

# Understanding the change in mindset when switching from WinForms to WPF

<https://rachel53461.wordpress.com/2012/10/12/switching-from-winforms-to-wpfmvvm/>

One of the biggest hurdles I find WinForms developers struggling with when learning WPF and the MVVM design pattern, is understanding the shift in thought process that is used for WPF/MVVM development.

Here's the best way I can summarize the difference:

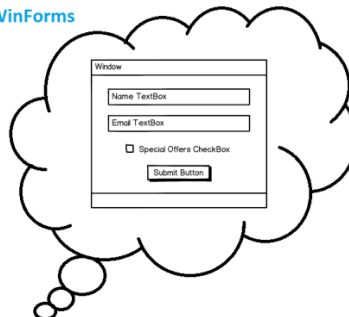
- In WinForms, your forms and UI objects are your application.
- In WPF/MVVM, the class objects you build are your application, while UI objects are nothing more than a user-friendly interface for interacting with your application objects.

## Example

For example, suppose we were asked to make a simple Registration form that takes down a user's Name, Email Address, and asks them if they want to receive Special Offers or not.

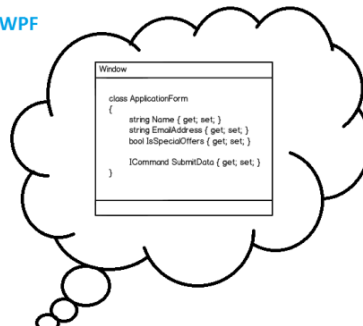
A WinForms developer's thought process might be to think "I need a Window that contains a TextBox for name, TextBox for email, CheckBox for if they want special offers or not, and a Button to submit the form. When the Submit button is pressed, the data is taken from the UI objects, and saved to the database."

WinForms



In contrast, a WPF developer's thought process is "I need a Window that contains a RegistrationForm object. This class needs a string for the Name, a string for the EmailAddress, a boolean property for if they want special offers or not, and a Command to submit the data."

WPF



But of course WPF doesn't know how to draw a class of type `RegistrationForm`, so we'll have to tell it how to draw the `RegistrationForm` object using a `Template`. This template would probably render the `Name` and `EmailAddress` properties using `TextBoxes`, the `IsSpecialOffers` property with a `CheckBox`, and provide a `Button` to run the `SaveDataCommand`, although that doesn't have to be the case and is something that can be figured out later.

With WPF, the UI layer and the application layer are so completely separated that you don't actually need the UI layer at all. If you wanted, you could run your application entirely by test scripts, or through a console window. The actual application layer never needs to reference any UI object to get its data.

## Using this concept on a larger scale

In fact, this thought process is used on a much broader scale for the entire WPF/MVVM application.

Want a `LoginWindow` to come up, and on successful login switch to the `MainWindow`? No problem, simply start the program with your `LoginClass`, and on successful `Login` return the `ApplicationClass`.

Want to have an application with separate `Windows` for `Products`, `Orders`, and `Customers`? No problem, create your `ApplicationClass` to hold a `List<T>` of available "Window" objects containing the classes representing each "Window", and include a `SelectedWindow` property which identifies which window object is currently active.

Of course, WPF needs to know how to draw these class objects, so you'll need to tell it to use a specific `DataTemplate`, `UserControl`, `Page`, or maybe even `Window` when rendering these objects.


## Summary

This is not meant to be an in-depth look at how WPF is different from `WinForms` or how to switch from `WinForms` to WPF, but is more of a brief summary on the change in mindset needed when moving from `WinForms` to WPF.

With WPF, your application consists of the objects you create, and you use `DataTemplates` and UI objects to tell WPF how to draw your application components. That's the opposite of `WinForms` where you build your application out of UI objects, and then supply them with the data needed.

Once you grasp this key difference, WPF becomes much easier to understand and work with.

## 13 Responses to Understanding the change in mindset when switching from WinForms to WPF

1.  Aaron Gasaway says:  
[March 7, 2015 at 7:59 pm](#)  
Old post, but i'm just now seeing it.

I think this article summarizes the essential difference between MVVM and `Winforms` very nicely. It also points out why so many coming from a `Winforms` background have trouble with WPF. Most of the code examples out there are MVVM-oriented in nature, and MVVM assumes a division of labor (or at least process) between building the UI controls and the data objects they display.

Unfortunately, in the business world, there is rarely such division of labor between creating the UI and the data. I know plenty of app developers who develop GUI's and program, but don't know a single graphic artist that develops GUIs without programming. I'm sure in the world of video games and sleek commercial gui's, there are some. Yet in the world of finance & insurance I inhabit, there are next to none.

