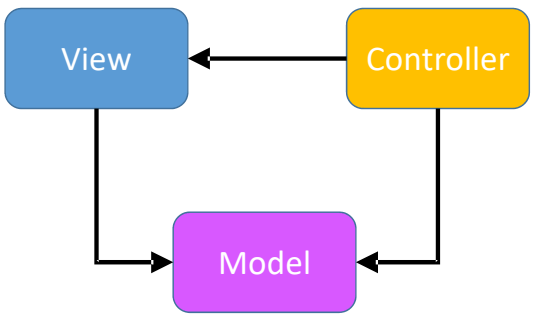
Architectural Patterns

MVC, MVP, MVVM

Clean Architecture

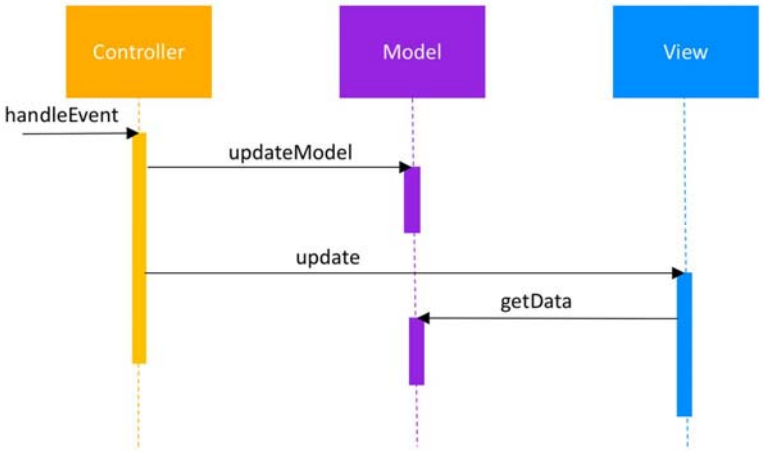
MVC

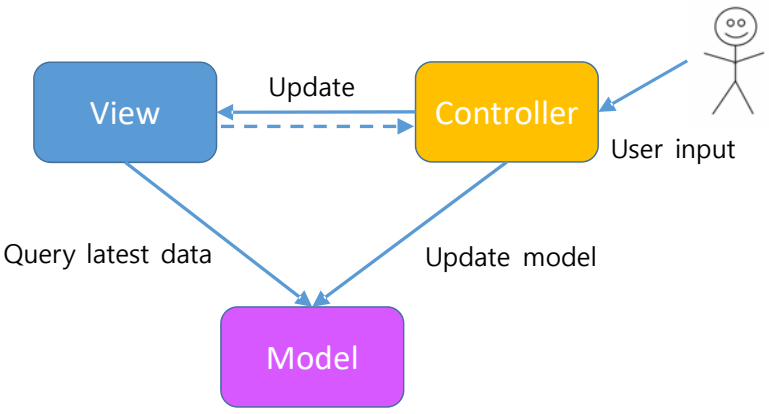
- **Model** - the data layer, responsible for managing the ***business logic*** and handling network or database API.
- **View** - the UI layer - a visualization of the data from the Model.
- **Controller** - the ***logic layer***, gets notified of the user's behavior and updates the Model as needed.



Passive Model

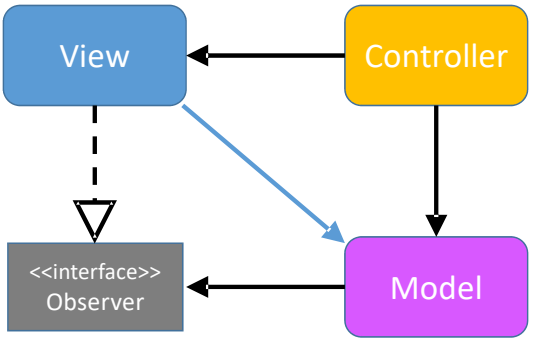
- **Controller** is the only class that manipulates the **Model**

