I find myself answering questions related to the presentation patterns repeatedly. Since I am a lazy person, I am writing this post to illustrate my understanding and interpretation of the Model-View-\* patterns and their variations so I can refer back to this post later.

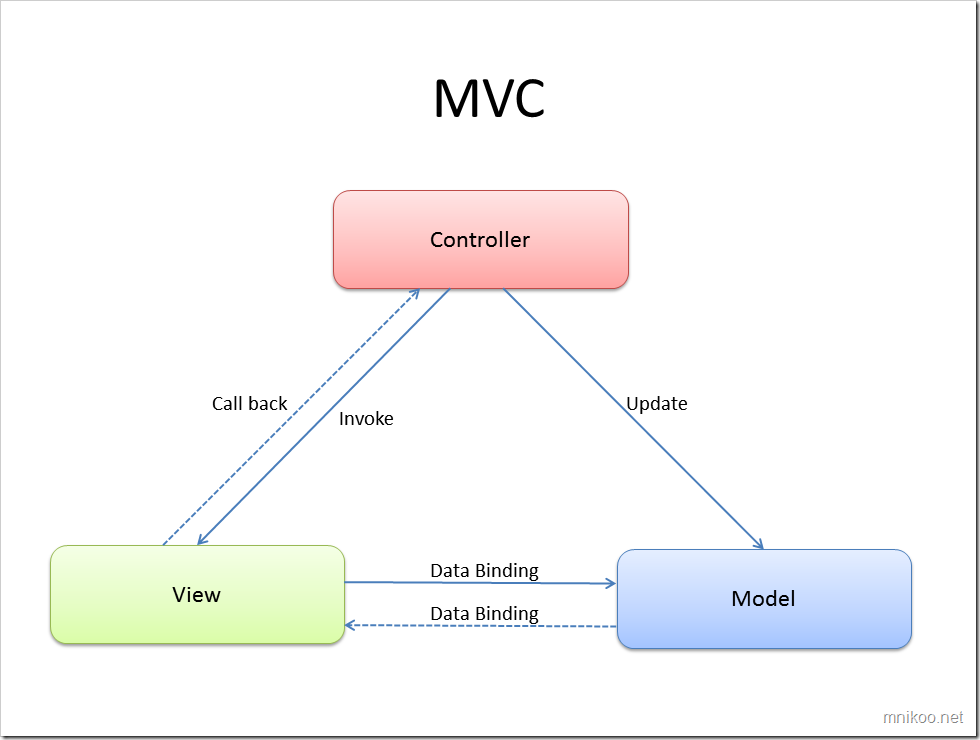
[Update] I have discussed these patterns and their contexts with my colleagues [Josh](http://www.google.com/url?q=http%3A%2F%2Fwww.thejoyofcode.com%2F&sa=D&sntz=1&usg=AFQjCNFrKvbn1bWIDSLK-idOAqRBJ7nExA) and [Rupert](http://www.google.com/url?q=http%3A%2F%2Fphazed.com%2F&sa=D&sntz=1&usg=AFQjCNGtF4KVmUj7w7LT8VHiygX5Jc-P2A) and have updated the post the reflect the result of our conversation. Thanks guys!

I understand the concerns the Model-View-\* purists might have around the correctness and/or pureness of these diagrams so I do apologize for potential inaccuracies in advance and I welcome any suggestions / comments!

I am not going to explain every single Model-View-\* pattern as you can find better explanations in other places. Instead, I will focus on the relationship between participant components in each pattern.

Let’s start with a list of patterns we are going to cover:

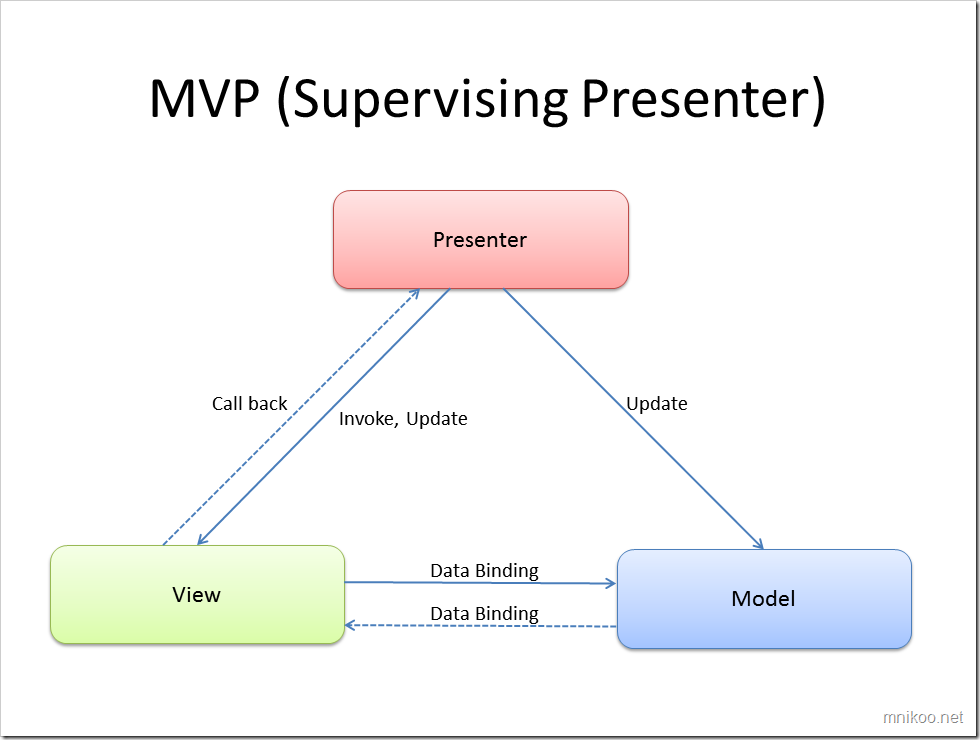
* **MVC**: Model-View-Controller
* **MVP (Supervising Presenter)**: Model-View-Presenter with Supervising Presenter
* **MVP (Passive View)**: Model-View-Presenter with Passive View
* **MVVM**: Model-View-ViewModel
* **MVPoo**: Model-View-ViewModel with some unwanted stuff (in the ViewModel)



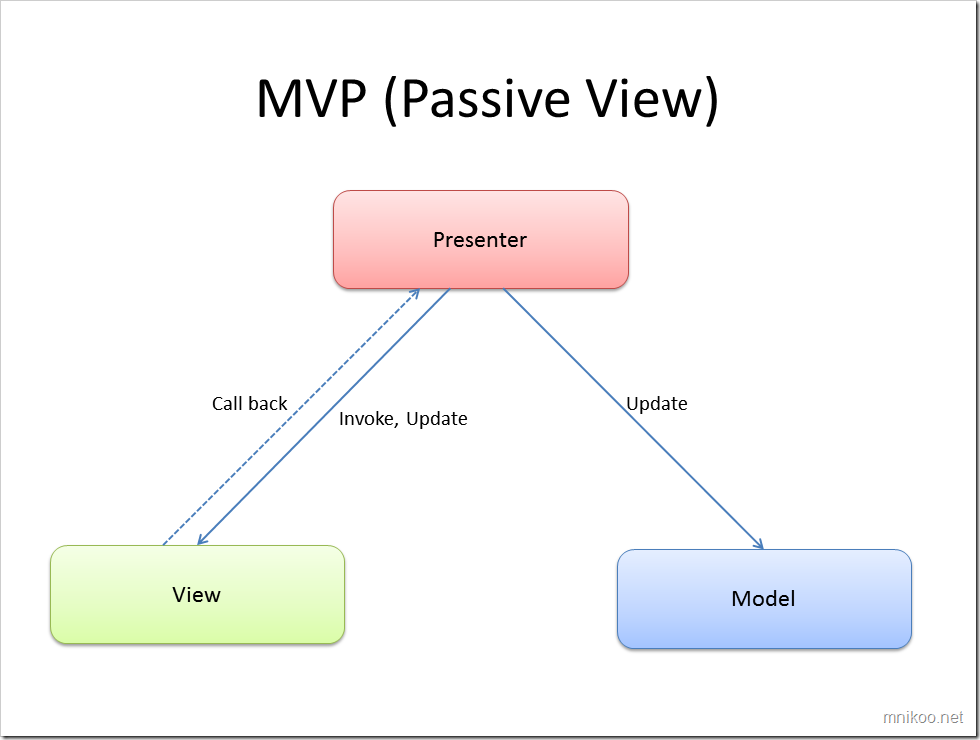
This is a typical implementation of the good old MVC pattern. The Controller displays the View, make changes to its state and manage the navigation. The Controller can also update the Model (or ask the Model to update itself) and these updates are subsequently reflected in the View.