My world is where the majority of developers live and toil. As I write now in the first part of 2015, WPF is still firmly the platform of choice for desktop LOB apps or those deployed to a Citrix server. Few in the corporate world feel comfortable with greenfield winforms projects; Win RT/Metro is not an option and won't be for some time (businesses are slow to adopt new versions of Windows—especially ones like Windows 8 that mean a big change for users). What would be nice is if some of the XAML crowd that champion MVVM also opened their eyes to the needs of the majority of businesses. It's a crowd that Winforms and ASP.NET seem to respect and cater to. By comparison, I could not, for example, locate a single complete, functional example of two-way binding to a WPF datagrid with dynamically-generated columns.

Is that just not possible using MVVM? If not, what good is MVVM to businesses such as mine? 'Name' and 'EmailAddress' will nearly always be coming from database fields; hardcoding them as string properties on window-by-window basis seems not only overly time-consuming, but a maintenance nightmare..

Sorry if this comes across as a rant—it's not meant to be. I've found your articles to be some of the best and most accessible for WPF newbs like me. It's just frustrating to hear MVVM preached in every post out there, and the old Winforms approaches villified, and then not find a single MVVM solution that seem to address the most fundamental needs.

Regards,

Aaron

[Reply](#)



2. *Fred says:*

[December 13, 2014 at 10:56 pm](#)

I know this is a rather old post at this point and I'm waaay late to the party. But it is a misnomer to say "WinForm programmers struggle".

A better phrase would be "un-disciplined programmers struggle". I say that because there is nothing about WinForms that prevents a programmer from using MVVM and MVVM wasn't invented alongside WPF.

I've been designing my apps using MVVM since the late 90's when I was still doing VB 6 apps. The fact of the matter is that regardless of the technology or language being used you will have programmers who employ best practices and those who don't.

The ones who struggle are those who never used those best practices and moved to a technology that strongly enforces certain design principles (i.e. WPF enforcing an MVVM approach). But there are plenty of us who moved to WPF and it wasn't a struggle at all. Quite

the contrary, it felt like a homecoming

[Reply](#)



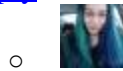
3. *K Aravind says:*

[March 7, 2014 at 4:37 am](#)

Hi guys I am new to windows forms and also WPF sorry if my question is noob....I wanted to create effects like that of windows 8 tiles and transitions and the rich UI like in windows 8 but i didnt how to do it when i googled it told me that u can metro styles which provide windows 8 kind effects..How can i do it..?? neither do i have much knowledge in windows forms nor in

WPF so please help me out..Which one should i start learning..?? Should i start with WPF or Windows Forms...???

[Reply](#)



○ [Rachel](#) says:

[August 24, 2015 at 2:23 pm](#)

You'll probably want to start researching WPF Animations for these kinds of effects

[Reply](#)

4.  [Sangram](#) says:

[September 14, 2013 at 7:36 am](#)

After spending 3 years in Winforms and now a beginner in WPF, it's quite confusing for me to understand how this mvvm model works.

Person who has not worked in Winforms may understand it quickly because every damn i try to do, i tend to think like a winform developer.

Thanks a ton for this awesome article.

[Reply](#)

5.  [John](#) says:

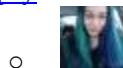
[June 13, 2013 at 5:17 pm](#)

I've been working with MVVVM for a few years now, and I still don't see the big 'why' of it all.

Writing business logic modular and independent of UI has been easy for decades. I've always been able to do it in Windows and Unixes without any trouble in languages from C to Java and C#.

My impression of MVVM is that it is extra froth, 3 files for every 2 that did the same thing the old way. Terms like 'separation of concerns' are presented as if they are new ideas, when modularity has been a rule of good programming since 1970.

[Reply](#)



○ [Rachel](#) says:

[June 15, 2013 at 4:07 pm](#)

Hi John,

The big advantage is how separate the UI and the application are. You can have a team build and test the entire application without the UI at all, and you can have a UI design team build the entire UI without the application data.

Separating data from UI isn't a new concept, however with WPF's binding framework, it makes the separation much easier. MVVM is an easy way to understand and work with that separation.

[Reply](#)

6.  [Adolfo Perez](#) says:

[April 25, 2013 at 6:00 pm](#)

I agree with you, the powerful WPF binding mechanism is what makes this UI/Business Layer separation possible. I have developed many WinForms apps in the past, some using MVP pattern and still found it very limited and troublesome. However, the transition to WPF is not that easy, the first big challenge is, like you said, is start thinking the WPF way.

