Model-View-ViewModel (MVVM) Explained

Jeremy Likness, 8 Aug 2010 CPOL

An introduction to the Model-View-ViewModel (MVVM) pattern.

Introduction

The purpose of this post is to provide an introduction to the Model-View-ViewModel (MVVM) pattern. While I've participated in lots of discussions online about MVVM, it occurred to me that beginners who are learning the pattern have very little to go on and a lot of conflicting resources to wade through in order to try to implement it in their own code. I am not trying to introduce dogma, but wanted to pull together key concepts in a single post to make it easy and straightforward to understand the value of the pattern and how it can be implemented. MVVM is really far simpler than people make it out to be.

Background

Why should you, as a developer, even care about the Model-View-ViewModel pattern? There are a number of benefits this pattern brings to both WPF and Silverlight development. Before you go on, ask yourself:

- Do you need to share a project with a designer, and have the flexibility for design work and development work to happen near-simultaneously?
- Do you require thorough unit testing for your solutions?
- Is it important for you to have reusable components, both within and across projects in your organization?
- Would you like more flexibility to change your user interface without having to refactor other logic in the code base?

If you answered "yes" to any of these questions, these are just a few of the benefits that using the MVVM model can bring for your project.

I've been amazed at some conversations I've read online. Things like, "MVVM only makes sense for extremely complex UI", or "MVVM always adds a lot of overhead and is too much for smaller applications". The real kicker was, "MVVM doesn't scale". In my opinion, statements like this speak to knowledge and implementation of MVVM, not MVVM itself. In other words, if you think it takes hours to wire up MVVM, you're not doing it right. If your application isn't scaling, don't blame MVVM, blame how you are using MVVM. Binding 100,000 items to a list box can be just silly regardless of what pattern you are following.

So the quick disclaimer: this is MVVM as I know it, not MVVM as a universal truth. I encourage you to share your thoughts, experiences, feedback, and opinions using the comments. If you feel something is incorrect, let me know and I'll do my best to keep this post updated and current.

The Model

The model is what I like to refer to as the domain object. The model represents the actual data and/or information we are dealing with. An example of a model might be a contact (containing name, phone number, address, etc.) or the characteristics of a live streaming publishing point.

The key to remember with the model is that it holds the information, but not behaviors or services that manipulate the information. It is not responsible for formatting text to look pretty on the screen, or fetching a list of items from a remote server (in fact, in that list, each item would most likely be a model of its own). Business logic is typically kept separate from the model, and encapsulated in other classes that act on the model. This is not always true: for example, some models may contain validation.

It is often a challenge to keep a model completely "clean". By this, I mean a true representation of "the real world". For example, a contact record may contain a last modified date and the identity of the modifying user (auditing information), and a unique identifier (database or persistence information). The modified date has no real meaning for a contact in the real world, but is a function of how the model is used, tracked, and persisted in the system.

Here is a sample model for holding contact information:

```
namespace MVVMExample {
    public class ContactModel : INotifyPropertyChanged {
        private string _firstName;
        public string FirstName {
            get { return _firstName; }
            set {
                _firstName = value;
                RaisePropertyChanged("FirstName");
                RaisePropertyChanged("FullName");
            }
        }
        private string _lastName;
        public string LastName {
            get { return _lastName; }
            set {
                _lastName = value;
                RaisePropertyChanged("LastName");
                RaisePropertyChanged("FullName");
            }
        }
        public string FullName {
            get { return string.Format("{0} {1}", FirstName, LastName); }
        }
        private string _phoneNumber;
        public string PhoneNumber {
```

```
            get { return _phoneNumber; }
            set {
                _phoneNumber = value;
                RaisePropertyChanged("PhoneNumber");
            }
        }
        protected void RaisePropertyChanged(string propertyName) {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null) {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
        public event PropertyChangedEventHandler PropertyChanged;
        public override bool Equals(object obj) {
            return obj is ContactModel && ((ContactModel) obj).FullName.Equals(FullName);
        }
        public override int GetHashCode() {
            return FullName.GetHashCode();
        }
    }
```

The View

The view is what most of us are familiar with and the only thing the end user really interacts with. It is the presentation of the data. The view takes certain liberties to make this data more presentable. For example, a date might be stored on the model as number of seconds since midnight on January 1, 1970 (Unix Time). To the end user, however, it is presented with the month name, date, and year in their local time zone. A view can also have behaviors associated with it, such as accepting user input. The view manages input (key presses, mouse movements, touch gestures, etc.) which ultimately manipulates properties of the model.

