Patterns

WPF Apps With The Model-View-ViewModel Design Pattern

Josh Smith

This article discusses:

- Patterns and WPF
- MVP pattern
- Why MVVM is better for WPF
- Building an application with MVVM

This article uses the following technologies:
WPF, data binding

Code download available from the MSDN Code Gallery
Browse the Code Online

Contents

Developing the user interface of a professional software application is not easy. It can be a murky blend of data, interaction design, visual design, connectivity, multithreading, security, internationalization, validation, unit testing, and a touch of voodoo. Considering that a user interface exposes the underlying system and must satisfy the unpredictable stylistic requirements of its users, it can be the most volatile area of many applications.

There are popular design patterns that can help to tame this unwieldy beast, but properly separating and addressing the multitude of concerns can be difficult. The more complicated the patterns are, the more likely that shortcuts will be used later on which undermine all previous efforts to do things the right way.

It is not always the design patterns at fault. Sometimes we use complicated design patterns, which require writing a lot of code because the UI platform in use does not lend itself well to a simpler pattern. What's needed is a platform that makes it easy to build UIs using simple, time-tested, developer-approved design patterns. Fortunately, Windows Presentation Foundation (WPF) provides exactly that.

As the software world continues to adopt WPF at an increasing rate, the WPF community has been developing its own ecosystem of patterns and practices. In this article, I'll review some of those best practices for designing and implementing client applications with WPF. By leveraging some core features of WPF in conjunction with the Model-View-ViewModel (MVVM) design pattern, I will walk through an example program that demonstrates just how simple it can be to build a WPF application the "right way."

By the end of this article, it will be clear how data templates, commands, data binding, the resource system, and the MVVM pattern all fit together to create a simple, testable, robust framework on which any WPF application can thrive. The demonstration program that accompanies this article can serve as a template for a real WPF application that uses MVVM as its core architecture. The unit tests in the demo solution show how easy it is to test the functionality of an application's user interface when that functionality exists in a set of ViewModel classes. Before diving into the details, let's review why you should use a pattern like MVVM in the first place.

Order vs. Chaos

It is unnecessary and counterproductive to use design patterns in a simple "Hello, World!" program. Any competent developer can understand a few lines of code at a glance. However, as the number of features in a program increases, the number of lines of code and moving parts increase accordingly. Eventually, the complexity of a system, and the recurring problems it contains, encourages developers to organize their code in such a way that it is easier to comprehend, discuss, extend, and troubleshoot. We diminish the cognitive chaos of a complex system by applying well-known names to certain entities in the source code. We determine the name to apply to a piece of code by considering its functional role in the system.

Developers often intentionally structure their code according to a design pattern, as opposed to letting the patterns emerge organically. There is nothing wrong with either approach, but in this article, I examine the benefits of explicitly using MVVM as the architecture of a WPF application. The names of certain classes include well-known terms from the MVVM pattern, such as ending with "ViewModel" if the class is an abstraction of a view. This approach helps avoid the cognitive chaos mentioned earlier. Instead, you can happily exist in a state of controlled chaos, which is the natural state of affairs in most professional software development projects!

The Evolution of Model-View-ViewModel

Ever since people started to create software user interfaces, there have been popular design patterns to help make it easier. For example, the Model-View-Presenter (MVP) pattern has enjoyed popularity on various UI programming platforms. MVP is a variation of the Model-View-Controller pattern, which has been around for decades. In case you have never used the MVP pattern before, here is a simplified explanation. What you see on the screen is the View, the data it displays is the model, and the Presenter hooks the two together. The view relies on a Presenter to populate it with model data, react to user input, provide input validation (perhaps by delegating to the model), and other such tasks. If you would like to learn more about the Model View Presenter, I suggest you read Jean-Paul Boodhoo's August 2006 Design Patterns column.

Back in 2004, Martin Fowler published an article about a pattern named Presentation Model (PM). The PM pattern is similar to MVP in that it separates a view from its behavior and state. The interesting part of the PM pattern is that an abstraction of a view is created, called the Presentation Model. A view, then, becomes merely a rendering of a Presentation Model. In Fowler's explanation, he shows that the Presentation Model frequently updates its View, so that the two stay in sync with each other. That synchronization logic exists as code in the Presentation Model classes.

In 2005, John Gossman, currently one of the WPF and Silverlight Architects at Microsoft, unveiled the Model-View-ViewModel (MVVM) pattern on his blog. MVVM is identical to Fowler's Presentation Model, in that both patterns feature an abstraction of a View, which contains a View's state and behavior. Fowler introduced Presentation Model as a means of creating a UI platform-independent abstraction of a View, whereas Gossman introduced MVVM as a standardized way to leverage core features of WPF to simplify the creation of user interfaces. In that sense, I consider MVVM to be a specialization of the more general PM pattern, tailor-made for the WPF and Silverlight platforms.

In Glenn Block's excellent article "Prism: Patterns for Building Composite Applications with WPF" in the September 2008 issue, he explains the Microsoft Composite Application Guidance for WPF. The term ViewModel is never used. Instead, the term Presentation Model is used to describe the abstraction of a view. Throughout this article, however, I'll refer to the pattern as MVVM and the abstraction of a view as a ViewModel. I find this terminology is much more prevelant in the WPF and Silverlight communities.

Unlike the Presenter in MVP, a ViewModel does not need a reference to a view. The view binds to properties on a ViewModel, which, in turn, exposes data contained in model objects and other state specific to the view. The bindings between view and ViewModel are simple to construct because a ViewModel object is set as the DataContext of a view. If property values in the ViewModel change, those new values automatically propagate to the view via data binding. When the user clicks a button in the View, a command on the ViewModel executes to perform the requested action. The ViewModel, never the View, performs all modifications made to the model data.