Another problem i faced is understanding all the things happening behind the scenes like bindings, style inheritance, event bubbling, etc.

[Reply](#)



7. [Mark Freedman](#) says:

[April 2, 2013 at 12:11 pm](#)

The big advantage I see to using WPF and MVVM is that the view-model is actually unit testable! You can test the behavior of the view-model without even creating a view yet. You can even execute your ICommands because those are public properties on the ViewModel. Thanks for your help on SO, by the way.

[Reply](#)



8. [Michael Rushton](#) says:

[October 15, 2012 at 8:29 pm](#)

Well designed Winforms applications can use MVP, a predecessor to MVVM. So the “thought process” can be the same whether WinForm/MVP or WPF/MVVM

[Reply](#)



- o [Rachel](#) says:

[October 17, 2012 at 4:18 pm](#)

Its true, but not everyone knows MVP and I think MVP still has to directly reference UI controls, which is something you don’t need to do with WPF’s binding system.

I like to answer WPF questions on StackOverflow and see a lot of developers there struggling to understand WPF because they’re stuck in a WinForms mindset, so this post was mostly written with them in mind =)

[Reply](#)



9. [Michael Ledin](#) says:

[October 12, 2012 at 8:04 pm](#)

Hi, Rachel! Unfortunately, in real world it’s not so simple...

[Reply](#)



- o [Rachel](#) says:

[October 12, 2012 at 8:11 pm](#)

Hi Michael,

You’re right, the real world is not nearly as simple, however once you understand the different thought process used with WPF & MVVM, it becomes much simpler =)

[Reply](#)

## 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 `NotificationObject` or MVVM Light's `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, and 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 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" VerticalAlignment="Center" />
                <TextBox Grid.Row="0" Grid.Column="1" Text="{Binding ProductId}" />

                <TextBlock Grid.Column="0" Grid.Row="1" Text="Name" VerticalAlignment="Center" />
                <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding ProductName}" />

                <TextBlock Grid.Column="0" Grid.Row="2" Text="Unit Price" VerticalAlignment="Center" />
                <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 Id"
                    VerticalAlignment="Center" />

                <TextBox Margin="10,2" Width="50" VerticalAlignment="Center" Text="{Binding
                    Path=ProductId, UpdateSourceTrigger=PropertyChanged}" />

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

                <Button Content="Get Product" DockPanel.Dock="Right" Margin="10,2"
                    VerticalAlignment="Center"
                    Command="{Binding Path=GetProductCommand}" IsDefault="True" Width="100" />
            </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.

## 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, 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 ??

## Navigation with MVVM

When I first started out with MVVM, I was lost about how you should navigate between pages. I'm a firm believer in using ViewModels to do everything (unless it's View-specific code), and that the UI is simply a user-friendly interface for your ViewModels. I did not want to create a button on a page that has any kind of code-behind to switch pages, and I didn't like the idea of my navigation being spread out throughout all the ViewModels.

I finally came to realize the solution was simple: I needed a ViewModel for the Application itself, which contained the application state, such as the `CurrentPage`.

Here is an example that builds on the [Simple MVVM Example](#).

### The ViewModel

Usually I name the ViewModel `ApplicationViewModel` or `ShellViewModel`, but you can call it whatever you want. It is the startup page of the application, and it is usually the only page or window object in my project.

It usually contains

```
List<ViewModelBase> PageViewModels
ViewModelBase CurrentPage
ICommand ChangePageCommand
```

Here is an example `ApplicationViewModel` that I would use to go with the [Simple MVVM Example](#).

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

    private ICommand _changePageCommand;

    private IPageViewModel _currentPageViewModel;
    private List<IPageViewModel> _pageViewModels;

    #endregion

    public ApplicationViewModel()
    {
        // Add available pages
        PageViewModels.Add(new HomeViewModel());
        PageViewModels.Add(new ProductsViewModel());

        // Set starting page
        CurrentPageViewModel = PageViewModels[0];
    }

    #region Properties / Commands

    public ICommand ChangePageCommand
    {
        get
        {
            if (_changePageCommand == null)
            {
```

```

        _changePageCommand = new RelayCommand(
            p => ChangeViewModel((IPageViewModel)p),
            p => p is IPageViewModel);
    }

    return _changePageCommand;
}

public List<IPageViewModel> PageViewModels
{
    get
    {
        if (_pageViewModels == null)
            _pageViewModels = new List<IPageViewModel>();

        return _pageViewModels;
    }
}

public IPageViewModel CurrentPageViewModel
{
    get
    {
        return _currentPageViewModel;
    }
    set
    {
        if (_currentPageViewModel != value)
        {
            _currentPageViewModel = value;
            OnPropertyChanged("CurrentPageViewModel");
        }
    }
}

