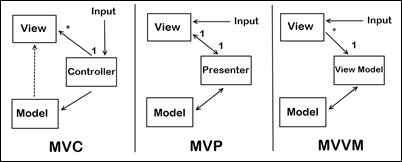
[](http://gwb.blob.core.windows.net/dlussier/WindowsLiveWriter/MVVMComparedToMVCandMVP_EFCB/image_4.png)

**MVC – Model View Controller**

Let’s look at MVC first. You’ll notice a few things about the diagram:

The input is directed at the Controller first, not the view. That input might be coming from a user interacting with a page, but it could also be from simply entering a specific url into a browser. In either case, its a Controller that is interfaced with to kick off some functionality.

There is a many-to-one relationship between the Controller and the View. That’s because a single controller may select different views to be rendered based on the operation being executed.

Note the one way arrow from Controller to View. This is because the View doesn’t have any knowledge of or reference to the controller.

The Controller does pass back the Model, so there is knowledge between the View and the expected Model being passed into it, but not the Controller serving it up.

**MVP – Model View Presenter**

Now let’s look at the MVP pattern. It looks very similar to MVC, except for some key distinctions:

The input begins with the View, not the Presenter.

There is a one-to-one mapping between the View and the associated Presenter.

The View holds a reference to the Presenter. The Presenter is also reacting to events being triggered from the View, so its aware of the View its associated with.

The Presenter updates the View based on the requested actions it performs on the Model, but the View is not Model aware.

**MVVM – Model View View Model**

So with the MVC and MVP patterns in front of us, let’s look at the MVVM pattern and see what differences it holds:

The input begins with the View, not the View Model.

While the View holds a reference to the View Model, the View Model has no information about the View. This is why its possible to have a one-to-many mapping between various Views and one View Model…even across technologies. For example, a WPF View and a Silverlight View \*could\* share the same View Model. However, my own feeling is that this is a bad practice and creates Franken-ViewModels that have too many responsibilities. It’s better to keep it as a one-to-one mapping instead.

You’ll also notice that the View has no idea about the Model in the MVVM pattern. This is because, as far as the View knows, its “Model” IS the View Model (hence its name). Because of how data-binding and other features like commanding work in WPF and Silverlight, there is rich communication between the View and View Model, isolating the View from having to know anything about what’s really happening behind the scenes.

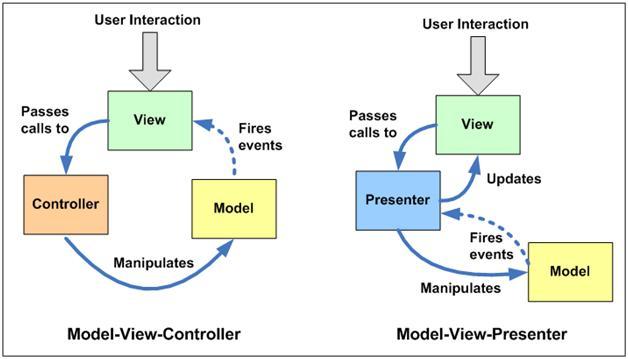
**Conclusion**

We could have gone deeper with this discussion, talking about the two different variations of MVP that Martin Fowler describes, or bring in other associated patterns like Front Controller. But at a high level, I think this gives us a good idea of the major differences between the three patterns.

# MVC vs. MVP vs. MVVM

[nirajrules](https://nirajrules.wordpress.com/author/nirajrules/" \o "Posts by nirajrules)  [Architecture Design](https://nirajrules.wordpress.com/category/architecture-design/), [Windows Presentation Foundation](https://nirajrules.wordpress.com/category/windows-presentation-foundation/)  July 18, 2009 4 Minutes

An important FAQ. The answer actually depends on where the person is coming from. MVC is a fundamental pattern which has been tweaked quite a bit to fit into various platforms. For instance if you had asked anybody how to implement an MVC in ASP.NET (prior to release of ASP.NET MVC framework) you would get very different answers. So let’s start with basic. The common motivation behind all 3 is separation of concerns, cutting flab from UI (good for UI designers), swapping UIs (for instance windows to web), make UI easy for Unit Testing, etc. Have a look at the below diagram, I have taken it from [CAB](http://www.codeplex.com/smartclient/Wiki/View.aspx?title=Composite%20UI%20Application%20Block) documentation.



**MVC:** Three components – View (your UI), Model (your business entities / data – that view is displaying) & Controller (contains the logic that alters the model depending on the action triggered by UI, typically implementing a Use Case). It’s widely known that MVC is a compound pattern (View and Controller have Strategy implementation, View itself can be a Composite implementation & View and Model are synched through Observer). In this case Controller doesn’t know anything about View, and the idea is that a View can switch Controllers (for instance depending upon who has logged to the system) & a single controller can be used by multiple Views. View subscribes to the changes done to the model & hence both are sync from the data perspective. One of the disadvantages of MVC is that it’s difficult to unit test. Controller manipulates the data but how about asserting those changes from a view perspective. For instance on click of a button you raise an event to controller, and controller modifies the value in model. This value modification changes the font size / color in View. Unit testing this scenario is slightly difficult in MVC.

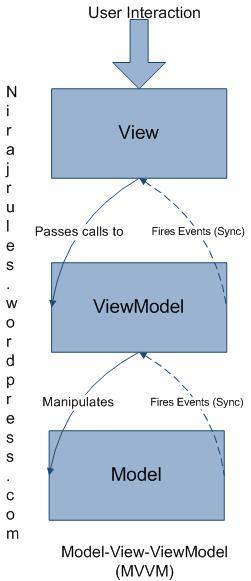
**MVP:** Again three components. But dependencies change (look at arrows in the diagram). Over here we replace Controller with Presenter (one which presents the changes done in model back to view). The main difference between both is that Presenter refers back to the view while Controller doesn’t. Normal pattern found here is to create an abstraction of the View (in terms of properties / events) & Presenter refers to it. This makes the [mocking](https://nirajrules.wordpress.com/tag/rhino-mocks/) of View much easier & hence the Unit Testing aspect. Presenter here hence takes the responsibility of not only manipulating model but also updating the view. Of course the implementations of MVP differ in real world in terms of how much thin the view is, some prefer keeping basic logic still inside view & taking complex logic in presenter, while others prefer keeping the entire logic in Presenter. Martin fowler describes 2 variations on MVP on these lines namely – Supervising Controller & Passive View described below

(A [Passive View](http://martinfowler.com/eaaDev/PassiveScreen.html) handles this by reducing the behavior of the UI components to the absolute minimum by using a controller that not just handles responses to user events, but also does all the updating of the view. This allows testing to be focused on the controller with little risk of problems in the view.

REPORT THIS AD

[Supervising Controller](http://martinfowler.com/eaaDev/SupervisingPresenter.html) uses a controller both to handle input response but also to manipulate the view to handle more complex view logic. It leaves simple view behavior to the declarative system, intervening only when effects are needed that are beyond what can be achieved declaratively.)

**MVVM:** Model–View-ViewModel talks of creating a new model (in addition to your domain model). This model normally adds additonal properties from the prespective of View (as we understand that View has controls in addition to data which it’s displaying). For instance if View had a property IsChecked and Presenter was setting in classic MVP, in MVVM Presenter will have that IsChecked Property which View will sync up with (doesn’t it look like Strategy pattern has been replaced with Observer?). So now a Presenter becomes more like a combo of – View Properties & Model properties which would be synchronized with View. So why not rename Presenter to ViewModel? Do that and you get MVVM. MVVM is attractive for platforms which support bi-directional binding with less effort. Also a minor tradeoff is ViewModel unlike Presenter can stand on its own (Presenter normally requires a View’s interface). Martin fowler describes similar pattern called [Presentation Model](http://martinfowler.com/eaaDev/PresentationModel.html) & Josh Smith captures MVVM implementation for [WPF](https://nirajrules.wordpress.com/category/windows-presentation-foundation/) / Silverlight in this [article](http://msdn.microsoft.com/en-us/magazine/dd419663.aspx).



**ASP.NET MVC:** So what has MVC got to do with ASP.NET MVC? First, Web works on a different model. Here, user interacts with HTML in browser and send a request back to the server for processing (for client side Ajax you might go just for data). As the interaction is normally stateless, when the request comes back to the server we need to recreate our View, load the model back & manipulate both of them as required. There are 2 variations on how handle this recreation – [Page Controller](http://martinfowler.com/eaaCatalog/pageController.html) & [Front Controller](http://martinfowler.com/eaaCatalog/frontController.html). Make Page the decision maker – in this widely implemented pattern HTTP request is specific to physical page on server (.aspx for instance) & page in turn creates itself (builds the view from postback data) decides what model it needs and triggers the manipulation (events in codebehind file) it requires. As you see here the distinction between View & Controller becomes blur & is little difficult to separate. This where ASP.NET MVC comes in which behaves like a Front Controller – where Controller is the decision maker. Here all HTTP requests are mapped to methods on the Controller Class. Controller class recreates the model & view as required and does the manipulations. This makes unit testing easier as we can directly instantiate the front controller class & invoke methods on it to perform the assertions.

I can add code snippets to above explanations if you feel they would help you understand things better. I will look forward to your comments  .

# MVC Versus MVP Versus MVVM

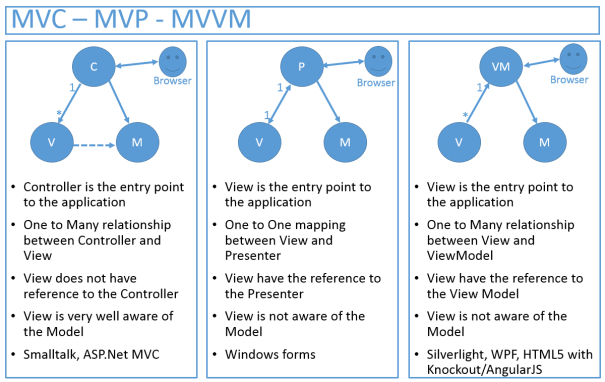
Both MVP and MVVM are derivatives of MVC. The key difference between MVC and its derivatives is the dependency each layer has on other layers, as well as how tightly bound they are to each other.

