# MVC

The MVC pattern suggests splitting the code into 3 components. While creating the class/file of the application, the developer must categorize it into one of the following three layers:

1. Model: This component stores the application data. It has no knowledge about the interface. The model is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

2. View: It is the UI(User Interface) layer that holds components that are visible on the screen. Moreover, it provides the visualization of the data stored in the Model and offers interaction to the user.

3. Controller: This component establishes the relationship between the View and the Model. It contains the core application logic and gets informed of the user’s response and updates the Model as per the need.

# MVC vs MVP vs MVVM

1. MVC is one of the oldest software architecture.

1-1. MVP was developed as the second iteration of software architecture which is advanced from MVC.

1-2. MVVM is Industry-recognized architecture pattern for applications.

2. MVC, UI(View) and data-access mechanism(Model) are tightly coupled.

2-1. MVP resolves the problem of having a dependent View by using Presenter as a communication channel between Model and View.

2-2. MVVM is more event-driven as it uses data binding and thus makes easy separation of core business logic from the View.

3. MVC, Controller and View exist with the one-to-many relationship. One Controller can select a different view based upon the required operation.

3-1. MVP, A one-to-one relationship exists between Presenter and View as one Presenter class manages one View at a time.

3-2. MVVM, Multiple View can be mapped with a single ViewModel and thus, the one-to-many relationship exists between View and ViewModel.

4. MVC, The View has no knowledge about the Controller.

4-1. MVP, The View has references to the Presenter.

4-2. MVVM, The View has references to the ViewModel

5. MVC, Difficult to make changes and modify the app features as the code layers are tightly coupled.

5-1. MVP, Code layers are loosely coupled and thus it is easy to carry out modifications/changes in the application code.

5-2. MVVM, Easy to make changes in the application. However, if data binding logic is too complex, it will be a little harder to debug the application.

6. MVC, User Inputs are handled by the Controller.

6-1. MVP, The View is the entry point to the Application.

6-2. MVVM, The View takes the input from the user and acts as the entry point of the application.

7. MVC is ideal for small scale projects only.

7-1. MVP is ideal for simple and complex applications.

7-2. MVVM is not ideal for small scale projects.

8. MVC, Limited support to Unit testing.

8-1. MVP, Easy to carry out Unit testing but a tight bond of View and Presenter can make it slightly difficult.

8-2. MVVM, Unit testability is highest in this architecture.

9. MVC has a high dependency on Android APIs.

9-1. MVP has a low dependency on the Android APIs.

9-2. MVVM Has low or no dependency on the Android APIs.

10. MVC does not follow the modular and single responsibility principle.

10-1. MVP follows modular and single responsibility principle.

10-2. MVVM follows modular and single responsibility principle.

# MVVM

Model — View — ViewModel (MVVM) is the industry-recognized software architecture pattern that overcomes all drawbacks of MVP and MVC design patterns. MVVM suggests separating the data presentation logic(Views or UI) from the core business logic part of the application.

The separate code layers of MVVM are:

1. Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

2. View: The purpose of this layer is to inform the ViewModel about the user’s action. This layer observes the ViewModel and does not contain any kind of application logic.

3. ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.

MVVM pattern has some similarities with the MVP(Model — View — Presenter) design pattern as the Presenter role is played by ViewModel. However, the drawbacks of the MVP pattern has been solved by MVVM in the following ways:

1. ViewModel does not hold any kind of reference to the View.

2. Many to-1 relationships exist between View and ViewModel.

3. No triggering methods to update the View.

There are 2 ways to implement MVVM design pattern in Android projects:

1. Using the DataBinding library released by Google

2. Using any tool like RxJava for DataBinding.

# MVP

MVP pattern overcomes the challenges of MVC and provides an easy way to structure the project codes. The reason why MVP is widely accepted is that it provides modularity, testability, and a more clean and maintainable codebase. It is composed of the following three components:

1. Model: Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

2. View: UI(User Interface) layer. It provides the visualization of the data and keep a track of the user’s action in order to notify the Presenter.

3. Presenter: Fetch the data from the model and applies the UI logic to decide what to display. It manages the state of the View and takes actions according to the user’s input notification from the View.

# MVC pros/cons

pros:

1. No business logic in UI

2. Scalable

3. Easy to maintain, test, upgrade

4. Friendly development process

5. Great for complex application

6. Suitable for team base work

cons:

1. Doesn't scale. Separates UI, but not model

2. Controller often grows too big

3. Violates Single Responsibility (Controller responsible for almost everything), Interface Segregation SOLID principles

# MVVM pros/cons

pros:

1. MVVM facilitates easier parallel development of a UI and the building blocks that power it.

2. MVVM abstracts the View and thus reduces the quantity of business logic (or glue) required in the code behind it.

3. The ViewModel can be easier to unit test than in the case of event-driven code.

4. The ViewModel (being more Model than View) can be tested without concerns of UI automation and interaction.

cons:

1. For simpler UIs, MVVM can be overkill.

2. While data bindings can be declarative and nice to work with, they can be harder to debug than imperative code where we simply set breakpoints.

3. Data bindings in nontrivial applications can create a lot of bookkeeping. We also don’t want to end up in a situation where bindings are heavier than the objects being bound to.

4. In larger applications, it can be more difficult to design the ViewModel up front to get the necessary amount of generalization.

# MVP pros/cons

pros:

1. Complex Tasks split into simpler tasks

2. Smaller objects, less bugs, easier to debug

3. Testable

cons:

1. Model can't be resued, tied to specific use case

2. View and Presenter are tied to data objects since they share the same type of object with the model

# MVVM over MVC

I think that MVVM is a better choice for developing Android apps for a few reasons.

First, it allows for a more separation of concerns between the different parts of the app, which can make development and maintenance simpler.

Second, it can make it easier to bind data to the UI, since the ViewModel can act as a mediator between the data and the View. Finally, it can help to improve performance, since the ViewModel can handle tasks that would otherwise need to be done on the UI thread.

# two-way vs one-way data binding

Two-way data binding means that changes to either the model or the view will update the other automatically. One-way data binding means that changes to the model will update the view, but not vice versa.

# @Bindable

One way to improve performance when using data binding is to use the @Bindable annotation. This annotation tells the data binding system to only update the UI when the data has actually changed. Without this annotation, the data binding system will update the UI every time any data is changed, even if the UI doesn’t need to be updated.

# Observables vs LiveData

1. LiveData is an observable data holder class. It is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

2. Observables, on the other hand, are not lifecycle-aware. This means they will continue to emit data even if the observer is no longer in an active state, such as when an activity has been destroyed. This can lead to memory leaks if the observer is not unregistered when no longer needed.

# reactive programming

Reactive programming is a programming paradigm that is concerned with data streams and the propagation of change. This means that when a piece of data changes, all observers of that data are notified and updated automatically. This is in contrast to the more traditional approach of having to manually keep track of what data has changed and then update the observers accordingly.

# reactive programming vs event handling

Reactive programming is a programming paradigm that is concerned with data streams and the propagation of changes. Event handling, on the other hand, is a way of responding to events that occur in a program. In the context of Android development, reactive programming can be used to handle events such as user input, while event handling is used to respond to events such as button clicks.

# observables, observers

In reactive programming, observables are objects that emit a stream of data, and observers are objects that consume that data. In the context of Android development, observables can be used to emit data from a ViewModel to a View, and observers can be used to consume that data and update the UI.