#endregion

#region Methods

private void ChangeViewModel(IPageViewModel viewModel)
{
    if (!PageViewModels.Contains(viewModel))
        PageViewModels.Add(viewModel);

    CurrentPageViewModel = PageViewModels
        .FirstOrDefault(vm => vm == viewModel);
}

#endregion
}

```

This won't compile right away because I've made some changes to it. For one, all my PageViewModels now inherit from an IPageViewModel interface so they can have some common properties, such as a Name.

I also created a new HomeViewModel and HomeView since its hard to demonstrate navigation unless you have at least 2 pages. The HomeViewModel is a blank class that inherits from IPageViewModel, and the HomeView is just a blank UserControl.

In addition, I added an `s` to `ProductsViewModel` since it really deals with multiple products, not a single one.

An added advantage to having a `ViewModel` to control the application state is that it can also be used to handle other application-wide objects, such as `Current User`, or `Error Messages`.

## The View

I also need an `ApplicationView` for my `ApplicationViewModel`. It needs to contain some kind of `Navigation` that shows the list of `PageViewModels`, and clicking on a `PageViewModel` should execute the `ChangePage` command.

It also needs to contain a control to display the `CurrentPage` property, and I usually use a `ContentControl` for that. This allows me to use `DataTemplates` to tell WPF how to draw each `IPageViewModel`.