In MVVM, the view is active. As opposed to a passive view which has no knowledge of the model and is completely manipulated by a controller/presenter, the view in MVVM contains behaviors, events, and data-bindings that ultimately require knowledge of the underlying model and viewmodel. While these events and behaviors might be mapped to properties, method calls, and commands, the view is still responsible for handling its own events, and does not turn this completely over to the viewmodel.

One thing to remember about the view is that it is not responsible for maintaining its state. Instead, it will synchronize this with the viewmodel.

Here is an example view, expressed as XAML:

```
<UserControl x:Class="MVVMExample.DetailView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White"
                DataContext="{Binding CurrentContact}">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <TextBlock Text="Name:" HorizontalAlignment="Right" Margin="5"/>
        <TextBlock Text="{Binding FullName}"
                HorizontalAlignment="Left" Margin="5" Grid.Column="1"/>
        <TextBlock Text="Phone:" HorizontalAlignment="Right"
                Margin="5" Grid.Row="1"/>
        <TextBlock Text="{Binding PhoneNumber}"
                HorizontalAlignment="Left" Margin="5"
                Grid.Row="1" Grid.Column="1"/>
    </Grid>
</UserControl>
```

Note that the various bindings are the integration/synchronization points with the viewmodel.

The View Model

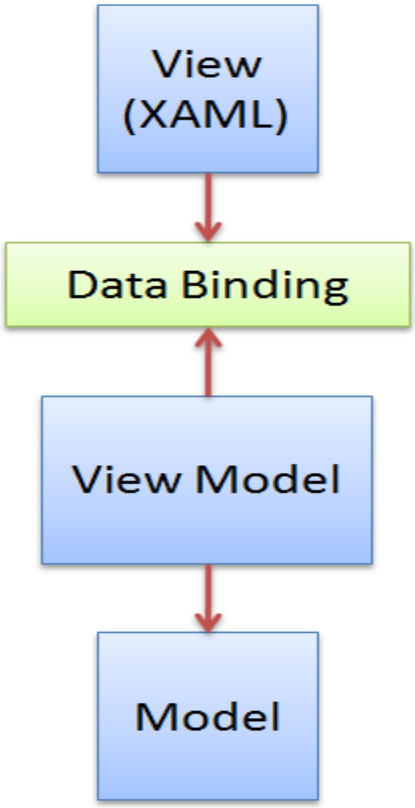
The viewmodel is a key piece of the triad because it introduces *Presentation Separation*, or the concept of keeping the nuances of the view separate from the model. Instead of making the model aware of the user's view of a date, so that it converts the date to the display format, the model simply holds the data, the view simply holds the formatted date, and the controller acts as the liaison between the two. The controller might take input from the view and place it on the model, or it might interact with a service to retrieve the model, then translate properties and place it on the view.

The viewmodel also exposes methods, commands, and other points that help maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself.

MVVM, while it evolved "behind the scenes" for quite some time, was introduced to the public in 2005 via Microsoft's John Gossman blog post about Avalon (the code name for Windows Presentation Foundation, or WPF). The blog post is entitled, Introduction to Model/View/ViewModel pattern for building WPF Apps, and generated quite a stir judging from the comments as people wrapped their brains around it.

I've heard MVVM described as an implementation of Presentation Model designed specifically for WPF (and later, Silverlight).

The examples of the pattern often focus on XAML for the view definition, and data-binding for commands and properties. These are more *implementation details* of the pattern rather than intrinsic to the pattern itself, which is why I offset data-binding with a different color:



Here is what a sample view model might look like. We've created a `BaseINPC` class (for "`INotifyPropertyChanged`") that has a method to make it easy for raising the property changed event.

