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!

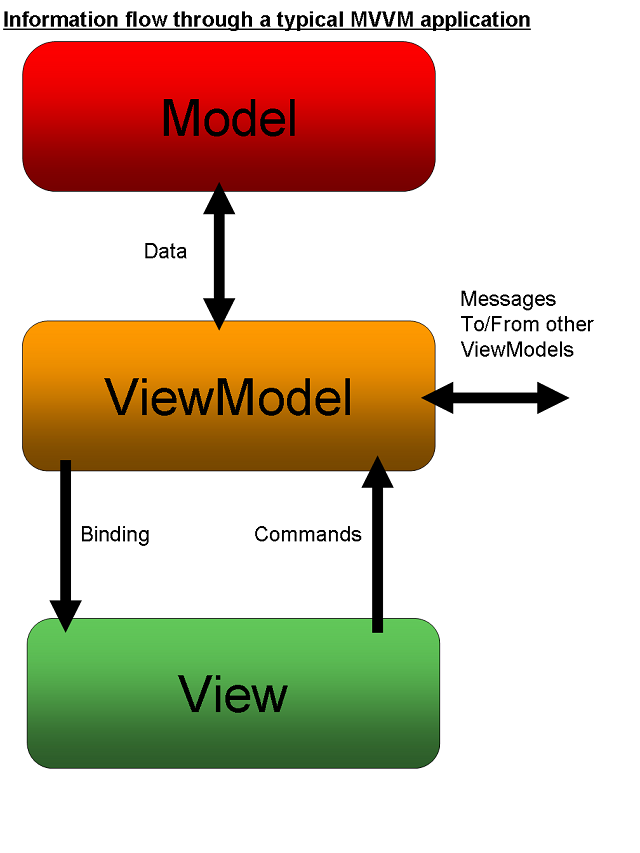
There are so many articles, written by the extremely knowledgeable through to the novices, and it is really hard to sort the wheat from the chaff. Debts are due to the WPF Disciples in general, Josh and Marlon in particular, and especially Pete Hanlon who has responded to my often cryptic questions on 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.

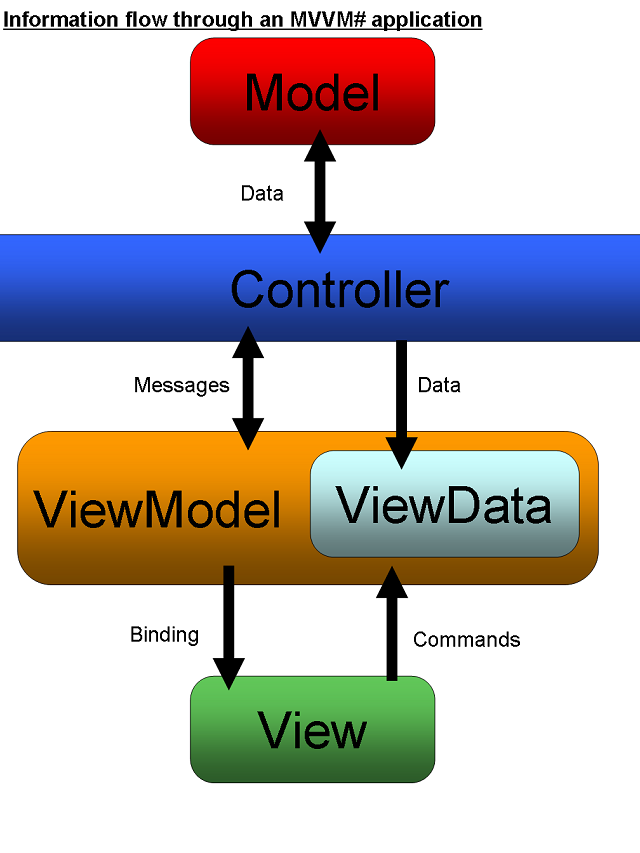
So our Customer Selection can ask to be informed every time the Customer Updated message is sent, and our CustomerEdit can send a CustomerUpdated message whenever it updates a Customer.

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.

# MVVM # Episode 2



[**\_Maxxx\_**](https://www.codeproject.com/script/Membership/View.aspx?mid=4693125), 3 Dec 2013 [CPOL](http://www.codeproject.com/info/cpol10.aspx)

|  |  |
| --- | --- |
| https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-fill-md.png  https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-empty-md.png | 4.96 (28 votes) |
|  | |  |  |  | | --- | --- | --- | | Rate: | [vote 1vote 2vote 3vote 4vote 5](https://www.codeproject.com/Articles/173630/MVVM-sharp-Episode-2) |  | |

Using an extended MVVM pattern for real world LOB applications: Part 2

https://www.codeproject.com/images/warning48.png**Is your email address OK?** You are signed up for our newsletters but your email address is either unconfirmed, or has not been reconfirmed in a long time. Please **click**[**here**](https://www.codeproject.com/script/Membership/SendConfirmRequest.aspx?rp=%2fArticles%2f173630%2fMVVM-sharp-Episode-2)**to have a confirmation email sent** so we can confirm your email address and start sending you newsletters again. Alternatively, you can [**update your subscriptions**](https://www.codeproject.com/script/Membership/Subscribe.aspx?rp=%2fArticles%2f173630%2fMVVM-sharp-Episode-2).

* [**Download source code - 66.19 KB**](https://www.codeproject.com/KB/WPF/MVVMEpisode2/MVVMSharp_Code.zip)

##### Other Articles In This Series:

##### [Part One](http://www.codeproject.com/Articles/173618/MVVM-sharp-Episode-1) [Part Two](http://www.codeproject.com/Articles/173630/MVVM-sharp-Episode-2) [Part Three](http://www.codeproject.com/Articles/173661/MVVM-sharp-Episode-3) [Part Four](http://www.codeproject.com/Articles/173673/MVVM-sharp-Episode-4)

## Introduction

In the [first article in this series](https://www.codeproject.com/KB/WPF/MVVMEpisode1.aspx), I looked at some of the issues I was having with the implementation of MVVM in a WPF application, and suggested some improvements to existing frameworks.

In this article, I will begin a walkthrough creating an MVVM application (albeit a small one) using the techniques I touched on last time.

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.

We'll just create the generic pre-requisites here, the 'framework' if you like. Next time, we'll continue with the actual project.

Just because one has to give these things a name, I thought I'd call this MVVM#. mainly because MVMVDCV looked too much like a roman date  " src="http://www.codeproject.com/script/Forums/Images/smiley\_wink.gif" />

### Pre-requisites and Caveats

I'm building this using VS2010 and C#, .NET 4.0. It should be translatable to VB.NET.

I'm NOT using TDD here. Although one of the advantages of MVVM is the enhanced testability of your application, I'm steering clear of both to keep the article short, and to avoid revealing my lack of knowledge on the subject of TDD!

I'm assuming familiarity with VS2010 and C# - so no 'Click this, drop down that" however, I do try not to make assumptions, so even novice users should be able to follow along - and the source of the completed application is available for download.

## Specification

I wanted this to be fairly realistic within the constraints of an article - so here's the spec:

* We want the user to be able to see a list of Customers.
* They need to be able to filter the list by State.
* They need to select a Customer and edit their details.
* They need to be able to save their changes.

Well, that's the sort of spec I'm sure many of you are used to! Obviously, we'll have to make lots of assumptions, and design decisions ourselves - but that's all good; we're going to be agile and show the users where we are, frequently, and re-engineer as necessary.

## Let's Get Started!

Create a new WPF C# application. I'm calling mine "CustomerMaintenance". We don't want the default MainWindow - so delete it.

Now, we'll create all of the projects we're going to need. Because each of the things we're dealing with should be independent of one another, I like to have each in its own project - that way cross-contamination is harder, easily documented by the 'references' in each project, and I can split development amongst multiple developers more easily if I need to. So, create new Class Library projects as below:

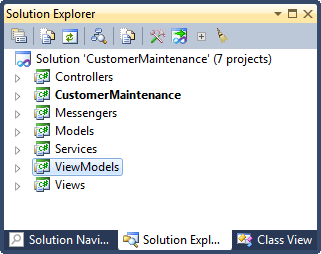
* Controllers
* Messengers
* Models
* Services
* ViewModels

We also want to create a project for our Views. This should not be a Class Library, but a WPF User Control Library.

You can either delete the default classes created, or just rename them when we get around to creating them.

Your VS Solution explorer should look like this.

#### Solution Explorer



### Foundation Classes

We're going to be using messaging to communicate around our application. We could use events (so long as we're very careful to remove handlers when classes are disposed). In this implementation, I'm using a Messengerclass derived from the [MVVM Foundation](http://mvvmfoundation.codeplex.com/SourceControl/changeset/view/32506#531061) classes. The Messenger class source code is part of the download for this article. It's a utility class that would be used in all my MVVM projects as-is.

The Messenger class is in its own project (Messengers) so I can keep it independent.

There's a real problem with the implementation of some Messenger systems that can cause unexpected behaviour, and be really difficult to track down. to describe it, imagine this scenario:

ViewModels VMA and VMB are both instantiated. VMA sends messages to which VMB subscribe, but the two know nothing about one another. When VMB handles the message, it performs some function (writes to the database, for example).

VMA sends the message, VMB receives it, writes to the DB and all is good. Now the user closed VMB (if it's in a window, maybe they close the window) so the system removes all 'Strong' references to VMB. There is still a weak reference to it - in the Messenger - but that won't stop it being Garbage Collected, so all is good. Of course, because VMA knows nothing about what's happening with VMB it continues to send the message - after all, other ViewModels may subscribe to the message too...

Now, if you recall Garbage Collection 101, the Garbage Collector doesn't necessarily remove items from memory immediately they are free of all hard references - the Garbage Collector runs when it goddamn feels like it. so if you have plenty of memory available, especially if VMB is small, it's still sitting there. Then VMA sends the message. You can see where this is going, can't you? Yep - VMB STILL RECEIVES THE MESSAGE even though there's no references to it! And it still writes merrily away to the database! Even though the user closed it!

The answer to this is for VMB to unsubscribe from the message before it is closed. But wait, wasn't part of the reason for using weak references that we wouldn't have to worry about the memory leaks associated with forgetting to remove event handlers? (yes it was!) But now we still have to remember to remove message handlers or we risk unexpected behaviour which may actually be harder to track down than memory leaks!