In MVC, the View sits on top of our architecture with the controller beside it. Models sit below the controller, so our Views know about our controllers and controllers know about Models. Here, our Views have direct access to Models. Exposing the complete Model to the View, however, may have security and performance costs, depending on the complexity of our application. MVVM attempts to avoid these issues.

In MVP, the role of the controller is replaced with a Presenter. Presenters sit at the same level as views, listening to events from both the View and model, and mediating the actions between them. Unlike MVVM, there isn’t a mechanism for binding Views to ViewModels, so we instead rely on each View implementing an interface allowing the Presenter to interact with the View.

MVVM consequently allows us to create View-specific subsets of a Model, which can contain state and logic information, avoiding the need to expose the entire Model to a View. Unlike MVP’s Presenter, a ViewModel is not required to reference a View. The View can bind to properties on the ViewModel, which in turn expose data contained in Models to the View. As we’ve mentioned, the abstraction of the View means there is less logic required in the code behind it.

One of the downsides to this however is that a level of interpretation is needed between the ViewModel and the View, and this can have performance costs. The complexity of this interpretation can also vary: it can be as simple as copying data or as complex as manipulating it to a form we would like the View to see. MVC doesn’t have this problem, as the whole Model is readily available and such manipulation can be avoided.



[https://martinfowler.com/mf-name-white.png](https://martinfowler.com/)

Top of Form



Bottom of Form

* [Intro](https://martinfowler.com/intro.html)
* [Videos](https://martinfowler.com/videos.html)
* [Design](https://martinfowler.com/design.html)
* [Agile](https://martinfowler.com/agile.html)
* [Refactoring](http://refactoring.com/)
* [FAQ](https://martinfowler.com/faq.html)
* [About Me](https://martinfowler.com/aboutMe.html)
* [All Sections](https://martinfowler.com/eaaDev/uiArchs.html#bottom-navmenu)
* [ThoughtWorks](http://www.thoughtworks.com/)

# GUI Architectures

*There have been many different ways to organize the code for a rich client system. Here I discuss a selection of those that I feel have been the most influential and introduce how they relate to the patterns.*

18 July 2006



[**Martin Fowler**](https://martinfowler.com/)

**Translations:**[Greek](http://anguerde.com/uiarchs/)

## Contents

* [Forms and Controls](https://martinfowler.com/eaaDev/uiArchs.html#FormsAndControls)
* [Model View Controller](https://martinfowler.com/eaaDev/uiArchs.html#ModelViewController)
* [VisualWorks Application Model](https://martinfowler.com/eaaDev/uiArchs.html#VisualworksApplicationModel)
* [Model-View-Presenter (MVP)](https://martinfowler.com/eaaDev/uiArchs.html#Model-view-presentermvp)
* [Humble View](https://martinfowler.com/eaaDev/uiArchs.html#HumbleView)

This is part of the [Further Enterprise Application Architecture development](https://martinfowler.com/eaaDev) writing that I was doing in the mid 2000’s. Sadly too many other things have claimed my attention since, so I haven’t had time to work on them further, nor do I see much time in the foreseeable future. As such this material is very much in draft form and I won’t be doing any corrections or updates until I’m able to find time to work on it again.

Graphical user interfaces have become a familiar part of our software landscape, both as users and as developers. Looking at it from a design perspective they represent a particular set of problems in system design - problems that have led to a number of different but similar solutions.

My interest is identifying common and useful patterns for application developers to use in rich-client development. I've seen various designs in project reviews and also various designs that have been written in a more permanent way. Inside these designs are the useful patterns, but describing them is often not easy. Take Model-View-Controller as an example. It's often referred to as a pattern, but I don't find it terribly useful to think of it as a pattern because it contains quite a few different ideas. Different people reading about MVC in different places take different ideas from it and describe these as 'MVC'. If this doesn't cause enough confusion you then get the effect of misunderstandings of MVC that develop through a system of Chinese whispers.

In this essay I want to explore a number of interesting architectures and describe my interpretation of their most interesting features. My hope is that this will provide a context for understanding the patterns that I describe.

To some extent you can see this essay as a kind of intellectual history that traces ideas in UI design through multiple architectures over the years. However I must issue a caution about this. Understanding architectures isn't easy, especially when many of them change and die. Tracing the spread of ideas is even harder, because people read different things from the same architecture. In particular I have not done an exhaustive examination of the architectures I describe. What I have done is referred to common descriptions of the designs. If those descriptions miss things out, I'm utterly ignorant of that. So don't take my descriptions as authoritative. Furthermore there are things I've left out or simplified if I didn't think they were particularly relevant. Remember my primary interest is the underlying patterns, not in the history of these designs.

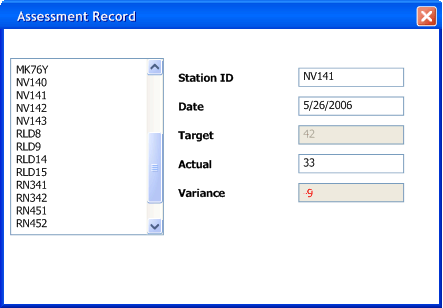
(There is something of an exception here, in that I did have access to a running Smalltalk-80 to examine MVC. Again I wouldn't describe my examination of it as exhaustive, but it did reveal things that common descriptions of it failed to - which even further makes me cautious about descriptions of other architectures that I have here. If you are familiar with one of these architectures and you see I have something important that is incorrect and missing I'd like to know about it. I also think that a more exhaustive survey of this territory would be a good object of academic study.)

## Forms and Controls

I shall begin this exploration with an architecture that is both simple and familiar. It doesn't have a common name, so for the purposes of this essay I shall call it "Forms and Controls". It's a familiar architecture because it was the one encouraged by client-server development environments in the 90's - tools like Visual Basic, Delphi, and Powerbuilder. It continues to be commonly used, although also often vilified by design geeks like me.

To explore it, and indeed the other architectures, I'll use a common example. In New England, where I live, there is a government program that monitors the amount of ice-cream particulate in the atmosphere. If the concentration is too low, this indicates that we aren't eating enough ice-cream - which poses a serious risk to our economy and public order. (I like to use examples that are no less realistic as you usually find in books like this.)

To monitor our ice-cream health, the government has set up monitoring stations all over the New England states. Using complex atmospheric modeling the department sets a target for each monitoring station. Every so often staffers go out on an assessment where they go to various stations and note the actual ice-cream particulate concentrations. This UI allows them to select a station, and enter the date and actual value. The system then calculates and displays the variance from the target. The system highlights the variance in red when it is 10% or more below the target, or in green when 5% or more above the target.

**

*Figure 1: The UI I'll use as an example.*

As we look at this screen we can see there is an important division as we put it together. The form is specific to our application, but it uses controls that are generic. Most GUI environments come with a hefty bunch of common controls that we can just use in our application. We can build new controls ourselves, and often it's a good idea to do so, but there is still a distinction between generic reusable controls and specific forms. Even specially written controls can be reused across multiple forms.

The form contains two main responsibilities:

* Screen layout: defining the arrangement of the controls on the screen, together with their hierarchic structure with one other.
* Form logic: behavior that cannot be easily programmed into the controls themselves.

Most GUI development environments allow the developer to define screen layout with a graphical editor that allows you to drag and drop the controls onto a space in the form. This pretty much handles the form layout. This way it's easy to setup a pleasing layout of controls on the form (although it isn't always the best way to do it - we'll come to that later.)

The controls display data - in this case about the reading. This data will pretty much always come from somewhere else, in this case let's assume a SQL database as that's the environment that most of these client-server tools assume. In most situations there are three copies of the data involved:

* One copy of data lies in the database itself. This copy is the lasting record of the data, so I call it the **record state**. The record state is usually shared and visible to multiple people via various mechanisms.
* A further copy lies inside in-memory [Record Sets](https://martinfowler.com/eaaCatalog/recordSet.html) within the application. Most client-server environments provided tools which made this easy to do. This data was only relevant for one particular session between the application and the database, so I call it **session state**. Essentially this provides a temporary local version of the data that the user works on until they save, or commit it, back to the database - at which point it merges with the record state. I won't worry about the issues around coordinating record state and session state here: I did go into various techniques in [[P of EAA]](https://martinfowler.com/books.html#eaa).
* The final copy lies inside the GUI components themselves. This, strictly, is the data they see on the screen, hence I call it the **screen state**. It is important to the UI how screen state and session state are kept synchronized.

Keeping screen state and session state synchronized is an important task. A tool that helped make this easier was [Data Binding](https://martinfowler.com/eaaDev/DataBinding.html). The idea was that any change to either the control data, or the underlying record set was immediately propagated to the other. So if I alter the actual reading on the screen, the text field control effectively updates the correct column in the underlying record set.

In general data binding gets tricky because if you have to avoid cycles where a change to the control, changes the record set, which updates the control, which updates the record set.... The flow of usage helps avoid these - we load from the session state to the screen when the screen is opened, after that any changes to the screen state propagate back to the session state. It's unusual for the session state to be updated directly once the screen is up. As a result data binding might not be entirely bi-directional - just confined to initial upload and then propagating changes from the controls to the session state.

[Data Binding](https://martinfowler.com/eaaDev/DataBinding.html) handles much of the functionality of a client-sever application pretty nicely. If I change the actual value the column is updated, even changing the selected station alters the currently selected row in the record set, which causes the other controls to refresh.

Much of this behavior is built in by the framework builders, who look at common needs and make it easy to satisfy them. In particular this is done by setting values, usually called properties, on the controls. The control binds to a particular column in a record set by having its column name set through a simple property editor.

Using data binding, with the right kind of parameterization, can take you a long way. However it can't take you all the way - there's almost always some logic that won't fit with the parameterization options. In this case calculating the variance is an example of something that doesn't fit in this built in behavior - since it's application specific it usually lies in the form.