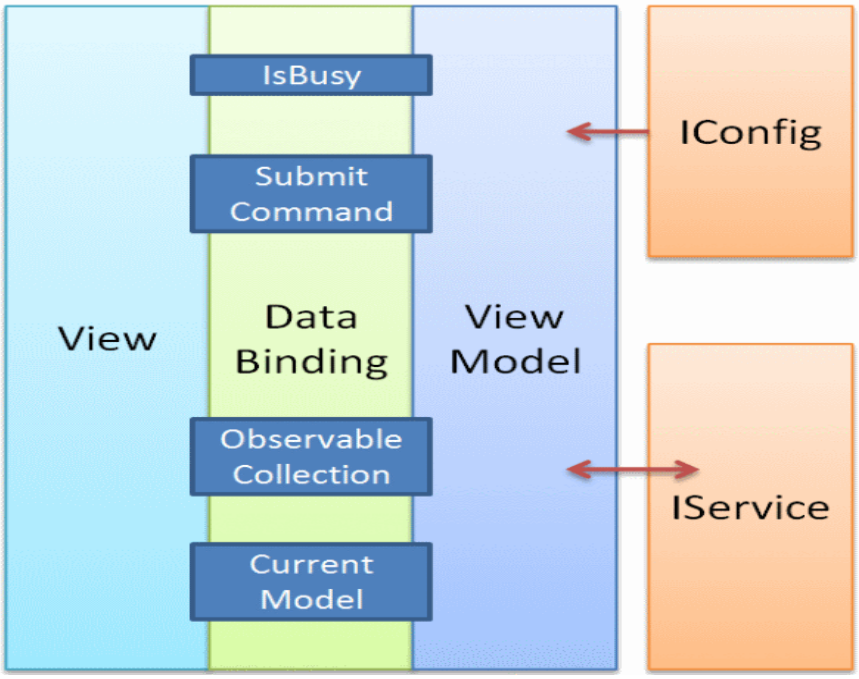
```
namespace MVVMExample {
    public class ContactViewModel : BaseINPC {
        public ContactViewModel() {
            Contacts = new ObservableCollection<ContactModel>();
            Service = new Service();
            Service.GetContacts(_PopulateContacts);
            Delete = new DeleteCommand(
                Service,
                ()=>CanDelete,
                contact => {
                        CurrentContact = null;
                        Service.GetContacts(_PopulateContacts);
                    });
        }
        private void _PopulatContacts(IEnumerable>ContactModel> contacts) {
            Contacts.Clear();
            foreach(var contact in contacts) {
                Contacts.Add(contact);
            }
        }
        public IService Service { get; set; }
        public bool CanDelete {
            get { return _currentContact != null; }
        }
        public ObservableCollection<ContactModel> Contacts { get; set; }
        public DeleteCommand Delete { get; set; }
```

```
        private ContactModel _currentContact;
        public ContactModel CurrentContact {
            get { return _currentContact; }
            set {
                _currentContact = value;
                RaisePropertyChanged("CurrentContact");
                RaisePropertyChanged("CanDelete");
                Delete.RaiseCanExecuteChanged();
            }
        }
    }
}
```

This view model is obviously designed to manage a list of contacts. It also exposes a delete command and a flag to indicate whether delete is allowed (thus maintaining state for the view). Often, the flag would be part of the command object, but the example is in Silverlight 3, which does not have native support for command binding, and I wanted to show a simple solution that didn't require a fancy framework. (Take a look at this video to see how easy it is to convert the example from Silverlight 3 to Silverlight 4 and use native commanding instead). The view model here makes a concrete reference to the service.

For larger applications, I prefer to wire in that reference externally or use a dependency injection framework. What's nice is we have the flexibility to build it like this initially and then refactor as needed - again, you do not have to use any of these frameworks to take advantage of the pattern, as you can see from this example. It fetches the list of "contacts" right away, which is a hard-coded list of me and someone a little more popular. The phone numbers, of course, are faked.