The view classes have no idea that the model classes exist, while the ViewModel and model are unaware of the view. In fact, the model is completely oblivious to the fact that the ViewModel and view exist. This is a very loosely coupled design, which pays dividends in many ways, as you will soon see.

Why WPF Developers Love MVVM

Once a developer becomes comfortable with WPF and MVVM, it can be difficult to differentiate the two. MVVM is the lingua franca of WPF developers because it is well suited to the WPF platform, and WPF was designed to make it easy to build applications using the MVVM pattern (amongst others). In fact, Microsoft was using MVVM internally to develop WPF applications, such as Microsoft Expression Blend, while the core WPF platform was under construction. Many aspects of WPF, such as the look-less control model and data templates, utilize the strong separation of display from state and behavior promoted by MVVM.

The single most important aspect of WPF that makes MVVM a great pattern to use is the data binding infrastructure. By binding properties of a view to a ViewModel, you get loose coupling between the two and entirely remove the need for writing code in a ViewModel that directly updates a view. The data binding system also supports input validation, which provides a standardized way of transmitting validation errors to a view.

Two other features of WPF that make this pattern so usable are data templates and the resource system. Data templates apply Views to ViewModel objects shown in the user interface. You can declare templates in XAML and let the resource system automatically locate and apply those templates for you at run time. You can learn more about binding and data templates in my July 2008 article, "Data and WPF: Customize Data Display with Data Binding and WPF."

If it were not for the support for commands in WPF, the MVVM pattern would be much less powerful. In this article, I will show you how a ViewModel can expose commands to a View, thus allowing the view to consume its functionality. If you aren't familiar with commanding, I recommend that you read Brian Noyes's comprehensive article, "Advanced WPF: Understanding Routed Events and Commands in WPF," from the September 2008 issue.

In addition to the WPF (and Silverlight 2) features that make MVVM a natural way to structure an application, the pattern is also popular because ViewModel classes are easy to unit test. When an application's interaction logic lives in a set of ViewModel classes, you can easily write code that tests it. In a sense, Views and unit tests are just two different types of ViewModel consumers. Having a suite of tests for an application's ViewModels provides free and fast regression testing, which helps reduce the cost of maintaining an application over time.

In addition to promoting the creation of automated regression tests, the testability of ViewModel classes can assist in properly designing user interfaces that are easy to skin. When you are designing an application, you can often decide whether something should be in the view or the ViewModel by imagining that you want to write a unit test to consume the ViewModel. If you can write unit tests for the ViewModel without creating any UI objects, you can also completely skin the ViewModel because it has no dependencies on specific visual elements.

Lastly, for developers who work with visual designers, using MVVM makes it much easier to create a smooth designer/developer workflow. Since a view is just an arbitrary consumer of a ViewModel, it is easy to just rip one view out and drop in a new view to render a ViewModel. This simple step allows for rapid prototyping and evaluation of user interfaces made by the designers.

The development team can focus on creating robust ViewModel classes, and the design team can focus on making user-friendly Views. Connecting the output of both teams can involve little more than ensuring that the correct bindings exist in a view's XAML file.

The Demo Application

At this point, I have reviewed MVVM's history and theory of operation. I also examined why it is so popular amongst WPF developers. Now it is time to roll up your sleeves and see the pattern in action. The demo application that accompanies this article uses MVVM in a variety of ways. It provides a fertile source of examples to help put the concepts into a meaningful context. I created the demo application in Visual Studio 2008 SP1, against the Microsoft .NET Framework 3.5 SP1. The unit tests run in the Visual Studio unit testing system.

The application can contain any number of "workspaces," each of which the user can open by clicking on a command link in the navigation area on the left. All workspaces live in a TabControl on the main content area. The user can close a workspace by clicking the Close button on that workspace's tab item. The application has two available workspaces: "All Customers" and "New Customer." After running the application and opening some workspaces, the UI looks something like **Figure 1**.

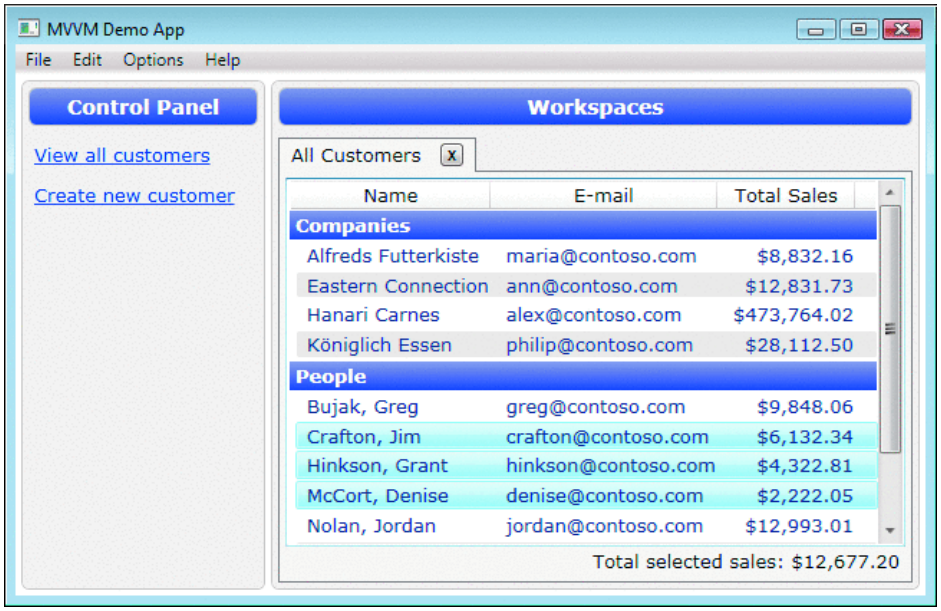

Figure 1 **Workspaces**

Only one instance of the "All Customers" workspace can be open at a time, but any number of "New Customer" workspaces can be open at once. When the user decides to create a new customer, she must fill in the data entry form in **Figure 2**.

