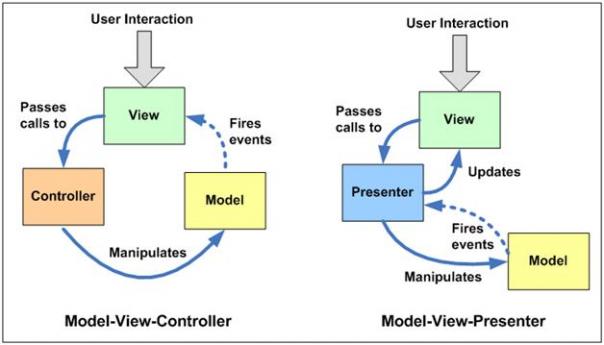
**MVC vs. MVP vs. MVVM** Posted by [nirajrules](http://nirajrules.wordpress.com/author/nirajrules/)

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.

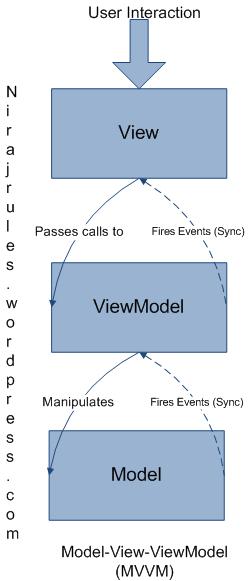


**MVC:** Three components – View (your UI), Model (your business entities / data – that view is displaying) & Controller (contains the logic that alters the model depending on the action triggered by UI, typically implementing a Use Case). It’s widely known that MVC is a compound pattern (View and Controller 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](http://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.

[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](http://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 :).