User selects a customer in the View. This sends a Command to the ViewModel. The ViewModel tells the Controller that a Customer has been selected (and, of course, tells the Controller which customer it is). The Controller now:

* Gets the Data for this particular Customer (if it doesn't already have all the data)
* Instantiates the CustomerEditViewModel, and gives it the Customer Data
* Instantiates the CustomerEditView and sets its DataContext to the CustomerEditViewModel
* Shows the CustomerEditView
* Wait...

Now, the user changes data in the View and clicks the save button (or, depending on our designer's whim, performs some other action) which sends a 'Save' command to the ViewModel. The ViewModel asks the Controller to save the Customer. The Controller now:

* Saves the data
* Sends out a Message that the Customer has been saved

1. Data vs Function

The ViewModel in most MVVM examples I've seen has both functionality (handling Commands, retrieving and updating data) and Data (Observable properties to be bound to). Each class, ideally, should have only a single function and these ViewModels have two.

My solution is to introduce a new class - the ViewData.

A ViewModel will have a property of type ViewData. This ViewData is the thing that now holds all of the Observable Properties bound to the View. (I say 'All' here but I really mean 'most'. The ViewData object will contain ObservableProperties for every piece of data to be bound to the View *that originates in the Data being used*. There ViewModel may still have additional ObservableProperties, bound to by the View, that are required for *functionality*.

In the project that accompanies this series of articles, for example, in the CustomerSelectionViewModel, there is an ObservableProperty called StateFilter which is used to allow the user to filter the list by State.

Want a 'read only' view of a customer as well as a Customer Edit? Same ViewData, different ViewModel.

The ViewModel is now much more of a model of a view - it's describing our specification quite well, don't you think? *"Present a list of Customers to the user, who can select one to view or modify. When they've selected one, they can change the details and save them."* The List of Customers is our ViewData, the ViewModel handles getting the list and the user selecting an item from the list. When we edit a Customer, the CustomerEditViewData has properties for all of the editable fields for a customer, while the CustomerEditViewModel handles the process (when the user clicks Save, for example).

Incidentally, this helps to answer one of the questions alluded to earlier - when should a ViewModel present Properties to a View that are GUI oriented rather than Data oriented? Now, we have a way of separating the two concepts. The ViewData object should be *purely* data-centric. If we want to present data to the View that is View-Centric, but we don't want to mess about with Convertors, for example, then the ViewModel can have properties bound to by the View. An example:

We have CustomerViewData which maps the Customer Model properties to Observable properties for the view to bind to - such as CustomerFirstName, CustomerSurname, CustomerAddressLine1, etc.

We have a CustomerDisplayViewModel that 'contains' a CustomerViewData instance.

All is good, but the designer asks if we can't format the Address nicely - the designer can't handle (for example) having a blank second line of address easily - even using Converters.

Now's our opportunity to add an observable property to our ViewModel - string FormattedAddress. Our designer can bind directly to this property, and the separation of this from the ViewData class makes it obvious what we're looking at.

1. Windows

In my view (ho ho!), a View is a discrete piece of visual functionality. It might be in a Window all by itself; it might be in a Window, docked to a part of that Window; it might be in a modal dialog; it might be loaded at run time; it might be positioned at design time. So, I might design a View for selecting a Customer from a list of Customers, and design it as a Window. My ViewModel gets the collection of Customers to view, and handles the command invoked when the User selects a customer for editing.   
All good.

Now, the boss comes along and says "Nah! We want that selection thingo to be on the same window as the Edit whatsit. You know, a bit like that Visual Studio thingo you showed me, with a tree of files down the right that you can double-click on and edit." Of course, you change the View to no longer be a window but a UserControl - a simple change. Oh - and you now have to put that UserControl on a Window - so that new window better be a new View with a ViewModel.

Of course, when the Boss sees it, he changes his mind and you have to change it all back.

**Q.** Why should there be a difference in a View depending on whether it is a UserControl or a Window?

**A.** There*shouldn't* be a difference.

I want to design EVERY view as a UserControl with its ViewModel. And I want my Controller to determine whether any particular view at any particular time needs to be shown in a Modal Window, a non-Modal Window or as a UserControl on some container (which, granted, ultimately will be contained in a window).

So - I design my Selection as a UserControl not a Window. My Controller shows it in a Window by itself. When the Controller handles the 'Customer Selected' event, it instantiates the CustomerEditView and shows *that* in Window, all by itself.