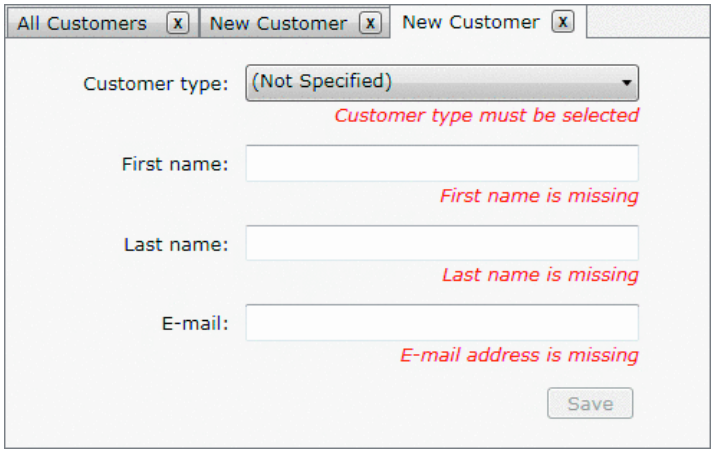

Figure 2 **New Customer Data Entry Form**

After filling in the data entry form with valid values and clicking the Save button, the new customer's name appears in the tab item and that customer is added to the list of all customers. The application does not have support for deleting or editing an existing customer, but that functionality, and many other features similar to it, are easy to implement by building on top of the existing application architecture. Now that you have a high-level understanding of what the demo application does, let's investigate how it was designed and implemented.

Relaying Command Logic

Every view in the app has an empty codebehind file, except for the standard boilerplate code that calls InitializeComponent in the class's constructor. In fact, you could remove the views' codebehind files from the project and the application would still compile and run correctly. Despite the lack of event handling methods in the views, when the user clicks on buttons, the application reacts and satisfies the user's requests. This works because of bindings that were established on the Command property of Hyperlink, Button, and MenuItem controls displayed in the UI. Those bindings ensure that when the user clicks on the controls, ICommand objects exposed by the ViewModel execute. You can think of the command object as an adapter that makes it easy to consume a ViewModel's functionality from a view declared in XAML.

When a ViewModel exposes an instance property of type ICommand, the command object typically uses that ViewModel object to get its job done. One possible implementation pattern is to create a private nested class within the ViewModel class, so that the command has access to private members of its containing ViewModel and does not pollute the namespace. That nested class implements the ICommand interface, and a reference

to the containing ViewModel object is injected into its constructor. However, creating a nested class that implements ICommand for each command exposed by a ViewModel can bloat the size of the ViewModel class. More code means a greater potential for bugs.

In the demo application, the RelayCommand class solves this problem. RelayCommand allows you to inject the command's logic via delegates passed into its constructor. This approach allows for terse, concise command implementation in ViewModel classes. RelayCommand is a simplified variation of the DelegateCommand found in the Microsoft Composite Application Library. The RelayCommand class is shown in **Figure 3**.

Figure 3 The RelayCommand Class

```
public class RelayCommand : ICommand
{
    #region Fields

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

    #endregion // Fields

    #region Constructors

    public RelayCommand(Action<object> execute)
    : this(execute, null)
    {
    }

    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 parameter)
    {
        return _canExecute == null ? true : _canExecute(parameter);
    }

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

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

    #endregion // ICommand Members
}
```

The CanExecuteChanged event, which is part of the ICommand interface implementation, has some interesting features. It delegates the event subscription to the CommandManager.RequerySuggested event. This ensures that the WPF commanding infrastructure asks all RelayCommand objects if they can execute whenever it asks the built-in commands. The following code from the CustomerViewModel class, which I will examine in-depth later, shows how to configure a RelayCommand with lambda expressions:

```
RelayCommand _saveCommand;
public ICommand SaveCommand
{
    get
    {
        if (_saveCommand == null)
        {
            _saveCommand = new RelayCommand(param => this.Save(),
                param => this.CanSave );
        }
        return _saveCommand;
    }
}
```

ViewModel Class Hierarchy

Most ViewModel classes need the same features. They often need to implement the INotifyPropertyChanged interface, they usually need to have a user-friendly display name, and, in the case of workspaces, they need

the ability to close (that is, be removed from the UI). This problem naturally lends itself to the creations of a ViewModel base class or two, so that new ViewModel classes can inherit all of the common functionality from a base class. The ViewModel classes form the inheritance hierarchy seen in **Figure 4**.