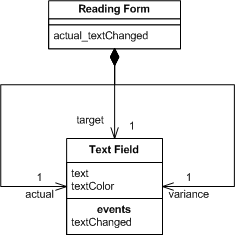
In order for this to work the form needs to be alerted whenever the value of the actual field changes, which requires the generic text field to call some specific behavior on the form. This is a bit more involved than taking a class library and using it through calling it as Inversion of Control is involved.

There are various ways of getting this kind of thing to work - the common one for client-server toolkits was the notion of events. Each control had a list of events it could raise. Any external object could tell a control that it was interested in an event - in which case the control would call that external object when the event was raised. Essentially this is just a rephrasing of the [Observer](http://www.amazon.com/exec/obidos/tg/detail/-/0201633612) pattern where the form is observing the control. The framework usually provided some mechanism where the developer of the form could write code in a subroutine that would be invoked when the event occurred. Exactly how the link was made between event and routine varied between platform and is unimportant for this discussion - the point is that some mechanism existed to make it happen.

Once the routine in the form has control, it can then do whatever is needed. It can carry out the specific behavior and then modify the controls as necessary, relying on data binding to propagate any of these changes back to the session state.

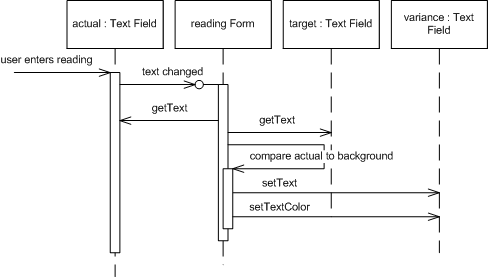
This is also necessary because data binding isn't always present. There is a large market for windows controls, not all of them do data binding. If data binding isn't present then it's up to the form to carry out the synchronization. This could work by pulling data out of the record set into the widgets initially, and copying the changed data back to the record set when the save button was pressed.

Let's examine our editing of the actual value, assuming that data binding is present. The form object holds direct references to the generic controls. There'll be one for each control on the screen, but I'm just interested in the actual, variance, and target fields here.

**

*Figure 2: Class diagram for forms and controls*

The text field declares an event for text changed, when the form assembles the screen during initialization it subscribes itself to that event, binding it a method on itself - here actual\_textChanged.

**

*Figure 3: Sequence diagram for changing a genre with forms and controls.*

When the user changes the actual value, the text field control raises its event and through the magic of framework binding the actual\_textChanged is run. This method gets the text from the actual and target text fields, does the subtraction, and puts the value into the variance field. It also figures out what color the value should be displayed with and adjusts the text color appropriately.

We can summarize the architecture with a few soundbites:

* Developers write application specific forms that use generic controls.
* The form describes the layout of controls on it.
* The form observes the controls and has handler methods to react to interesting events raised by the controls.
* Simple data edits are handled through data binding.
* Complex changes are done in the form's event handling methods.

## Model View Controller

Probably the widest quoted pattern in UI development is Model View Controller (MVC) - it's also the most misquoted. I've lost count of the times I've seen something described as MVC which turned out to be nothing like it. Frankly a lot of the reason for this is that parts of classic MVC don't really make sense for rich clients these days. But for the moment we'll take a look at its origins.

As we look at MVC it's important to remember that this was one of the first attempts to do serious UI work on any kind of scale. Graphical User Interfaces were not exactly common in the 70's. The Forms and Controls model I've just described came after MVC - I've described it first because it's simpler, not always in a good way. Again I'll discuss Smalltalk 80's MVC using the assessment example - but be aware that I am taking a few liberties with the actual details of Smalltalk 80 to do this - for start it was a monochrome system.

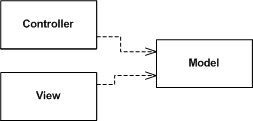
At the heart of MVC, and the idea that was the most influential to later frameworks, is what I call [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html). The idea behind [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html) is to make a clear division between domain objects that model our perception of the real world, and presentation objects that are the GUI elements we see on the screen. Domain objects should be completely self contained and work without reference to the presentation, they should also be able to support multiple presentations, possibly simultaneously. This approach was also an important part of the Unix culture, and continues today allowing many applications to be manipulated through both a graphical and command-line interface.

In MVC, the domain element is referred to as the model. Model objects are completely ignorant of the UI. To begin discussing our assessment UI example we'll take the model as a reading, with fields for all the interesting data upon it. (As we'll see in a moment the presence of the list box makes this question of what is the model rather more complex, but we'll ignore that list box for a little bit.)

In MVC I'm assuming a [Domain Model](https://martinfowler.com/eaaCatalog/domainModel.html) of regular objects, rather than the [Record Set](https://martinfowler.com/eaaCatalog/recordSet.html)notion that I had in Forms and Controls. This reflects the general assumption behind the design. Forms and Controls assumed that most people wanted to easily manipulate data from a relational database, MVC assumes we are manipulating regular Smalltalk objects.

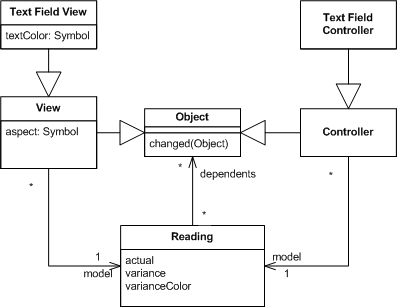
The presentation part of MVC is made of the two remaining elements: view and controller. The controller's job is to take the user's input and figure out what to do with it.

At this point I should stress that there's not just one view and controller, you have a view-controller pair for each element of the screen, each of the controls and the screen as a whole. So the first part of reacting to the user's input is the various controllers collaborating to see who got edited. In this case that's the actuals text field so that text field controller would now handle what happens next.

**

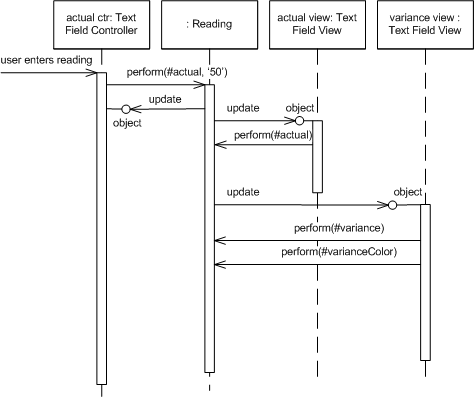
*Figure 4: Essential dependencies between model, view, and controller. (I call this essential because in fact the view and controller do link to each other directly, but developers mostly don't use this fact.)*

Like later environments, Smalltalk figured out that you wanted generic UI components that could be reused. In this case the component would be the view-controller pair. Both were generic classes, so needed to be plugged into the application specific behavior. There would be an assessment view that would represent the whole screen and define the layout of the lower level controls, in that sense similar to a form in Forms and Controllers. Unlike the form, however, MVC has no event handlers on the assessment controller for the lower level components.

**

*Figure 5: Classes for an MVC version of an ice-cream monitor display*

The configuration of the text field comes from giving it a link to its model, the reading, and telling it what what method to invoke when the text changes. This is set to '#actual:' when the screen is initialized (a leading '#' indicates a symbol, or interned string, in Smalltalk). The text field controller then makes a reflective invocation of that method on the reading to make the change. Essentially this is the same mechanism as occurs for [Data Binding](https://martinfowler.com/eaaDev/DataBinding.html), the control is linked to the underlying object (row) and told which method (column) it manipulates.

**

*Figure 6: Changing the actual value for MVC.*

So there is no overall object observing low level widgets, instead the low level widgets observe the model, which itself handles many of the decision that would be made by the form. In this case, when it comes to figuring out the variance, the reading object itself is the natural place to do that.

Observers do occur in MVC, indeed it's one of the ideas credited to MVC. In this case all the views and controllers observe the model. When the model changes, the views react. In this case the actual text field view is notified that the reading object has changed, and invokes the method defined as the aspect for that text field - in this case #actual - and sets its value to the result. (It does something similar for the color, but this raises its own specters that I'll get to in a moment.)

You'll notice that the text field controller didn't set the value in the view itself, it updated the model and then just let the observer mechanism take care of the updates. This is quite different to the forms and controls approach where the form updates the control and relies on data binding to update the underlying record-set. These two styles I describe as patterns: [Flow Synchronization](https://martinfowler.com/eaaDev/FlowSynchronization.html) and [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html). These two patterns describe alternative ways of handling the triggering of synchronization between screen state and session state. Forms and Controls do it through the flow of the application manipulating the various controls that need to be updated directly. MVC does it by making updates on the model and then relying of the observer relationship to update the views that are observing that model.

[Flow Synchronization](https://martinfowler.com/eaaDev/FlowSynchronization.html) is even more apparent when data binding isn't present. If the application needs to do synchronization itself, then it was typically done at important point in the application flow - such as when opening a screen or hitting the save button.

One of the consequences of [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) is that the controller is very ignorant of what other widgets need to change when the user manipulates a particular widget. While the form needs to keep tabs on things and make sure the overall screen state is consistent on a change, which can get pretty involved with complex screens, the controller in [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) can ignore all this.

This useful ignorance becomes particularly handy if there are multiple screens open viewing the same model objects. The classic MVC example was a spreadsheet like screen of data with a couple of different graphs of that data in separate windows. The spreadsheet window didn't need to be aware of what other windows were open, it just changed the model and [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) took care of the rest. With [Flow Synchronization](https://martinfowler.com/eaaDev/FlowSynchronization.html) it would need some way of knowing which other windows were open so it tell them to refresh.

While [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) is nice it does have a downside. The problem with [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) is the core problem of the observer pattern itself - you can't tell what is happening by reading the code. I was reminded of this very forcefully when trying to figure out how some Smalltalk 80 screens worked. I could get so far by reading the code, but once the observer mechanism kicked in the only way I could see what was going on was via a debugger and trace statements. Observer behavior is hard to understand and debug because it's implicit behavior.