Let's get a little more specific and look at how this would be implemented in a sample application. Here is what an X-ray of a sample MVVM set up may look like:



So what can we gather from this snapshot?
First, the `IConfig` represents a configuration service (in a newsreader, it may contain the account information and feeds that are being fetched), while the `IService` is "some service" - perhaps the interface to fetch feeds from RSS sources in a news reader application.

The View and the ViewModel

- The view and the viewmodel communicate via data-binding, method calls, properties, events, and messages
- The viewmodel exposes not only models, but other properties (such as state information, like the "is busy" indicator) and commands
- The view handles its own UI events, then maps them to the viewmodel via commands
- The models and properties on the viewmodel are updated from the view via two-way databinding

Two mechanisms that often factor into implementations of the pattern are triggers (especially data triggers) in WPF, and the Visual State Manager (VSM) in Silverlight. These mechanisms help implement the pattern by binding UI behaviors to the underlying models. In Silverlight, the VSM should be the primary choice for coordination of transitions and animations. Learn more about VSM.

The ViewModel and the Model

The viewmodel becomes wholly responsible for the model in this scenario. Fortunately, it's not alone:
The viewmodel may expose the model directly, or properties related to the model, for data-binding
The viewmodel can contain interfaces to services, configuration data, etc., in order to fetch and manipulate the properties it exposes to the view

The Chicken or the Egg?

You might have heard discussion about *view first* or *viewmodel first*. In general, I believe most developers agree that a view should have exactly one viewmodel. There is no need to attach multiple viewmodels to a single view. If you think about separation of concerns, this makes sense, because if you have a "contact widget" on the screen bound to a "contact viewmodel", and a "company widget" bound to a "company viewmodel", these should be separate views, not a single view with two viewmodels.

A view may be composed of other views, each with its own viewmodel. Viewmodels might compose other viewmodels when necessary (often, however, I see people composing and aggregating viewmodels, when in fact what they really want is messaging between viewmodels).

While a view should only have one viewmodel, a single viewmodel might be used by multiple views (imagine a wizard, for example, that has three views but all bind to the

same viewmodel that drives the process).

View First

View first simply means the view is what drives the creation or discovery of the view model. In view first scenarios, the view typically binds to the view model as a resource, uses a locator pattern, or has the view model injected via MEF, Unity, or some other means. This is a very common method for managing views and view models. Here are some of my posts on the topic:

- ViewModel Binding with MEF
- MVVM Composition in Silverlight with Prism
- Prism, MEF, and MVVM

The example I've included with this post is view-first. The view is created, then the view model attached. In the `App` object, it looks like this:

```
private void Application_Startup(object sender, StartupEventArgs e) {
    var shell = new MainPage();
    shell.LayoutRoot.DataContext = new ContactViewModel();
    RootVisual = shell;
}
```

In this example, I'm keeping it simple, and not using any frameworks to wire in interfaces and implementations.

ViewModel First

ViewModel first is another method to wire the framework together. In this scenario, the viewmodel is responsible for creating the view and binding itself to the view. You can see an example of this in Rob Eisenberg's convention-based framework he discussed at MIX: Build your own MVVM Framework.

The take away here is there are multiple ways to skin the cat.

A Basic MVVM Framework

In my opinion, a basic MVVM framework really only requires two things:

1. A class that is either a `DependencyObject` or implements `INotifyPropertyChanged`to fully support data-binding, and
2. Some sort of commanding support.

The second issue exists in Silverlight 3 because the `ICommand` interface is provided, but not implemented. In Silverlight 4, commanding is more "out of the box". Commands facilitate binding of events from the view to the viewmodel. These are implementation details that make it easier to use the MVVM pattern.

Keep in mind, there is very rich support for binding and behaviors in Blend and the free

Blend SDK. You can watch my video, MVVM with MEF in Silverlight, to get an idea of how easy it really is to implement the MVVM pattern even without an existing framework in place. The post, MEF instead of Prism for Silverlight 3, shows how to build your own command objects.