**\* What Happened to the MVP Pattern? \***

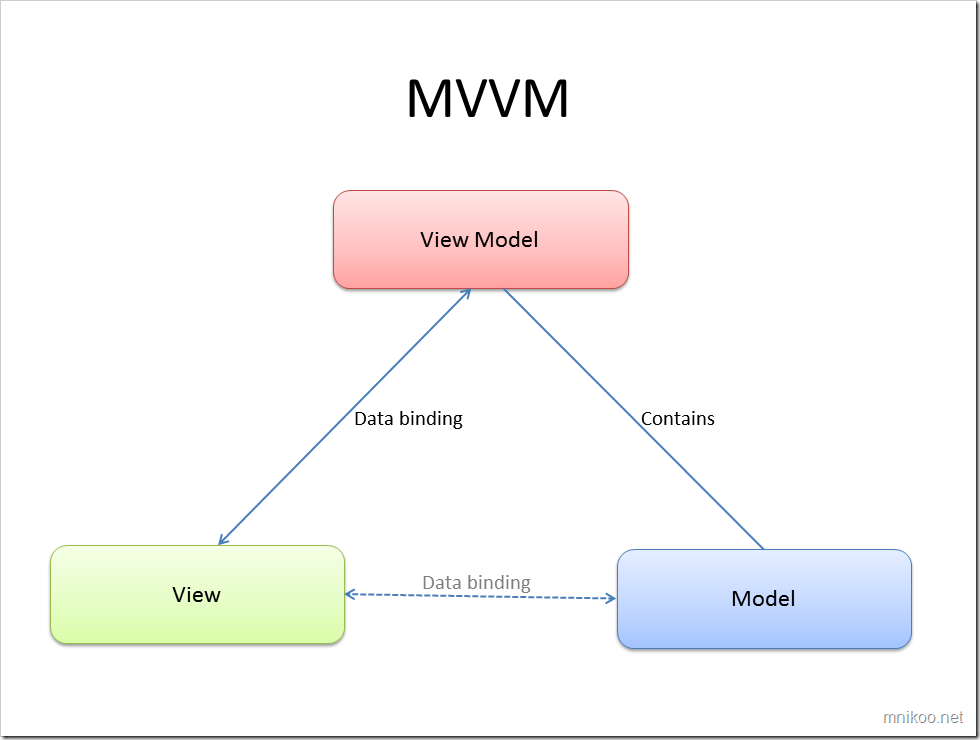
As you may have noticed, the plain MVP (Model-View-Presenter) pattern is not in the above list. If you want to know why, read [Martin Fowler’s note on retirement of the MVP pattern](http://www.google.com/url?q=http%3A%2F%2Fwww.martinfowler.com%2FeaaDev%2FModelViewPresenter.html&sa=D&sntz=1&usg=AFQjCNE9m41P2vCiV1hPBIiko1wzOUQj5Q). Basically, he has split the MVP pattern into two: Supervising Presenter and Passive View. By looking at the next two diagrams, you will notice why they need to be separated.



In the [Supervising Presenter variation of the MVP pattern](http://www.google.com/url?q=http%3A%2F%2Fmartinfowler.com%2FeaaDev%2FSupervisingPresenter.html&sa=D&sntz=1&usg=AFQjCNHlItWJiSYnOKeEgJq1XPncHqoSwg), Controller becomes responsible for implementing the complex view logic where the data binding between the View and the Model is not able to support that complex logic. Data binding is still the primary mechanism for the communication between the View and the Model and the Controller is used only in those scenarios where data binding support falls short.

