Rachel Lim's Blog

Simplifying programming into something I can understand

A Simple MVVM Example

In my opinion, if you are using WPF or Silverlight you should be using the MVVM design pattern. It is perfectly suited to the technology and allows you to keep your code clean and easy to maintain.

The problem is, there are a lot of online resources for MVVM, each with their own way of implementing the design pattern and it can be overwhelming. I would like to present MVVM in the simplest way possible using just the basics.

So lets start at the beginning.

MVVM

MVVM is short for Model-View-ViewModel.

Models are simple class objects that hold data. They should only contain properties and property validation. They are not responsible for getting data, saving data, click events, complex calculations, business rules, or any of that stuff.

Views are the UI used to display data. In most cases, they can be DataTemplates which is simply a template that tells the application how to display a class. It is OK to put code behind your view **IF** that code is related to the View only, such as setting focus or running animations.

ViewModels are where the magic happens. This is where the majority of your code-behind goes: data access, click events, complex calculations, business rules validation, etc. They are typically built to reflect a View. For example, if a View contains a ListBox of objects, a Selected object, and a Save button, the ViewModel will have an ObservableCollection ObectList, Model SelectedObject, and ICommand SaveCommand.

MVVM Example

I've put together a small sample showing these 3 layers and how they relate to each other. You'll notice that other than property/method names, none of the objects need to know anything about the others. Once the interfaces have been designed, each layer can be built completely independent of the others.

Sample Model

For this example I've used a Product Model. You'll notice that the only thing this class contains is properties and change notification code.

Usually I would also implement IDataErrorInfo here for property validation, however I have left this out for now.

```
public class ProductModel : ObservableObject
{
    #region Fields

    private int _productId;
    private string _productName;
    private decimal _unitPrice;

    #endregion // Fields

    #region Properties

    public int ProductId
    {
        get { return _productId; }
        set
        {
            if (value != _productId)
            {
                _productId = value;
                OnPropertyChanged("ProductId");
            }
        }
    }

    public string ProductName
    {
        get { return _productName; }
        set
        {
            if (value != _productName)
            {
                _productName = value;
                OnPropertyChanged("ProductName");
            }
        }
    }

    public decimal UnitPrice
    {
        get { return _unitPrice; }
        set
```

```
        {
            if (value != _unitPrice)
            {
                _unitPrice = value;
                OnPropertyChanged("UnitPrice");
            }
        }
    }

    #endregion // Properties
}
```

The class inherits from ObservableObject, which is a custom class I use to avoid having to rewrite the property change notification code repeatedly. I would actually recommend looking into Microsoft PRISM's (http://msdn.microsoft.com/en-us/library/gg406140.aspx) NotificationObject or MVVM Light's (http://mvvmlight.codeplex.com/) ViewModelBase which does the same thing once you are comfortable with MVVM, but for now I wanted to keep 3rd party libraries out of this and to show the code.

```
public abstract class ObservableObject : INotifyPropertyChanged
{
    #region INotifyPropertyChanged Members

    /// <summary>
    /// Raised when a property on this object has a new value.
    /// </summary>
    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// Raises this object's PropertyChanged event.
    /// </summary>
    /// <param name="propertyName">The property that has a new value.</param>
    protected virtual void OnPropertyChanged(string propertyName)
    {
        this.VerifyPropertyName(propertyName);

        if (this.PropertyChanged != null)
        {
            var e = new PropertyChangedEventArgs(propertyName);
            this.PropertyChanged(this, e);
        }
    }

    #endregion // INotifyPropertyChanged Members

    #region Debugging Aides

    /// <summary>
    /// Warns the developer if this object does not have
    /// a public property with the specified name. This
    /// method does not exist in a Release build.
    /// </summary>
    [Conditional("DEBUG")]
    [DebuggerStepThrough]
    public virtual void VerifyPropertyName(string propertyName)
    {
        // Verify that the property name matches a real,
        // public, instance property on this object.
        if (TypeDescriptor.GetProperties(this)[propertyName] == null)
        {
```

```
            string msg = "Invalid property name: " + propertyName;

            if (this.ThrowOnInvalidPropertyName)
                throw new Exception(msg);
            else
                Debug.Fail(msg);
        }
    }

    /// <summary>
    /// Returns whether an exception is thrown, or if a Debug.Fail() is used
    /// when an invalid property name is passed to the VerifyPropertyName method.
    /// The default value is false, but subclasses used by unit tests might
    /// override this property's getter to return true.
    /// </summary>
    protected virtual bool ThrowOnInvalidPropertyName { get; private set; }

    #endregion // Debugging Aides
}
```

In addition to the INotifyPropertyChanged methods, there is also a debug method to validate the PropertyName. This is because the PropertyChange notification gets passed in as a String, and I have caught myself forgetting to change this string when I change the name of a Property.

Note: The PropertyChanged notification exists to alert the View that a value has changed so it knows to update. I have seen suggestions to drop it from the Model and to expose the Model's properties to the View from the ViewModel instead of the Model, however I find in most cases this complicates things and requires extra coding. Exposing the Model to the View via the ViewModel is much simpler, although either method is valid.