With this example, I created a base class to handle the property changed events:

```
namespace MVVMExample {
    public abstract class BaseINPC : INotifyPropertyChanged {
        protected void RaisePropertyChanged(string propertyName) {
            var handler = PropertyChanged;
            if (handler != null) {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
        public event PropertyChangedEventHandler PropertyChanged;
    }
}
```

I also implemented a command. Typically, you would have a more generic type of command to handle different situations, but again, for the sake of illustration, I simply created a delete command specific to the function it performs. I am using a message box to confirm the delete. If you require something more elegant like a `ChildWindow`, read the scenarios I describe below to better understand how to integrate a dialog box as a service within MVVM.

```
namespace MVVMExample {
    public class DeleteCommand : ICommand {
        private readonly IService _service;
        private readonly Func<bool> _canExecute;
        private readonly Action<ContactModel> _deleted;

        public DeleteCommand(IService service, Func<bool> canExecute,
                                        Action<ContactModel> deleted) {
            _service = service;
            _canExecute = canExecute;
            _deleted = deleted;
        }

        public bool CanExecute(object parameter) {
            return _canExecute();
        }

        public void Execute(object parameter) {
            if (CanExecute(parameter)) {
                var contact = parameter as ContactModel;
                if (contact != null) {
                    var result = MessageBox.Show(
                       "Are you sure you wish to delete the contact?",
                       "Confirm Delete", MessageBoxButton.OKCancel);

                    if (result.Equals(MessageBoxResult.OK)) {
                        _service.DeleteContact(contact);
                        if (_deleted != null)
                        {
                            _deleted(contact);
```

```
                        }
                    }
                }
            }
        }

        public void RaiseCanExecuteChanged() {
            var handler = CanExecuteChanged;
            if (handler != null) {
                handler(this, EventArgs.Empty);
            }
        }

        public event EventHandler CanExecuteChanged;
    }
}
```

This particular command uses a delegate to callback when it is done, but this would only allow for a single subscriber. A multicast delegate or event will be required if multiple consumers (or viewmodels) for the command exist.

In Silverlight 4, I can simply bind a button to the command using the `Command` tag. I built the example in Silverlight 3, which does not have native support. To create the binding, I made a simple trigger - again, specific to this project and for the sake of illustration - to invoke the command, so I can easily bind it in the XAML.

For the examples I've provided here, you can view the sample application. It is very simple and contains exactly one service and one view model with a "mock database". Two views bind to the same viewmodel, and you can click on a contact to see its details. You can also delete a contact.

I often receive complaints that blog examples are too simple. This is sufficiently complex to show a full app without depending on other frameworks, but certainly doesn't show multiple pages and types of views. The reason you don't see these as often from me is simply because I am a consultant and contractor, so I am constantly building these line of business frameworks and applications for customers, and am not at liberty to share their code. While I can build small examples for posts, I simply don't have the time to build a larger working model. It's something I'd certainly like to do, but just wasn't practical for the timing of this post.

I think the easiest way to learn the pattern is by seeing a full application being built. I demonstrate this in MVVM with MEF in Silverlight. In that video, I build some simple viewmodels and show a view that is dynamically swapped based on user selection, and use MEF to wire everything up. A more complex scenario is then introduced in Part 2.

So what about those more complicated line of business solutions ... the ones that actually have more than one button, multiple views, and complex logic? That is beyond the scope of this post to cover in detail, but I'd like to tackle a few common scenarios and how I've solved them with MVVM.

Common MVVM Scenarios

In my experience, the idea of binding both commands and models or properties is straightforward. It's when you hit specific situations such as showing a dialog box or triggering an animation that MVVM may seem confusing. How do we solve these common problems?

List with Selection

How do you handle a combo box used for selection of a single, or multiple, items with MVVM? It's actually fairly straightforward. In fact, imagine a scenario where you have a combo-box that has a list of contact names, and another view on the same page that shows the contact details when selected. The ViewModel would look something like this, assuming I'm using MEF to wire dependencies (to show you a different way from the reference application):

```
public class ContactViewModel : BaseViewModel,
                IPartImportsSatisfiedNotification {
    [Import]
    public IContactService Service { get; set; }

    public ContactViewModel() {
        Contacts = new ObservableCollection<Contact>();
    }

    public ObservableCollection<Contact> Contacts { get; set; }

    private Contact _currentContact;

    public Contact CurrentContact {
        get { return _currentContact; }
        set {
            _currentContact = value;
            RaisePropertyChanged("CurrentContact");
        }
    }

    public void OnImportsSatisfied() {
        Service.FetchContacts(list => {
                foreach(var contact in list) {
                    Contacts.Add(contact);
                }
                CurrentContact = Contacts[0];
            });
    }
}
```

