An Introduction to Model View Presenter on Android

by Tin Megali 30 Mar 2016
Difficulty: Intermediate Length: Medium Languages: English ▼

Android Android SDK Java Design Patterns Mobile Development Theory How-To

This post is part of a series called How to Adopt Model View Presenter on Android.

How to Adopt Model View Presenter on Android

When we are developing a complex application, we generally come across challenges that were probably tackled before and that already have some pretty great solutions. Such solutions are often referred to as patterns. We generally talk about design patterns and architectural patterns. They simplify development and we should use them whenever it is appropriate to use them.

This tutorial should help you understand the importance of a well designed project and why Android's standard architecture isn't always sufficient. We discuss a few potential problems that you may encounter when developing Android applications and I show you how to address those issues by improving the app's testability and reliability through the **Model View Presenter** (MVP) pattern.

In this tutorial, we explore:

- the value of applying known architectural patterns in software projects
- why it can be a good idea to change Android's standard architecture
- the key concepts behind the Model View Presenter (MVP) pattern
- the differences between MVC and MVP
- how MVP fits the Android SDK

In the first part of this tutorial, we focus on the theory of the MVP pattern. The second part of this tutorial is more hands-on.

1. Android Architecture

The design of a project should be a concern from the very beginning. One of the first things we should consider is the architecture that we plan to adopt as it will define how different elements of our application relate to one another. It will also establish some ground rules to guide us during development.

In general, a framework or SDK expects things to be done a certain way, but that isn't always the right one for a project. Sometimes, there is no predefined or correct way of doing things, leaving design decisions up to the developer. The Android SDK expects things to be done a certain way, but that isn't always sufficient or the best choice.

Although Android offers an excellent SDK, its architectural patterns are quite unusual and can easily get in your way during development, especially when building complex applications that need to be tested and maintained for a long time. Fortunately, we can choose from multiple architectural solutions to solve this issue.

What Is the Problem?

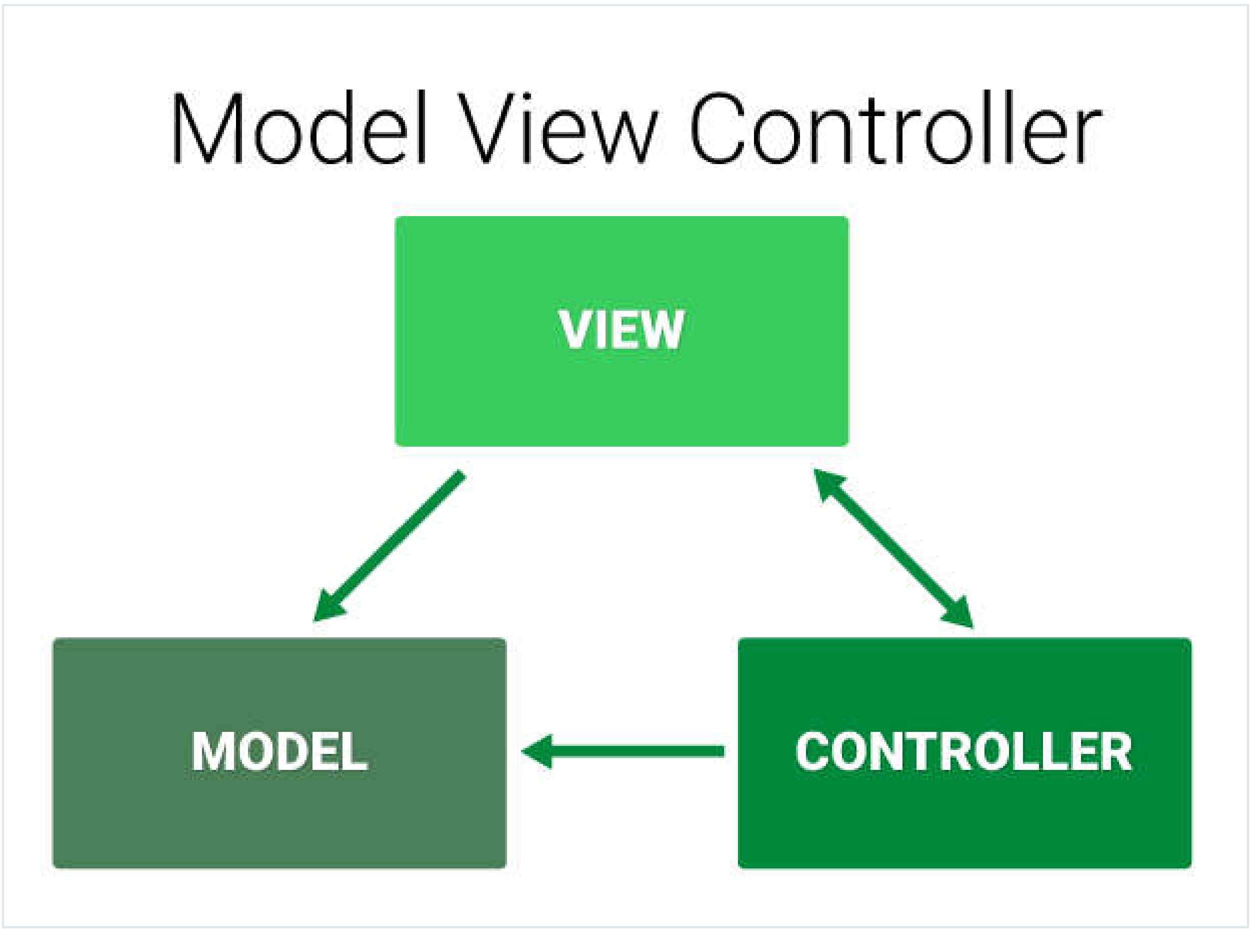
This is a tricky question. Some could say that there aren't any issues with the architecture offered by Android. Certainly, it gets the job done. But can we do better? I strongly believe that we can.

