**Flux**:

It has three major parts: the dispatcher that sends every action to all the stores in the application. The stores that accept updates and reconcile them as appropriate, rather than depending on something external to update its data in a consistent way and are responsible for a domain of the application, and only update themselves in response to actions. Nothing outside the store has any insight into how it manages the data for its domain, helping to keep a clear separation of concerns. And the view propagates an action through a central dispatcher, to the various stores that hold the application's data and business logic, which updates all of the views that are affected.

​​When a user clicks on something, the view creates actions. Action can create new data and send it to the dispatcher. The dispatcher then dispatches the action result to the appropriate store. The store updates the state based on the result and sends an update to the view.

**MVC**:

* Model: Maintains the data and behavior of an application
* View: Displays the model in the UI
* Controller: Serves as an interface between view & model components

**MVP**:

* 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.
* View**:** UI(User Interface) layer. It provides the visualization of the data and keeps a track of the user’s action in order to notify the Presenter.
* Presenter**:** Fetch the data from the model and apply 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.

Communication between View-Presenter and Presenter-Model happens via an interface(also called Contract).

One Presenter class manages one View at a time i.e., there is a one-to-one relationship between Presenter and View.

Model and View class doesn’t have knowledge about each other’s existence.

**MVVM**:

The view is responsible for defining the structure, layout, and appearance of what the user sees on screen.

The view model implements properties and commands to which the view can data bind to, and notifies the view of any state changes through change notification events. The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be displayed. Model classes are non-visual classes that encapsulate the app's data. Therefore, the model can be thought of as representing the app's domain model, which usually includes a data model along with business and validation logic. Model classes are typically used in conjunction with services or repositories that encapsulate data access and caching.

This pattern helps to cleanly separate the business and presentation logic of an application from its user interface (UI). Maintaining a clean separation between application logic and the UI helps to address numerous development issues and can make an application easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of an app.

**VIPER**:

**View:** Class that has all the code to show the app interface to the user and get their responses. Upon receiving a response View alerts the Presenter.

**Presenter:** Nucleus of a module. It gets user response from the View and works accordingly. The only class to communicate with all the other components. Calls the router for wire-framing, Interactor to fetch data (network calls or local data calls), view to update the UI.

**Interactor:** Has the business logic of an app. e.g if business logic depends on making network calls then it is Interactor’s responsibility to do so.

**Router:** Does the wire-framing. Listens from the presenter about which screen to present and executes that. Controls the navigation flow between screens or between modules by changing the root of the current Window.

**Entity:** Contains plain model classes used by the Interactor. It’s the information and data used on the functionality.

It is a delegation-driven architecture. Most of the communication between different layers executes through delegation. One layer calls another through a protocol. The calling layer calls a function from a protocol. The listening layer conforms to that protocol and implements the function.

Flux works with actions, dispatchers and stores; actions send new data to the dispatcher and this sends to the store, store updates the state based on the result and sends an update to the view.

The controller receives the request from the user, it uses the appropriate Model & View and generates the response sending it back to the user.

MVP works with Presenters that fetch the data from the model and apply the UI logic to decide what to display according to the user’s input notification from the view.

MVVM works with view models that implement properties and commands that notify the view of any state changes through change notification events.

VIPER works with a presenter who is the only class to communicate with all the other components. It gets user response from the view and works accordingly