In this case, we import a service for getting contacts, wire in the list, and set the current contact. The drop down binds to the `Contacts` collection. What's important, however, is that the selected item is also bound (this is where the view model maintains state). The binding would look like this:

```
...
<ComboBox ItemsSource="{Binding Contacts}"
        SelectedItem="{Binding CurrentContact,Mode=TwoWay}"/>
...
```

This ensures whenever something is selected in the list, the current contact is updated. Remember that I mentioned multiple views might share the same viewmodel? In this case, the view for the contact details can use this same view model, and simply bind to the`CurrentContact` property.

Navigation

Navigation is a common issue to tackle. How do you manage navigation from an MVVM application? Most examples show only a single button or widget on the screen and don't tackle composite applications with multiple pages.

The short answer is that regardless of how you navigate (whether you use your own engine to pull in views, you use the navigation framework supplied by Silverlight, you use region management with Prism, or a combination of all of these), you should abstract the mechanism behind an interface. By defining `INavigation` or something similar, navigation no longer becomes an MVVM problem. However you solve it, your viewmodel can import `INavigation`and simply navigate to or trigger the transition as needed.

My post on MEF instead of Prism shows how to do this with the Managed Extensibility Framework. Auto-discoverable views using a Fluent interface covers mapping views to regions, and Dynamic module loading with Prism has a full solution using the navigation framework.

Dynamic Modules

This follows navigation. What if you have an extremely large application? It often doesn't make sense to load everything at once. You want the main menu and screen to appear, and then load other modules dynamically as they are needed. This cuts down on the initial time to get the application up and running, and also respects the user's browser and/or desktop memory and CPU.

The issue of dynamic modules isn't really specific to MVVM, but messaging between viewmodels and across modules is, of course, important. For these, I do believe it makes more sense to look at existing frameworks like MEF and Prism that solve the specific issue. Prism has modules that can be loaded "on demand", and MEF offers a deployment catalog that allows for dynamic loading of XAP files. Prism's solution for messaging across the application is the event aggregator. I talk more about frameworks and solutions for these types of problems in the appendix when I cover existing frameworks that are available to use "out of the box".

Dialog

A common UI pattern is the dialog box (similar to the message box, but expects a reply). I've seen a few people trip over how this can be implemented using both MVVM and the restriction by Silverlight that all code must be asynchronous.

The easiest solution in my opinion is to abstract the dialog behind an interface and provide a callback for the response. The view model can import the dialog, then based

on some change in state or a command, trigger the dialog service. The callback will return the response, and then the view can process accordingly.

Animations

This is a very common problem to tackle: how can changes triggered either in the UI or the backend kick off animations and other transitions?
There are several solutions to the problem. Here are a few examples of how to solve the problem:

- The Visual State Aggregator allows you to bind animations based on UI events without involving the viewmodel at all. When you do need to involve it, something like:
- Animation delegates can do the trick. Want something more abstract? Try
- nRoute's reverse ICommand implementation, or
- A sample MVVM framework that uses the visual state manager.

Configuration or Global Values

Another issue I see raised quite often is how to deal with global variables and configuration information. Again, this is less an MVVM problem and more a general architecture consideration. In most cases, you can expose configuration with an interface (`IConfiguration`) and then wire up an implementation with your configuration values. Any viewmodel that requires the information simply imports the implementation, whether via MEF, Unity, or some other mechanism, and only one copy of the class is kept (Singleton pattern, although most likely managed by the container and not the class itself).

Asynchronous Processes

One point of confusion with Silverlight is that it forces service calls to be asynchronous. This can seem strange when building a viewmodel: when do you trigger the call, and how do you know it is complete? Typically, this is managed by registered to an event when the process is complete, and binding the results. I prefer to hide the implementation details of the events behind a simple `Action` function. Read Simplifying Asynchronous Calls in Silverlight using Action for an example of this.