What I have implemented to try to overcome this issue, is a Deregister method, that takes a ViewModel as a parameter, and removes all Message subscriptions to it. I then call this method from my Base ViewModel class in a CloseViewModel method.

I've also added two enumerations to the source, MessageHandledStatus and NotificationResult.

The first, MessageHandledStatus, was added to allow the message handlers (our View Models) to communicate back to the Messenger system.

The default value (NotHandled) tells the system that the ViewModel hasn't handled the message (so the Messenger should keep sending it to any other ViewModel registered as a recipient).

If a ViewModel sets the value to HandledContinue, this tells the Messenger to continue sending out the message, but on completion it will know that something has handled the message.

The HandledCompleted value tells the Messenger not to send the message out to any further recipients as it has been handled

Finally, the NotHandledAbort message tells the Messenger that although the message has not been handled, it should not send it to further recipients.

The NotificationResult enumeration is used to return a some information to the ViewModel that sends the message. This can be used, for example, to instantiate a new ViewModel to handle some event if there are currently no handlers registered to handle it.

My version of the messenger class also uses a class, Message, which is carried around with the message. The class looks like this...

#### Message.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

namespace Messengers

{

public class Message

{

#region Public Properties

/// *<summary>*

/// *Has the message been handled*

/// *</summary>*

public MessageHandledStatus HandledStatus

{

get;

set;

}

/// *<summary>*

/// *What type of message is this*

/// *</summary>*

private MessageTypes messageType;

public MessageTypes MessageType

{

get

{

return messageType;

}

}

/// *<summary>*

/// *The payload for the message*

/// *</summary>*

public object Payload

{

get;

set;

}

#endregion

#region Constructor

public Message(MessageTypes messageType)

{

this.messageType = messageType;

}

#endregion

}

}

This Message object is passed around with every message - so every message handler has the opportunity to look at, or modify, the HandledStatus and look at the MessageType. This, for example, allows a single message handler to cater for many different message types, and to set the HandledStatus appropriately.

The Message object also contains a 'Payload'. This is some object you want passed around with that message, so when we have saved a Customer, for example, we send a message using:

Hide   Copy Code

Messenger.NotifyColleagues(MessageTypes.MSG\_CUSTOMER\_SAVED, data);

where the data passed is the CustomerEditViewData, which contains all the information just updated - so if some ViewModel somewhere wants to act, it already has the information at its fingertips, so to speak.

We're also going to be using two other classes from the MVVMFoundation project - namely ObservableObject and RelayCommand. Both of these I create in the ViewModels project in a folder called BaseClasses, as all ViewModel and ViewData classes derive from ObservableObject, and ViewModelsare the RelayCommand handlers.

You will also need to add a reference in the ViewModels project, to PresentationCore.

To complete this section, we should add our enumeration for the messages. So add a new file to the Messengers project, called Messages...

#### MessageTypes.cs

Hide   Copy Code

namespace Messengers

{

/// *<summary>*

/// *Use an enumeration for the messages to ensure consistency.*

///

/// *</summary>*

public enum MessageTypes

{

MSG\_CUSTOMER\_SELECTED\_FOR\_EDIT,*// Sent when a Customer is selected for editing*

MSG\_CUSTOMER\_SAVED *// Sent when a Customer is updated to the repository*

};

}

These are the only two messages our application is going to handle - so adding them now is no problem. In a larger application, we'd be adding new messages as the functionality is specified - it's a good place to go to ensure the functionality provided matches the requirements.

This, if you like, is the basic 'framework' for my MVVM# application. You can, of course, use any implementation you like of the Mediator pattern (our Messenger class). The ObservableObject and RelayCommand classes can also be replaced with some other version providing similar functionality.

### ViewModel

The other bit-players in our scenario can now get created too.

We're using a Controller to manage the application - so let's create its interface. Again, this can go in the BaseClasses folder in the ViewModels project. You can see that the interface for the base controller just specifies that it has a Messenger property.

#### IController.cs

Hide   Copy Code

using Messengers;

namespace ViewModel

{

public interface IController

{

Messenger Messenger

{

get;

}

}

}

Now, we also need base classes for our ViewData and ViewModel classes. Here, some controversy creeps into my implementation. We're going to declare an IView interface for use in our ViewModel.

What!!! I can hear the gasps! Our ViewModels shouldn't know anything about our Views! Well, I live in the real world. I need to be able to tell the Views to activate themselves and to close themselves. More accurately, I need to be able to tell a View when its ViewModel is closing, or when its ViewModel is activating - so giving the View the option to handle these events.

You'll see that is all the IView interface is - definitions of a couple of methods to allow the views to hook into events raised by the ViewModel. It additionally specifies the DataContext property - which every view, as a descendent of UserControl, will have anyway. You'll see this property used in the ViewModel constructor.

#### IView.cs

Hide   Copy Code

namespace ViewModel

{

public interface IView

{

void ViewModelClosingHandler(bool? dialogResult);

void ViewModelActivatingHandler();

object DataContext{get;set;}

}

}

The BaseViewdata is about as simple a class as you could want...

#### BaseViewData.cs

Hide   Copy Code

namespace ViewModels

{

/// *<summary>*

/// *The base class from which all View Data objects inherit.*

/// *Just an Observable Object right now -*

/// *but a separate abstract class in case we want to add*

/// *to it while not modifying ObservableObject itself.*

/// *</summary>*

public abstract class BaseViewData : ObservableObject

{

}

}

The BaseViewModel source also declares the two delegates that use the methods defined in the IViewinterface. We also keep, in the BaseViewModel, a collection of Child BaseViewModels. Keeping this list allows each ViewModel to ensure that each of its children unsubscribe from all of their messages (and release any other resources) when the 'parent' is being 'closed'. Incidentally, rather than use the name "Parent" for variables I have used "daddy" - to avoid any possible confusion with other uses of the Parent name. Those of a feminist bent, feel free to rename this 'mummy'.

The BaseViewModel contains a BaseViewData property. The BaseViewData is business data bound to the View, while any other properties of the ViewModel that may be bound by the View provide functionalityrather than just data.

In the constructor, a ViewModel is passed an IController and IView reference. This is logical - every ViewModel is going to require a Controller to service it, and a View to provide a GUI. You can see that the constructor is where the methods defined in the IView are wired up to the Event Handlers defined in the BaseViewModel - and you can take note that the ViewModel does not retain any other reference to the View.

Finally two methods, CloseViewModel and ActivateViewModel, are provided.

#### BaseViewModel.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System.Collections.Generic;

namespace ViewModel

{

/// *<summary>*

/// *When the VM is closed, the associated V needs to close too*

/// *</summary>*

/// *<param name="sender"></param>*

public delegate void ViewModelClosingEventHandler(bool? dialogResult);

/// *<summary>*

/// *When a pre-existing VM is activated the View needs to activate itself*

/// *</summary>*

public delegate void ViewModelActivatingEventHandler();

/// *<summary>*

/// *A base class for all view models*

/// *</summary>*

public abstract class BaseViewModel : ObservableObject

{

public event ViewModelClosingEventHandler ViewModelClosing;

public event ViewModelActivatingEventHandler ViewModelActivating;

/// *<summary>*

/// *Keep a list of any children ViewModels so we can safely*

/// *remove them when this ViewModel gets closed*

/// *</summary>*

private List<BaseViewModel> childViewModels = new List<BaseViewModel>();

public List<BaseViewModel> ChildViewModels

{

get { return childViewModels; }

}

#region Bindable Properties

#region ViewData

private BaseViewData viewData;

public BaseViewData ViewData

{

get

{

return viewData;

}

set

{

if (value != viewData)

{

viewData = value;

base.RaisePropertyChanged("ViewData");

}

}

}

#endregion

#endregion

#region Controller

/// *<summary>*

/// *If the ViewModel wants to do anything, it needs a controller*

/// *</summary>*

protected IController Controller

{

get;

set;

}

#endregion

#region Constructor

/// *<summary>*

/// *Parameterless Constructor required for support of DesignTime*

/// *versions of View Models*

/// *</summary>*

public BaseViewModel()

{

}

/// *<summary>*

/// *A view model needs a controller reference*

/// *</summary>*

/// *<param name="controller"></param>*

public BaseViewModel(IController controller)

{

Controller = controller;

}

/// *<summary>*

/// *Create the View Model with a Controller and a FrameworkElement (View) injected.*

/// *Note that we do not keep a reference to the View -*

/// *just set its data context and*

/// *subscribe it to our Activating and Closing events...*

/// *Of course, this means there are references -*

/// *that must be removed when the view closes,*

/// *which is handled in the BaseView*

/// *</summary>*

/// *<param name="controller"></param>*

/// *<param name="view"></param>*

*//public BaseViewModel(IController controller, FrameworkElement view)*

public BaseViewModel(IController controller, IView view)

: this(controller)

{

if (view != null)

{

view.DataContext = this;

ViewModelClosing += view.ViewModelClosingHandler;

ViewModelActivating += view.ViewModelActivatingHandler;

}

}

#endregion

#region public methods

/// *<summary>*

/// *De-Register the VM from the Messenger to avoid non-garbage*

/// *collected VMs receiving messages*

/// *Tell the View (via the ViewModelClosing event) that we are closing.*

/// *</summary>*

public void CloseViewModel(bool? dialogResult)

{

Controller.Messenger.DeRegister(this);

if (ViewModelClosing != null)

{

ViewModelClosing(dialogResult);

}

foreach (var childViewModel in childViewModels)

{

childViewModel.CloseViewModel(dialogResult);

}

}

public void ActivateViewModel()

{

if (ViewModelActivating != null)

{

ViewModelActivating();

}

}

#endregion

}

}

### Controller

Because our Controllers will have some common functionality, we'll use a BaseController class from which our controller(s) will inherit. In this case, the only common functionality, in fact, is a reference to the singleton instance of our Messenger class, as defined in the IController interface.

So we just need to create a new class in the Controllers project, called BaseController. As usual, I create it in a folder called Base Classes.