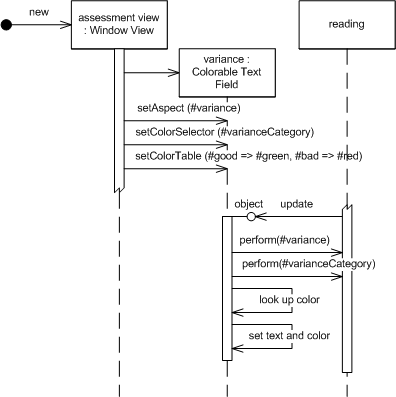
While the different approaches to synchronization are particularly noticeable from looking at the sequence diagram, the most important, and most influential, difference is MVC's use of [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html). Calculating the variance between actual and target is domain behavior, it is nothing to do with the UI. As a result following [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html) says we should place this in the domain layer of the system - which is exactly what the reading object represents. When we look at the reading object, the variance feature makes complete sense without any notion of the user interface.

At this point, however, we can begin to look at some complications. There's two areas where I've skipped over some awkward points that get in the way of MVC theory. The first problem area is to deal with setting the color of the variance. This shouldn't really fit into a domain object, as the color by which we display a value isn't part of the domain. The first step in dealing with this is to realize that part of the logic is domain logic. What we are doing here is making a qualitative statement about the variance, which we could term as good (over by more than 5%), bad (under by more than 10%), and normal (the rest). Making that assessment is certainly domain language, mapping that to colors and altering the variance field is view logic. The problem lies in where we put this view logic - it's not part of our standard text field.

This kind of problem was faced by early smalltalkers and they came up with some solutions. The solution I've shown above is the dirty one - compromise some of the purity of the domain in order to make things work. I'll admit to the occasional impure act - but I try not to make a habit of it.

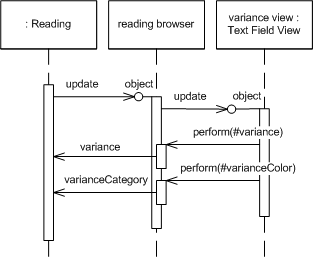
We could do pretty much what Forms and Controls does - have the assessment screen view observe the variance field view, when the variance field changes the assessment screen could react and set the variance field's text color. Problems here include yet more use of the observer mechanism - which gets exponentially more complicated the more you use it - and extra coupling between the various views.

A way I would prefer is to build a new type of the UI control. Essentially what we need is a UI control that asks the domain for a qualitative value, compares it to some internal table of values and colors, and sets the font color accordingly. Both the table and message to ask the domain object would be set by the assessment view as it's assembling itself, just as it sets the aspect for the field to monitor. This approach could work very well if I can easily subclass text field to just add the extra behavior. This obviously depends on how well the components are designed to enable sub-classing - Smalltalk made it very easy - other environments can make it more difficult.

**

*Figure 7: Using a special subclass of text field that can be configured to determine the color.*

The final route is to make a new kind of model object, one that's oriented around around the screen, but is still independent of the widgets. It would be the model for the screen. Methods that were the same as those on the reading object would just be delegated to the reading, but it would add methods that supported behavior relevant only to the UI, such as the text color.

**

*Figure 8: Using an intermediate*[*Presentation Model*](https://martinfowler.com/eaaDev/PresentationModel.html)*to handle view logic.*

This last option works well for a number of cases and, as we'll see, became a common route for Smalltalkers to follow - I call this a [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) because it's a model that is really designed for and thus part of the presentation layer.

The [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) works well also for another presentation logic problem - presentation state. The basic MVC notion assumes that all the state of the view can be derived from the state of the model. In this case how do we figure out which station is selected in the list box? The [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) solves this for us by giving us a place to put this kind of state. A similar problem occurs if we have save buttons that are only enabled if data has changed - again that's state about our interaction with the model, not the model itself.

So now I think it's time for some soundbites on MVC.

* Make a strong separation between presentation (view & controller) and domain (model) - [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html).
* Divide GUI widgets into a controller (for reacting to user stimulus) and view (for displaying the state of the model). Controller and view should (mostly) not communicate directly but through the model.
* Have views (and controllers) observe the model to allow multiple widgets to update without needed to communicate directly - [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html).

## VisualWorks Application Model

As I've discussed above, Smalltalk 80's MVC was very influential and had some excellent features, but also some faults. As Smalltalk developed in the 80's and 90's this led to some significant variations on the classic MVC model. Indeed one could almost say that MVC disappeared, if you consider the view/controller separation to be an essential part of MVC - which the name does imply.

The things that clearly worked from MVC were [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html) and [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html). So these stayed as Smalltalk developed - indeed for many people they were the key element of MVC.

Smalltalk also fragmented in these years. The basic ideas of Smalltalk, including the (minimal) language definition remained the same, but we saw multiple Smalltalks develop with different libraries. From a UI perspective this became important as several libraries started using native widgets, the controls used by the Forms and Controls style.

Smalltalk was originally developed by Xerox Parc labs and they span off a separate company, ParcPlace, to market and develop Smalltalk. ParcPlace Smalltalk was called VisualWorks and made a point of being a cross-platform system. Long before Java you could take a Smalltalk program written in Windows and run it right away on Solaris. As a result VisualWorks didn't use native widgets and kept the GUI completely within Smalltalk.

In my discussion of MVC I finished with some problems of MVC - particularly how to deal with view logic and view state. VisualWorks refined its framework to deal with this by coming up with a construct called the Application Model - a construct that moves towards [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html). The idea of using something like a [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) wasn't new to VisualWorks - the original Smalltalk 80 code browser was very similar, but the VisualWorks Application Model baked it fully into the framework.

A key element of this kind of Smalltalk was the idea of turning properties into objects. In our usual notion of objects with properties we think of a Person object having properties for name and address. These properties may be fields, but could be something else. There is usually a standard convention for accessing the properties: in Java we would see temp = aPerson.getName() andaPerson.setName("martin"), in C# it would temp = aPerson.name and aPerson.name = "martin".

A **Property Object** changes this by having the property return an object that wraps the actual value. So in VisualWorks when we ask for a name we get back a wrapping object. We then get the actual value by asking the wrapping object for its value. So accessing a person's name would use temp = aPerson name value and aPerson name value: 'martin'

Property objects make the mapping between widgets and model a little easier. We just have to tell the widget what message to send to get the corresponding property, and the widget knows to access the proper value using value and value:. VisualWorks's property objects also allow you to set up observers with the message onChangeSend: aMessage to: anObserver.

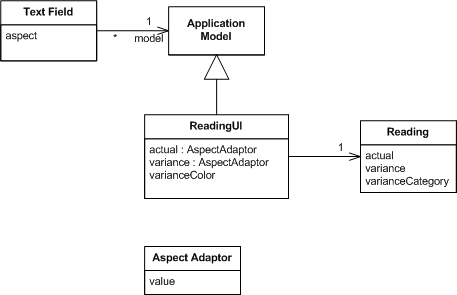
You won't actually find a class called property object in Visual Works. Instead there were a number of classes that followed the value/value:/onChangeSend: protocol. The simplest is the ValueHolder - which just contains its value. More relevant to this discussion is the AspectAdaptor. The AspectAdaptor allowed a property object to wrap a property of another object completely. This way you could define a property object on a PersonUI class that wrapped a property on a Person object by code like

adaptor := AspectAdaptor subject: person

adaptor forAspect: #name

adaptor onChangeSend: #redisplay to: self

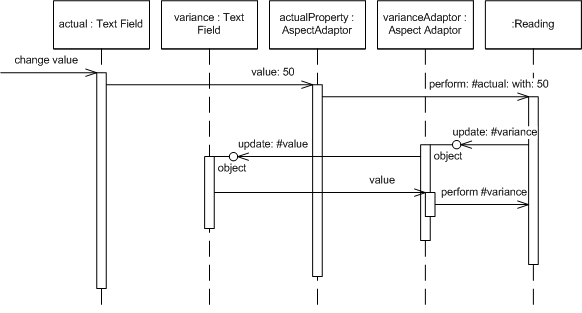
So let's see how the application model fits into our running example.

**

*Figure 9: Class diagram for visual works application model on the running example*

The main difference between using an application model and classic MVC is that we now have an intermediate class between the domain model class (Reader) and the widget - this is the application model class. The widgets don't access the domain objects directly - their model is the application model. Widgets are still broken down into views and controllers, but unless you're building new widgets that distinction isn't important.

When you assemble the UI you do so in a UI painter, while in that painter you set the aspect for each widget. The aspect corresponds to a method on the application model that returns a property object.

**

*Figure 10: Sequence diagram showing how updating the actual value updates the variance text.*

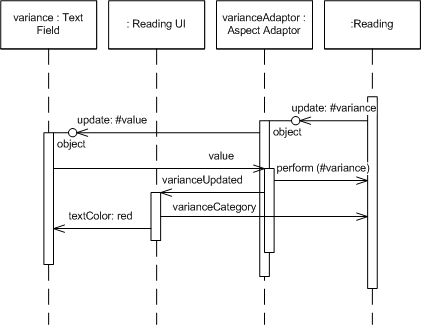
[Figure 10](https://martinfowler.com/eaaDev/uiArchs.html#uiArchs_appModelNoColor-seq.gif) shows how the basic update sequence works. When I change a value in text field, that field then updates the value in the property object inside the application model. That update follows through to the underlying domain object, updating its actual value.

At this point the observer relationships kick in. We need to set things up so that updating the actual value causes the reading to indicate that it has changed. We do this by putting a call in the modifier for actual to indicate that the reading object has changed - in particular that the variance aspect has changed. When setting up the aspect adaptor for variance it's easy to tell it to observe the reader, so it picks up the update message which it then forwards to its text field. The text field then initiates getting a new value, again through the aspect adaptor.

Using the application model and property objects like this helps us wire up the updates without having to write much code. It also supports fine-grained synchronization (which I don't think is a good thing).

Application models allow us to separate behavior and state that's particular to the UI from real domain logic. So one of the problems I mentioned earlier, holding the currently selected item in a list, can be solved by using a particular kind of aspect adaptor that wraps the domain model's list and also stores the currently selected item.