Sometimes you may have a more complex workflow that requires multiple asynchronous calls to execute and complete prior to continuing. If that is the case, you might want to look into a mechanism for making it easy to code and read the sequential workflow. This post will help you understand one solution using co-routines, and this post describes how to use an existing robust framework to manage those calls with thread safety, error handling, and more.

Huge Datasets

Finally, I've heard claims that MVVM doesn't handle large datasets well. I would argue it is certain implementations that have this issue, not the pattern itself. The solution is often to page the data, but I find many people approach the problem incorrectly. For some reason, developers want to insist paging is a function of the database and should be isolated to the data access layer. The simple fact that you have a UI element with "current

page" and "total pages" suggested it is not just an artifact of the database, but participates in all layers of the application, and should be managed as such.

In the most primitive form, you can create a collection that grows as the user pages. If your data is small, you might pull a very large collection and keep it in the Silverlight client, but use a virtualized panel to display the information (the problem with some panels is that they create a control for every bound data element, which can crush performance - virtualized panels only create enough controls to fill the visible window on the screen).

Technologies like WCF RIA support LINQ queries. These queries contain extension methods that allow you to grab only the first few items in a list, rather than fetching the full list at once. The framework also provides helper classes like the `PagedCollectionView` to help filter, sort, and page data.

What MVVM Isn't

No discussion would be complete unless we talked about what MVVM isn't.

MVVM isn't a complete framework. It's a pattern, and might be part of a framework, but it's only a piece of the overall solution for your application architecture. It doesn't address, and doesn't really care, about what happens on your server or how your services are put together. It does stress separation of concerns, which is nice.

I bet that nowhere in this article did you read a rule that stated, "With MVVM, code-behind is not allowed". This is a raging debate, but the pattern itself doesn't tell you how to implement your view, whether that is with XAML, code-behind, or a combination of the two. I would suggest that if you are spending days writing something just to avoid minutes of code-behind, your approach is wrong.

It is not required for Silverlight or WPF. I believe that line-of-business, data-driven, and forms-based applications are prime candidates for MVVM. Games, entertainment websites, paint programs, and others may not make sense. Be sure you are using the right tool for the right job.

MVVM is not supposed to slow you down! All new patterns and frameworks come with a learning curve. You'll have to accept that your developers need to learn and understand the pattern, but you should not accept that your entire process suddenly takes longer or becomes delayed. The pattern is useful when it accelerates development, improves stability and performance, reduces risk, and so forth. When it slows development, introduces problems, and has your developers cringing whenever they hear the phrase "design pattern", you might want to rethink your approach.

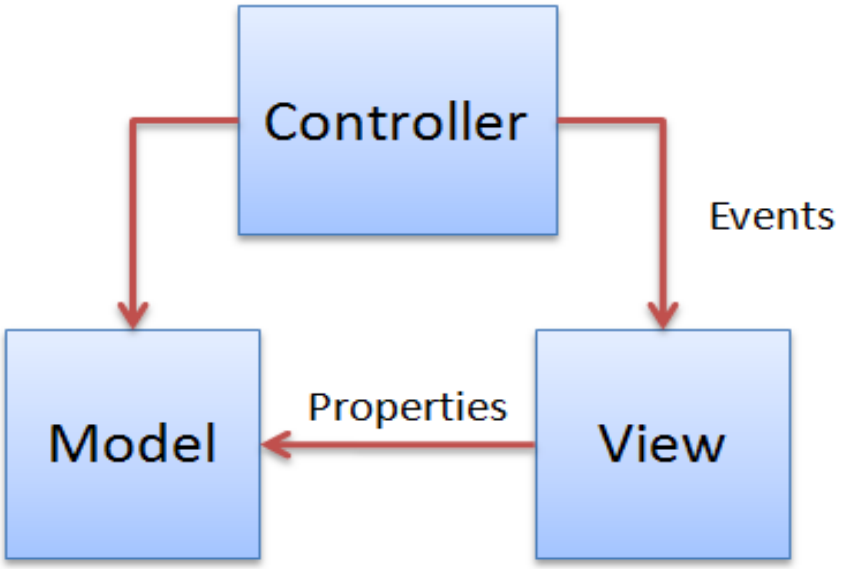
Conclusion

OK, we're done! That's it. I hope you've learned why MVVM is so powerful for Silverlight and WPF applications, what the pattern looks like, and even examples of solutions for common problems that MVVM can solve.