```
1 <Window x:Class="SimpleMVVMExample.ApplicationView"
2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4       xmlns:local="clr-namespace:SimpleMVVMExample"
5       Title="Simple MVVM Example" Height="350" Width="525">
6
7     <Window.Resources>
8       <DataTemplate DataType="{x:Type local:HomeViewModel}">
9         <local:HomeView />
10      </DataTemplate>
11      <DataTemplate DataType="{x:Type local:ProductsViewModel}">
12        <local:ProductsView />
13      </DataTemplate>
14    </Window.Resources>
15
16    <DockPanel>
17      <Border DockPanel.Dock="Left" BorderBrush="Black"
18        BorderThickness="0,0,1,0">
19        <ItemsControl ItemsSource="{Binding PageViewModels}">
20          <ItemsControl.ItemTemplate>
21            <DataTemplate>
22              <Button Content="{Binding Name}"
23                Command="{Binding DataContext.ChangePageCommand,
24                  RelativeSource={RelativeSource AncestorType={x:Type Window}}}"
25                CommandParameter="{Binding }"
26                Margin="2,5"/>
27            </DataTemplate>
28          </ItemsControl.ItemTemplate>
29        </ItemsControl>
30      </Border>
31
32      <ContentControl Content="{Binding CurrentPageViewModel}" />
33    </DockPanel>
34  </Window>
```

In this example, I'm using an [ItemsControl](#) to display my `PageViewModels`. Each item is drawn using a `Button`, and the `Button`'s `Command` property is bound to the `ChangePageCommand`.

Since the `Button`'s `DataContext` is the `PageViewModel`, I used a `RelativeSource` binding to find the `ChangePageCommand`. I know that my `Window` is the `ApplicationView`, and its `DataContext` is the

ApplicationViewModel, so this binding looks up the VisualTree for the Window tag, and gets bound to Window.DataContext.ChangePageCommand.

Also note that I am putting DataTemplates in Window.Resources to tell WPF how to draw each IPageViewModel. By default, if WPF encounters an object in it's visual tree that it doesn't know how to handle, it will draw it using a TextBlock containing the .ToString() method of the object. By defining a DataTemplate, I am telling WPF to use a specific template instead of defaulting to a TextBlock.

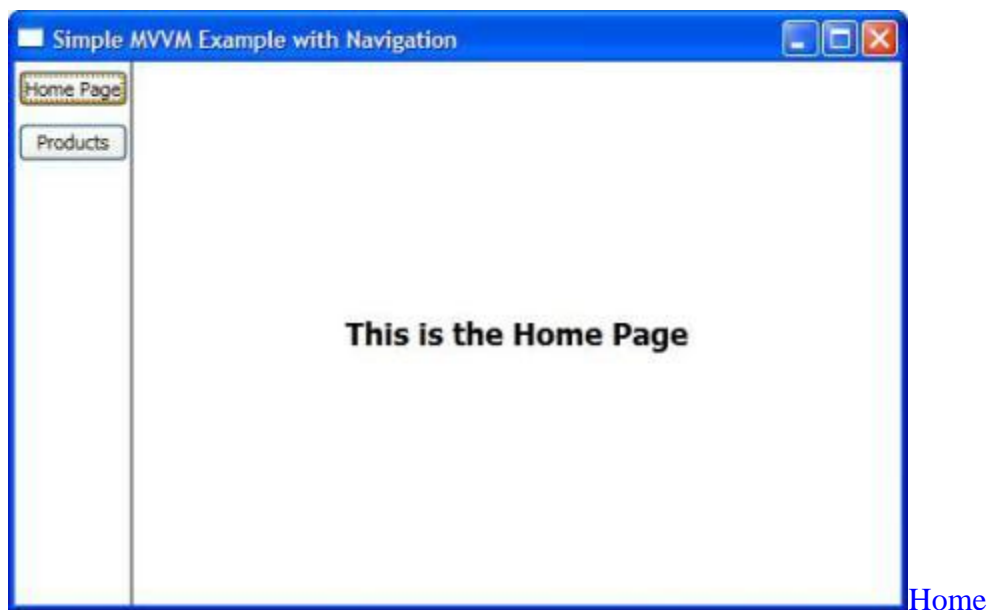
If you are continuing from the Simple MVVM Example, I moved the ProductView out of a ResourceDictionary and into a UserControl to make this simpler.

## Starting the Example

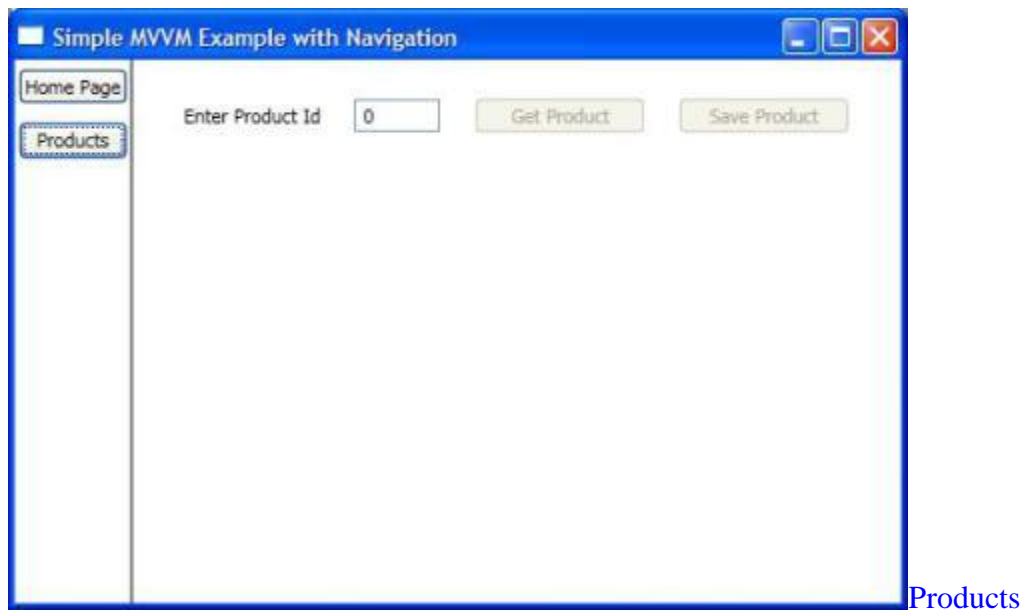
The last thing to do is change App.xaml to make ApplicationView and ApplicationViewModel our startup, instead of ProductView/ProductViewModel.

```
public partial class App : Application
1 {
2     protected override void OnStartup(StartupEventArgs e)
3     {
4         base.OnStartup(e);
5
6         ApplicationView app = new ApplicationView();
7         ApplicationViewModel context = new ApplicationViewModel();
8         app.DataContext = context;
9         app.Show();
10    }
11 }
12 }
```

Run the project and you should see something that looks like the images below, which quickly switches the CurrentPage when clicking on the Navigation buttons.







## Summary

And there you have it. A simple navigation example with MVVM.

You can download the source code for this sample from [here](#).

Once you get more comfortable with WPF, I would recommend looking into using a Messaging System, such as [MVVM Light's Messenger](#), or [Microsoft Prism's EventAggregator](#) to broadcast ChangePage commands from any ViewModel so you wouldn't need to find the ApplicationViewModel to execute the ChangePageCommand, however that's for another day.