#### BaseController.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using Messengers;

using ViewModel;

namespace Controllers

{

/// *<summary>*

/// *The base controller class.*

/// *</summary>*

public abstract class BaseController : IController

{

/// *<summary>*

/// *Retain a reference to the single instance of the*

/// *Messenger class for convenience*

/// *as it means we can use Controller.Messenger.blah rather than*

/// *Controller.Messenger.Instance.blah*

/// *In a large system this also allows us to use multiple Messengers*

/// *(e.g. for different parts of a system*

/// *that have no need to communicate between them)*

/// *by making a single change here to return a different Messenger*

/// *</summary>*

public Messenger Messenger

{

get

{

return Messenger.Instance;

}

}

}

}

### View

In our Views project, we need to create two items. Firstly, we're going to create a Window. This window will be used by our Views when we want to show them in a window - as you'll see later.

Because we're creating base classes (well, the Window isn't actually a base class, but it sort of fits the idea) I create a folder called Base Classes, then create within it a new Window called ViewWindow.

Because when we display our views, we need to put them on some surface, we'll add a DockPanel to the window. This is the surface on which all of our views will be placed when shown in a window. It must be named WindowDockPanel. Note that you can 'pretty up' your window as much as you like - just so long as it has a WindowDockPanel. (and you can change that functionality if you want, by changing the code that puts Views onto the window - it's all defined in Base classes, so the implementation can be changed to suit your preferences.)

Next we create the BaseView class.

#### BaseView.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Windows;

using System.Windows.Controls;

using ViewModels;

namespace Views

{

/// *<summary>*

/// *A delegate to allow the window closed event to be handled (if required)*

/// *</summary>*

/// *<param name="o"></param>*

/// *<param name="e"></param>*

public delegate

void OnWindowClose(

Object sender, EventArgs e);

/// *<summary>*

/// *This is the basis of all views.*

/// *It cannot be Abstract because of design time issues when*

/// *it tries to instantiate this class.*

/// *Note that this 'view' doesn't have any XAML*

/// *(because you can't inherit XAML)*

/// *</summary>*

public partial class BaseView : UserControl, IDisposable, IView

{

private ViewWindow viewWindow;

*// If shown on a window, the window in question*

private OnWindowClose onWindowClosed = null;

#region Closing

/// *<summary>*

/// *The view is closing, so clean up references*

/// *</summary>*

public void ViewClosed()

{

*// In order to handle the case where the*

*// user closes the window*

*// (rather than us controlling the close via a ViewModel)*

*// we need to check that the DataContext is not null*

*// (which would mean this ViewClosed has already been done)*

if (DataContext != null)

{

((BaseViewModel)DataContext).ViewModelClosing -=

ViewModelClosingHandler;

((BaseViewModel)DataContext).ViewModelActivating -=

ViewModelActivatingHandler;

this.DataContext = null; *// Make sure we don't*

*// have a reference to the VM any more.*

}

}

/// *<summary>*

/// *Handle the Window Closed event*

/// *</summary>*

/// *<param name="sender"></param>*

/// *<param name="e"></param>*

void ViewsWindow\_Closed(object sender, EventArgs e)

{

if (onWindowClosed != null)

{

onWindowClosed(sender, e);

}

((BaseViewModel)DataContext).CloseViewModel(false);

}

#endregion

#region IView implementations

/// *<summary>*

/// *Tell the View to close itself. Handle the case*

/// *where we're in a window and the window needs closing.*

/// *</summary>*

/// *<param name="dialogResult"></param>*

public void ViewModelClosingHandler(bool? dialogResult)

{

if (viewWindow == null)

{

System.Windows.Controls.Panel panel =

this.Parent as System.Windows.Controls.Panel;

if (panel != null)

{

panel.Children.Remove(this);

}

}

else

{

viewWindow.Closed -= ViewsWindow\_Closed;

if (viewWindow.IsDialogWindow)

{

*// If the window is a Dialog and is not*

*// active it must be in the process of*

*// being closed*

if (viewWindow.IsActive)

{

viewWindow.DialogResult =

dialogResult;

}

}

else

{

viewWindow.Close();

}

viewWindow = null;

}

*// Process the ViewClosed method to cater for if this has*

*// been instigated by the user closing a window,*

*// rather than by*

*// the close being instigated by a ViewModel*

ViewClosed();

}

public void ViewModelActivatingHandler()

{

if (viewWindow != null)

{

viewWindow.Activate();

}

}

#endregion

#region Constructor

public BaseView()

{

}

#endregion

#region Window

/// *<summary>*

/// *The Window on which the View is displayed*

/// *(if it is displayed on a Window)*

/// *The Window will be created by the View on demand*

/// *(if required) or may be*

/// *supplied by the application.*

/// *</summary>*

private ViewWindow ViewWindow

{

get

{

if (viewWindow == null)

{

viewWindow = new ViewWindow();

viewWindow.Closed += ViewsWindow\_Closed;

}

return viewWindow;

}

}

#endregion

#region Showing methods

/// *<summary>*

/// *Show this control in a window, sized to fit, with this title*

/// *</summary>*

/// *<param name="windowTitle"></param>*

public void ShowInWindow(bool modal, string windowTitle)

{

ShowInWindow(modal, windowTitle, 0, 0, Dock.Top, null);

}

/// *<summary>*

/// *Show this control in an existing window, by default docked top.*

/// *</summary>*

/// *<param name="window"></param>*

public void ShowInWindow(bool modal, ViewWindow window)

{

ShowInWindow(modal, window, window.Title,

window.Width, window.Height,

Dock.Top, null);

}

/// *<summary>*

/// *Maximum Flexibility of Window Definition version of Show In Window*

/// *</summary>*

/// *<param name="window">The Window in which to show this View</param>*

/// *<param name="windowTitle"> A Title for the Window</param>*

/// *<param name="windowWidth">The Width of the Window</param>*

/// *<param name="windowHeight">The Height of the Window </param>*

/// *<param name="dock">How should the View be Docked </param>*

/// *<param name="onWindowClosed">Event handler for when the window*

/// *is closed </param>*

public void ShowInWindow(

bool modal, ViewWindow window,

string windowTitle, double windowWidth,

double windowHeight,

Dock dock, OnWindowClose onWindowClose)

{

this.onWindowClosed = onWindowClose;

viewWindow = window;

viewWindow.Title = windowTitle;

DockPanel.SetDock(this, dock);

*// The viewWindow must have a dockPanel*

*// called WindowDockPanel.*

*// If you want to change this to use some*

*// other container on the window, then*

*// the below code should be the only place*

*// it needs to be changed.*

viewWindow.WindowDockPanel.Children.Add(this);

if (windowWidth == 0 && windowHeight == 0)

{

viewWindow.SizeToContent =

SizeToContent.WidthAndHeight;

}

else

{

viewWindow.SizeToContent = SizeToContent.Manual;

viewWindow.Width = windowWidth;

viewWindow.Height = windowHeight;

}

if (modal)

{

viewWindow.ShowDialog();

}

else

{

viewWindow.Show();

}

}

/// *<summary>*

/// *Show the View in a New Window*

/// *</summary>*

/// *<param name="windowTitle">Give the Window a Title</param>*

/// *<param name="windowWidth">Set the Window's Width</param>*

/// *<param name="windowHeight">Set the Window's Height</param>*

/// *<param name="dock">How to Dock the View in the Window</param>*

/// *<param name="onWindowClosed">Event handler for*

/// *when the Window closes</param>*

public void ShowInWindow(

bool modal, string windowTitle,

double windowWidth, double windowHeight,

Dock dock, OnWindowClose onWindowClose)

{

ShowInWindow(modal, ViewWindow, windowTitle,

windowWidth, windowHeight, dock, onWindowClose);

}

#endregion

#region IDisposable Members

void IDisposable.Dispose()

{

*// Remove any events from our window to prevent any*

*// memory leakage.*

if (viewWindow != null)

{

viewWindow.Closed -= this.ViewsWindow\_Closed;

}

}

#endregion

}

}

You'll need to add a reference to the ViewModel project in order for this to compile.

This is a reasonably basic BaseView - there's a few different methods that can be used for showing our View, but the list is not complete by any means. I've tried to give the basic requirements in this version. It is certainly open to expansion. You will also see the implementation of our event handlers for when the ViewModel closes, and when it is Activated.

### End of the Second Part

We've now completed the project up to the point where we need to start creating application specific code. In other words, we've created our framework - but I really don't want to use that word - I don't see this as a framework, but just as a bunch of classes, put together to allow me to develop a WPF MVVM# application.

The application should build - if you're typing it in rather than downloading it, check your namespaces, as VS does tend to add folder names to namespaces just to annoy me.

Next time, we'll start developing the application proper - and finally have something to run.

## License

## Introduction

In [part one](https://www.codeproject.com/KB/WPF/MVVMEpisode1.aspx) of this series of articles, I introduced my take on MVVM pattern, and discussed some shortfalls I felt existed in some implementations and, indeed, with the model itself.

In [part two](https://www.codeproject.com/KB/WPF/MVVMEpisode2.aspx), I introduced the base classes and interfaces I use in my implementation that, for want of a better title, I'm calling MVVM#.

In this part of the series, I will add the application specific classes to give us a (very) simple running application

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.

## Models

Whether we're dealing with a legacy system or a new one, I tend to think about the data first and foremost - after all, if you don't have the right data, it doesn't matter how cool the application is! (AKA GIGO).

We're just dealing with Customers in our example application. So we will need a Customer class. This would be the full details of a customer and may, in a real system, have a lot of data. When we're just dealing with a selection list, though, we really don't want to have a huge collection of large Customer objects, just to display a customer Name. For this purpose, I create 'ListData' classes. The CustomerListData class will hold just the basic details of a customer that I want to show in my selection list.

Because the CustomerListData is a subset of the full Customer data, I actually inherit my Customer data from the CustomerListData for convenience. It means that I can always replace a collection of CustomerListDatawith a collection of Customer if I want.

#### CustomerListData.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

namespace Model

{

/// *<summary>*

/// *Summary information for a Customer*

/// *As a 'cut down' version of Customer information, this class is used*

/// *for lists of Customers, for example, to avoid having to get a complete*

/// *Customer object*

/// *</summary>*

public class CustomerListData

{

/// *<summary>*

/// *The unique Id assigned to this Customer in the Data Store*

/// *</summary>*

public int? Id

{

get;

set;

}

/// *<summary>*

/// *The Business name of the Customer*

/// *</summary>*

public string Name

{

get;

set;

}

/// *<summary>*

/// *Which State the Customer is in*

/// *</summary>*

public string State

{

get;

set;

}

}

}

#### Customer.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

namespace Model

{

/// *<summary>*

/// *A Customer*

/// *This inherits from the CustomerSummary class, which contains the basic Customer*

/// *information provided in lists.*

/// *In real implementations this class may use lazy loading to get transactions*

/// *</summary>*

public class Customer : CustomerListData

{

/// *<summary>*

/// *The address of the customer.*

/// *</summary>*

public string Address

{

get;

set;

}

public string Suburb

{

get;

set;

}

public string PostCode

{

get;

set;

}

public string Phone

{

get;

set;

}

public string Email

{

get;

set;

}

}

}

I've stripped my Model class down to the bare bones for this article.