Sample ViewModel

I am doing the ViewModel next because I need it before I can create the View. This should contain everything the User would need to interact with the page. Right now it contains 4 properties: a ProductModel, a GetProduct command, a SaveProduct command, an a ProductId used for looking up a product.

```
public class ProductViewModel : ObservableObject
{
    #region Fields

    private int _productId;
    private ProductModel _currentProduct;
    private ICommand _getProductCommand;
    private ICommand _saveProductCommand;

    #endregion

    #region Public Properties/Commands

    public ProductModel CurrentProduct
    {
        get { return _currentProduct; }
```

```
        set
        {
            if (value != _currentProduct)
            {
                _currentProduct = value;
                OnPropertyChanged("CurrentProduct");
            }
        }
    }

    public ICommand SaveProductCommand
    {
        get
        {
            if (_saveProductCommand == null)
            {
                _saveProductCommand = new RelayCommand(
                    param => SaveProduct(),
                    param => (CurrentProduct != null)
                );
            }
            return _saveProductCommand;
        }
    }

    public ICommand GetProductCommand
    {
        get
        {
            if (_getProductCommand == null)
            {
                _getProductCommand = new RelayCommand(
                    param => GetProduct(),
                    param => ProductId > 0
                );
            }
            return _getProductCommand;
        }
    }

    public int ProductId
    {
        get { return _productId; }
        set
        {
            if (value != _productId)
            {
                _productId = value;
                OnPropertyChanged("ProductId");
            }
        }
    }

    #endregion

    #region Private Helpers

    private void GetProduct()
    {
        // You should get the product from the database
        // but for now we'll just return a new object
```

```
        ProductModel p = new ProductModel();
        p.ProductId = ProductId;
        p.ProductName = "Test Product";
        p.UnitPrice = 10.00;
        CurrentProduct = p;
    }

    private void SaveProduct()
    {
        // You would implement your Product save here
    }

    #endregion
}
```

There is another new class here: the RelayCommand. This is essential for MVVM to work. It is a command that is meant to be executed by other classes to run code in this class by invoking delegates. Once again, I'd recommend checking out the MVVM Light Toolkit's (http://mvvmlight.codeplex.com/) version of this command when you are more comfortable with MVVM, but I wanted to keep this simple so have included this code here.

```
/// <summary>
/// A command whose sole purpose is to relay its functionality to other
/// objects by invoking delegates. The default return value for the
/// CanExecute method is 'true'.
/// </summary>
public class RelayCommand : ICommand
{
    #region Fields

    readonly Action<object> _execute;
    readonly Predicate<object> _canExecute;

    #endregion // Fields

    #region Constructors

    /// <summary>
    /// Creates a new command that can always execute.
    /// </summary>
    /// <param name="execute">The execution logic.</param>
    public RelayCommand(Action<object> execute)
        : this(execute, null)
    {
    }

    /// <summary>
    /// Creates a new command.
    /// </summary>
    /// <param name="execute">The execution logic.</param>
    /// <param name="canExecute">The execution status logic.</param>
    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        if (execute == null)
            throw new ArgumentNullException("execute");

        _execute = execute;
        _canExecute = canExecute;
    }
```

```
    #endregion // Constructors

    #region ICommand Members

    [DebuggerStepThrough]
    public bool CanExecute(object parameters)
    {
        return _canExecute == null ? true : _canExecute(parameters);
    }

    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }

    public void Execute(object parameters)
    {
        _execute(parameters);
    }

    #endregion // ICommand Members
}
```

Sample View

And now the Views. These are DataTemplates which define how a class should be displayed to the User. There are many ways to add these templates to your application, but the simplest way is to just add them to the startup window's Resources.

```
<Window.Resources>
    <DataTemplate DataType="{x:Type local:ProductModel}">
        <Border BorderBrush="Black" BorderThickness="1" Padding="20">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition />
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>

                <TextBlock Grid.Column="0" Grid.Row="0"  Text="ID" VerticalAlignme
                <TextBox Grid.Row="0" Grid.Column="1"  Text="{Binding ProductId}"

                <TextBlock Grid.Column="0" Grid.Row="1"  Text="Name" VerticalAlign
                <TextBox Grid.Row="1" Grid.Column="1"  Text="{Binding ProductName}

                <TextBlock Grid.Column="0" Grid.Row="2"  Text="Unit Price" Vertica
                <TextBox Grid.Row="2" Grid.Column="1"  Text="{Binding UnitPrice}"

            </Grid>
        </Border>
    </DataTemplate>
```

```
    <DataTemplate DataType="{x:Type local:ProductViewModel}">
        <DockPanel Margin="20">
            <DockPanel DockPanel.Dock="Top">
                <TextBlock Margin="10,2" DockPanel.Dock="Left" Text="Enter Product

                <TextBox Margin="10,2" Width="50" VerticalAlignment="Center" Text=

                <Button Content="Save Product" DockPanel.Dock="Right" Margin="10,2
                        Command="{Binding Path=SaveProductCommand}" Width="100" />

                <Button Content="Get Product" DockPanel.Dock="Right" Margin="10,2"
                        Command="{Binding Path=GetProductCommand}" IsDefault="True
            </DockPanel>

            <ContentControl Margin="20,10" Content="{Binding Path=CurrentProduct}"
        </DockPanel>
    </DataTemplate>
</Window.Resources>
```