The tools offered by Android, with layouts, Activities, and data structures, seem to steer us in the direction of the Model View Controller (MVC) pattern. MVC is a solid, established pattern that aims to isolate the different roles of an application. This is known as separation of concerns.

This architecture creates three layers:

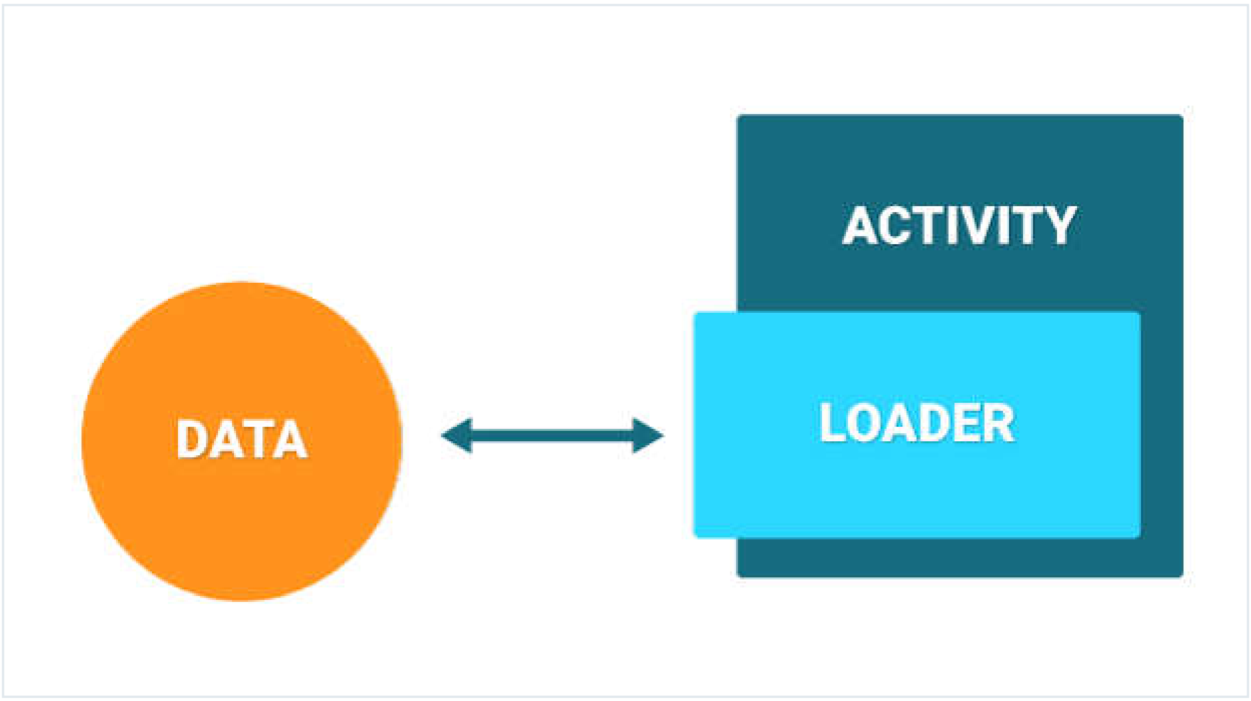
- Model
- View
- Controller

Each layer is responsible for an aspect of the app. **Model** responds to business logic, **View** is the user interface, and **Controller** mediates **View** access to **Model**.



But if we closely analyze Android's architecture, especially the relation between **View** (Activities, Fragments, etc.) and Model (data structures), we can conclude that it cannot be considered MVC. It is quite different from MVC and follows its own rules. It can certainly get in your way when your goal is to create the best application possible.

Being more specific, if we think about the symbiotic connection between loaders and Activities or Fragments, you can understand why we should pay close attention to Android's architecture. An Activity or Fragment is responsible for calling the Loader, who should fetch data and return it to its parent. Its existence is completely tied to its parent and there is no separation between the View role (Activity/Fragment) and business logic performed by the Loader.



How can we use unit testing, in an application in which data (Loader) is so tightly coupled with the View (Activity or Fragment) if the very essence of unit testing is to test the smallest possible piece of code? If you are working in a team and you need to change something in someone else's code, how can you find the issue if the project doesn't stick to a known architectural pattern and anything could be literally anywhere?

What Is the Solution?

We can solve this by implementing a different architectural pattern and, fortunately, the Android SDK allows us to choose between multiple solutions. We can narrow down our options to solutions that are most suitable for Android. The Model View Controller (MVC) pattern is a good choice, but an even better one is the closely related Model View Presenter (MVP) pattern. MVP was developed using the same premises as MVC, but with a more modern paradigm that creates an even better separation of concerns and maximizes the application's testability.

With Android's architecture in mind (or the lack of one), we can conclude that:

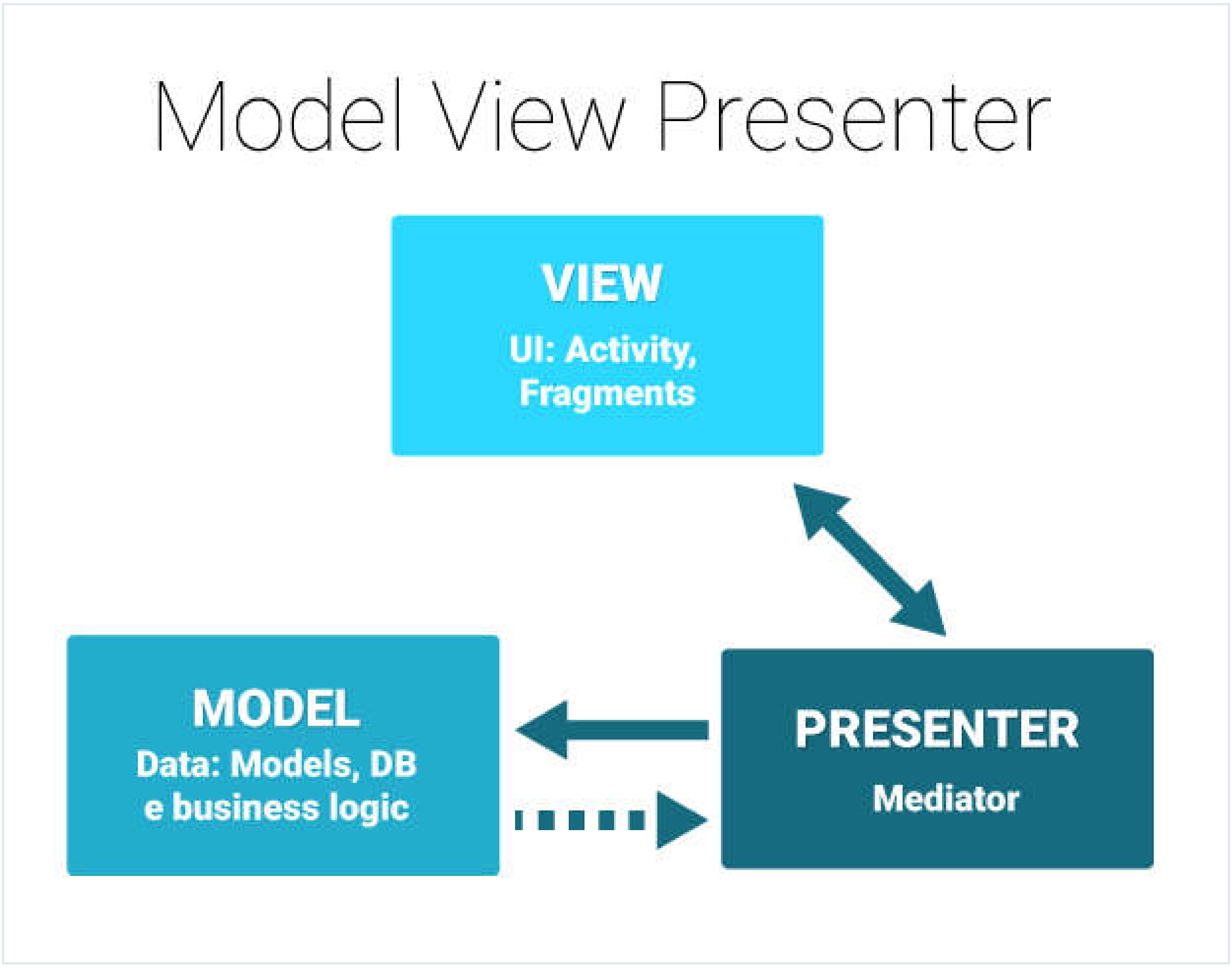
- Android doesn't worry too much about a separation of concerns
- it is best to leave Android's architecture for what it is as it is could lead to problems in the future
- the lack of a proper architectural pattern could make unit testing a real agony
- Android allows us to choose between several alternative architectural patterns
- Model View Presenter (MVP) is one of the best solutions available for Android

2. Model View Presenter on Android

As I mentioned earlier, separation of concerns isn't Android's strongest point. Fortunately, the Model View Presenter pattern improves this significantly. MVP separates the application into three layers:

- Model
- View
- Presenter

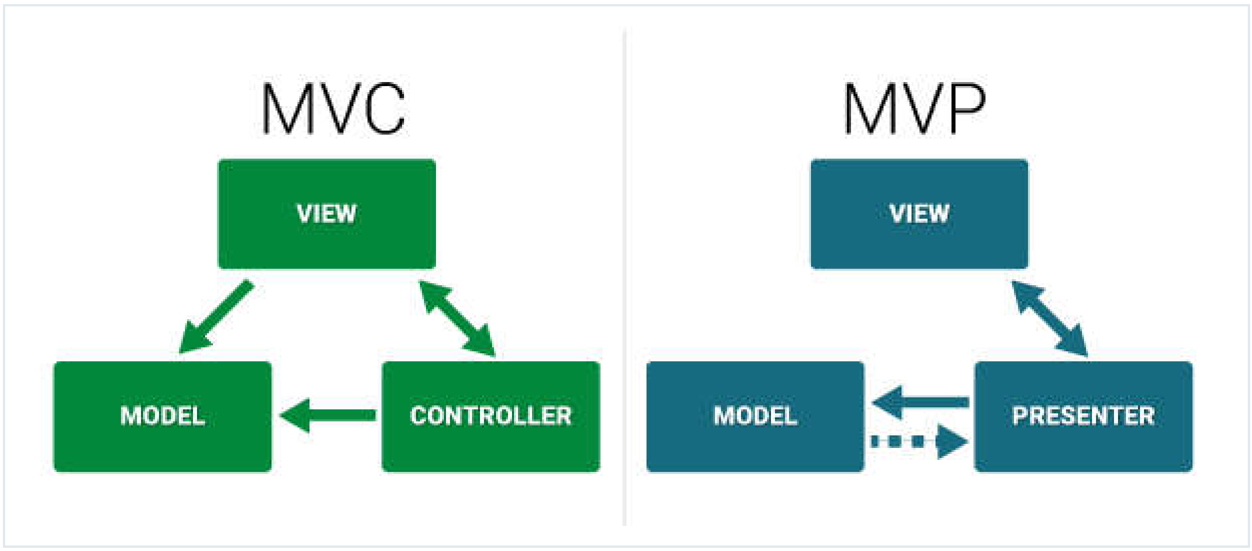
Each one has its responsibilities and communication between these layers is managed by the Presenter. The Presenter works as a mediator between the different parts.



- The Model holds the business logic of the application. It controls how data can be created, stored, and modified.
- The View is a passive interface that displays data and routes user actions to the Presenter.
- The Presenter acts as the middle man. It retrieves data from the Model and shows it in the View. It also processes user actions forwarded to it by the View.

Differences Between MVC and MVP

The Model View Presenter pattern is based on the Model View Controller pattern. Since they share several concepts, it can be hard to differentiate them. The Presenter and the Controller have a similar role. They are responsible for the communication between Model and View. That said, the Controller doesn't manage Model and View as strictly as the Presenter does.



In the MVC pattern, the View layer is somewhat intelligent and can retrieve data directly from the Model. In the MVP pattern, the View is completely passive and data is always delivered to the View by the Presenter. Controllers in MVC can also be shared between multiple Views. In MVP, the View and the Presenter have a one-to-one relationship, therefore, the Presenter is tied to one View.

- In MVP, the View cannot access the Model.
- The Presenter is tied to a single View.
- The View is completely passive in the MVP pattern.

These conceptual differences make that the MVP pattern guarantees a better separation of concerns and it also considerably increases the application's testability by promoting a greater separation of the three core layers.

Activity, Fragment, and View Objects

There are several interpretations of how MVP can be implemented on Android. In general, though, Activities and Fragments are assigned the role of View and are responsible for creating the Presenter and the Model. The View is also responsible for maintaining the Model and the Presenter between configuration changes, informing them about the eventual destruction of the View.

Other View objects, such as RecyclerView, can also be considered part of the View layer in MVP. There is a one-to-one relation between the View and the Presenter and sometimes complex situations may ask for multiple Presenters.

What We Know So Far

- By using architectural and design patterns, we can make development a lot easier and transparent.
- Android lacks a well structured architectural pattern.
- Without the use of established design patterns, we may run into a number of difficulties along the way, especially problems related to maintainability and testability.
- The Model View Presenter pattern increases the separation of concerns and facilitates unit testing.
- The Presenter mediates the communication between the View and the Model.
- The View displays data and directs user interaction to the Presenter.
- The Model is in charge of the application's business logic.
- The View role is mostly assumed by an Activity or Fragment.

Conclusion

In the next tutorial, we implement the Model View Presenter pattern on Android. We put to test the concepts we learned in this tutorial and further explore the pattern's complexities and what that means for Android.

At the end of this series, you are able to implement MVP in your own projects, create your own framework, or adopt other known solutions. I hope to see you in the next tutorial.