### Services

Now we have some data objects, we need some way to retrieve and store them into our data store (be that a database, a text file, some XML files, a web service or whatever). So in the Services project, we need to create our Service Interface ...

#### IcustomerService.cs

Hide   Copy Code

using System.Collections.Generic;

using Model;

namespace Service

{

public interface ICustomerService

{

/// *<summary>*

/// *Return the Customer for the given id*

/// *</summary>*

/// *<param name="id"></param>*

/// *<returns></returns>*

Customer GetCustomer(int id);

/// *<summary>*

/// *Return a list of Customers' List Data filtered by State*

/// *</summary>*

/// *<returns></returns>*

List<CustomerListData> GetListOfCustomers(string stateFilter);

/// *<summary>*

/// *Update a customer in the data store*

/// *</summary>*

/// *<param name="?"></param>*

void UpdateCustomer(Customer data);

}

}

That will give us what we need for this application. So let's implement the Interface...

#### CustomerService.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System.Collections.Generic;

using Model;

namespace Service

{

/// *<summary>*

/// *Provide services for retrieving and storing Customer information*

/// *</summary>*

public class CustomerService : ICustomerService

{

/// *<summary>*

/// *A fake database implementation so we can store and retrieve customers*

/// *</summary>*

private List<Customer> fakeDatabaseOfCustomers;

public CustomerService()

{

*// Add some data to our database*

fakeDatabaseOfCustomers = new List<Customer>();

fakeDatabaseOfCustomers.Add(DummyCustomerData(1));

fakeDatabaseOfCustomers.Add(DummyCustomerData(2));

fakeDatabaseOfCustomers.Add(DummyCustomerData(3));

fakeDatabaseOfCustomers.Add(DummyCustomerData(4));

fakeDatabaseOfCustomers.Add(DummyCustomerData(5));

fakeDatabaseOfCustomers.Add(DummyCustomerData(6));

fakeDatabaseOfCustomers.Add(DummyCustomerData(7));

}

/// *<summary>*

/// *Make a fake customer*

/// *</summary>*

/// *<param name="id"></param>*

/// *<returns></returns>*

private Customer DummyCustomerData(int id)

{

Customer customer = new Customer()

{

Id = id,

Address = id.ToString() + " High Street",

Suburb = "Nether Wallop",

State = (id % 2) == 0 ? "Qld" : "NSW",

Email = "Customer" + id.ToString() + "@BigFoot.Com",

Phone = "07 3333 4444",

Name = "Customer Number " + id.ToString()

};

return customer;

}

#region ICustomerService

public Customer GetCustomer(int id)

{

return fakeDatabaseOfCustomers[id - 1];

}

public List<CustomerListData> GetListOfCustomers(string stateFilter)

{

List<CustomerListData> list = new List<CustomerListData>();

foreach (var item in fakeDatabaseOfCustomers)

{

if (string.IsNullOrEmpty(stateFilter) ||

item.State.ToUpper() == stateFilter.ToUpper())

{

list.Add(new CustomerListData()

{

Id = item.Id,

Name = item.Name,

State = item.State

});

}

}

return list;

}

public void UpdateCustomer(Customer data)

{

fakeDatabaseOfCustomers[(int)data.Id - 1] = data;

}

#endregion

}

}

You'll see that the CustomerService class creates a 'fake' collection (fakeDatabaseOfCustomer) - no saving to any repository - but it serves the purposes for this demonstration. It's just there to help us get an application running with some data without having to populate a database - don't confuse it with design-time data which will be discussed later.

Don't forget to add a reference to the Models project!

### ViewData

So, we have our data objects (in Models) and we have some services to store and retrieve the data. Now we need to think about the actual presentation. Remember that our ViewData need to have Observable properties for each of the properties the user needs to see.

We'll think first about what data is going to be used.

* We'll need a class containing all of the editable properties of the Customer (**CustomerEditViewData**)
* We'll need a class containing a minimal set of data for displaying lists of Customer information (**CustomerListItemViewData**)
* We'll need a class containing a collection of CustomerListItemViewData so we can show a list (**CustomerSelectionViewData**)

These classes have a more or less 1-1 relationship with the ViewModels (and thus the Views) we'll be creating. In this case, there's also a (more or less)1-1 relationship between the ViewData and the Model objects - but that's not necessarily going to be the case in larger more complex applications.

I say 'more or less' because while the CustomerEditViewData maps to the CustomerEditViewModel, and the CustomerSelectionViewData maps to the CustomerSelectionViewModel, CustomerSelectionViewData is really just a collection of CustomerListItemViewData - which doesn't have its own ViewModel at all.

These classes all live in the ViewModel's project, in their own sub folder, **ViewData.**

They all inherit from BaseViewData, and use their base's RaisePropertyChanged method to notify the View(s) of any changes.

So, let's start off with the CustomerListItemViewData:

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System.Windows;

namespace ViewModels

{

/// *<summary>*

/// *A minimalist view of a Customer - for displaying in lists*

/// *</summary>*

public class CustomerListItemViewData : BaseViewData

{

#region Private Fields

private string customerName;

private int? customerId;

private string state;

#endregion

#region Observable Properties

/// *<summary>*

/// *The Id of the Customer represented by this item.*

/// *</summary>*

public int? CustomerId

{

get

{

return customerId;

}

set

{

if (value != customerId)

{

customerId = value;

base.RaisePropertyChanged("CustomerId");

}

}

}

public string CustomerName

{

get

{

return customerName;

}

set

{

if (value != customerName)

{

customerName = value;

base.RaisePropertyChanged("CustomerName");

}

}

}

public string State

{

get

{

return state;

}

set

{

if (value != state)

{

state = value;

base.RaisePropertyChanged("State");

}

}

}

#endregion

#region Constructor

#endregion

}

}

That's all fairly simple - so let's move on to the CustomerSelectionViewData, which, as we've said, is just a collection of CustomerListItemViewData using System.Collections.ObjectModel.

Hide   Copy Code

namespace ViewModels

{

public class CustomerSelectionViewData : BaseViewData

{

private ObservableCollection<CustomerListItemViewData> customers;

public ObservableCollection<CustomerListItemViewData> Customers

{

get

{

return customers;

}

set

{

if (value != customers)

{

customers = value;

base.RaisePropertyChanged("Customers");

}

}

}

}

}

Well, now we're getting somewhere!

We also need CustomerEditViewData - this is a big one, but still pretty simple in concept.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

namespace ViewModels

{

/// *<summary>*

/// *Editable Customer Info*

/// *</summary>*

public class CustomerEditViewData : BaseViewData

{

#region Private Fields

private string name;

private int? customerId;

private string address;

private string suburb;

private string email;

private string postCode;

private string phone;

private string state;

#endregion

#region Observable Properties

public int? CustomerId

{

get

{

return customerId;

}

set

{

if (value != customerId)

{

customerId = value;

base.RaisePropertyChanged("CustomerId");

}

}

}

public string Name

{

get

{

return name;

}

set

{

if (value != name)

{

name = value;

base.RaisePropertyChanged("Name");

}

}

}

public string Address

{

get

{

return address;

}

set

{

if (value != address)

{

address = value;

base.RaisePropertyChanged("Address");

}

}

}

public string Suburb

{

get

{

return suburb;

}

set

{

if (suburb != value)

{

suburb = value;

base.RaisePropertyChanged("Suburb");

}

}

}

public string Email

{

get

{

return email;

}

set

{

if (email != value)

{

email = value;

base.RaisePropertyChanged("Email");

}

}

}

public string PostCode

{

get

{

return postCode;

}

set

{

if (postCode != value)

{

postCode = value;

base.RaisePropertyChanged("PostCode");

}

}

}

public string Phone

{

get

{

return phone;

}

set

{

if (phone != value)

{

phone = value;

base.RaisePropertyChanged("Phone");

}

}

}

public string State

{

get

{

return state;

}

set

{

if (state != value)

{

state = value;

base.RaisePropertyChanged("State");

}

}

}

#endregion

#region Constructor

#endregion

}

}

Let's finish there with the ViewData and move over to our Controller.

### ICustomerController

We need now to look at our CustomerController. What functionality do we need it to perform?

1. Provide a CustomerSelectionViewData object to be shown to the user
2. Handle the selection of a Customer
3. Handle the request to edit a Customer
4. Handle updating a Customer when changes are saved

It is worth just looking closely at items 2 and 3. In a simplistic view, you might think that we don't need both of these - after all, when a Customer is selected, we're going to edit it; but there's actually two steps here - the selection and the editing - even though the selection in this case is specifically for editing.

What we're going to be doing is sending a message when the customer is selected - and that will be the end of the job for the CustomerSelectionViewModel. The Controller will send a message informing anyone that's interested that the Customer has been selected for editing. If there's nothing our there that has both registered to receive the message, and that confirms they can handle editing this specific customer, then the controller will need to take steps to edit the customer itself - by instantiating a new CustomerEditViewModel and CustomerEditView.

It may sound overly complicated but what I had in mind here was allowing us to have several CustomerEditViews open at one time - each editing a different customer. So, if the user selected a customer, all of the CustomerEditViewModels would receive the message telling them that Customer 1234 has been selected for editing. Most ViewModels would ignore the message - but one that is currently editing that very customer, could then 'make itself known'.

So - here's our ICustomerController interface. This is in the ViewModels project - under BaseClasses(yeah, I know it's not a class, but if you're worried, change the folder name to BaseClassesIntrefacesAndOtherNonProjectSpecificClasses or something!