The View defines two DataTemplates: one for the ProductModel, and one for the ProductViewModel. You'll need to add a namespace reference to the Window definition pointing to your Views/ViewModels so you can define the DataTypes. Each DataTemplate only binds to properties belonging to the class it is made for.

In the ViewModel template, there is a ContentControl that is bound to ProductViewModel.CurrentProduct. When this control tries to display the CurrentProduct, it will use the ProductModel DataTemplate.

Starting the Sample

And finally, to start the application add the following on startup:

```
MainWindow app = new MainWindow();
ProductViewModel viewModel = new ProductViewModel();
app.DataContext = viewModel;
app.Show();
```

This is found in the code behind the startup file – usually App.xaml.cs.

This creates your Window (the one with the DataTemplates defined in Window.Resources), creates a ViewModel, and it sets the Window's DataContext to the ViewModel.

And there you have it. A basic look at MVVM.

UPDATE
Sample code can be found here (http://www.mediafire.com/?2a8gwjtq58hsj86).

Notes

There are many other ways to do the things shown here, but I wanted to give you a good starting point before you start diving into the confusing world of MVVM.

The important thing to remember about using MVVM is your Forms, Pages, Buttons, TextBoxes, etc (the Views) are NOT your application. Your ViewModels are. The Views are merely a user-friendly way to interact with your ViewModels.

So if you want to change pages, you should not be changing pages in the View, but instead you should be setting something like the AppViewModel.CurrentPage = YourPageViewModel. If you want to run a Save method, you don't put that behind a button's Click event, but rather bind the Button.Command to a ViewModel's ICommand property.

I started with Josh Smith's article on MVVM (http://msdn.microsoft.com/en-us/magazine/dd419663.aspx), which was a good read but for a beginner like me, some of these concepts flew right over my head.

I've never done a blog or tutorial before, but I noticed there is a lot of confusion about what MVVM is and how to use it. Since I struggled through the maze of material online to figure out what MVVM is and how its used, I thought I'd try and write a simpler explanation. I hope this clarifies things a bit and doesn't make it worse 🙂

>> Next – Navigation with MVVM (https://rachel53461.wordpress.com/2011/12/18/navigation-with-mvvm-2/)

This entry was posted on Sunday, May 8th, 2011 at 2:24 am and is filed under MVVM. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

71 Responses to *A Simple MVVM Example*

Tanya says:
June 6, 2016 at 10:44 am
Thank you for one of the best descriptions of the MVVM.

Reply
Ricardo says:
April 9, 2016 at 4:42 pm
Hi rachel,

I'm learning MVVM and I noticed that you instanciate the Model in the view. Is that a neat implementation of the MVVM?

How? Can you explain?

Regards,

Reply
Rachel says:
May 12, 2016 at 3:58 pm

Hi Ricardo, typically I only initialize the main AppViewModel or ShellViewModel in the OnStartup code of the application. Any other ViewModel or Model creation is done from within the ViewModels somewhere. I do not ever recommend creating Models/ViewModels from the View layer except for this one instance in AppStartup. Hope that helps explain things!

Reply

Andrew Simpson says:
May 13, 2016 at 5:37 am
Hi Rachel. You were kind enough to reply to my SO question(s). I have followed this example – bu, I am not interested (at this stage) of using the ViewModel to manage my navigation. i am interested in having multiple ViewModels to manage my entire application. For this to work – as it does not yet do I still have to include the DataTemplates you have declared in ApplicationView to make it work? Thanks

Reply

Rachel says:
May 13, 2016 at 2:38 pm
Hi Andrew, DataTemplates are just a way to tell WPF how to draw objects in the UI. For example, if I have a CustomerModel inserted in the UI via something like a ListBox or ContentControl, I can tell WPF to draw it using the CustomerDataTemplate. If you like, I'll be in the WPF chat room on stack overflow for a bit and you can stop by there and ask any questions you have.

Andrew Simpson says:
May 14, 2016 at 9:23 am
By the way thanks for all your help. it is really good of you to give up so much of your time

Saltyy says:
February 5, 2016 at 12:55 am
Very informative and useful code for staring MVVM design. I'll be using it as a starter for a new project at work.

Reply

dbnex B says:
December 24, 2015 at 3:53 am
This, by far, is the best MVVM/WPF explanation I have read so far. I have one question. When I run your example, all works; however, following your example, I wrote my own (since I have to do it to learn it) and I have a problem I can't figure out why it is happening.