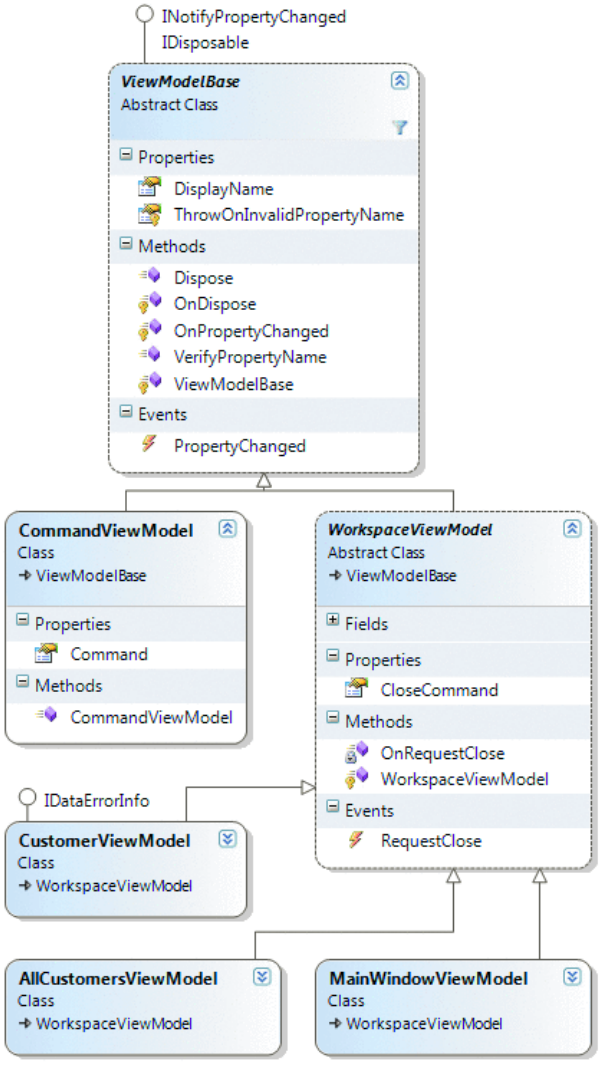

Figure 4 **Inheritance Hierarchy**

Having a base class for all of your ViewModels is by no means a requirement. If you prefer to gain features in your classes by composing many smaller classes together, instead of using inheritance, that is not a problem. Just like any other design pattern, MVVM is a set of guidelines, not rules.

ViewModelBase Class

ViewModelBase is the root class in the hierarchy, which is why it implements the commonly used INotifyPropertyChanged interface and has a DisplayName property. The INotifyPropertyChanged interface contains an event called PropertyChanged. Whenever a property on a ViewModel object has a new value, it can raise the PropertyChanged event to notify the WPF binding system of the new value. Upon receiving that notification, the binding system queries the property, and the bound property on some UI element receives the new value.

In order for WPF to know which property on the ViewModel object has changed, the PropertyChangedEventArgs class exposes a PropertyName property of type String. You must be careful to pass the correct property name into that event argument; otherwise, WPF will end up querying the wrong property for a new value.

One interesting aspect of ViewModelBase is that it provides the ability to verify that a property with a given name actually exists on the ViewModel object. This is very useful when refactoring, because changing a property's name via the Visual Studio 2008 refactoring feature will not update strings in your source code that happen to contain that property's name (nor should it). Raising the PropertyChanged event with an incorrect property name in the event argument can lead to subtle bugs that are difficult to track down, so this little feature can be a huge timesaver. The code from ViewModelBase that adds this useful support is shown in **Figure 5**.

Figure 5 Verifying a Property

```
// In ViewModelBase.cs
public event PropertyChangedEventHandler PropertyChanged;

protected virtual void OnPropertyChanged(string propertyName)
{
    this.VerifyPropertyName(propertyName);

    PropertyChangedEventHandler handler = this.PropertyChanged;
```

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

    [Conditional("DEBUG")]
    [DebuggerStepThrough]
    public 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);
        }
    }
```

CommandViewModel Class

The simplest concrete ViewModelBase subclass is CommandViewModel. It exposes a property called Command of type ICommand. MainWindowViewModel exposes a collection of these objects through its Commands property. The navigation area on the left-hand side of the main window displays a link for each CommandViewModel exposed by MainWindowViewModel, such as "View all customers" and "Create new customer." When the user clicks on a link, thus executing one of those commands, a workspace opens in the TabControl on the main window. The CommandViewModel class definition is shown here:

```
public class CommandViewModel : ViewModelBase
{
    public CommandViewModel(string displayName, ICommand command)
    {
        if (command == null)
            throw new ArgumentNullException("command");

        base.DisplayName = displayName;
        this.Command = command;
    }

    public ICommand Command { get; private set; }
}
```

In the MainWindowResources.xaml file there exists a DataTemplate whose key is "CommandsTemplate". MainWindow uses that template to render the collection of CommandViewModels mentioned earlier. The template simply renders each CommandViewModel object as a link in an ItemsControl. Each Hyperlink's Command property is bound to the Command property of a CommandViewModel. That XAML is shown in **Figure 6**.

Figure 6 Render the List of Commands

```
<!-- In MainWindowResources.xaml -->
<!--
This template explains how to render the list of commands on
the left side in the main window (the 'Control Panel' area).
-->
<DataTemplate x:Key="CommandsTemplate">
  <ItemsControl ItemsSource="{Binding Path=Commands}">
    <ItemsControl.ItemTemplate>
      <DataTemplate>
        <TextBlock Margin="2,6">
          <Hyperlink Command="{Binding Path=Command}">
            <TextBlock Text="{Binding Path=DisplayName}" />
          </Hyperlink>
        </TextBlock>
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ItemsControl>
</DataTemplate>
```

MainWindowViewModel Class

As previously seen in the class diagram, the WorkspaceViewModel class derives from ViewModelBase and adds the ability to close. By "close," I mean that something removes the workspace from the user interface at run time. Three classes derive from WorkspaceViewModel: MainWindowViewModel, AllCustomersViewModel, and CustomerViewModel. MainWindowViewModel's request to close is handled by the App class, which creates the MainWindow and its ViewModel, as seen in **Figure 7**.

Figure 7 Create the ViewModel

```
// In App.xaml.cs
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);

    MainWindow window = new MainWindow();

    // Create the ViewModel to which
    // the main window binds.
    string path = "Data/customers.xml";
    var viewModel = new MainWindowViewModel(path);

    // When the ViewModel asks to be closed,
    // close the window.
    viewModel.RequestClose += delegate
    {
        window.Close();
    };

    // Allow all controls in the window to
    // bind to the ViewModel by setting the
    // DataContext, which propagates down
    // the element tree.
    window.DataContext = viewModel;

    window.Show();
}
```

MainWindow contains a menu item whose Command property is bound to the MainWindowViewModel's CloseCommand property. When the user clicks on that menu item, the App class responds by calling the window's Close method, like so:

```
<!-- In MainWindow.xaml -->
<Menu>
  <MenuItem Header="_File">
    <MenuItem Header="_Exit" Command="{Binding Path=CloseCommand}" />
  </MenuItem>
  <MenuItem Header="_Edit" />
  <MenuItem Header="_Options" />
  <MenuItem Header="_Help" />
</Menu>
```

MainWindowViewModel contains an observable collection of WorkspaceViewModel objects, called Workspaces. The main window contains a TabControl whose ItemsSource property is bound to that collection. Each tab item has a Close button whose Command property is bound to the CloseCommand of its corresponding WorkspaceViewModel instance. An abridged version of the template that configures each tab item is shown in the code that follows. The code is found in MainWindowResources.xaml, and the template explains how to render a tab item with a Close button:

```
<DataTemplate x:Key="ClosableTabItemTemplate">
  <DockPanel Width="120">
    <Button
      Command="{Binding Path=CloseCommand}"
      Content="X"
      DockPanel.Dock="Right"
      Width="16" Height="16"
      />
    <ContentPresenter Content="{Binding Path=DisplayName}" />
  </DockPanel>
</DataTemplate>
```

When the user clicks the Close button in a tab item, that WorkspaceViewModel's CloseCommand executes, causing its RequestClose event to fire. MainWindowViewModel monitors the RequestClose event of its workspaces and removes the workspace from the Workspaces collection upon request. Since the Main-Window's TabControl has its ItemsSource property bound to the observable collection of WorkspaceViewModels, removing an item from the collection causes the corresponding workspace to be removed from the TabControl. That logic from MainWindowViewModel is shown in **Figure 8**.

Figure 8 Removing Workspace from the UI

```
// In MainWindowViewModel.cs

ObservableCollection<WorkspaceViewModel> _workspaces;

public ObservableCollection<WorkspaceViewModel> Workspaces
{
    get
    {
        if (_workspaces == null)
```

```
        {
            _workspaces = new ObservableCollection<WorkspaceViewModel>();
            _workspaces.CollectionChanged += this.OnWorkspacesChanged;
        }
        return _workspaces;
    }
}

void OnWorkspacesChanged(object sender, NotifyCollectionChangedEventArgs e)
{
    if (e.NewItems != null && e.NewItems.Count != 0)
        foreach (WorkspaceViewModel workspace in e.NewItems)
            workspace.RequestClose += this.OnWorkspaceRequestClose;

    if (e.OldItems != null && e.OldItems.Count != 0)
        foreach (WorkspaceViewModel workspace in e.OldItems)
            workspace.RequestClose -= this.OnWorkspaceRequestClose;
}

void OnWorkspaceRequestClose(object sender, EventArgs e)
{
    this.Workspaces.Remove(sender as WorkspaceViewModel);
}
```

In the UnitTests project, the MainWindowViewModelTests.cs file contains a test method that verifies that this functionality is working properly. The ease with which you can create unit tests for ViewModel classes is a huge selling point of the MVVM pattern, because it allows for simple testing of application functionality without writing code that touches the UI. That test method is shown in **Figure 9**.

Figure 9 The Test Method

```
// In MainWindowViewModelTests.cs
[TestMethod]
public void TestCloseAllCustomersWorkspace()
{
    // Create the MainWindowViewModel, but not the MainWindow.
    MainWindowViewModel target =
        new MainWindowViewModel(Constants.CUSTOMER_DATA_FILE);

    Assert.AreEqual(0, target.Workspaces.Count, "Workspaces isn't empty.");

    // Find the command that opens the "All Customers" workspace.
    CommandViewModel commandVM =
        target.Commands.First(cvm => cvm.DisplayName == "View all customers");

    // Open the "All Customers" workspace.
    commandVM.Command.Execute(null);
    Assert.AreEqual(1, target.Workspaces.Count, "Did not create viewmodel.");

    // Ensure the correct type of workspace was created.
    var allCustomersVM = target.Workspaces[0] as AllCustomersViewModel;
    Assert.IsNotNull(allCustomersVM, "Wrong viewmodel type created.");

    // Tell the "All Customers" workspace to close.
    allCustomersVM.CloseCommand.Execute(null);
    Assert.AreEqual(0, target.Workspaces.Count, "Did not close viewmodel.");
}
```

Applying a View to a ViewModel

MainWindowViewModel indirectly adds and removes WorkspaceViewModel objects to and from the main window's TabControl. By relying on data binding, the Content property of a TabItem receives a ViewModelBase-derived object to display. ViewModelBase is not a UI element, so it has no inherent support for rendering itself. By default, in WPF a non-visual object is rendered by displaying the results of a call to its ToString method in a TextBlock. That clearly is not what you need, unless your users have a burning desire to see the type name of our ViewModel classes!

You can easily tell WPF how to render a ViewModel object by using typed DataTemplates. A typed DataTemplate does not have an x:Key value assigned to it, but it does have its DataType property set to an instance of the Type class. If WPF tries to render one of your ViewModel objects, it will check to see if the resource system has a typed DataTemplate in scope whose DataType is the same as (or a base class of) the type of your ViewModel object. If it finds one, it uses that template to render the ViewModel object referenced by the tab item's Content property.

The MainWindowResources.xaml file has a ResourceDictionary. That dictionary is added to the main window's resource hierarchy, which means that the resources it contains are in the window's resource scope. When a tab item's content is set to a ViewModel object, a typed DataTemplate from this dictionary supplies a view (that is, a user control) to render it, as shown in **Figure 10**.

Figure 10 Supplying a View

```
<!--
This resource dictionary is used by the MainWindow.
-->
<ResourceDictionary
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:vm="clr-namespace:DemoApp.ViewModel"
    xmlns:vw="clr-namespace:DemoApp.View"
    >

    <!--
    This template applies an AllCustomersView to an instance
    of the AllCustomersViewModel class shown in the main window.
    -->
    <DataTemplate DataType="{x:Type vm:AllCustomersViewModel}">
      <vw:AllCustomersView />
    </DataTemplate>