Hide   Copy Code

namespace ViewModels

{

public interface ICustomerController : IController

{

/// *<summary>*

/// *Return a collection of Customer information to be displayed in a list*

/// *</summary>*

/// *<returns>A collection of Customers</returns>*

CustomerListViewData GetCustomerSelectionViewData(string stateFilter);

/// *<summary>*

/// *Do whatever needs to be done when a Customer is selected (i.e. edit it)*

/// *</summary>*

/// *<param name="customerId"></param>*

void CustomerSelectedForEdit(CustomerListItemViewData data, BaseViewModel daddy);

/// *<summary>*

/// *Edit this customer Id*

/// *</summary>*

/// *<param name="customerId"></param>*

void EditCustomer(int customerId, BaseViewModel daddy);

/// *<summary>*

/// *Update Customer data in the repository*

/// *</summary>*

/// *<param name="data"></param>*

void UpdateCustomer(CustomerEditViewData data);

}

}

### ViewModel

Well, now we have the basics, let's start thinking about our first ViewModel. At last!

The CustomerSelectionViewModel needs to display a list of customers, and allow the user to select one. Initially, that's it, so let's write the CustomerSelectionViewModel class.

#### CustomerSelectionViewModel

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Windows.Input;

using Messengers;

namespace ViewModels

{

/// *<summary>*

/// *This view model expects the user to be able to select from a list of*

/// *Customers, sending a message when one is selected.*

/// *On selection, the Controller will be asked to show*

/// *the details of the selected Customer*

/// *</summary>*

public class CustomerSelectionViewModel : BaseViewModel

{

#region Properties

/// *<summary>*

/// *Just to save us casting the base class's*

/// *IController to ICustomerController all the time...*

/// *</summary>*

private ICustomerController CustomerController

{

get

{

return (ICustomerController)Controller;

}

}

#region Observable Properties

private CustomerListItemViewData selectedItem;

public CustomerListItemViewData SelectedItem

{

get

{

return selectedItem;

}

set

{

if (value != selectedItem)

{

selectedItem = value;

RaisePropertyChanged("SelectedItem");

}

}

}

#endregion

#endregion

#region Commands

#region Command Relays

private RelayCommand userSelectedItemCommand;

public ICommand UserSelectedItemCommand

{

get

{

return userSelectedItemCommand ??

(userSelectedItemCommand = new RelayCommand(() =>

ObeyUserSelectedItemCommand()));

}

}

#endregion

#region Command Handlers

private void ObeyUserSelectedItemCommand()

{

CustomerController.CustomerSelectedForEdit

(this.SelectedItem, this);

}

#endregion

#endregion

#region Constructors

/// *<summary>*

/// *Required to allow our DesignTime version to be instantiated*

/// *</summary>*

protected CustomerSelectionViewModel()

{

}

public CustomerSelectionViewModel(ICustomerController controller,

string stateFilter = "")

: this(controller, null, stateFilter)

{

}

/// *<summary>*

/// *Use the base class to store the controller*

/// *and set the Data Context of the view (view)*

/// *Initialise any data that needs initialising*

/// *</summary>*

/// *<param name="controller"></param>*

/// *<param name="view"></param>*

public CustomerSelectionViewModel(ICustomerController controller,

IView view, string stateFilter = "")

: base(controller, view)

{

controller.Messenger.Register(MessageTypes.MSG\_CUSTOMER\_SAVED,

new Action<Message>(RefreshList));

*// Leave it for half a second before filtering on State*

RefreshList();

}

#endregion

#region Private Methods

private void RefreshList(Message message)

{

RefreshList();

message.HandledStatus = MessageHandledStatus.HandledContinue;

}

/// *<summary>*

/// *Ask for an updated list of customers based on the filter*

/// *</summary>*

private void RefreshList()

{

ViewData =

CustomerController.GetCustomerSelectionViewData("");

}

#endregion

}

}

A few things to note in the CustomerSelectionViewModel...

First, to save me having to cast the BaseViewModel's Controller property to ICustomerController all the time, I've added a private property CustomerController. Much like the flushable toilet, it's just a convenience thing.

We have an Observable property of SelectedItem. This is the CustomerListItemViewData that is currently selected from the list presented to the user - so whatever binds to this property needs to tell us via that binding what is currently selected.

We have a UserSelectedItemCommand. As its name suggests, this is the Command that our View will send when the user has selected an item. It is up to the designer whether this is on the press of a button, or as each row on a grid list is clicked, or via some quirky user interface dreamed up over several pints of Guinness.

There is a parameterless constructor. This is required because I want to be able to provide design-time support for data - and design time support demands a parameterless constructor. Every ViewModel requires a Controller, so the other constructors require an ICustomerController parameter. I'm also allowing the constructor to (optionally) provide a State Filter. This isn't implemented in the listing above, but the aim is to allow a CustomerSelectionViewModel to be created, filtering the customers to only show those from a particular state - perhaps the state the operator is in.

The other constructor allows us to create the ViewModel without an injected View. But what good is a ViewModel without a View? Well, no good at all - but good question, it shows you're paying attention! The non-view variant of our constructor will allow us to instantiate a ViewModel for a View that is created at design time - for example, if the designer decides that the Customer selection and edit should both appear together on a 'parent' view, she can design it like that, and we'll need to create a parent ViewModel that instantiates the CustomerSelectionViewModel and assigns it to the DataContext of the design-time created view.

Notice that the constructor also registers our ViewModel to receive messages of type MSG\_CUSTOMER\_SAVEDand, when it does so, it uses the RefreshList method to ask the Controller to provide an updated list of Customers. This way, whenever a customer is updated somewhere, our list will reflect any changes.

When it's instantiated, our ViewModel also calls its Refresh() method to get the initial data for display. I sometimes struggle with the "right" way to do this - should the ViewModel get the data when it's instantiated, or should the Controller feed in the data? There's pros and cons for each school of thought, and in this case I chose to use the 'pull' method - where the ViewModel pulls the data from the Controller - rather than the 'push' method - where the Controller pushes the data into the ViewModel.

### View

We really should think about creating a View now - so we not only have something to see, but also so our highly paid designer can have something to do!

Create a new WPF UserControl in the Views project, called CustomerSelectionView. You'll need to change the base class in the .cs file to BaseView (from UserControl). Then, do your design. Here's my XAML. (I'm not a designer!)

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

<view:BaseView x:Class="Views.CustomerSelectionView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:view="clr-namespace:Views"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

Background="#FF190000"

Margin="0"

Padding="1"

Height="304"

Width="229"

d:DataContext="{d:DesignInstance

Type=view:DesignTimeCustomerSelectionViewModel,

IsDesignTimeCreatable=true}">

<view:BaseView.Resources>

<view:NullToFalseBooleanConverter x:Key="NullToFalseBooleanConverter" />

<view:NullToHiddenVisibilityConverter

x:Key="NullToHiddenVisibilityConverter" />

</view:BaseView.Resources>

<StackPanel Background="#FF0096C8">

<StackPanel Orientation="Horizontal"

Margin="20,20,20,2"

Height="20">

<TextBlock>State:</TextBlock>

<TextBox Width="80"

Margin="10,0,0,0"

Text="{Binding Path=StateFilter,

UpdateSourceTrigger=PropertyChanged}"></TextBox>

</StackPanel>

<DataGrid AutoGenerateColumns="False"

Height="186"

Margin="4"

ItemsSource="{Binding ViewData.Customers}"

SelectedItem="{Binding Path=SelectedItem}"

Background="#FFE0C300"

CanUserReorderColumns="False"

AlternatingRowBackground="#E6FCFCB8"

CanUserAddRows="False"

CanUserDeleteRows="False"

CanUserResizeRows="False"

SelectionMode="Single"

IsReadOnly="True">

<DataGrid.Columns>

<DataGridTextColumn Header="Customer"

Binding="{Binding Path=CustomerName}"

Width="\*" />

<DataGridTextColumn Header="State"

Binding="{Binding Path=State}" />

</DataGrid.Columns>

</DataGrid>

<TextBlock Visibility="{Binding Path=SelectedItem,

Converter={StaticResource NullToHiddenVisibilityConverter}}">

<TextBlock.Text>

<MultiBinding StringFormat="{}Selected {0} with Id {1}">

<Binding Path="SelectedItem.CustomerName" />

<Binding Path="SelectedItem.CustomerId" />

</MultiBinding>

</TextBlock.Text></TextBlock>

<Button Content="Edit Customer"

Command="{Binding Path=UserSelectedItemCommand,

Mode=OneTime}"

Width="Auto"

HorizontalAlignment="Right"

Margin="4"

Padding="8,0,8,0"

IsEnabled="{Binding Path=SelectedItem,

Converter={StaticResource NullToFalseBooleanConverter}}" />

</StackPanel>

</view:BaseView>

If you're following along rather than downloading the project, you'll see that there's a couple of errors in the XAML.

In our resources section, we have two resources referenced that we've not written yet; NullToFalseBooleanConverter and NullToHiddenVisibilityConverter. The reason for the first is that my designer wants to display the Id and Name of the currently selected customer in a text block - so obviously if nothing is currently selected,  she wants the TextBlock to be hidden. The second is used because the designer wants the Edit Customer button to be disabled when no customer is selected.

I stick all my converters into a single source file, in a converters folder in the Views project - so we can go ahead and write these two simple converters now.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Globalization;

using System.Windows;

using System.Windows.Data;

namespace Views

{

*/\**

*\* This source file contains all the converters used.*

*\*/*

/// *<summary>*

/// *Returns false if the object is null, true otherwise.*

/// *handy for using when something needs to be enabled or disabled depending on*

/// *whether a value has been selected from a list.*

/// *</summary>*

[ValueConversion(typeof(object), typeof(bool))]

public class NullToFalseBooleanConverter : IValueConverter

{

public object Convert(object value, Type targetType,

object parameter, CultureInfo culture)

{

return (value != null);

}

public object ConvertBack(object value, Type targetType,

object parameter, CultureInfo culture)

{

return null;

}

}

/// *<summary>*

/// *Returns false if the object is null, true otherwise.*

/// *handy for using when something needs to be enabled or disabled depending on*

/// *whether a value has been selected from a list.*

/// *</summary>*

[ValueConversion(typeof(object), typeof(Visibility))]

public class NullToHiddenVisibilityConverter : IValueConverter

{

public object Convert(object value, Type targetType,

object parameter, CultureInfo culture)

{

if (value == null)

{

return Visibility.Hidden;

}

else

{

return Visibility.Visible;

}

}

public object ConvertBack(object value, Type targetType,

object parameter, CultureInfo culture)

{

return null;

}

}

}

When these two converters are written, we're left with a single compile error. The line:

Hide   Copy Code

d:DataContext="{d:DesignInstance Type=view:DesignTimeCustomerSelectionViewModel,

IsDesignTimeCreatable=true}">

can't find the DesignTimeCustomerSelectionViewModel class. which is fair enough, as we haven't written it yet!