When I enter a product ID button Get Product is enabled, when I hit it, nothing happens. I traced the code and I found that this call in ObservableObject.OnPropertyChanged:

PropertyChangedEventHandler handler = this.PropertyChanged;
handler is always null

, is always ending with handler == null.

Same will happen in your example IF I put breakpoint in OnPropertyChanged method. If I dont, then your example will show the CurrentProduct.

What could be causing my behavor to not show Current Product details?

Why if I debug OnPropertyChanged, your example will not show it either?

Much appreciated

Reply

Rachel says:
January 20, 2016 at 4:01 pm
Hi dbnex, I'd recommend posting a question on http://www.stackoverflow.com with the relevant code and an explanation of the problem. You should get an answer quickly, but if not feel free to leave me a link to your question and I'll try to take a look when I have a minute. Goodluck!

Reply

Piotr Kundu says:
October 19, 2015 at 11:40 am
Rachel, am I getting this right? You use "INotifyPropertyChanged" to notify the ViewModel (VM) that properties in the Model (M) have changed just the same way you notify the View (V) of changes in VM??

Reply

Rachel says:
October 22, 2015 at 3:30 pm
Yes, INotifyPropertyChanged should be implemented anywhere you want change notifications. So if you want to bind an object to the View, it should implement INotifyPropertyChanged. And if you want another class (such as a VM) to do something when a property changes on another class (such as a M), that class (M) should also implement INotifyPropertyChanged so the first class (VM) can hook up to the PropertyChanged event.

Reply

Sunil says:
August 3, 2015 at 10:00 am
Dear Rachel,

I have read many articles but I could not understand MVVM.
But your article is simple and straight forward to understand the concept of MVVM. Thank you very much.

Reply

Harry Moloney says:
May 2, 2015 at 7:22 am
I also would like to thank you! Am new to mvvm and wpf. Have a question regarding the models. Say I have a customer model with the following fields, FirstName, LastName, Address, Town, Country. On the form the last two fields Town and Country require their own table and need to be displayed in a combobox. Obviously the list of towns or countries is not part of the model, yet the comboboxes must appear on the same form but are not part of the model. Not sure how to deal with this with mvvm.

Reply

Rachel says:
May 11, 2015 at 3:04 pm
Hi Harry, typically the ItemsSource for that dropdown would bind to something other than the current data model, such as a static resource collection, or the a property on the DataContext on an item further up on the View.

Reply

Silvia Sarte says:
April 7, 2015 at 5:43 pm
I would like to say thank you! thank you!
I was looking for some example on how to navigate between views, but i can't find a good example.
I'm having a hard time doing this mvvm thing but i find your blog easy to understand for beginners like me.
thank you!

Reply

Manvendra Singh says:
March 1, 2015 at 10:22 am
Hi Ma'am,
First of all thank you for this ultimate example and description.
This was the best example of MVVM I've found on the net and after searching for 2 to 3 days, I can say that this example gets you to "up an running" in the best way possible. And It is one of the very few examples which runs properly as many of the examples on the web don't run and add to confusion.

In your example I have two confusions.
->this may be a silly point but what is productview.xaml and how is this used.
->I can't see a link between productview.xaml and mainview.xaml.
->why are we binding to both model and view model in productview.xaml(what I read in basics is that you bind a viewmodel to a view..ie view model represents a view)

I am new to MVVM and desktop application development so I might be missing some points.

Thank You Again.

Reply

Rachel says:
August 24, 2015 at 2:13 pm
Hi Manvendra,

In the SampleCode, ProductView.xaml is just a ResourceDictionary that contains the templates for how to draw both the ProductModel and ProductViewModel objects. If you look at the code in App.xaml, you'll see a line where it is including the resources (styles, templates, etc) found in ProductView.xaml in the application. With these templates, we can insert ProductModel or ProductViewModel objects directly into the UI, and WPF will draw them using the templates defined in ProductView.xaml.

It is just one way of keeping the View code separated into individual xaml files, which makes it easier to maintain or debug code later on.

Thanks,
Rachel

Reply

Richard Barrs says:
February 22, 2015 at 1:13 am
I would like to add my thanks to those before me for your blog posts, Rachel.

Today has been one of those frustrating learning days…it was neat, and a little energizing, to read you were also homeschooled, and mostly self-taught.

Well, back to rebuilding what I thought was my MVVM app, but is apparently more like a weird, quasi, half-breed M-VM model.

Thanks again!

Reply
Einar Adolfsen says:
August 28, 2014 at 1:26 pm
Thanks for a great article, which help me getting up to speed implementing MVVM Pattern.

I consulted the prism MVVM online documentation(Developer's Guide to Microsoft Prism Library 5.0 for WPF) chapter 5, the first figure illustrates the MVVM, in this diagram model is responsible for data as well as business logic (and validation). I thinks the ViewModel is just the glue between the UI and the Model. The Model represent the model in the real world(domain), and should contain data and business logic in my opinion.

It is just a pattern after all, I'm sure there is many views on how to implement it.

