MVP for Android: how to organize the presentation layer

by Antonio Leiva | Apr 15, 2014 | Blog, Development | 129 comments

MVP (Model View Presenter) pattern is a **derivative from the well known MVC** (Model View Controller), which for a while now is gaining importance in the development of Android applications. There are more and more people talking about it, but yet very few reliable and structured information. That is why I wanted to use this blog to encourage the discussion and bring all our knowledge to apply it in the best possible way to our projects.

What is MVP?

The MVP pattern allows **separate the presentation layer from the logic**, so that everything about how the interface works is separated from how we represent it on screen. Ideally the MVP pattern would achieve that same logic might have completely different and interchangeable views.

First thing to clarify is that MVP **is not an architectural pattern**, it's only responsible for the presentation layer . In any case it is always better to use it for your architecture that not using it at all.

Why use MVP?

In Android we have a problem arising from the fact that Android activities are closely coupled to both interface and data access mechanisms. We can find extreme examples such as CursorAdapter, which mix adapters, which are part of the view, with cursors, something that should be relegated to the depths of data access layer .

For an application to be easily extensible and maintainable we need to define well separated layers. What do we do tomorrow if, instead of retrieving the same data from a database, we need to do it from a web service? We would have to redo our entire view .

MVP makes views independent from our data source. We divide the application into at least three different layers, which let us test them independently. With MVP we are able to take most of logic out from the activities so that we can test it without using instrumentation tests.

How to implement MVP for Android

Well, this is where it all starts to become more diffuse. There are many variations of MVP and everyone can adjust the pattern idea to their needs and the way they feel more comfortable. The pattern varies depending basically on the amount of responsibilities that we delegate to the presenter.

Is the view responsible to enable or disable a progress bar, or should it be done by the presenter? And who decides which actions should be shown in the Action Bar? That's where the tough decisions begin . I will show how I usually work, but I want this article to be more a place for discussion that strict guidelines on how to apply MVP, because up to know there is no "standard" way to implement it .

The presenter

The presenter is responsible to act as **the middle man between view and model**. It retrieves data from the model and returns it formatted to the view. But unlike the typical MVC, it also decides what happens when you interact with the view.

The View

The view, usually implemented by an Activity (it may be a Fragment, a View… depending on how the app is structured), will contain a reference to the presenter. Presenter will be ideally provided by a dependency injector such as Dagger, but in case you don't use something like this, it will be responsible for creating the presenter object. The only thing that the view will do is calling a method from the presenter every time there is an interface action (a button click for example).

The model

In an application with a good layered architecture, this model would only be the gateway to the domain layer or business logic. If we were using the Uncle Bob clean architecture , the model would probably be an interactor that implements a use case. But this is another topic that I'd like to discuss in future articles. For now, it is enough to see it as the provider of the data we want to display in the view.

An example

As it is a bit lengthy to explain , I created an MVP example on Github consisting of a login screen that validates the data and allows access to a home with a list of items that are retrieved from the model. This article does not explain any code because it is quite simple, but if you see that you find it difficult to understand I may create another article explaining it in detail.

Go to MVP example on Github

Conclusion

Separating interface from logic in Android is not easy, but the Model-View-Presenter pattern makes a little easier to prevent our activities end up degrading into very coupled classes consisting on hundreds or even thousands of lines. In large applications it is essential to organize our code well. If not, it becomes impossible to maintain and extend