This class is the design-time only class that I can populate with some realistic-looking data to allow my designer to see what she's dealing with. So much nicer for her to see real data rather than an empty grid.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System.Collections.ObjectModel;

using ViewModels;

namespace Views

{

/// *<summary>*

/// *This class allows us to see design time customers. Blendability R Us*

/// *</summary>*

public class DesignTimeCustomerSelectionViewModel : CustomerSelectionViewModel

{

public DesignTimeCustomerSelectionViewModel()

{

ViewData = new CustomerListViewData();

var customers = new ObservableCollection<CustomerListItemViewData>();

customers.Add(new CustomerListItemViewData()

{

CustomerId = 1,

CustomerName = "First Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 2,

CustomerName = "2nd Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 3,

CustomerName = "Third Customer",

State = "NSW"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 4,

CustomerName = "Fourth Customer",

State = "SA"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 1,

CustomerName = "First Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 2,

CustomerName = "2nd Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 3,

CustomerName = "Third Customer",

State = "NSW"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 4,

CustomerName = "Fourth Customer",

State = "SA"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 1,

CustomerName = "First Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 2,

CustomerName = "2nd Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 3,

CustomerName = "Third Customer",

State = "NSW"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 4,

CustomerName = "Fourth Customer",

State = "SA"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 1,

CustomerName = "First Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 2,

CustomerName = "2nd Customer",

State = "Qld"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 3,

CustomerName = "Third Customer",

State = "NSW"

});

customers.Add(new CustomerListItemViewData()

{

CustomerId = 4,

CustomerName = "Fourth Customer",

State = "SA"

});

((CustomerListViewData)ViewData).Customers = customers;

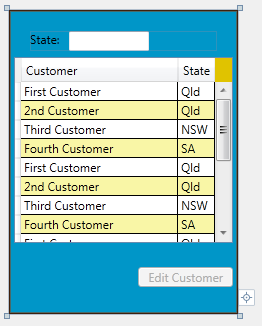
}

}

}

The class itself inherits from the 'real' CustomerSelectionViewModel, and just has a constructor that creates some dummy data for the designer to use.

Once this is written, rebuild and you should see the design time data, at design time! As Designed!



It seems we're so close to having a running program - just a few bits to do, so why not give the View to your designer to pretty up while we do the technical stuff?

### Controller

Remember we created the ICustomerController interface earlier? Well, now we have to do some real implementation. In any large system, the Controller can become a bit of a large beast, so I tend to split mine into several partial classes. The main one called CustomerController, then others called CustomerController\_DataRetrieval and CustomerController\_ViewManagement. this is one of those things that I find useful, and you may like it, or use different partial classes, or just lump code into one source file with lots of #regions - whatever takes your fancy. The thing I like about the logical separation into partial classes is in a multi-developer environment it allows me to assign a developer to write one area of the controller without affecting other developers who may work on other areas of the Controller.

Because the Controller is the central hub of the system, it requires references to all the other projects, and also PresentationCore, PresentationFramework, WindowsBase, and System.Xaml - add 'em now or wait to see the errors if you don't believe me. https://www.codeproject.com/script/Forums/Images/smiley\_wink.gif

#### CustomerController.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using Messengers;

using Service;

using ViewModels;

using Views;

namespace Controllers

{

/// *<summary>*

/// *The controller 'is' the application.*

/// *Everything is controlled by this :*

/// *it instantiates Views and ViewModels*

/// *it retrieves and stores customers via services*

///

/// *But it does all this only in response to requests*

/// *made by the ViewModels.*

///

/// *e.g. a ViewModel may request a list of customers*

/// *e.g. a ViewModel may want to save changes to a customer*

///

/// *set up as a partial class for convenience*

/// *</summary>*

public partial class CustomerController : BaseController, ICustomerController

{

private static ICustomerService CustomerService;

#region Constructors

/// *<summary>*

/// *Private constructor - we must pass a service to the constructor*

/// *</summary>*

private CustomerController()

{

}

/// *<summary>*

/// *The controller needs a reference to the service layer to enable it*

/// *to make service calls*

/// *</summary>*

/// *<param name="customerService"></param>*

public CustomerController(ICustomerService customerService)

{

CustomerService = customerService;

}

#endregion

#region Public Methods

/// *<summary>*

/// *Main entry point of the Controller.*

/// *Called once (from App.xaml.cs) this will initialise the application*

/// *</summary>*

public void Start()

{

ShowViewCustomerSelection();

}

/// *<summary>*

/// *Edit the customer with the Id passed*

/// *</summary>*

/// *<param name="customerId">Id of the customer to be edited</param>*

/// *<param name="daddy">The 'parent' ViewModel who will own the*

/// *ViewModel that controls the Customer Edit</param>*

public void EditCustomer(int customerId, BaseViewModel daddy = null)

{

*//BaseView view = GetCustomerEditView(customerId, daddy);*

*//view.ShowInWindow(false, "Edit Customer");*

}

/// *<summary>*

/// *A Customer has been selected to be edited*

/// *</summary>*

/// *<param name="data">The CustomerListItemViewData of the selected customer*

/// *</param>*

/// *<param name="daddy">The parent ViewModel</param>*

public void CustomerSelectedForEdit(CustomerListItemViewData data,

BaseViewModel daddy = null)

{

*// Check in case we get a null sent to us*

if (data != null && data.CustomerId != null)

{

NotificationResult result = Messenger.NotifyColleagues

(MessageTypes.MSG\_CUSTOMER\_SELECTED\_FOR\_EDIT, data);

if (result == NotificationResult.MessageNotRegistered ||

result == NotificationResult.MessageRegisteredNotHandled)

{

*// Nothing was out there that handled our message,*

*// so we'll do it ourselves!*

EditCustomer((int)data.CustomerId, daddy);

}

}

}

#endregion

}

}

The main CustomerController source will show a couple of build errors until we complete the other partial classes. notice also here I've commented out code in the EditCustomer method - as we haven't yet created the ViewModel or View to perform this function.

#### CustomerController\_Dataretrieval.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System.Collections.ObjectModel;

using Messengers;

using Model;

using Service;

using ViewModels;

namespace Controllers

{

public partial class CustomerController

{

/// *<summary>*

/// *Get a collection of Customers and return an Observable*

/// *collection of CustomerListItemViewData*

/// *for display in a list.*

/// *You could bypass this conversion if you wanted to present a*

/// *list of Customers by binding directly to*

/// *the Customer object.*

/// *</summary>*

/// *<returns></returns>*

public CustomerListViewData GetCustomerSelectionViewData(string stateFilter)

{

CustomerListViewData vd = new CustomerListViewData();

vd.Customers = new ObservableCollection<CustomerListItemViewData>();

foreach (var customer in CustomerService.GetListOfCustomers(stateFilter))

{

vd.Customers.Add(new CustomerListItemViewData()

{

CustomerId = (int)customer.Id,

CustomerName = customer.Name,

State = customer.State

});

}

return vd;

}

/// *<summary>*

/// *Get the Edit View Data for the Customer Id specified*

/// *</summary>*

/// *<param name="customerId"></param>*

/// *<returns></returns>*

public CustomerEditViewData GetCustomerEditViewData(int customerId)

{

var customer = CustomerService.GetCustomer(customerId);

return new CustomerEditViewData()

{

CustomerId = customer.Id,

Name = customer.Name,

Address = customer.Address,

Suburb = customer.Suburb,

PostCode = customer.PostCode,

State = customer.State,

Phone = customer.Phone,

Email = customer.Email

};

}

public void UpdateCustomer(CustomerEditViewData data)

{

Customer item = new Customer()

{

Id = data.CustomerId,

Address = data.Address,

Name = data.Name,

Suburb = data.Suburb,

PostCode = data.PostCode,

Email = data.Email,

Phone = data.Phone,

State = data.State

};

CustomerService.UpdateCustomer(item);

Messenger.NotifyColleagues(MessageTypes.MSG\_CUSTOMER\_SAVED, data);

}

}

}

#### CustomerController\_ViewManagement.cs

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using ViewModels;

using Views;

namespace Controllers

{

public partial class CustomerController : ICustomerController

{

/// *<summary>*

/// *The ShowView methods are private.*

/// *A ViewModel may request some action to take place,*

/// *but the Controller will decide whether this action will result*

/// *in some view being shown.*

/// *e.g. clicking a 'Search' button on a form may result*

/// *in a Command being sent from the*

/// *View (via binding) to the ViewModel; the Command handler*

/// *then asks the Controller to*

/// *Search for whatever.*

/// *The controller may (for example) use a service to*

/// *return a collection of objects. if there*

/// *is only a single object, then it may return a single object*

/// *rather than popping up a search*

/// *view only to have the User be presented with a single action*

/// *from which to select.*

/// *</summary>*

#region Show Views

private void ShowViewCustomerSelection()

{

CustomerSelectionView v = GetCustomerSelectionView();

v.ShowInWindow(false);

}

#endregion

#region Get Views

private CustomerSelectionView GetCustomerSelectionView(BaseViewModel daddy = null)

{

CustomerSelectionView v = new CustomerSelectionView();

CustomerSelectionViewModel vm = new CustomerSelectionViewModel(this, v);

if (daddy != null)

{

daddy.ChildViewModels.Add(vm);

}

return v;

}

private BaseView GetCustomerEditView(int customerId, BaseViewModel daddy)

{

*//CustomerEditView v = new CustomerEditView();*

*//CustomerEditViewModel vm = new CustomerEditViewModel(this, v);*

*//vm.ViewData = GetCustomerEditViewData(customerId);*

*//if (daddy != null)*

*//{*

*// daddy.ChildViewModels.Add(vm);*

*//}*

*//return v;*

return new BaseView();

}

#endregion

}

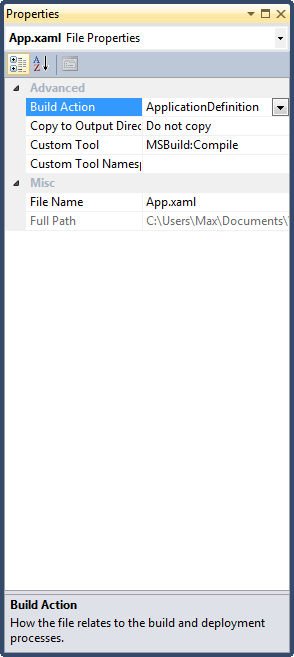
}

And again, I've fiddled the GetCustomerEditView method as we've not written the View or ViewModel yet.