Reply
Jeremy Wilkins says:
August 24, 2015 at 4:06 pm
Rachel indicated that usually she would implement the IDataErrorInfo interface on the model code which is the correct implementation and does move validation into the Model code where it belongs. Business logic also belongs within the model code for similar reasons.

The model is where you want your validation to occur because many different ViewModels may manipulate the same model objects affecting your data. You want your model to behave appropriately when invalid data is being sent regardless of the source ViewModel otherwise you will risk two scenarios:

1) You are duplicating the model validation or business logic in multiple ViewModels (breaks the "do not repeat yourself" [DRY] principle)

2) You risk forgetting to copy that code and some ViewModel can now break your model and now your data is compromised. (Not very secure or repeatable)

Reply
Jeremy Wilkins says:
August 24, 2015 at 4:20 pm
After careful review I see what you mean. I disagree with Rachel regarding where the business rules should be. The Microsoft documentation is correct about where business rules belong.

Reply
Tung Nguyen Thanh says:
August 17, 2014 at 3:53 pm
Great introduction about MVVM pattern. Thanks for sharing.

Reply
Michael Garrison says:
March 27, 2014 at 12:23 pm
Great blog, Rachel; your site's one of my first go-to places!

A question for you — I'm using DataTemplates in the app.xaml file to associate my views to my viewmodels, and so far everything's working great. The question is, is it possible to do a similar approach for my application view/viewmodel? (For the app, I'm setting the datacontext in the application's shell to the application's viewmodel, similar to how you do yours.) Just curious if there's a way I could do it so that *all* views and viewmodels use the same approach — I'd the application view not being the one exception.

Again, thanks for all the information — it's really been very helpful!

Reply

Rachel says:
August 24, 2015 at 2:22 pm
Hi Michael,

I'm not sure, but I would guess not. I have never really bothered to look as it made sense to me to set the highest-level DataContext to initialize the data for the application, and I've always been a person who believes design guidelines are not strict rules that must be followed at all times. It didn't seem worth the effort to investigate such a thing.

Glad my site has helped you though!

Regards,
Rachel

Reply

Jon Wells says:
February 26, 2014 at 10:07 am
Really useful read, thanks. I've not been working with WPF/MVVM for long and this helped cement some of the fundamentals

Reply

Hal says:
October 10, 2013 at 1:10 pm
Hey Rachel

I learning MVVM toghether with WPF, as the only way to go with it. I have a question. I plan to do database connected application that use Entity Framework (base first) model.

Please tell me whats the best way to wire up entity framework model to viewmodel?

Should I just simply wire up entity model in constructor of ViewModel?

Like in that example?

```
//constructor of ViewModel
public ViewModel()
{
db = new PackageShipmentDBEntities(); // Entity Framework generated class

ListFromDb = new ObservableCollection(db.Packs.Where(w => w.IsSent == false));
}
```

Or this approach is not the best?

Thanks for the answer.

Reply

Ronald says:
August 20, 2013 at 4:31 pm
Hi Rachel. What you say about Models and ViewModels in the beginning of your article looks to me to sharply contradict what I've read elsewhere. It appears that either you or they have gotten the two mixed up. What I've read so far has it that the ViewModel is the one that "should only contain properties and property validation" and the Model is the one that contains all the business logic–the opposite of what you explain. Have I just missed something here?

Reply

Rachel says:
August 22, 2013 at 4:52 pm
Hi Ronald,

Models are typically either plain data models (think of something like a database row), or they are domain models which represents the state of something at run time. They usually only contain properties and basic validation related to that single model object itself. They are the building blocks used by the rest of the code when constructing the application.

ViewModels are meant to be code representations of the Views. This means if the View needs a Name property, the ViewModel may contain a string property called Name, although that string property is probably stored on a Model inside the ViewModel. So although the ViewModel may contain the properties the View needs, it usually will be using a Model object to store those values. ViewModels also contain more advanced validation, such as checking if the user has access to something, or verifying that an item is unique.

It should be noted here that some people prefer to return the actual Model object to the view in its entirety, such as I have, rather than create separate properties on the ViewModel that get/set properties on the Model. This often is faster to code and more practical, although it is not the "MVVM purist" approach. Both methods are acceptable when using MVVM.

Reply

Jeremy Wilkins says:
August 24, 2015 at 4:17 pm
This is where I do disagree Rachel. Models should contain your business and validation logic otherwise you risk compromising your Model when many ViewModels come into play on the same Models.

Certainly for small applications you can "get away" with having business logic in your ViewModel, but that will quickly fall apart once your model gets used a lot more.

I do agree that general UI level input validation and reporting can occur in the ViewModel, but it should never be left up to the ViewModel to enforce the business rules.

$('#Anil') (@NoobDeveloper) says:
May 1, 2013 at 9:34 am
I am learning WP development and wanted to try out MVVM in a sample app. This blog post helped me tremendously because of its simplicity. I extended my app to use repository pattern in model.