The limitation of all this, however, is that for more complex behavior you need to construct special widgets and property objects. As an example the provided set of objects don't provide a way to link the text color of the variance to the degree of variance. Separating the application and domain models does allow us to separate the decision making in the right way, but then to use widgets observing aspect adapters we need to make some new classes. Often this was seen as too much work, so we could make this kind of thing easier by allowing the application model to access the widgets directly, as in [Figure 11](https://martinfowler.com/eaaDev/uiArchs.html#uiArchs_appModelColoring-seq.gif).

**

*Figure 11: Application Model updates colors by manipulating widgets directly.*

Directly updating the widgets like this is not part of [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html), which is why the visual works application model isn't truly a [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html). This need to manipulate the widgets directly was seen by many as a bit of a dirty workaround and helped develop the Model-View-Presenter approach.

So now the soundbites on Application Model

* Followed MVC in using [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html) and [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html).
* Introduced an intermediate application model as a home for presentation logic and state - a partial development of [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html).
* Widgets do not observe domain objects directly, instead they observe the application model.
* Made extensive use of Property Objects to help connect the various layers and to support the fine grained synchronization using observers.
* It wasn't the default behavior for the application model to manipulate widgets, but it was commonly done for complicated cases.

## Model-View-Presenter (MVP)

MVP is an architecture that first appeared in IBM and more visibly at Taligent during the 1990's. It's most commonly referred via the [Potel](http://www.wildcrest.com/Potel/Portfolio/mvp.pdf) paper. The idea was further popularized and described by the developers of [Dolphin Smalltalk](http://www.object-arts.com/papers/TwistingTheTriad.PDF). As we'll see the two descriptions don't entirely mesh but the basic idea underneath it has become popular.

To approach MVP I find it helpful to think about a significant mismatch between two strands of UI thinking. On the one hand is the Forms and Controller architecture which was the mainstream approach to UI design, on the other is MVC and its derivatives. The Forms and Controls model provides a design that is easy to understand and makes a good separation between reusable widgets and application specific code. What it lacks, and MVC has so strongly, is [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html) and indeed the context of programming using a [Domain Model](https://martinfowler.com/eaaCatalog/domainModel.html). I see MVP as a step towards uniting these streams, trying to take the best from each.

The first element of [Potel](http://www.wildcrest.com/Potel/Portfolio/mvp.pdf) is to treat the view as a structure of widgets, widgets that correspond to the controls of the Forms and Controls model and remove any view/controller separation. The view of MVP is a structure of these widgets. It doesn't contain any behavior that describes how the widgets react to user interaction.

The active reaction to user acts lives in a separate presenter object. The fundamental handlers for user gestures still exist in the widgets, but these handlers merely pass control to the presenter.

The presenter then decides how to react to the event. [Potel](http://www.wildcrest.com/Potel/Portfolio/mvp.pdf) discusses this interaction primarily in terms of actions on the model, which it does by a system of commands and selections. A useful thing to highlight here is the approach of packaging all the edits to the model in a command - this provides a good foundation for providing undo/redo behavior.

As the Presenter updates the model, the view is updated through the same [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) approach that MVC uses.

The [Dolphin](http://www.object-arts.com/papers/TwistingTheTriad.PDF) description is similar. Again the main similarity is the presence of the presenter. In the Dolphin description there isn't the structure of the presenter acting on the model through commands and selections. There is also explicit discussion of the presenter manipulating the view directly. Potel doesn't talk about whether presenters should do this or not, but for Dolphin this ability was essential to overcoming the kind of flaw in Application Model that made it awkward for me to color the text in the variation field.

One of the variations in thinking about MVP is the degree to which the presenter controls the widgets in the view. On one hand there is the case where all view logic is left in the view and the presenter doesn't get involved in deciding how to render the model. This style is the one implied by [Potel](http://www.wildcrest.com/Potel/Portfolio/mvp.pdf). The direction behind [Bower and McGlashan](http://www.object-arts.com/papers/TwistingTheTriad.PDF) was what I'm calling [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html), where the view handles a good deal of the view logic that can be described declaratively and the presenter then comes in to handle more complex cases.

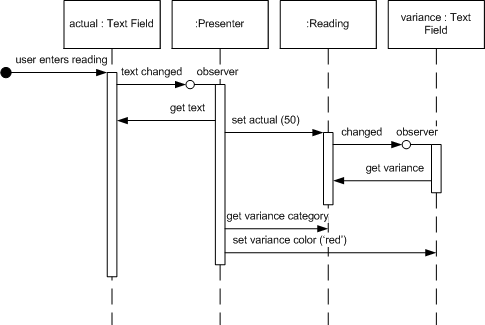
You can also move all the way to having the presenter do all the manipulation of the widgets. This style, which I call [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html) isn't part of the original descriptions of MVP but got developed as people explored testability issues. I'm going to talk about that style later, but that style is one of the flavors of MVP.

Before I contrast MVP with what I've discussed before I should mention that both MVP papers here do this too - but not quite with the same interpretation I have. Potel implies that MVC controllers were overall coordinators - which isn't how I see them. Dolphin talks a lot about issues in MVC, but by MVC they mean the VisualWorks Application Model design rather than classic MVC that I've described (I don't blame them for that - trying to get information on classic MVC isn't easy now let alone then.)

So now it's time for some contrasts:

* Forms and Controls: MVP has a model and the presenter is expected to manipulate this model with [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) then updating the view. Although direct access to the widgets is allowed, this should be in addition to using the model not the first choice.
* MVC: MVP uses a [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html) to manipulate the model. Widgets hand off user gestures to the [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html). Widgets aren't separated into views and controllers. You can think of presenters as being like controllers but without the initial handling of the user gesture. However it's also important to note that presenters are typically at the form level, rather than the widget level - this is perhaps an even bigger difference.
* Application Model: Views hand off events to the presenter as they do to the application model. However the view may update itself directly from the domain model, the presenter doesn't act as a [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html). Furthermore the presenter is welcome to directly access widgets for behaviors that don't fit into the [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html).

There are obvious similarities between MVP presenters and MVC controllers, and presenters are a loose form of MVC controller. As a result a lot of designs will follow the MVP style but use 'controller' as a synonym for presenter. There's a reasonable argument for using controller generally when we are talking about handling user input.

**

*Figure 12: Sequence diagram of the actual reading update in MVP.*

Let's look at an MVP ([Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html)) version of the ice-cream monitor ([Figure 12](https://martinfowler.com/eaaDev/uiArchs.html#uiArchs_mvp-seq.gif)). It starts much the same as the Forms and Controls version - the actual text field raises an event when its text is changed, the presenter listens to this event and gets the new value of the field. At this point the presenter updates the reading domain object, which the variance field observes and updates its text with. The last part is the setting of the color for the variance field, which is done by the presenter. It gets the category from the reading and then updates the color of the variance field.

Here are the MVP soundbites:

* User gestures are handed off by the widgets to a [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html).
* The presenter coordinates changes in a domain model.
* Different variants of MVP handle view updates differently. These vary from using [Observer Synchronization](https://martinfowler.com/eaaDev/MediatedSynchronization.html) to having the presenter doing all the updates with a lot of ground in-between.

## Humble View

In the past few years there's been a strong fashion for writing self-testing code. Despite being the last person to ask about fashion sense, this is a movement that I'm thoroughly immersed in. Many of my colleagues are big fans of xUnit frameworks, automated regression tests, Test-Driven Development, Continuous Integration and similar buzzwords.

When people talk about self-testing code user-interfaces quickly raise their head as a problem. Many people find that testing GUIs to be somewhere between tough and impossible. This is largely because UIs are tightly coupled into the overall UI environment and difficult to tease apart and test in pieces.

Sometimes this test difficulty is over-stated. You can often get surprisingly far by creating widgets and manipulating them in test code. But there are occasions where this is impossible, you miss important interactions, there are threading issues, and the tests are too slow to run.

As a result there's been a steady movement to design UIs in such a way that minimizes the behavior in objects that are awkward to test. Michael Feathers crisply summed up this approach in [The Humble Dialog Box](http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf). [Gerard Meszaros](http://xunitpatterns.com/) generalized this notion to idea of a **Humble Object** - any object that is difficult to test should have minimal behavior. That way if we are unable to include it in our test suites we minimize the chances of an undetected failure.

[The Humble Dialog Box](http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf) paper uses a presenter, but in a much deeper way than the original MVP. Not just does the presenter decide how to react to user events, it also handles the population of data in the UI widgets themselves. As a result the widgets no longer have, nor need, visibility to the model; they form a [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html), manipulated by the presenter.

This isn't the only way to make the UI humble. Another approach is to use[Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html), although then you do need a bit more behavior in the widgets, enough for the widgets to know how to map themselves to the [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html).

The key to both approaches is that by testing the presenter or by testing the presentation model, you test most of the risk of the UI without having to touch the hard-to-test widgets.

With [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) you do this by having all the actual decision making made by the [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html). All user events and display logic is routed to the [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html), so that all the widgets have to do is map themselves to properties of the [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html). You can then test most of the behavior of the [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) without any widgets being present - the only remaining risk lies in the widget mapping. Provided that this is simple you can live with not testing it. In this case the screen isn't quite as humble as with the [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html) approach, but the difference is small.

Since [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html) makes the widgets entirely humble, without even a mapping present, [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html) eliminates even the small risk present with [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html). The cost however is that you need a [Test Double](http://xunitpatterns.com/Test%20Double.html) to mimic the screen during your test runs - which is extra machinery you need to build.

A similar trade-off exists with [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html). Having the view do simple mappings introduces some risk but with the benefit (as with [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html)) of being able to specify simple mapping declaratively. Mappings will tend to be smaller for [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html) than for [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) as even complex updates will be determined by the [Presentation Model](https://martinfowler.com/eaaDev/PresentationModel.html) and mapped, while a [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html) will manipulate the widgets for complex cases without any mapping involved.