Give that a build and it should be clean. Try to run it, though, and you will see an unhandled IO exception "cannot locate resource 'mainwindow.xaml'".

Fear not - this is expected. Remember we created a WPF application which expected us to use a main window WPF window - which we got rid of?  But we didn't tell the application that we didn't need it! Let's do that now. We'll need to add a reference to the Controllers project from the CustomerMaintenance project - so the application knows where to find its controllers, and a reference to the Services project, as the Controllers require a Service injected into their constructor. we also need a reference to ViewModels because that's where the CustomerController interface is located.

You also need to ensure that the Build Action property of the App.Xaml file to be 'ApplicationDefinition'.



Open up your App.xaml file in the Customermaintenance project, and change it to look like this...

Hide   Copy Code

<Application x:Class="MyMVVMApplication.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Startup="Application\_Startup">

<Application.Resources>

</Application.Resources>

</Application>

The Startup= attribute needs to point to our Event Handler that will start the whole thing going.

Finally, open up the App.xaml.cs file and change it to look like this...

Hide   Copy Code

using System.Windows;

using Controllers;

using Service;

using System;

namespace CustomerMaintenance

{

/// *<summary>*

/// *Interaction logic for App.xaml*

/// *</summary>*

public partial class App : Application

{

private void Application\_Startup(object sender, StartupEventArgs e)

{

CustomerController controller = new CustomerController(new CustomerService());

controller.Start();

}

}

}

Well - what are you waiting for? Press F5!

The program runs, a form appears with the Customer selection on it, showing a list of customers.

Astute WPF programmers always check the Output window when they run an application. Just in case you are one of them, I will point out that, in fact, there is an error:

Hide   Copy Code

System.Windows.Data Error: 40 : BindingExpression path error: 'StateFilter'

property not found on 'object' ''CustomerSelectionViewModel' (HashCode=13304725)'.

BindingExpression:Path=StateFilter; DataItem='CustomerSelectionViewModel'

(HashCode=13304725); target element is 'TextBox' (Name='');

target property is 'Text' (type 'String')

That's just because I've left the StateFilter TextBox on the View but omitted any property in the ViewModel to actually handle it.

But let's not dwell on the negatives, put on your party frock and celebrate - we've a working MVVM# application!

Next time, we'll add the filtering, and create CustomerEditViewModel and associated View so we'll end up with a small, but functional, application.

## License

## Introduction

In [part one](https://www.codeproject.com/KB/WPF/MVVMEpisode1.aspx) of this series of articles, I introduced my take on MVVM pattern, and discussed some of the shortfalls I felt existed in some implementations and, indeed, with the model itself.

In [part two](https://www.codeproject.com/KB/WPF/MVVMEpisode2.aspx), I introduced the base classes and interfaces I use in my implementation that, for want of a better title, I'm calling MVVM#

In [part three](https://www.codeproject.com/KB/WPF/MVVMEpisode3.aspx), I presented the code for enough of the application to get us up and running, displaying a Customer Selection View in a Form, containing data at both run time and design time.

In this part of the series, I will build upon this basic application to show real functionality.

## Filtering

The specification for our filtering requirements is pretty simple. Allow the user to type in a State code, and filter the list to include only those customers in that state. (I'm an ex-pat pom living in Aus - so I'm using Australian state codes here - the full set is QLD, NSW, SA, WA, TAS, NT, ACT, VIC if you're interested.) (Actually, that's the full set whether you're interested or not)

As usual, a million and one ways to solve this, but my designer wanted to do this as a "Search As you Type".

The first part to achieve this is actually already in the XAML of our CustomerSelectionView:

Hide   Copy Code

Text="{Binding Path=StateFilter, UpdateSourceTrigger=PropertyChanged}"></TextBox>

This line binds the state text box to a StateFilter property, and the UpdateSourceTrigger attribute tells WPF to update the property whenever it is changed.

A reminder may be in order here; this property, while an Observable property, is part of the functionality rather than part of the data being acted upon - so the property resides in the ViewModel rather than the ViewDataobject.

So, what will happen is, the user types something into the TextBox, that action sets the property on the ViewModel, which will ask the Controller to provide a newly filtered set of data. As that data is bound to the View, the DataGrid in the View will update.

But wait! If we implement that, as soon as a single letter is typed, the list will blank, as no States have a single letter code. we could do the filtering as a 'starts-with' filter - but then typing 'N' for NT would bring up all the NSW customers as well. OK - it's not such a problem really, but there could be an issue with the time it takes to get the filtered results - so we don't want to keep refreshing results as the user presses a key.

So, what I want to do is introduce a delay. No filtering will take place for, say, half a second after the user has typed something - then after half a second, the list will be filtered using the contents of the TextBox, unless another key is pressed, in which case the half-second countdown starts again.

So, we need a Timer - I'm using a DispatcherTimer, so a reference to WindowsBase needs to be added to the ViewModels project.

Then, we can add a new private field:

Hide   Copy Code

DispatcherTimer stateFilterTimer;

And we need to add our ObservableProperty StateFilter.

Hide   Copy Code

private string stateFilter;

public string StateFilter

{

get

{

return stateFilter;

}

set

{

if (value != stateFilter)

{

stateFilterTimer.Stop();

stateFilter = value;

RaisePropertyChanged("StateFilter");

stateFilterTimer.Start();

}

}

}

In the constructor, we need to instantiate the timer, and set its timespan to half a second, and give it an event handler method so we can handle the event when the timer reaches zero.

Hide   Copy Code

/// *<summary>*

/// *Use the base class to store the controller and set the Data Context of the view (view)*

/// *Initialise any data that needs initialising*

/// *</summary>*

/// *<param name="controller"></param>*

/// *<param name="view"></param>*

public CustomerSelectionViewModel(ICustomerController controller,

IView view, string stateFilter = "")

: base(controller, view)

{

controller.Messenger.Register(MessageTypes.MSG\_CUSTOMER\_SAVED,

new Action<Message>(RefreshList));

*// Leave it for half a second before filtering on State*

stateFilterTimer = new DispatcherTimer()

{

Interval = new TimeSpan(0, 0, 0, 0, 500)

};

stateFilterTimer.Tick += StateFilterTimerTick;

StateFilter = stateFilter;

RefreshList();

}

Of course, then we need to write that event handler...

Hide   Copy Code

/// *<summary>*

/// *Event handler for the timer used for 'filter as you type' on the State filter.*

/// *When the timer triggers, filter the list with the existing filter.*

/// *</summary>*

/// *<param name="sender"></param>*

/// *<param name="e"></param>*

void StateFilterTimerTick(object sender, EventArgs e)

{

stateFilterTimer.Stop();

RefreshList();

}

Then all that is required is to change the RefreshList method so that we pass the StateFilter property, rather than an empty string.

Hide   Copy Code

/// *<summary>*

/// *Ask for an updated list of customers based on the filter*

/// *</summary>*

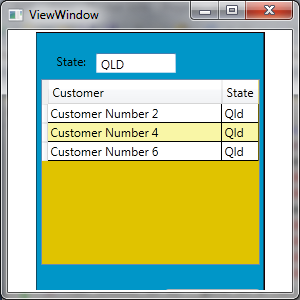
private void RefreshList()

{

ViewData = CustomerController.GetCustomerSelectionViewData(StateFilter);

}

That should do us - give it a run. Remembering in my test data I only used two states - Qld and NSW.



## Editing

Well, the whole point of this application is to be able to edit customer details, so let's get started by creating our CustomerEditViewModel.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Windows.Input;

using Messengers;

namespace ViewModels

{

/// *<summary>*

/// *A ViewModel for a view that allows a Customer to be modified*

/// *</summary>*

public class CustomerEditViewModel : BaseViewModel

{

#region Private Fields

#endregion

#region Properties

/// *<summary>*

/// *Just to save us casting the base class's IController*

/// *to ICustomerController all the time...*

/// *</summary>*

private ICustomerController CustomerController

{

get

{

return (ICustomerController)Controller;

}

}

#region Observable Properties

#endregion

#endregion

#region Commands

#region Command Relays

private RelayCommand<IView> cancelledCommand;

private RelayCommand<IView> saveCommand;

public ICommand CancelledCommand

{

get

{

return cancelledCommand ?? (cancelledCommand =

new RelayCommand<IView>(param => ObeyCancelledCommand(param),

param => CanObeyCancelledCommand(param)));

}

}

public ICommand SaveCommand

{

get

{

return saveCommand ?? (saveCommand = new RelayCommand<IView>

(param => ObeySaveCommand(param), param => CanObeySaveCommand(param)));

}

}

#endregion // Command Relays

#region Command Handlers

/// *<summary>*

/// *</summary>*

/// *<returns></returns>*

private bool CanObeyCancelledCommand(IView view)

{

return true;

}

private void ObeyCancelledCommand(IView view)

{

CloseViewModel(false);

}

private bool CanObeySaveCommand(IView view)

{

return true;

}

private void ObeySaveCommand(IView view)

{

CustomerController.UpdateCustomer((CustomerEditViewData)ViewData);

CloseViewModel(true);

}

#endregion // Command Handlers

#endregion // Commands

#region Constructor

public CustomerEditViewModel(ICustomerController controller)

: this(controller, null)

{

}

public CustomerEditViewModel(ICustomerController controller, IView view)

: base(controller, view)

{

controller.Messenger.Register(MessageTypes.MSG\_CUSTOMER\_SELECTED\_FOR\_EDIT,

new Action<Message>(HandleCustomerSelectedForEditMessage));

}

#endregion

/// *<summary>*

/// *If somewhere someone selects a customer for editing and this*

/// *Edit ViewModel is already*

/// *Editing that customer, then abort the message, and make the View active*

/// *</summary>*

/// *<param name="message"></param>*

private void HandleCustomerSelectedForEditMessage(Message message)

{

CustomerListItemViewData customer =

message.Payload as CustomerListItemViewData;

if (customer != null && customer.CustomerId ==

((CustomerEditViewData)ViewData).CustomerId)

{

message.HandledStatus = MessageHandledStatus.HandledCompleted;

ActivateViewModel();

}

}

}

}

Let's spend a minute looking through this code to make sure we understand what's what.

