**Architecture**

Android architecture components are a collection of libraries that help you design robust, testable, and maintainable apps. Start with classes for managing your UI component lifecycle and handling data persistence.

**Standard android ( Model – View – Controller)**

Is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

* + - Model : it includes all the data and its related logic.
    - View: The frontend or graphical user interface (GUI).
    - Controller: It is the logic layer. It gets notify the user’s behavior and updates to the Model as required.

The MVC pattern helps you break up the frontend and backend code into separate components. This way, it's much easier to manage and make changes to either side without them interfering with each other.

MVC is just a pattern that applies well to the web. In particular it applies well to the idea that an application is accessed in the following way:

The user asks for some resource

Some underlying data is retrieved from somewhere.

A template is then applied to that data in order to show it to the user.

**Clean architecture (Model – View – Presenter)**

MVP is a derivation of the MVC (Model View Controller example) architectural pattern. It is used for building user interfaces. The MVP pattern allows separating the presentation layer from the logic. It means that everything from how the interface works to how it represents on the screen. MVP is a user interface architectural pattern, which eases automated unit testing and it is responsible to provide clean code

* Model: which holds the business logic as well as controls how data is created, stored and modified. In Android, it is a data access layer, for example. The Model consists of components that are responsible for functionalities like for generating, storing, exposing and fetching the data. All these functionalities usually perform in the background thread.
* View: It is a passive interface, which displays data, and the routes user actions to Presenter. In Android, the View is represented by Activity, Fragment or View. The View contains the UI and it does not contain any logic or knowledge of the displayed data.
* Presenter: The Presenter is in between Model and the View. And, it triggers the business logic, and lets to know ‘the View’ when to update. It recovers data received from the Model and shows it in the View. It interacts with the Model, then fetches and transforms the data from the Model to update the view.

In Android, the application should be easily extensible and maintainable. Therefore,

in order to maintain the level, it is important to define separated layers well. And, MVP makes things easier for developers and it makes the views independent of the data source.

MVP (Model – View – Presenter) is highly recommended for Android developers to build up an agile and scalable android application with minimal cost as compared to MVC.

**Data-binding MVVM (Model-View-ViewModel)**

Is a Model-View-ViewModel architecture that removes the tight coupling between each component. Most importantly, in this architecture, the children don't have the direct reference to the parent, they only have the reference by observables.

* + - Model: It represents the data and the business logic of the Android Application. It consists of the business logic - local and remote data source, model classes, repository.
    - View: It consists of the UI Code(Activity, Fragment), XML. It sends the user action to the ViewModel but does not get the response back directly. To get the response, it has to subscribe to the observables which ViewModel exposes to it.
    - ViewModel: It is a bridge between the View and Model(business logic). It does not have any clue which View has to use it as it does not have a direct reference to the View.

MVVM guides us how to distribute responsibilities between classes in a GUI application (or between layers - more about this later), with the goal of having a small number of classes, while keeping the number of responsibilities per class small and well defined. Proper MVVM assumes at least a moderately complex application, which deals with data it gets from "somewhere". It may get the data from a database, a file, a web service, or from a myriad of other sources.

**Solid**

**S**ingle responsibility principle

A class should have one and only one reason to change, meaning that a class should have only one job.

**O**pen-Closed Principle

Objects or entities should be open for extension but closed for modification.

**L**iskov Substitution Principle

If class A is a subtype of class B, we should be able to replace B with A without disrupting the behavior of our program.

**I**nterface Segregation Principle

It simply means that larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

**D**ependency Inversion Principle

Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

SOLID Principles is a coding standard that all developers should have a clear concept for developing software properly to avoid a bad design. When applied properly it makes your code more extendable, logical, and easier to read.

When the developer builds software following a bad design, the code can become inflexible and more brittle. Small changes in the software can result in bugs. For these reasons, we should follow SOLID Principles.

**Design pattern**

Design Pattern is a general solution to solve common problems when designing software in object-oriented programming

Why should use Design pattern ?

* Speed up software development
* Limit potential errors
* Support code reuse
* Make the code easier to read

Creational design patterns

This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

1. Abstract Factory

Creates an instance of several families of classes.

1. Builder

Separates object construction from its representation.

1. Factory Method

Creates an instance of several derived classes.

1. Object Pool

Avoid expensive acquisition and release of resources by recycling objects that are no longer in use.

1. Prototype

A fully initialized instance to be copied or cloned.

1. Singleton

A class of which only a single instance can exist.

Structural design patterns

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

1. Adapter

Match interfaces of different classes

1. Bridge

Separates an object’s interface from its implementation

1. Composite

A tree structure of simple and composite objects

1. Decorator

Add responsibilities to objects dynamically

1. Facade

A single class that represents an entire subsystem

1. Flyweight

A fine-grained instance used for efficient sharing

1. Private Class Data

Restricts accessor/mutator access

1. Proxy

An object representing another object

Behavioral design patterns

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

1. Chain of responsibility

A way of passing a request between a chain of objects

1. Command

Encapsulate a command request as an object

1. Interpreter

A way to include language elements in a program

1. Iterator

Sequentially access the elements of a collection

1. Mediator

Defines simplified communication between classes

1. Memento

Capture and restore an object's internal state

1. Null Object

Designed to act as a default value of an object

1. Observer

A way of notifying change to a number of classes

1. State

Alter an object's behavior when its state changes

1. Strategy

Encapsulates an algorithm inside a class

1. Template method

Defer the exact steps of an algorithm to a subclass

1. Visitor

Defines a new operation to a class without change