

**Model-View-Presenter**

In **MVP**, the Presenter contains the UI business logic for the View. All invocations from the View delegate directly to Presenter. The Presenter is also decoupled directly from the View and talks to it through an interface. This is to allow mocking of the View in a unit test. One common attribute of MVP is that there has to be a lot of two-way dispatching. For example, when someone clicks the "Save" button, the event handler delegates to the Presenter's "OnSave" method. Once the save is completed, the Presenter will then call back the View through its interface so that the View can display that the save has completed.

**Two primary variations**

**Passive View:** The View is as dumb as possible and contains almost zero logic. The Presenter is a middle man that talks to the View and the Model. The View and Model are completely shielded from one another. The Model may raise events, but the Presenter subscribes to them for updating the View. In Passive View there is no direct data binding, instead the View exposes setter properties which the Presenter uses to set the data. All state is managed in the Presenter and not the View.

**Supervising Controller:** The Presenter handles user gestures. The View binds to the Model directly through data binding. In this case it's the Presenter's job to pass off the Model to the View so that it can bind to it. The Presenter will also contain logic for gestures like pressing a button, navigation, etc.

**Model-View-Controller**

In the **MVC**, the Controller is responsible for determining which View to display in response to any action including when the application loads. This differs from MVP where actions route through the View to the Presenter. In MVC, every action in the View correlates with a call to a Controller along with an action. In the web each action involves a call to a URL on the other side of which there is a Controller who responds. Once that Controller has completed its processing, it will return the correct View. The sequence continues in that manner throughout the life of the application:

Action in the View

-> Call to Controller

-> Controller Logic

-> Controller returns the View.

One other big difference about MVC is that the View does not directly bind to the Model. The view simply renders, and is completely stateless. In implementations of MVC the View usually will not have any logic in the code behind. This is contrary to MVP where it is absolutely necessary because, if the View does not delegate to the Presenter, it will never get called.

**Presentation Model**

One other pattern to look at is the **Presentation Model** pattern. In this pattern there is no Presenter. Instead the View binds directly to a Presentation Model. The Presentation Model is a Model crafted specifically for the View. This means this Model can expose properties that one would never put on a domain model as it would be a violation of separation-of-concerns. In this case, the Presentation Model binds to the domain model, and may subscribe to events coming from that Model. The View then subscribes to events coming from the Presentation Model and updates itself accordingly. The Presentation Model can expose commands which the view uses for invoking actions. The advantage of this approach is that you can essentially remove the code-behind altogether as the PM completely encapsulates all of the behaviour for the view. This pattern is a very strong candidate for use in WPF applications and is also called [Model-View-ViewModel](http://msdn.microsoft.com/en-us/magazine/dd419663.aspx).

**Here are the key differences between the patterns:**

***MVP Pattern***

* View is more loosely coupled to the model. The presenter is responsible for binding the model to the view.
* Easier to unit test because interaction with the view is through an interface.
* Usually view to presenter map one to one. Complex views may have multi presenters.

***MVC Pattern***

* Controller are based on behaviors and can be shared across views.
* Can be responsible for determining which view to display.

**Which architectural pattern is more suitable for Android development – MVC or MVP?**

We’ve already seen that MVP and MVC architectural patterns (as defined in this post) are very similar. Still, since we discuss Android development, there are aspects of Android framework which makes one of them a more suitable choice for application’s architecture.

My personal opinion is that MVP is better for Android because it is simpler and cleaner to have independent view and model components.

If we allow the view and the model to communicate directly, we might end up in a situation when the view needs to become aware life-cycle events. Since these events are not directly related to UI management, making the view aware of them breaks the abstraction of application’s UI.

Since complete UI abstraction is very important for achieving “clean code”, I prefer MVP over MVC for Android development.