Again, I've defined a private property of type ICustomerController to return the IController defined in the BaseViewModel, just to save me casting it every time I use it.

There's no ObservableProperties. The ObservableProperties for the Customer data that we are editing are in the CustomerViewData - the absense of ObservableProperties in the ViewModel tells us that there is no additional bound functionality in this View.

We've defined two RelayCommands of type IView (cancelledCommand and saveCommand) instantiated when needed by the associated property Getter.

The CanObeyCancelledCommand method always returns true - so the user can always cancel.

The CanObeySaveCommand also always returns true. In the real world, of course, you'd probably only want to return true if the current CustomerEditViewData was 'dirty' - that is, the user had made changes - but this series of articles is long enough without adding the additional complexity of change tracking!

The ObeyCancelledCommand method (which is the method that gets called when the user cancels) uses the CloseViewModel method with a False parameter. This parameter determines, if a view is shown as a dialog,  whether the dialogresult is true or false.

The ObeySaveCommand asks the CustomerController to save the data in our ViewData (which is bound to the controls that the user is using to make changes, and so reflects those changes). It then uses the CloseViewModel method, passing 'True' so that, if the view is shown as a dialog, the DialogResult will be true.

In the constructor, you will see that the CustomerEditViewModel registers to receive messages of the type 'MSG\_CUSTOMER\_SELECTED\_FOR\_EDIT'. The reason for this is so that we can, if we so desire, open several CustomerEditViewModels, each editing its own Customer - but we really don't want the same customer to be edited in two Views - so every time a Customer is selected for Edit, the CustomerEditViewModel inspects the message and, if the customer selected matches the customer it's currently editing, it will set the message status to HandledCompleted and 'Activate' itself, which really means it will Activate the View - so the effect to the user will be that the window containing the View will become the Active window.)

This is all achieved in the HandleCustomerSelectedForEditMessage method. The Message object has a Payload property, which in the case of this message type, is a CustomerListItemViewData. (As an aside, I have implemented nothing in this version to either enforce that this message type's payload object is of the correct type, or made any attempt to automate this casting. It's not that hard to do so, if you want to - but the more you move down that route, the more you move toward a complicated Framework - and that's what I wanted to avoid.)

Okay - that's the ViewModel - let's knock up a View to suit - then we can ship it off to the designer to make it look pretty.

#### CustomerEditView.xaml

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

<views:BaseView x:Class="Views.CustomerEditView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:views="clr-namespace:Views"

mc:Ignorable="d"

d:DesignHeight="243"

d:DesignWidth="346"

d:DataContext="{d:DesignInstance

Type=views:DesignTimeCustomerEditViewModel,

IsDesignTimeCreatable=true}">

<StackPanel Margin="10">

<Grid >

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100\*" />

<ColumnDefinition Width="200\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="Name"

Grid.Column="0"

Margin="8" />

<TextBox Text="{Binding ViewData.Name}"

Grid.Column="1"

Margin="2" />

</Grid>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100\*" />

<ColumnDefinition Width="200\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="Address"

Grid.Column="0"

Grid.Row="0"

Margin="8" />

<TextBox Text="{Binding ViewData.Address}"

Grid.Column="1"

Grid.Row="0"

Margin="2" />

</Grid>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100\*" />

<ColumnDefinition Width="200\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="Suburb"

Grid.Column="0"

Grid.Row="0"

Margin="8" />

<TextBox Text="{Binding ViewData.Suburb}"

Grid.Column="1"

Grid.Row="0"

Margin="2" />

</Grid>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100\*" />

<ColumnDefinition Width="200\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="State"

Grid.Column="0"

Grid.Row="0"

Margin="8" />

<TextBox Text="{Binding ViewData.State}"

Grid.Column="1"

Grid.Row="0"

Margin="2" />

</Grid>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100\*" />

<ColumnDefinition Width="200\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="PostCode"

Grid.Column="0"

Grid.Row="0"

Margin="8" />

<TextBox Text="{Binding ViewData.PostCode}"

Grid.Column="1"

Grid.Row="0"

Margin="2" />

</Grid>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100\*" />

<ColumnDefinition Width="200\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="Phone"

Grid.Column="0"

Grid.Row="0"

Margin="8" />

<TextBox Text="{Binding ViewData.Phone}"

Grid.Column="1"

Grid.Row="0"

Margin="2" />

</Grid>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100\*" />

<ColumnDefinition Width="200\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="eMail"

Grid.Column="0"

Grid.Row="0"

Margin="8" />

<TextBox Text="{Binding ViewData.Email}"

Grid.Column="1"

Grid.Row="0"

Margin="2" />

</Grid>

<StackPanel Orientation="Horizontal"

FlowDirection="RightToLeft"

Height="35">

<Button Content="Save"

Command="{Binding Path=SaveCommand, Mode=OneTime}"

Height="23"

Width="75"

Margin="5,5,25,2" />

<Button Content="Cancel"

Command="{Binding Path=CancelledCommand, Mode=OneTime}"

Height="23"

Width="75"

Margin="5,5,25,2" />

</StackPanel>

</StackPanel>

</views:BaseView>

Nothing much to see in the edit view. A bunch of Text Blocks in pairs with TextBoxes. The TextBoxes are bound to the properties of the CustomerEditViewData.

There's two buttons - one to Cancel and one to Save, each bound to the appropriate Command. And that's about it!

So if we head back to the CustomerController\_ViewManagement.cs, the GetCustomerEditView can be un-commented as we now have a CustomerEditView so it will compile. Also in the CustomerController.cs source, the contents of the editCustomer method can be un-commented.

Before we go further, remember that part of our goal is Blendability - the ability to ship off our Views to be messed with by Designers to make them look pretty? In the CustomerEditView XAML, we have:

Hide   Copy Code

d:DataContext="{d:DesignInstance Type=views:DesignTimeCustomerEditViewModel,

IsDesignTimeCreatable=true}">

which tells the designer to instantiate an instance of a DesignTimeCustomerEditViewModel so our designer can see some data. So, we better create one.

#### DesignTimeCustomerEidtViewModel.cs

Hide   Copy Code

using ViewModels;

namespace Views

{

class DesignTimeCustomerEditViewModel : CustomerEditViewModel

{

public DesignTimeCustomerEditViewModel()

{

ViewData = new CustomerEditViewData()

{

Address = "23 Netherington on Wallop Street",

CustomerId = 123,

Email = "Oldhag@GeeMail.Com",

Name = "Betty Boop",

Phone = "0414 4142424",

PostCode = "4540",

State = "QLD",

Suburb = "Indooroopilly"

};

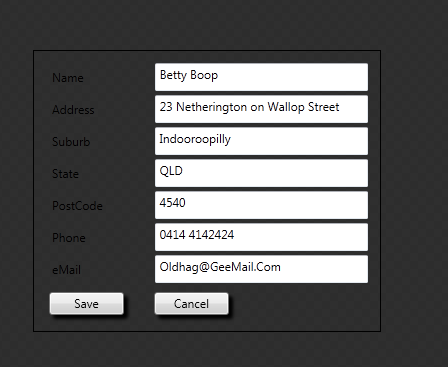
}

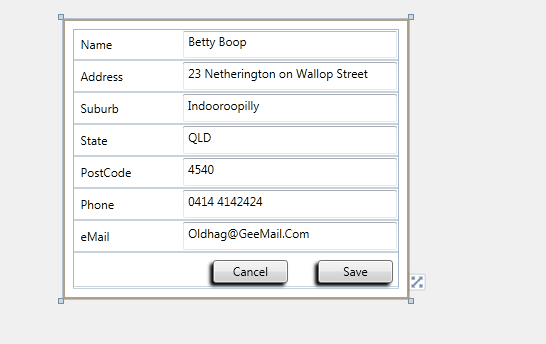
}

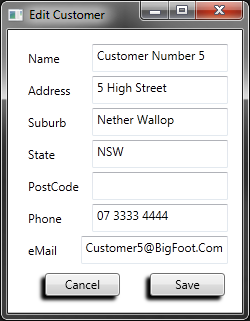
}

### Blendability - An Aside

Look at the three screenshots below. They're comparisons of what you see in VS2010, Expression Blend 4 and at Runtime with the customer form with a simple Drop Shadow effect added to the buttons (this effect is not included in the listings presented here, nor in the downloaded version.)







Interesting how Blend aligns the Buttons and the drop Shadow to the left, while VS2010 (and the runtime) aligns them both right! This sort of thing tends to upset Designers, but at least they can be mollified by the fact that they are looking at data rather than blank entry fields.

### On With the Show

Now, run the program. You should be able to select a customer, click the buton to edit it, which should open a window in which you can make changes to the customer. The window isn't modal, so you can go back and select another customer, which opens another window.

Close the Selection window, and all of the Edit windows will be closed too.

Make a change to one of the fields shown in the Selection list, save the customer, and the selection list refreshes to show the modified details.

Make a change to one of the same fields and cancel, and no changes are shown in the selection list.

Select the same customer twice, and the same window you opened the first time takes focus.

I reckon that fits the specification nicely (if you can remember back that far to part 1).

### Changes

Of course, we all know life ain't that simple! As soon as our project sponsor sees the application, he wants it changed. He doesn't like being able to have multiple edit windows open - far too confusing.

OK, let's make the change for him.

Pop into the CustomerController source, and find the EditCustomer method. This is where we do whatever we want to do when we want to edit a customer - so here's where we need to change that first parameter in the view.ShowInWindow from false to true.

Job done.

## Conclusion

We've come a long way over four instalments, but I hope this has been of some help to somebody.

As I have stressed, this is not a Framework, but just an example of how I have put together workable components to make an MVVM WPF application that works, and improves, for me, on some of the shortcomings of other solutions.

This might not be for you. You may want to use one of the frameworks out there - Cinch or MVVM Light or one of the gazzilion others - or you may want to roll your own.  Or, like me, you may want to do it your own way, implementing the thing you find works best in your environment.

Whatever you do, I'd really appreciate your feedback. I'm sure I've made mistakes along the way, and am always happy to learn from others to improve my own stuff.

Once again, many thanks to the giants on whose shoulders I've stood - especially Pete O'Hanlon.

## License

This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)