Reply

Scott says:
April 16, 2013 at 4:48 pm
Hello Rachel,

Very nice description of a simple MVVM, but I'm still confused. I'm *very* new to C# and WPF, having done mostly web-based applications.

I'm still confused as to how you are affecting changes with this pattern. I've gone through other tutorials describing this method and they seem to leave out the piece I'm not understanding.

When you change the ProductId from '0' to '1' the "Get Product" button becomes active. How? When you click on the "Get Product" button the ProductModel DataTemplate appears. How does it appear? The application works as described but I'm completely stumped as to the how.

It's clear to me that when you change the data in the "TextBox"es an event is triggered, but just can't see what else the delegates are doing.

Thanks much!

Reply

Rachel says:
April 16, 2013 at 6:18 pm
Hi Scott,

When you bind a DependencyProperty in WPF, you are telling the property to look somewhere else to get it's value instead of looking for the value locally on the object. So when you bind a TextBox.Text to a ProductId, you are telling WPF that it should go look at the ProductId property of whatever the TextBox's data layer is anytime the Text property is needed. (I have a blog post about the data layer here if you're interested in learning more about it)

Most bound objects implement the INotifyPropertyChanged interface, which WPF's binding system uses to know when a bound value has changed and needs to be updated. For example, changing the ProductId from 0 to 1 will trigger a PropertyChanged notification for the ProductId property, and any bindings that rely on that value get automatically re-evaluated and updated.

The RelayCommand I am using for the Get Product button is a special type of command that automatically re-evaluates CanExecute() anytime a property changes, so when ProductId changes, the GetProductCommand re-evalutes it's CanExecute method, which toggles the change of the button's Enabled properties (Buttons that are bound to a Command will automatically bind their IsEnabled to the Command.CanExecute).

Clicking the button merely populates the CurrentProduct property, which is bound to a ContentControl's Content property in the UI. When WPF receives notification that the CurrentProduct property has changed, it re-evaluates the binding and tries to draw the CurrentProduct object in the UI. Because I have an implicit DataTemplate defined in the Resources (A DataTemplate that has a Type, but not a Key), WPF will automatically use that DataTemplate when trying to draw an object of type Product.

Hope that helps clarify things, and that I didn't just confuse you more =)

Rachel

Reply

Scott says:
April 16, 2013 at 7:27 pm

Wow! Thank you! It definitely cleared things up for me. I wonder though, what is determining that a button Command.Execute happens on click and not, say, on hover or on focus?

Rachel says:
April 16, 2013 at 8:18 pm
WPF actually evaluates Command.CanExecute when the control is first rendered, and that determines if the button is Enabled or not.

I don't even think CanExecute is checked when you click on a Button that is bound to a Command, unless you perform the check manually.

The RelayCommand I used will re-evaluate CanExecute anytime a property change notification is raised, and that will alert any bindings to the command that they need to reevaluate.

There are other commands which behave differently. One example is Microsoft Prism's DelegateCommand, which does not automatically re-evaluate the CanExecute when any property changes (this is better on performance). To use their command, you have to re-evaluate CanExecute manually when a property you're interested in changes

Scott says:
April 7, 2013 at 3:24 pm
Late to the MVVM game. Your work shared here is wonderful. I was in the middle of my own DataTemplate vs UserControl war. You (and one other) have convinced me, unless I have some unusual data presentation or needed extension, I should stick with DataTemplates. Love your by-line. You've certainly helped me understand.

Reply
oscar says:
March 19, 2013 at 3:21 pm
Hello Rachel,
Thanks, your example helped me understand better the MVVM !

Reply
Ghufran says:
February 12, 2013 at 12:06 pm
Hi Rachel please guide me how to Handle Drag and drop in MVVM pattern, I have to drag usercontrol

Reply
Rachel says:
February 13, 2013 at 9:16 pm
Hi Ghufran, DragDrop is a UI-specific operation so should probably be handled by the UI code somewhere, such as by a custom UserControl or in the Code-Behind the UI.

If you Google "WPF Drag Drop" you'll probably find plenty of examples, although I personally like this blog article for dragging and dropping data bound items: http://www.zagstudio.com/blog/488

Good luck with your project!

Reply
Azhar says:

February 2, 2013 at 8:47 am
Really a nice Introduction to MVVM, I have started using MVVM after reading this

Reply
Chau Chee Yang says:
January 22, 2013 at 3:09 am
You state the business rules should code in view models instead of model. I personally feel that this is incorrect. View models should contain presentation logic which is specific to that particular application instance

Reply
Séddik Laraba says:
January 2, 2013 at 12:59 pm
very nice, mvvm is more clear to me now, thank u

Reply
Dipak says:
December 25, 2012 at 10:52 am
Rachel , Great post. Tell me is it necessary for create Relay command class? if yes why? and there is another method without using relay command class?

Reply
Rachel says:
December 29, 2012 at 8:00 pm
Hi Dipak,

The RelayCommand class uses delegates for Execute and CanExecute, so you can easily build Commands in your ViewModel without having to write a new command definition every time you want to add a Command to your ViewModel.