When the boss wants them both in a window together, a new CustomerSelectAndEditView is created, the two existing Views placed upon its surface and the Controller changed to show the single CustomerSelectAndEditView in a Window. No change to either View or ViewModel required.

When he changes his mind back again, similarly, only the Controller needs to change.

Of course, I make it sound simple - but how *does* the Controller show a View in a Window?

My solution is to have a base View class from which all Views inherit, which has the methods necessary to show *itself* in an existing container, or in a new window, modally or not.

So my controller can say "Hey, View, show yourself in a new modal window, please!" and let the View worry about how it's going to do so.

1. Inter-ViewModel Communication

Obviously, there will be times when you have two ViewModels, and changes in one need to be reflected in the other. In our Customer Selection/Edit example, the Selection will need to be refreshed when data is saved by the Edit function.

Many of the MVVM models I have seen use a Messenger or Mediator singleton paradigm. This is quite nice (depending on the exact implementation) as it allows ViewModels to say "I want to be informed if this message is sent from some other ViewModel" and for ViewModels to simply send messages in 'Fire and Forget' mode.

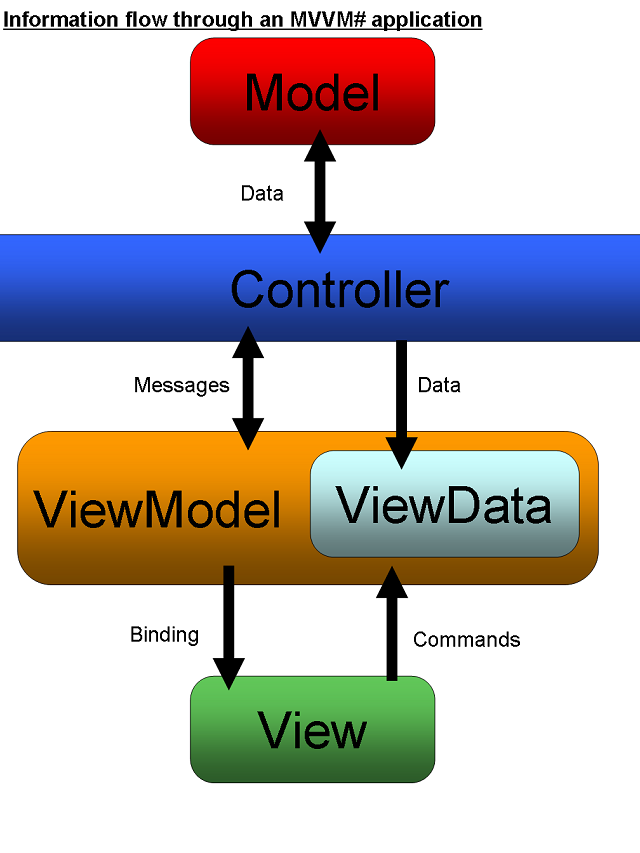
So our Customer Selection can ask to be informed every time the Customer Updated message is sent, and our CustomerEdit can send a CustomerUpdated message whenever it updates a Customer.

But I don't think the ViewModels have any business sending messages. When the ViewModel wants to perform a function, it asks the (singleton) Controller to perform the function on its behalf. The Controller can then handle sending messages. I find this allows better control of the messaging, as the Controller can more easily handle any logic surrounding the Message.

As an example, the CustomerEditViewModel knows only that the user has updated a Customer's data. It informs the Controller that 'Customer 123 has been updated with this data".

The Controller has a big advantage over the CustomerEditViewModel - it also has access to things like the collection of Customers currently visible in the CustomerSelectionViewModel (or, it *can* have that information if we want it to). The Controller could, therefore, decide not to send a CustomerUpdatedmessage at all - but just to refresh the list and inform the CustomerSelectionViewModel - or it might use logic to determine that the current selection does not *need* to be refreshed at all. So our Controller could send a CustomerUpdated message, or a CustomerListsNeedToBeRefreshed message, or no message at all, depending on our implementation requirements. The ViewModel simply doesn't have this option.

In principal, the Controller could also handle the receipt of messages, using methods on the ViewModels to exert its will, but this then relies on the Controller having more knowledge about a ViewModel than I'd like. So, in my world, ViewModels subscribe to messages.



1. Summary

In this article, I have explained what I understand by MVVM using WPF, and listed some of the problems with it that I encountered. I have then proposed some solutions to these problems. In the [next article](https://www.codeproject.com/KB/WPF/MVVMEpisode2.aspx), I will walk through developing the foundations of an application to demonstrate some of these features.