Appendix : Some Historical Patterns

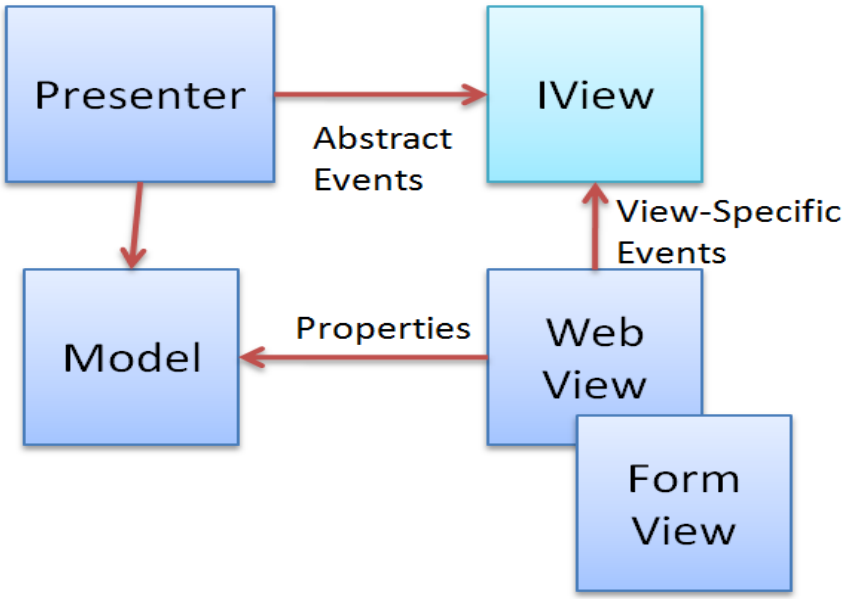
Model-View-Controller (MVC)

This software architecture pattern was first described in the context of Smalltalk at Xerox in 1979. If you are interested, you can download some of those original papers (PDF format) by clicking here (PDF).



Model-View-Presenter (MVP)

In 1996, the Model-View-Presenter pattern (PDF) was introduced to the world. This pattern builds on MVC, but places special constraints on the controller, now called the presenter. A general overview looks like this:



Martin Fowler describes this pattern with two flavors: the Supervising Controller/Presenter and the Passive View. Here is how Microsoft describes: MVP.

References

[1] Steve Sanderson. "KnockoutJS"

[2] Sencha Inc. "Are You Ready for Ext JS 5?"

[3] Massey, Simon. "Presentation Patterns In ZK". Retrieved 24 March 2012

[4] John Gossman. "Tales from the Smart Client: Introduction to Model/View/ViewModel pattern for building WPF apps"

[5] INTEL Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3, June 2009, pp. 3-1.

[6] Karl Shifflett. "Learning WPF M-V-VM."

[7] Martin Grund, Jan Schaffner, Jens Krueger, Jan Brunnert, and Alexander Zeier, "The Effects of Virtualization on Main Memory Systems," Proceedings of the Sixth International Workshop on Data Management on New Hardware, 2010, pp. 41-46, doi:10.1145/1869389.1869395.

[8] Peter J. Denning, "The locality principle," Communications of the ACM - Designing for the mobile device, vol. 48, Issue 7, July 2005, doi:10.1145/1070838.1070856.

[9] L. A. Belady, "A study of replacement algorithms for a virtualstorage computer," IBM Systems Journal, vol. 5, Issue 2, June 1966, pp.78-101, doi:10.1147/sj.52.0078.

[10] John Gossman. "Tales from the Smart Client: Advantages and disadvantages of M-V-VM."

[11] Wildermuth, Shawn. "Windows Presentation Foundation Data Binding: Part 1". Microsoft. Retrieved 24 March 2012.