Share:[https://martinfowler.com/t_mini-a.png](https://twitter.com/intent/tweet?url=https://martinfowler.com/eaaDev/uiArchs.html&text=GUI%20Architectures%20%E2%9E%99)[https://martinfowler.com/fb-icon-20.png](https://facebook.com/sharer.php?u=https://martinfowler.com/eaaDev/uiArchs.html)[https://martinfowler.com/gplus-16.png](https://plus.google.com/share?url=https://martinfowler.com/eaaDev/uiArchs.html)

if you found this article useful, please share it. I appreciate the feedback and encouragement

## Further Reading

For recent articles that develop these ideas further, take a look at [my bliki](https://martinfowler.com/bliki/UiPatternsReadings.html).

## Acknowledgements

Vassili Bykov generously let me have a copy of Hobbes - his implementation of Smalltalk-80 version 2 (from the early 1980's)which runs in modern VisualWorks. This provided me with a live example of Model-View-Controller which was extremely helpful in answering detailed questions of how it worked and how it was used in the default image. Many people in those days considered it impractical to use a virtual machine. I wonder what our prior selves would have thought to see me running Smalltalk 80 in a virtual machine written in VisualWorks running in the VisualWorks virtual machine on Windows XP running in a VMware virtual machine running on Ubuntu.

## Significant Revisions

*18 July 2006:*First publication in development website.

[https://martinfowler.com/TW_smb.jpg](http://www.thoughtworks.com/)

© Martin Fowler | [Privacy Policy](http://www.thoughtworks.com/privacy-policy) | [Disclosures](https://martinfowler.com/aboutMe.html#disclosures)

[https://martinfowler.com/mf-name-white.png](https://martinfowler.com/)

Top of Form

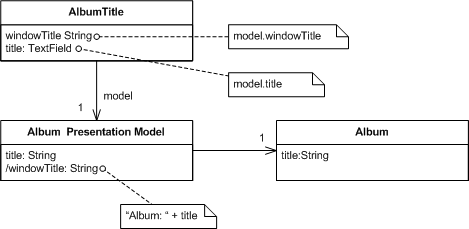


Bottom of Form

* [Intro](https://martinfowler.com/intro.html)
* [Videos](https://martinfowler.com/videos.html)
* [Design](https://martinfowler.com/design.html)
* [Agile](https://martinfowler.com/agile.html)
* [Refactoring](http://refactoring.com/)
* [FAQ](https://martinfowler.com/faq.html)
* [About Me](https://martinfowler.com/aboutMe.html)
* [All Sections](https://martinfowler.com/eaaDev/PresentationModel.html#bottom-navmenu)
* [ThoughtWorks](http://www.thoughtworks.com/)

# Presentation Model

*Represent the state and behavior of the presentation independently of the GUI controls used in the interface*



Also Known as: **Application Model**



[**Martin Fowler**](https://martinfowler.com/)

19 July 2004

This is part of the [Further Enterprise Application Architecture development](https://martinfowler.com/eaaDev) writing that I was doing in the mid 2000’s. Sadly too many other things have claimed my attention since, so I haven’t had time to work on them further, nor do I see much time in the foreseeable future. As such this material is very much in draft form and I won’t be doing any corrections or updates until I’m able to find time to work on it again.

GUIs consist of widgets that contain the state of the GUI screen. Leaving the state of the GUI in widgets makes it harder to get at this state, since that involves manipulating widget APIs, and also encourages putting presentation behavior in the view class.

Presentation Model pulls the state and behavior of the view out into a model class that is part of the presentation. The Presentation Model coordinates with the domain layer and provides an interface to the view that minimizes decision making in the view. The view either stores all its state in the Presentation Model or synchronizes its state with Presentation Model frequently

Presentation Model may interact with several domain objects, but Presentation Model is not a GUI friendly facade to a specific domain object. Instead it is easier to consider Presentation Model as an abstract of the view that is not dependent on a specific GUI framework. While several views can utilize the same Presentation Model, each view should require only one Presentation Model. In the case of composition a Presentation Model may contain one or many child Presentation Model instances, but each child control will also have only one Presentation Model.

Presentation Model is known to users of Visual Works Smalltalk as **Application Model**

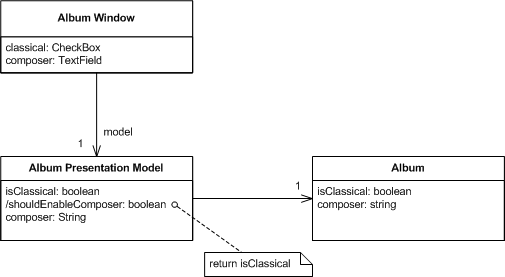
## How it Works

The essence of a Presentation Model is of a fully self-contained class that represents all the data and behavior of the UI window, but without any of the controls used to render that UI on the screen. A view then simply projects the state of the presentation model onto the glass.

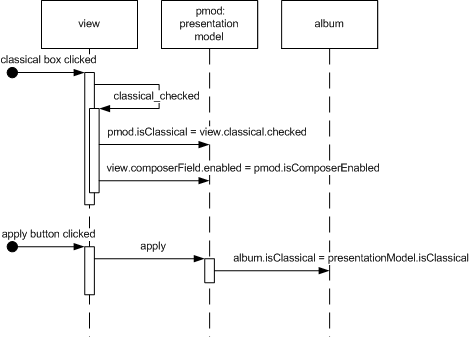
To do this the Presentation Model will have data fields for all the dynamic information of the view. This won't just include the contents of controls, but also things like whether or not they are enabled. In general the Presentation Model does not need to hold all of this control state (which would be lot) but any state that may change during the interaction of the user. So if a field is always enabled, there won't be extra data for its state in the Presentation Model.

Since the Presentation Model contains data that the view needs to display the controls you need to synchronize the Presentation Model with the view. This synchronization usually needs to be tighter than synchronization with the domain - screen synchronization is not sufficient, you'll need field or key synchronization.

To illustrate things a bit better, I'll use the aspect of the [running example](https://martinfowler.com/eaaDev/OrganizingPresentations.html#runningExample) where the composer field is only enabled if the classical check box is checked.

**

*Figure 1: Classes showing structure relevant to clicking the classical check box*

**

*Figure 2: How objects react to clicking the classical check box.*

When someone clicks the classical check box the check box changes its state and then calls the appropriate event handler in the view. This event handler saves the state of the view to Presentation Model and then updates itself from the Presentation Model (I'm assuming a coarse-grained synchronization here.) The Presentation Model contains the logic that says that the composer field is only enabled if the check box is checked, so the when the view updates itself from the Presentation Model, the composer field control changes its enablement state. I've indicated on the diagram that the Presentation Model would typically have a property specifically to mark whether the composer field should be enabled. This will, of course, just return the value of the isClassical property in this case - but the separate property is important because that property encapsulates how the Presentation Model determines whether the composer field is enabled - clearly indicating that that decision is the responsibility of the Presentation Model.

This small example is illustrates the essence of the idea of the Presentation Model - all the decisions needed for presentation display are made by the Presentation Model leaving the view to be utterly simple.

Probably the most annoying part of Presentation Model is the synchronization between Presentation Model and view. It's simple code to write, but I always like to minimize this kind of boring repetitive code. Ideally some kind of framework could handle this, which I'm hoping will happen some day with technologies like .NET's data binding.

A particular decision you have to make with synchronization in Presentation Modelis which class should contain the synchronization code. Often, this decision is largely based on the desired level of test coverage and the chosen implementation of Presentation Model. If you put the synchronization in the view, it won't get picked up by tests on the Presentation Model. If you put it in the Presentation Model you add a dependency to the view in the Presentation Model which means more coupling and stubbing. You could add a mapper between them, but adds yet more classes to coordinate. When making the decision of which implementation to use it is important to remember that although faults do occur in synchronization code, they are usually easy to spot and fix (unless you use fine-grained synchronization).

An important implementation detail of Presentation Model is whether the View should reference the Presentation Model or the Presentation Model should reference the View. Both implementations provide pros and cons.

A Presentation Model that references a view generally maintains the synchronization code in the Presentation Model. The resulting view is very dumb. The view contains setters for any state that is dynamic and raises events in response to user actions. The views implement interfaces allowing for easy stubbing when testing the Presentation Model. The Presentation Model will observe the view and respond to events by changing any appropriate state and reloading the entire view. As a result the synchronization code can be easily tested without needing the actual UI class.

A Presentation Model that is referenced by a view generally maintains the synchronization code in the view. Because the synchronization code is generally easy to write and easy to spot errors it is recommended that the testing occur on the Presentation Model and not the View. If you are compelled to write tests for the view this should be a clue that the view contains code that should belong in the Presentation Model. If you prefer to test the synchronization, a Presentation Modelthat references a view implementation is recommended.

## When to Use It

Presentation Model is a pattern that pulls presentation behavior from a view. As such it's an alternative to to [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html) and [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html). It's useful for allowing you to test without the UI, support for some form of multiple view and a separation of concerns which may make it easier to develop the user interface.

Compared to [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html) and [Supervising Controller](https://martinfowler.com/eaaDev/SupervisingPresenter.html), Presentation Model allows you to write logic that is completely independent of the views used for display. You also do not need to rely on the view to store state. The downside is that you need a synchronization mechanism between the presentation model and the view. This synchronization can be very simple, but it is required. [Separated Presentation](https://martinfowler.com/eaaDev/SeparatedPresentation.html)requires much less synchronization and [Passive View](https://martinfowler.com/eaaDev/PassiveScreen.html) doesn't need any at all.

## Example: Running Example (View References PM) (C#)

Here's a version of the [running example](https://martinfowler.com/eaaDev/OrganizingPresentations.html#runningExample), developed in C# with Presentation Model.

**

*Figure 3: The album window.*

I'll start discussing the basic layout from the domain model outwards. Since the domain isn't the focus of this example, it's very uninteresting. It's essentially just a data set with a single table holding the data for an album. Here's the code for setting up a few test albums. I'm using a strongly typed data set.

public static DsAlbum AlbumDataSet() {

DsAlbum result = new DsAlbum();

result.Albums.AddAlbumsRow(1, "HQ", "Roy Harper", false, null);

result.Albums.AddAlbumsRow(2, "The Rough Dancer and Cyclical Night", "Astor Piazzola", false, null);

result.Albums.AddAlbumsRow(3, "The Black Light", "Calexico", false, null);

result.Albums.AddAlbumsRow(4, "Symphony No.5", "CBSO", true, "Sibelius" );

result.AcceptChanges();

return result;

}

The Presentation Model wraps this data set and provides properties to get at the data. There's a single instance of the Presentation Model for the whole table, corresponding to the single instance of the window. The Presentation Model has fields for the data set and also keeps track of which album is currently selected.