Reply
mrothaus says:
November 17, 2012 at 1:21 pm
Rachel, great post. Tell me how you would handle this situation: You have another property in ProductModel, "bool OnSale". In your view, you want to display "Yes" if OnSale=true and "No" if OnSale is false. What I've been doing is just making another property in the ProductModel called OnSaleDisplay which is a string with just a "get" and no "set". This works well because I can raise a notification for OnSaleDisplay when OnSale changes (with an OnPropertyChanged("OnSaleDisplay") within the OnSale property.

However, after reading your post, I suspect OnSaleDisplay doesn't belong in ProductModel. How would you do this and who is responsible?

Reply
Rachel says:
November 17, 2012 at 7:07 pm
Hi mrothaus, I would actually create a converter in the View that converts the boolean value to a Yes/No value since this seems like something only the View would care about, and the converter could be re-used in other places where you want to convert a boolean to a Yes/No value

Reply
Kasper says:
November 11, 2012 at 5:28 pm

Good work! I would just like to add that in my models, I _do_ have business logic, validation etc. The reason is that I can then use my models stand alone, e.g. in apps where there is no view. The model knows how to persist data, usually through a repository interface.

Reply

Priyadarshini Soman says:
October 23, 2012 at 9:58 am
Thank you for this awesome post, This is just the thing I needed today

Reply

Tech Guido (@TechGuido) says:
October 14, 2012 at 12:26 am
Rachel, thanks for a simple explanation of MVVM. The one thing that gets me thinking in circles are reusable controls and view models. For example, if we need to reuse a phone number control over and over again in different views with the same logic behind the control, do we create a view model for this?

I can see how this would work for events, but how about the data itself that it uses? Please share your thoughts.

Reply

Rachel says:
October 15, 2012 at 4:52 pm
I usually have two kinds of UserControls: Very generic ones with no ViewModel at all that get passed any values they need via DependencyProperties, and more complex UserControls that are built to go with a specific ViewModel, and that ViewModel is expected to be used as the DataContext when the UserControl is used.

I'm a bit unclear about where your PhoneNumber example would land since I'm not sure what sort of functionality you're thinking a PhoneNumber would have.

If you want to use a UserControl as nothing more than a generic way to display the phone number, such as having 3 TextBoxes to separate out the phone number sections, then I'd probably do the first case of building a PhoneNumberUserControl and have a DependencyProperty for PhoneNumber that is a string that can be passed to the UserControl.

If you're doing something more complex, such as supporting various phone number types like international numbers, extensions, etc, and adjusting the View based on the type of number it is, I'd probably build a PhoneNumberViewModel and build my UserControl with the expectation that the DataContext will be of type PhoneNumberViewModel. Then the actual application classes would have to use a PhoneNumberViewModel anywhere they want to represent a phone number, and the View would tell WPF to draw the PhoneNumberViewModel using the PhoneNumberUserControl.

I hope I understood your question correctly and didn't confuse you more than you already were

Reply

Tech Guido (@TechGuido) says:
October 18, 2012 at 2:51 am
Yes that makes perfect sense using another View Model to represent the data rules of a user control such as a complex phone number control. I appreciate your input. Keep up the good work blogging. =)

Krishna says:
September 15, 2012 at 11:35 am
Good one. But can you please let me know who is listening to the PropertyChangedEvent and how the datacontext is refreshed while changing a property in ViewModel?

Reply

Rachel says:
September 15, 2012 at 9:38 pm
WPF's binding system listens for property change notifications (so a <TextBox Text="{Binding SomeProperty}" /> will re-evaluate SomeProperty for the Text anytime it hears a PropertyChange notification where e.PropertyName == "SomeProperty"), and you can also hook your own handlers to the PropertyChange notifications and do some custom logic yourself anytime a property changes.

Reply

Shan says:
August 16, 2012 at 9:27 pm
I changed the ObservableObject class from "public abstract class ObservableObject to be "public class ObservableObject" then it worked.

Reply

Shan says:
August 15, 2012 at 10:26 pm
Hi Rachel,

It is a good article and easy to understanding for beginner such as me. I am building a Windows phone application. I followed your article to create the class. However, I got the erorr during deserialization. The error is "The type 'CMSPhoneApp.ObservableObject' cannot be deserialized in partial trust because it does not have a public parameterless constructor." Would you have any suggestion how can I solve it?

Reply

steve says:
July 13, 2012 at 3:52 pm
Good article although too much fluff eg:
RaisePropertyChanged(string propertyName) is never called so can be omitted and in the OnPropertChanged method there is no need to create a new handler reference, you could just use:

```
if (PropertyChanged != null) {
var e = new PropertyChangedEventArgs(propertyName);
PropertyChanged(this, e);
}
```

But overall a very good demo of Mvvm

Reply

Rachel says:
July 13, 2012 at 7:33 pm
You're right, some of this stuff is unnecessary. I never really noticed it until you pointed it out because it doesn't really harm anything, and some of it is actually useful.