    <!--
    This template applies a CustomerView to an instance
    of the CustomerViewModel class shown in the main window.
    -->
    <DataTemplate DataType="{x:Type vm:CustomerViewModel}">
      <vw:CustomerView />
    </DataTemplate>

  <!-- Other resources omitted for clarity... -->

</ResourceDictionary>
```

You do not need to write any code that determines which view to show for a ViewModel object. The WPF resource system does all of the heavy lifting for you, freeing you up to focus on more important things. In more complex scenarios, it is possible to programmatically select the view, but in most situations that is unnecessary.

The Data Model and Repository

You have seen how ViewModel objects are loaded, displayed, and closed by the application shell. Now that the general plumbing is in place, you can review implementation details more specific to the domain of the application. Before getting deep into the application's two workspaces, "All Customers" and "New Customer," let's first examine the data model and data access classes. The design of those classes has almost nothing to do with the MVVM pattern, because you can create a ViewModel class to adapt just about any data object into something friendly to WPF.

The sole model class in the demo program is Customer. That class has a handful of properties that represent information about a customer of a company, such as their first name, last name, and e-mail address. It provides validation messages by implementing the standard IDataErrorInfo interface, which existed for years before WPF hit the street. The Customer class has nothing in it that suggests it is being used in an MVVM architecture or even in a WPF application. The class could easily have come from a legacy business library.

Data must come from and reside somewhere. In this application, an instance of the CustomerRepository class loads and stores all Customer objects. It happens to load the customer data from an XML file, but the type of external data source is irrelevant. The data could come from a database, a Web service, a named pipe, a file on disk, or even carrier pigeons: it simply does not matter. As long as you have a .NET object with some data in it, regardless of where it came from, the MVVM pattern can get that data on the screen.

The CustomerRepository class exposes a few methods that allow you to get all the available Customer objects, add new a Customer to the repository, and check if a Customer is already in the repository. Since the application does not allow the user to delete a customer, the repository does not allow you to remove a customer. The CustomerAdded event fires when a new Customer enters the CustomerRepository, via the AddCustomer method.

Clearly, this application's data model is very small, compared to what real business applications require, but that is not important. What is important to understand is how the ViewModel classes make use of Customer and CustomerRepository. Note that CustomerViewModel is a wrapper around a Customer object. It exposes the state of a Customer, and other state used by the CustomerView control, through a set of properties. CustomerViewModel does not duplicate the state of a Customer; it simply exposes it via delegation, like this:

```
public string FirstName
{
    get { return _customer.FirstName; }
    set
    {
        if (value == _customer.FirstName)
            return;
        _customer.FirstName = value;
        base.OnPropertyChanged("FirstName");
    }
}
```

When the user creates a new customer and clicks the Save button in the CustomerView control, the CustomerViewModel associated with that view will add the new Customer object to the CustomerRepository. That causes the repository's CustomerAdded event to fire, which lets the AllCustomersViewModel know that it should add a new CustomerViewModel to its AllCustomers collection. In a sense, CustomerRepository acts as a synchronization mechanism between various ViewModels that deal with Customer objects. Perhaps one might think of this as using the Mediator design pattern. I will review more of how this works in the upcoming sections, but for now refer to the diagram in **Figure 11** for a high-level understanding of how all the pieces fit together.

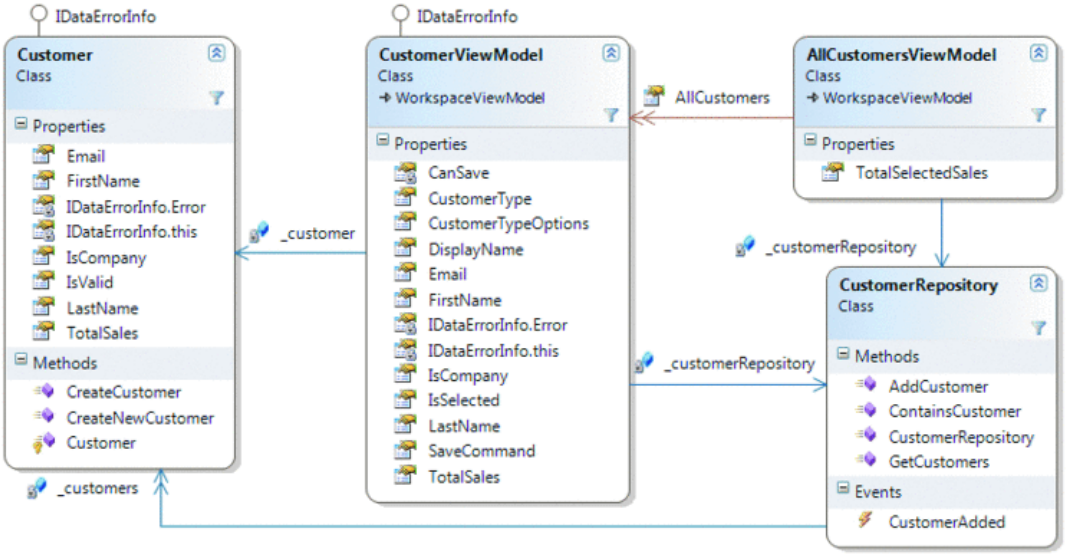

Figure 11 **Customer Relationships**

New Customer Data Entry Form

When the user clicks the "Create new customer" link, MainWindowViewModel adds a new CustomerViewModel to its list of workspaces, and a CustomerView control displays it. After the user types valid values into the input fields, the Save button enters the enabled state so that the user can persist the new customer information. There is nothing out of the ordinary here, just a regular data entry form with input validation and a Save button.

The Customer class has built-in validation support, available through its IDataErrorInfo interface implementation. That validation ensures the customer has a first name, a well-formed e-mail address, and, if the customer is a person, a last name. If the Customer's IsCompany property returns true, the LastName property cannot have a value (the idea being that a company does not have a last name). This validation logic might make sense from the Customer object's perspective, but it does not meet the needs of the user interface. The UI requires a user to select whether a new customer is a person or a company. The Customer Type selector initially has the value "(Not Specified)". How can the UI tell the user that the customer type is unspecified if the IsCompany property of a Customer only allows for a true or false value?

Assuming you have complete control over the entire software system, you could change the IsCompany property to be of type Nullable<bool>, which would allow for the "unselected" value. However, the real world is not always so simple. Suppose you cannot change the Customer class because it comes from a legacy library owned by a different team in your company. What if there is no easy way to persist that "unselected" value because of the existing database schema? What if other applications already use the Customer class and rely on the property being a normal Boolean value? Once again, having a ViewModel comes to the rescue.

The test method in **Figure 12** shows how this functionality works in CustomerViewModel. CustomerViewModel exposes a CustomerTypeOptions property so that the Customer Type selector has three strings to display. It also exposes a CustomerType property, which stores the selected String in the selector. When CustomerType is set, it maps the String value to a Boolean value for the underlying Customer object's IsCompany property. **Figure 13** shows the two properties.

Figure 12 The Test Method