*class PmodAlbum...*

public PmodAlbum(DsAlbum albums) {

this.\_data = albums;

\_selectedAlbumNumber = 0;

}

private DsAlbum \_data;

private int \_selectedAlbumNumber;

PmodAlbum provides properties to get at the data in the data set. Essentially I provide a property for each bit of information that the form needs to display. For those values which are just pulled directly out of the data set, this property is pretty simple.

*class PmodAlbum...*

public String Title {

get {return SelectedAlbum.Title;}

set {SelectedAlbum.Title = value;}

}

public String Artist {

get {return SelectedAlbum.Artist;}

set {SelectedAlbum.Artist = value;}

}

public bool IsClassical {

get {return SelectedAlbum.IsClassical;}

set {SelectedAlbum.IsClassical = value;}

}

public String Composer {

get {

return (SelectedAlbum.IsComposerNull()) ? "" : SelectedAlbum.Composer;

}

set {

if (IsClassical) SelectedAlbum.Composer = value;

}

}

public DsAlbum.AlbumsRow SelectedAlbum {

get {return Data.Albums[SelectedAlbumNumber];}

}

The title of the window is based on the album title. I provide this through another property.

*class PmodAlbum...*

public String FormTitle

{

get {return "Album: " + Title;}

}

I have a property to see if the composer field should be enabled.

*class PmodAlbum...*

public bool IsComposerFieldEnabled {

get {return IsClassical;}

}

This is just a call to the public IsClassical property. You may wonder why the form doesn't just call this directly - but this is the essence of the encapsulation that the Presentation Model provides. PmodAlbum decides what the logic is for enabling that field, the fact that it's simply based on a property is known to the Presentation Model but not to the view.

The apply and cancel buttons should only be enabled if the data has changed. I can provide this by checking the state of that row of the dataset, since data sets record this information.

*class PmodAlbum...*

public bool IsApplyEnabled {

get {return HasRowChanged;}

}

public bool IsCancelEnabled {

get {return HasRowChanged;}

}

public bool HasRowChanged {

get {return SelectedAlbum.RowState == DataRowState.Modified;}

}

The list box in the view shows a list of the album titles. PmodAlbum provides this list.

*class PmodAlbum...*

public String[] AlbumList {

get {

String[] result = new String[Data.Albums.Rows.Count];

for (int i = 0; i < result.Length; i++)

result[i] = Data.Albums[i].Title;

return result;

}

}

So that covers the interface that PmodAlbum presents to the view. Next I'll look at how I do the synchronization between the view and the Presentation Model. I've put the synchronization methods in the view and am using coarse-grained synchronization. First I have a method to push the state of the view into the Presentation Model.

*class FrmAlbum...*

private void SaveToPmod() {

model.Artist = txtArtist.Text;

model.Title = txtTitle.Text;

model.IsClassical = chkClassical.Checked;

model.Composer = txtComposer.Text;

}

This method is very simple, just assigning the mutable parts of the view to the Presentation Model. The load method is a touch more complicated.

*class FrmAlbum...*

private void LoadFromPmod() {

if (NotLoadingView) {

\_isLoadingView = true;

lstAlbums.DataSource = model.AlbumList;

lstAlbums.SelectedIndex = model.SelectedAlbumNumber;

txtArtist.Text = model.Artist;

txtTitle.Text = model.Title;

this.Text = model.FormTitle;

chkClassical.Checked = model.IsClassical;

txtComposer.Enabled = model.IsComposerFieldEnabled;

txtComposer.Text = model.Composer;

btnApply.Enabled = model.IsApplyEnabled;

btnCancel.Enabled = model.IsCancelEnabled;

\_isLoadingView = false;

}

}

private bool \_isLoadingView = false;

private bool NotLoadingView {

get {return !\_isLoadingView;}

}

private void SyncWithPmod() {

if (NotLoadingView) {

SaveToPmod();

LoadFromPmod();

}

}

The complication here is avoiding a infinite recursion since synchronizing causes fields on the form to update which triggers synchronization.... I guard against that with a flag.

With these synchronization methods in place, the next step is just to call the right bit of synchronization in event handlers for the controls. Most of the time this easy, just call SyncWithPmod when data changes.

*class FrmAlbum...*

private void txtTitle\_TextChanged(object sender, System.EventArgs e){

SyncWithPmod();

}

Some cases are more involved. When the user clicks on a new item in the list we need to navigate to a new album and show its data.

*class FrmAlbum...*

private void lstAlbums\_SelectedIndexChanged(object sender, System.EventArgs e){

if (NotLoadingView) {

model.SelectedAlbumNumber = lstAlbums.SelectedIndex;

LoadFromPmod();

}

}

*class PmodAlbum...*

public int SelectedAlbumNumber {

get {return \_selectedAlbumNumber;}

set {

if (\_selectedAlbumNumber != value) {

Cancel();

\_selectedAlbumNumber = value;

}

}

}

Notice that this method abandons any changes if you click on the list. I've done this awful bit of usability to keep the example simple, the form should really at least pop up a confirmation box to avoid losing the changes.

The apply and cancel buttons delegate what to do to the Presentation Model.

*class FrmAlbum...*

private void btnApply\_Click(object sender, System.EventArgs e) {

model.Apply();

LoadFromPmod();

}

private void btnCancel\_Click(object sender, System.EventArgs e){

model.Cancel();

LoadFromPmod();

}

*class PmodAlbum...*

public void Apply () {

SelectedAlbum.AcceptChanges();

}

public void Cancel() {

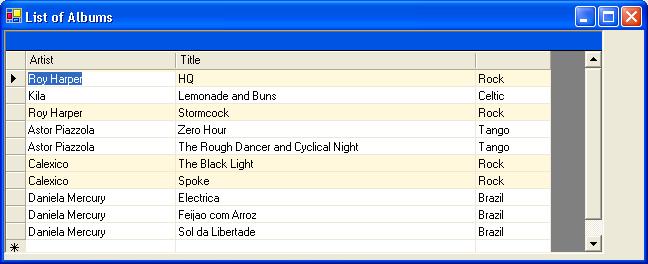
SelectedAlbum.RejectChanges();

}

So although I can move most of the behavior to the Presentation Model, the view still retains some intelligence. For the testing aspect of Presentation Model to work better, it would be nice to move more. Certainly you can move more into the Presentation Model by moving the synchronization logic there, at the expense of having the Presentation Model know more about the view.

## Example: Data Binding Table Example (C#)

As I first looked at Presentation Model in the .NET framework, it seemed that data binding provided excellent technology to make Presentation Model work simply. So far limitations in the current version of data binding holds it back from places that I'm sure it will eventually go. One area where data binding can work very well is with read-only data, so here is an example that shows this as well as how tables can fit in with a Presentation Model design.

**

*Figure 4: A list of albums with the rock ones highlighted.*

This is just a list of albums. The extra behavior is that each rock album should have it's row colored in cornsilk.

I'm using a slightly different data set to the other example. Here is the code for some test data.

public static AlbumList AlbumGridDataSet()

{

AlbumList result = new AlbumList();

result.Albums.AddAlbumsRow(1, "HQ", "Roy Harper", "Rock");

result.Albums.AddAlbumsRow(2, "Lemonade and Buns", "Kila", "Celtic");

result.Albums.AddAlbumsRow(3, "Stormcock", "Roy Harper", "Rock");

result.Albums.AddAlbumsRow(4, "Zero Hour", "Astor Piazzola", "Tango");

result.Albums.AddAlbumsRow(5, "The Rough Dancer and Cyclical Night", "Astor Piazzola", "Tango");

result.Albums.AddAlbumsRow(6, "The Black Light", "Calexico", "Rock");

result.Albums.AddAlbumsRow(7, "Spoke", "Calexico", "Rock");

result.Albums.AddAlbumsRow(8, "Electrica", "Daniela Mercury", "Brazil");

result.Albums.AddAlbumsRow(9, "Feijao com Arroz", "Daniela Mercury", "Brazil");

result.Albums.AddAlbumsRow(10, "Sol da Libertade", "Daniela Mercury", "Brazil");

Console.WriteLine(result);

return result;

}

The presentation model in this case reveals its internal data set as a property. This allows the form to data bind directly to the cells in the data set.

private AlbumList \_dsAlbums;

internal AlbumList DsAlbums {

get {return \_dsAlbums;}

}

To support the highlighting, the presentation model provides an additional method that indexes into the table.

internal Color RowColor(int row) {

return (Albums[row].genre.Equals("Rock")) ? Color.Cornsilk : Color.White;

}

private AlbumList.AlbumsDataTable Albums {

get {return DsAlbums.Albums;}

}

This method is similar to the ones in a simple example, the difference being that methods on table data need cell coordinates to pick out parts of the table. In this case all we need is a row number, but in general we may need row and column numbers.

From here on I can use the standard data binding facilities that come with visual studio. I can bind table cells easily to data in the data set, and also to data on thePresentation Model.

Getting the color to work is a little bit more involved. This is straying a little bit away from the main thrust of the example, but the whole thing gets its complication because there's no way to have row by row highlighting on the standard WinForms table control. In general the answer to this need is to buy a third party control, but I'm too cheap to do this. So for the curious here's what I did (the idea was mostly ripped off from http://www.syncfusion.com/FAQ/WinForms/). I'm going to assume you're familiar with the guts of WinForms from now on.

Essentially I made a subclass of DataGridTextBoxColumn which adds the color highlighting behavior. You link up the new behavior by passing in a delegate that handles the behavior.