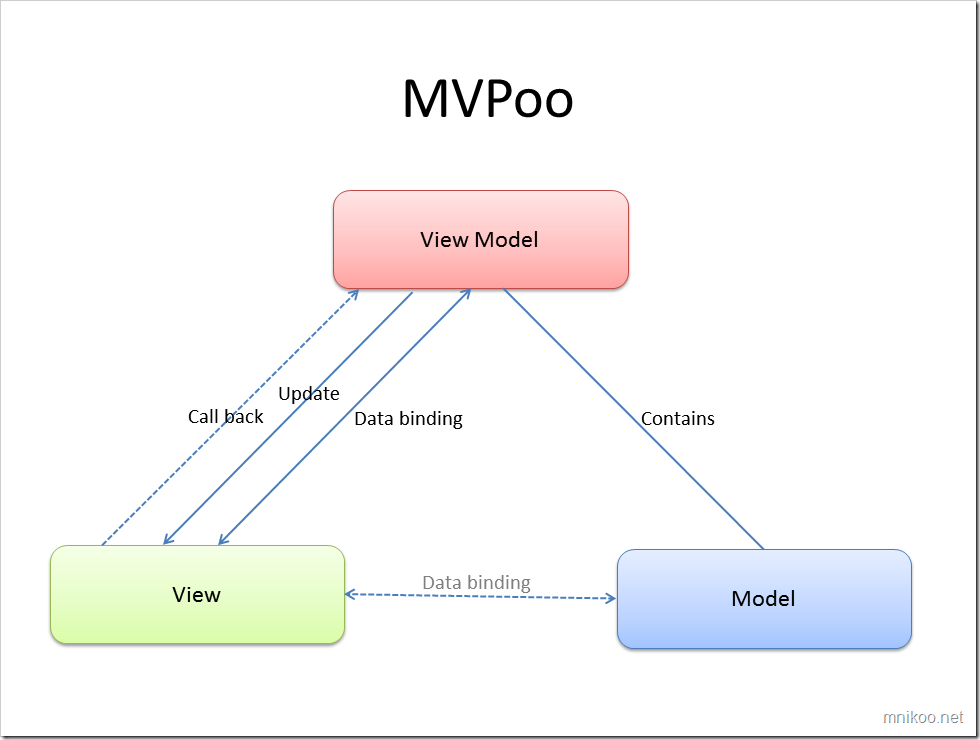


[Passive View](http://www.google.com/url?q=http%3A%2F%2Fmartinfowler.com%2FeaaDev%2FPassiveScreen.html&sa=D&sntz=1&usg=AFQjCNGb3MWLikpPkiUzOAtwrZ5R0fVWUw) is the other variation of the MVP pattern and it brings the level of supervision of the Controller to the next level. The View no longer updates itself based on the Model and Controller becomes responsible for updating the View based on the data in the Model.



In the MVVM pattern, the View Model inherits most of the functionality that would otherwise be in the Controller. The user gestures are captured by the View and the View uses the commanding feature of the underlying framework to invoke the operations defined in the View Model. Note that there is a dotted line showing the data binding between the View and the Model. This highlights the ability for the view to be bound to the Model via the View Model. Although the View Model can have its own properties the View binds to, but it is very likely for the View Model to publish the underlying Models as properties so the View is effectively bound to the Model (via the View Model).

MVVM makes an assumption that the framework the application is built on provides strong support for two-way data binding between the View and the Model. This diagram shows how simple the MVVM pattern is but this pattern would be really hard (if not impossible) to implement before WPF because of the lack of strong support for commanding and data binding. Even with WPF, implementing a non-trivial application that follows the design shown above can be tricky and will usually require a lot of additional helpers and constructs if we want to rely on data binding only.



The [MVPoo](http://www.google.com/url?q=http%3A%2F%2Fdrwpf.com%2Fblog%2F2008%2F01%2F23%2Fa-new-software-architecture-pattern-m-v-poo%2F&sa=D&sntz=1&usg=AFQjCNH09OtfIEb1Ajl_NYnJ5aAfr8jNrQ) (or M-V-poo to be more precise as defined by its creator, [Dr. WPF](http://www.google.com/url?q=http%3A%2F%2Fdrwpf.com%2Fblog&sa=D&sntz=1&usg=AFQjCNHZAayUSZickt4a-mMY5YzEZPpzVg)) pattern acknowledges the fact that there is a difference between the ideal world and the real world so that nice and clean implementation of the MVVM pattern is not always achievable. As a result, there will be scenarios where:

* View Model needs to update the View with mechanisms other than data binding.
* The user gesture captured by the View or the events raised by UI components needs to be handled by the View Model with mechanisms other than commanding
* …

So it is very likely that we will end up with a stinky version of the MVVM pattern. The MVPoo pattern acknowledges this fact and says this is okay but we need to try and make the components (with specific attention to the View Model!) less stinky as much as possible (by keeping the number of non-ideal elements/features in the View Model to a minimum).

Originally posted by Mehran Nikoo on June 14th here [http://mnikoo.net/2010/06/14/model-view-star/](http://www.google.com/url?q=http%3A%2F%2Fmnikoo.net%2F2010%2F06%2F14%2Fmodel-view-star%2F&sa=D&sntz=1&usg=AFQjCNGFurVqqF0pZyVi2obcUDAgt99q8A)