```
// In CustomerViewModelTests.cs
[TestMethod]
public void TestCustomerType()
{
    Customer cust = Customer.CreateNewCustomer();
    CustomerRepository repos = new CustomerRepository(
        Constants.CUSTOMER_DATA_FILE);
    CustomerViewModel target = new CustomerViewModel(cust, repos);

    target.CustomerType = "Company"
    Assert.IsTrue(cust.IsCompany, "Should be a company");

    target.CustomerType = "Person";
    Assert.IsFalse(cust.IsCompany, "Should be a person");

    target.CustomerType = "(Not Specified)";
    string error = (target as IDataErrorInfo)["CustomerType"];
    Assert.IsFalse(String.IsNullOrEmpty(error), "Error message should
        be returned");
}
```

Figure 13 CustomerType Properties

```
// In CustomerViewModel.cs

public string[] CustomerTypeOptions
{
```

```
    get
    {
        if (_customerTypeOptions == null)
        {
            _customerTypeOptions = new string[]
            {
                "(Not Specified)",
                "Person",
                "Company"
            };
        }
        return _customerTypeOptions;
    }
}
public string CustomerType
{
    get { return _customerType; }
    set
    {
        if (value == _customerType ||
            String.IsNullOrEmpty(value))
            return;

        _customerType = value;

        if (_customerType == "Company")
        {
            _customer.IsCompany = true;
        }
        else if (_customerType == "Person")
        {
            _customer.IsCompany = false;
        }

        base.OnPropertyChanged("CustomerType");
        base.OnPropertyChanged("LastName");
    }
}
```

The CustomerView control contains a ComboBox that is bound to those properties, as seen here:

```
<ComboBox
  ItemsSource="{Binding CustomerTypeOptions}"
  SelectedItem="{Binding CustomerType, ValidatesOnDataErrors=True}"
  />
```

When the selected item in that ComboBox changes, the data source's IDataErrorInfo interface is queried to see if the new value is valid. That occurs because the SelectedItem property binding has ValidatesOnDataErrors set to true. Since the data source is a CustomerViewModel object, the binding system asks that CustomerViewModel for a validation error on the CustomerType property. Most of the time, CustomerViewModel delegates all requests for validation errors to the Customer object it contains. However, since Customer has no notion of having an unselected state for the IsCompany property, the CustomerViewModel class must handle validating the new selected item in the ComboBox control. That code is seen in **Figure 14**.

Figure 14 Validating a CustomerViewModel Object

```
// In CustomerViewModel.cs
string IDataErrorInfo.this[string propertyName]
{
    get
    {
        string error = null;

        if (propertyName == "CustomerType")
        {
            // The IsCompany property of the Customer class
            // is Boolean, so it has no concept of being in
            // an "unselected" state. The CustomerViewModel
            // class handles this mapping and validation.
            error = this.ValidateCustomerType();
        }
        else
        {
            error = (_customer as IDataErrorInfo)[propertyName];
        }

        // Dirty the commands registered with CommandManager,
        // such as our Save command, so that they are queried
        // to see if they can execute now.
        CommandManager.InvalidateRequerySuggested();

        return error;
    }
}
```

```
string ValidateCustomerType()
{
    if (this.CustomerType == "Company" ||
       this.CustomerType == "Person")
        return null;

    return "Customer type must be selected";
}
```

The key aspect of this code is that CustomerViewModel's implementation of IDataErrorInfo can handle requests for ViewModel-specific property validation and delegate the other requests to the Customer object. This allows you to make use of validation logic in Model classes and have additional validation for properties that only make sense to ViewModel classes.

The ability to save a CustomerViewModel is available to a view through the SaveCommand property. That command uses the RelayCommand class examined earlier to allow CustomerViewModel to decide if it can save itself and what to do when told to save its state. In this application, saving a new customer simply means adding it to a CustomerRepository. Deciding if the new customer is ready to be saved requires consent from two parties. The Customer object must be asked if it is valid or not, and the CustomerViewModel must decide if it is valid. This two-part decision is necessary because of the ViewModel-specific properties and validation examined previously. The save logic for CustomerViewModel is shown in **Figure 15**.

Figure 15 The Save Logic for CustomerViewModel