*class ColorableDataGridTextBoxColumn...*

public ColorableDataGridTextBoxColumn (ColorGetter getcolorRowCol, DataGridTextBoxColumn original)

{

\_delGetColor = getcolorRowCol;

copyFrom(original);

}

public delegate Color ColorGetter(int row);

private ColorGetter \_delGetColor;

The constructor takes the original DataGridTextBoxColumn as well as the delegate. What I'd really like to do here is to use the decorator pattern to wrap the original but the original, like so many classes in WinForms, is all sealed up. So instead I copy over all the properties of the original into my subclass. This won't work is there are vital properties that can't be copied because you can't read or write to them. It seems to work here for now.

*class ColorableDataGridTextBoxColumn...*

void copyFrom (DataGridTextBoxColumn original) {

PropertyInfo[] props = original.GetType().GetProperties();

foreach (PropertyInfo p in props) {

if (p.CanWrite && p.CanRead)

p.SetValue(this, p.GetValue(original, null), null) ;

}

}

Fortunately the paint method is virtual (otherwise I would need a whole new data grid.) I can use it to insert the appropriate background color using the delegate.

*class ColorableDataGridTextBoxColumn...*

protected override void Paint(System.Drawing.Graphics g, System.Drawing.Rectangle bounds,

System.Windows.Forms.CurrencyManager source, int rowNum,

System.Drawing.Brush backBrush, System.Drawing.Brush foreBrush,

bool alignToRight)

{

base.Paint(g, bounds, source, rowNum, new SolidBrush(\_delGetColor(rowNum)), foreBrush, alignToRight);

}

To put this new table in place, I replace the columns of the data table in the page load after the controls have been built on the form.

*class FrmAlbums...*

private void FrmAlbums\_Load(object sender, System.EventArgs e){

bindData();

replaceColumnStyles();

}

private void replaceColumnStyles() {

ColorableDataGridTextBoxColumn.ReplaceColumnStyles(dgsAlbums,

new ColorableDataGridTextBoxColumn.ColorGetter(model.RowColor));

}

*class ColorableDataGridTextBoxColumn...*

public static void ReplaceColumnStyles(DataGridTableStyle grid, ColorGetter del) {

for (int i = 0; i < grid.GridColumnStyles.Count; i++) {

DataGridTextBoxColumn old = (DataGridTextBoxColumn) grid.GridColumnStyles[0];

grid.GridColumnStyles.RemoveAt(0);

grid.GridColumnStyles.Add(new ColorableDataGridTextBoxColumn(del, old));

}

}

It works, but I'll admit it's a lot more messy than I would like. If I were doing this for real, I'd want to look into a third party control. However I've seen this done in a production system and it worked just fine.

[https://martinfowler.com/TW_smb.jpg](http://www.thoughtworks.com/)

© Martin Fowler | [Privacy Policy](http://www.thoughtworks.com/privacy-policy) | [Disclosures](https://martinfowler.com/aboutMe.html#disclosures)

# Difference between MVC, MVP And MVVM

* Published on July 30, 2015

**[Swapneel Salunkhe](https://www.linkedin.com/in/swapneel-salunkhe-b44aa762/)**

## [Swapneel Salunkhe](https://www.linkedin.com/in/swapneel-salunkhe-b44aa762/)

Follow

Software Developer at ITCube Solutions Pvt.Ltd.

[**2 articles**](https://www.linkedin.com/in/swapneel-salunkhe-b44aa762/detail/recent-activity/posts/)

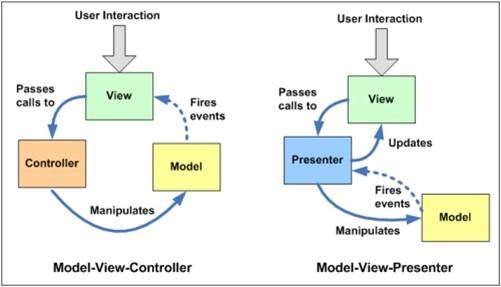
* Like158
* Comment**9**
* Share

MVC, MVP, and MVVM are some of the common patterns to guide programmers toward creating decoupled solutions.   
The software behaviors that are common to MVC, MVP, and MVVM are:   
  
1)Data Layer / Business Logic (Model): This is the behavior which applies the business logic to the application's data. A Domain Model usually represents the Model where the objects are used to mimic the real world entities.

2) Presentation Layer / UI ( View ): View is responsible for the visual presentation of the application. This behavior displays model information to the user.

3) Application Logic ( Controller, Presentation or View Model ): This behavior holds the logic that implements the interaction between the model and the view.   
  
**MVC**

* MVC consists of three layers Model, View, and Controller.
* MVC is a compound pattern
* It uses a Front Controller pattern that processes Web application requests through a single controller. This enables you to design an application that supports a rich routing infrastructure.
* It does not use view state or server-based forms. This makes the MVC framework ideal for developers who want full control over the behavior of an application.
* It provides better support for test-driven development (TDD).
* It works well for Web applications that are supported by large teams of developers and for Web designers who need a high degree of control over the application behavior
* Client-side library: Backbone.js, knockback.js, Spine.js, angular.js.
* Server-side library: ASP.NET MVC, Spring MVC,Ruby-on-Rails



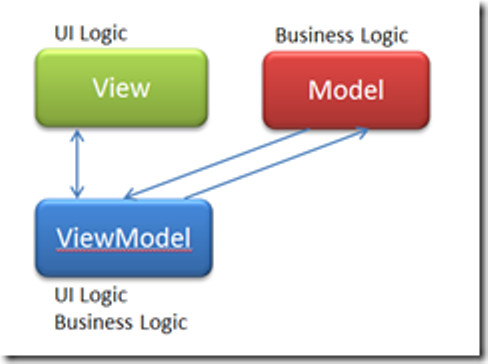
**MVP**

* MVP consists of three layers Model, View, and Presenter.
* In MVP, View and Model are more loosely coupled, providing a clearer separation of concerns.
* MVP, View is responsible for delegating the user input to the Presenter.
* MVP, Presenter and View should have a 1-1 relation, with each View having a reference to its Presenter through the interface.
* MVP, view binds to the Model directly through data binding.
* In MVP, unit testing is easier, as View knows Presenter through an interface which can easily be mocked.
* Client-side library:Riot.js,GWT
* Server-side library: Clas­sic ASP.NET,JSP Servlets.

**MVVM**

* MVVM pattern is a one of the best solutions to handle such problems for WPF and Silverlight application.
* When you use MVVM pattern for WPF, Silverlight the view wouldn't have the typical event handlers that's so common in UI code.
* MVVM provides a clear separation between the UI and application logic.
* Client-side library:Knockout.js, Kendo (MVVM)
* Server-side library:WPF (Desk­top) or Sil­verlight,Win­dows Phone apps (XAML),Adobe Flex
* The MVVM pattern includes three key parts:

1. Model (Business rule, data access, model classes)
2. View (User interface (XAML))
3. ViewModel (Agent or middle man between view and model)



## When to use which?

**MVP**

* Use in situations where binding via a datacontext is not possible.
* Windows Forms is a perfect example of this.  In order to separate the view from the model, a presenter is needed.  Since the view cannot directly bind to the presenter, information must be passed to it view an interface (IView).

**MVVM**

* Use in situations where binding via a datacontext is possible.  Why?  The various IView interfaces for each view are removed which means less code to maintain.
* Some examples where MVVM is possible include WPF and javascript projects using [Knockout](http://knockoutjs.com/).

**MVC**

* Use in situations where the connection between the view and the rest of the program is not always available (and you can’t effectively employ MVVM or MVP).
* This clearly describes the situation where a web API is separated from the data sent to the client browsers.  Microsoft’s ASP.NET MVC is a great tool for managing such situations and provides a very clear MVC framework.

[Report this](https://www.linkedin.com/pulse/difference-between-mvc-mvp-mvvm-swapneel-salunkhe/)

### **158 Likes**

**[Filzah Khairuddin](https://www.linkedin.com/in/filzah-khairuddin-a59065126/)**

**[Varsha Y](https://www.linkedin.com/in/varsha-y-9391b8bb/)**

**[Thomas O'Dea](https://www.linkedin.com/in/thomas-odea/)**

**[Nitish Sharma](https://www.linkedin.com/in/nitish-sharma-061628144/)**

**[Daniel Arant](https://www.linkedin.com/in/daniel-arant-a9769128/)**

**[subba rao kala](https://www.linkedin.com/in/subba-rao-kala-11504b33/)**

**[sai tulasi ram](https://www.linkedin.com/in/sai-tulasi-ram-20816520/)**

**[Manisha Kamble](https://www.linkedin.com/in/manisha-kamble-13691a70/)**

**[Rupesh Kuinkel](https://www.linkedin.com/in/rupesh-kuinkel-832144b0/)**

**[Bhawana Panday](https://www.linkedin.com/in/bhawana-panday/)**

* …

### 9 CommentsComments on Swapneel Salunkhe’s article

Top of Form

Add a comment…

Images

Bottom of Form

2y

**[Devang Oza](https://www.linkedin.com/in/devangoza/" \t "_self)**

### [Devang Oza](https://www.linkedin.com/in/devangoza/" \t "_self)**[Asp.net MVC | C# | Angular | Sql Server](https://www.linkedin.com/in/devangoza/" \t "_self)**

In MVC, user interactions pass call to controller or view ? what is the dotted line saying fire events in model ?

LikeLike Devang Oza’s commentReply

2y

**[James Wilkins](https://www.linkedin.com/in/jamesnw/" \t "_self)**

### [James Wilkins](https://www.linkedin.com/in/jamesnw/" \t "_self)**[I.T. Consultant and Software Development Services for Catmedis Healthcare and Holcim Canada](https://www.linkedin.com/in/jamesnw/" \t "_self)**

Not a good diagram for MVC vs MVP. The controller returns the view, same as the presenter on the MVP side. Models don't trigger events to return views, so that's wrong (unless you mean WebForms, and not ASP MVC, but even that is not really true).

LikeLike James Wilkins’ commentReply

1 Like1 Like on James Wilkins’ comment