I'll clean it up a bit though, thanks for the feedback

Reply

manish says:
June 5, 2012 at 6:13 am
This looks great.. Thanks for simpligying it. Looking forward for more explanations on other UI patterns..

Reply
RK76 says:
May 30, 2012 at 10:52 am
Reblogged this on Silverlight.

Reply
Antoine Jeanrichard says:
February 29, 2012 at 2:18 pm
Hi Rachel,
Nice blog with good explanations!

In most of the apps we write at my company, we have views corresponding to instances of classes of the domain model. Therefore, we wrote a framework which makes it easy to use our domain model classes as ViewModels. We do this for all our views, except the apps main views (as stated in your article about navigation with ViewModels).
What do you think about this?
For instance, in the example you wrote above, what would you think about putting the relay commands directly into your model, and therefore using it as datacontext of your view?

Thanks for your ideas!

Antoine

Reply
Rachel says:
February 29, 2012 at 5:38 pm
Hi Antoine,

Glad you enjoyed the article! There are many different frameworks or variations of MVVM, however I personally like to keep my Models as plain old data objects that only contain some raw data validation and PropertyChange notifications, and to put any application or business logic such as Command handlers in ViewModels.

I'll still use Models sometimes as the DataContext for my Views, however I try and keep any application-related functionality out of them to keep the layers separate and make maintenance easier.

```
<DataTemplate DataType={x:Type local:MyModel}">
    <local:MyView />
</DataTemplate>
```

Or

```
<DockPanel>
    <Button Content="Save"
            Command="{Binding SaveCommand}"
            DockPanel.Dock="Bottom" />
    <Grid DataContext="{Binding SomeModel}">
        ...
    </Grid>
</DockPanel>
```

Rachel

Reply

Antoine Jeanrichard says:
April 17, 2013 at 11:21 am
Hi Rachel,
I see this article is still read a lot I have a new need for insight : How do you do when your Data Objects are modified by other threads? Do you manage dispatcher calls in your ViewModels, using CheckAccess or something similar?
Do you do it everytime, for example in your abstract ViewModel class, or do you specifically implement this for the ViewModels that will likely wrap multi-threaded-modified data models?

Rachel says:
April 17, 2013 at 12:22 pm
I try to never create or modify my data objects from another thread. Typically I will get data from another thread, then populate my data objects on the main thread.

Although if I ever did need to modify a data object from a background thread, I would probably use the Dispatcher to pass the work to the main UI thread from my ViewModel.

Michel Keijzers says:
January 26, 2012 at 11:34 am
Thanks for this clear blog … I probably am going to convert my (too big) WPF app into a MVVM app. Till now, I only separated the Model and View, so the ViewModel logic is now shared between the Model and the View.

Reply

Mark Wiskochil says:
November 18, 2011 at 2:43 pm
Hi Rachel, This is a great article, I just have one question. When you run this app, there is textbox validation going on. Where is that coming from. I cannot find textbox validation code anywhere. Thanks in advance.

Reply

Rachel says:
November 18, 2011 at 4:55 pm
Hi Mark, there is some default validation that is built into the default WPF Control Templates. For example, if a TextBox's Text is bound to an int field, and the user enters some letters, it will cause an error since you cannot store text in an int field.

The default validation template is usually just a red border around the control, but this template can be modified. You can also modify the validation logic by making your class implement `IDataErrorInfo`.

Reply

Rohit Khare says:
July 19, 2011 at 6:03 pm
Hi Rachel,

Just a right time. I am planning to scale my old inventory application to a WPF app. and was looking for few easy samples on MVVM. Most of the MVVM explanation on the net uses very cryptic language that is hard to understand for beginners.

Thanks for the simple explanation and little sample.

Reply

B. Clay Shannon says:
June 10, 2011 at 9:09 pm
I think I've got a valid (no pun intended) problem with the code in the article.

I'm getting the err msg, "'SimpleMVVMExample.Model.ProductModel' does not contain a definition for 'IsValid' and no extension method 'IsValid' accepting a first argument of type 'SimpleMVVMExample.Model.ProductModel' could be found (are you missing a using directive or an assembly reference?)"

In the article, "IsValid" only appears once – on that line where I'm getting the err msg. What to do/how to solve this?

Reply

Rachel says:
June 14, 2011 at 3:08 pm
Sorry about that, my original sample had the Model's implementing IDataErrorInfo and doing some validation. I left that out for simplicity, and forgot to remove the IsVaild reference. Thanks for catching that.

Reply

B. Clay Shannon says:
June 10, 2011 at 4:39 pm
Thanks, Rachel!
I've been berating myself for being a doofus, not grokking MVVM. This helps.

Reply

Davros says:
June 10, 2011 at 2:26 pm
This looks great, is there sample code I can download to see it in action?
Thank you.

Reply

Rachel says:
June 13, 2011 at 8:19 pm
Great idea! Life's been a bit busy lately, but I'll see if I can get some sample code on here next weekend.

Reply