```
// In CustomerViewModel.cs
public ICommand SaveCommand
{
    get
    {
        if (_saveCommand == null)
        {
            _saveCommand = new RelayCommand(
                param => this.Save(),
                param => this.CanSave
                );
        }
        return _saveCommand;
    }
}

public void Save()
{
    if (!_customer.IsValid)
        throw new InvalidOperationException("...");

    if (this.IsNewCustomer)
        _customerRepository.AddCustomer(_customer);

    base.OnPropertyChanged("DisplayName");
}

bool IsNewCustomer
{
    get
    {
        return !_customerRepository.ContainsCustomer(_customer);
    }
}

bool CanSave
{
    get
    {
        return
            String.IsNullOrEmpty(this.ValidateCustomerType()) &&
            _customer.IsValid;
    }
}
```

The use of a ViewModel here makes it much easier to create a view that can display a Customer object and allow for things like an "unselected" state of a Boolean property. It also provides the ability to easily tell the customer to save its state. If the view were bound directly to a Customer object, the view would require a lot of code to make this work properly. In a well-designed MVVM architecture, the codebehind for most Views should be empty, or, at most, only contain code that manipulates the controls and resources contained within that view. Sometimes it is also necessary to write code in a View's codebehind that interacts with a ViewModel object, such as hooking an event or calling a method that would otherwise be very difficult to invoke from the ViewModel itself.

All Customers View

The demo application also contains a workspace that displays all of the customers in a ListView. The customers in the list are grouped according to whether they are a company or a person. The user can select one or more customers at a time and view the sum of their total sales in the bottom right corner.

The UI is the AllCustomersView control, which renders an AllCustomersViewModel object. Each ListViewItem

represents a CustomerViewModel object in the AllCustomers collection exposed by the AllCustomerViewModel object. In the previous section, you saw how a CustomerViewModel can render as a data entry form, and now the exact same CustomerViewModel object is rendered as an item in a ListView. The CustomerViewModel class has no idea what visual elements display it, which is why this reuse is possible.

AllCustomersView creates the groups seen in the ListView. It accomplishes this by binding the ListView's ItemsSource to a CollectionViewSource configured like **Figure 16**.

Figure 16 CollectionViewSource

```
<!-- In AllCustomersView.xaml -->
<CollectionViewSource
  x:Key="CustomerGroups"
  Source="{Binding Path=AllCustomers}"
  >
  <CollectionViewSource.GroupDescriptions>
    <PropertyGroupDescription PropertyName="IsCompany" />
  </CollectionViewSource.GroupDescriptions>
  <CollectionViewSource.SortDescriptions>
    <!--
    Sort descending by IsCompany so that the ' True' values appear first,
    which means that companies will always be listed before people.
    -->
    <scm:SortDescription PropertyName="IsCompany" Direction="Descending" />
    <scm:SortDescription PropertyName="DisplayName" Direction="Ascending" />
  </CollectionViewSource.SortDescriptions>
</CollectionViewSource>
```

The association between a ListViewItem and a CustomerViewModel object is established by the ListView's ItemContainerStyle property. The Style assigned to that property is applied to each ListViewItem, which enables properties on a ListViewItem to be bound to properties on the CustomerViewModel. One important binding in that Style creates a link between the IsSelected property of a ListViewItem and the IsSelected property of a CustomerViewModel, as seen here:

```
<Style x:Key="CustomerItemStyle" TargetType="{x:Type ListViewItem}">
  <!--   Stretch the content of each cell so that we can
  right-align text in the Total Sales column.  -->
  <Setter Property="HorizontalContentAlignment" Value="Stretch" />
  <!--
  Bind the IsSelected property of a ListViewItem to the
  IsSelected property of a CustomerViewModel object.
  -->
  <Setter Property="IsSelected" Value="{Binding Path=IsSelected,
    Mode=TwoWay}" />
</Style>
```

When a CustomerViewModel is selected or unselected, that causes the sum of all selected customers' total sales to change. The AllCustomersViewModel class is responsible for maintaining that value, so that the ContentPresenter beneath the ListView can display the correct number. **Figure 17** shows how AllCustomersViewModel monitors each customer for being selected or unselected and notifies the view that it needs to update the display value.

Figure 17 Monitoring for Selected or Unselected

```
// In AllCustomersViewModel.cs
public double TotalSelectedSales
{
    get
    {
        return this.AllCustomers.Sum(
            custVM => custVM.IsSelected ? custVM.TotalSales : 0.0);
    }
}

void OnCustomerViewModelPropertyChanged(object sender,
    PropertyChangedEventArgs e)
{
    string IsSelected = "IsSelected";

    // Make sure that the property name we're
    // referencing is valid.  This is a debugging
    // technique, and does not execute in a Release build.
    (sender as CustomerViewModel).VerifyPropertyName(IsSelected);

    // When a customer is selected or unselected, we must let the
    // world know that the TotalSelectedSales property has changed,
    // so that it will be queried again for a new value.
    if (e.PropertyName == IsSelected)
        this.OnPropertyChanged("TotalSelectedSales");
}
```

The UI binds to the TotalSelectedSales property and applies currency (monetary) formatting to the value. The ViewModel object could apply the currency formatting, instead of the view, by returning a String instead of a

Double value from the TotalSelectedSales property. The ContentStringFormat property of ContentPresenter was added in the .NET Framework 3.5 SP1, so if you must target an older version of WPF, you will need to apply the currency formatting in code:

```
<!-- In AllCustomersView.xaml -->
<StackPanel Orientation="Horizontal">
  <TextBlock Text="Total selected sales: " />
  <ContentPresenter
    Content="{Binding Path=TotalSelectedSales}"
    ContentStringFormat="c"
  />
</StackPanel>
```

Wrapping Up

WPF has a lot to offer application developers, and learning to leverage that power requires a mindset shift. The Model-View-ViewModel pattern is a simple and effective set of guidelines for designing and implementing a WPF application. It allows you to create a strong separation between data, behavior, and presentation, making it easier to control the chaos that is software development.

I would like to thank John Gossman for his help with this article.

Josh Smith is passionate about using WPF to create great user experiences. He was awarded the Microsoft MVP title for his work in the WPF community. Josh works for Infragistics in the Experience Design Group. When he is not at a computer, he enjoys playing the piano, reading about history, and exploring New York City with his girlfriend. You can visit Josh's blog at joshsmithonwpf.wordpress.com.