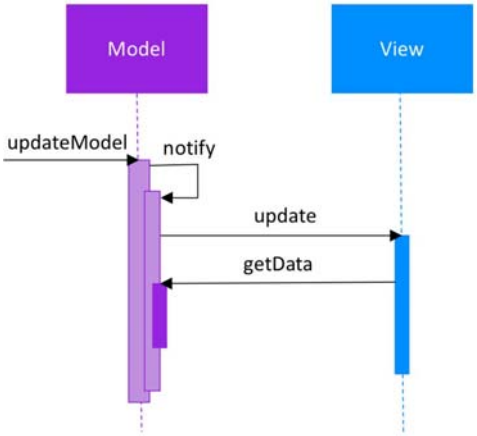


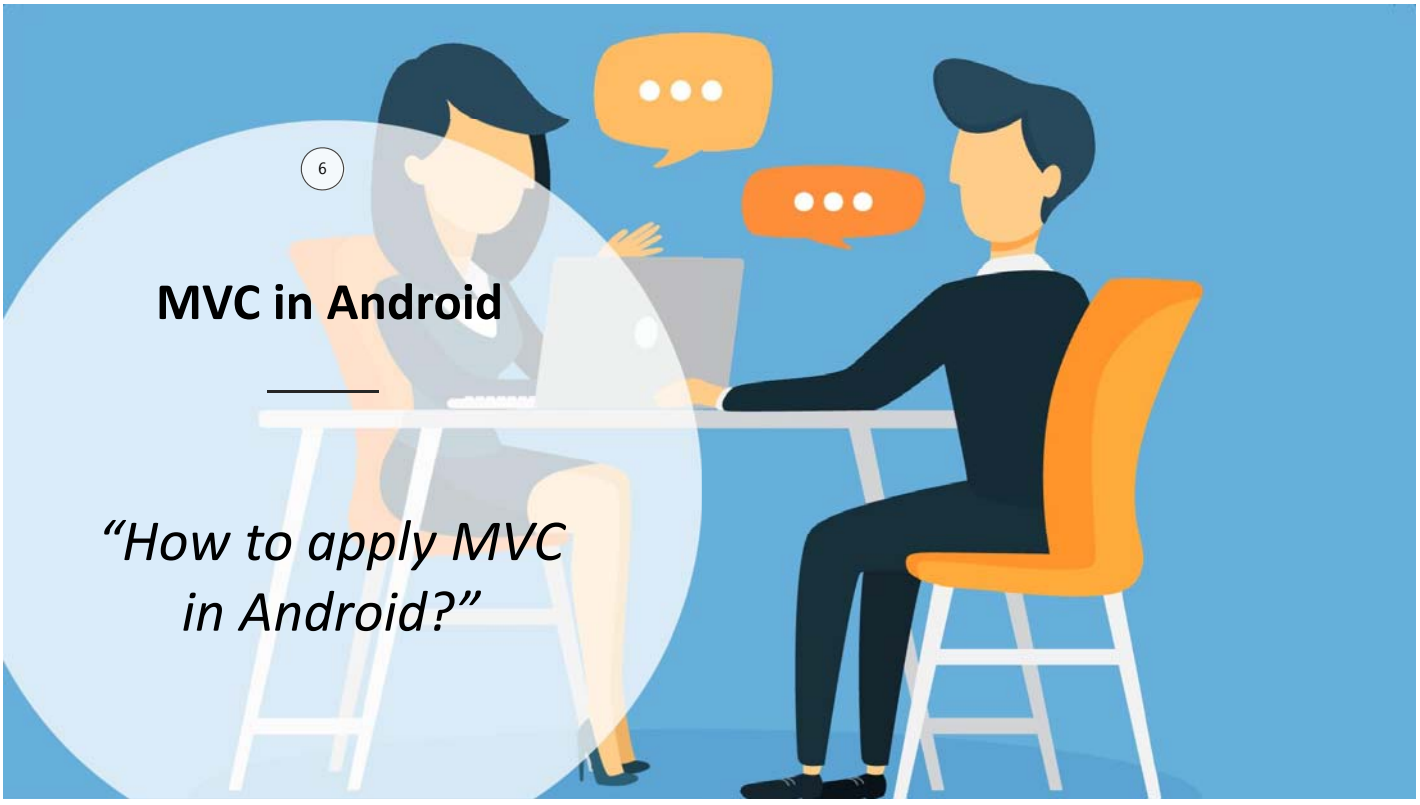


Active Model

- Observer pattern



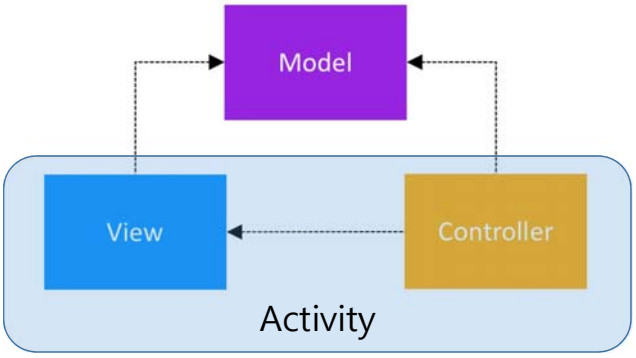




Looking back, this sounds almost crazy!

Typical answer:

An **Activity** is both the **View** and the **Controller**.



Issues with Legacy Apps

- They're extremely complex and difficult to read and modify.
- They contain all kinds of interdependences that can cause issues. So one thing breaks, likely many things will break.
- Dealing with lifecycle event is a pain because there's so many variables you need to account for.
- They are very difficult to unit test.

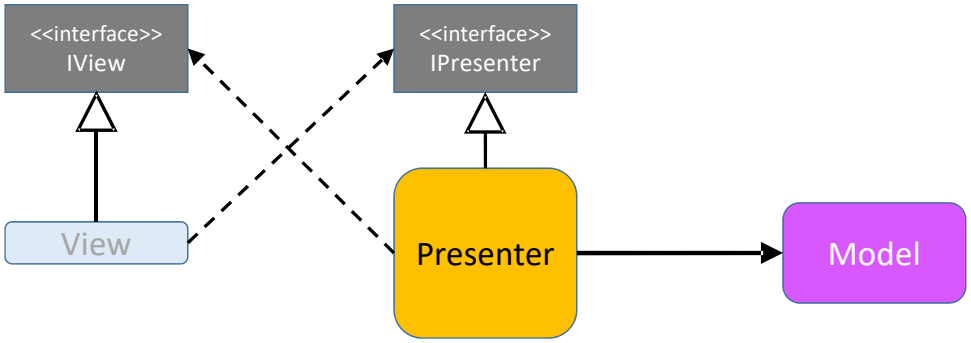
MVP

- **Model** - the data layer. Responsible for handling the business logic and communication with the network and database layers.
- **View** - the UI layer. Displays the data and notifies **Presenter** about user actions.
- **Presenter** - retrieves data from **Model**, applies the UI logic and manages the state of **View**, decides what to display and reacts to user input notifications from **View**.

It this more testable?

MVP (cont'd)

- **Model** - the data layer. Responsible for handling the business logic and communication with the network and database layers.
- **View** - the UI layer. Displays the data and notifies **Presenter** about user actions.
- **Presenter** - retrieves data from **Model**, applies the UI logic and manages the state of **View**, decides what to display and reacts to user input notifications from **View**.



MVP (cont'd)

- The relationship between the **Presenter** and its corresponding **View** is defined in a ***Contract*** interface class, making the code more readable and the connection between the two easier to understand.

```
public interface LoginContract {

    interface ILoginView extends BaseView<ILoginPresenter> {
        void showProgress();
        void hideProgress();
        void showUserNameEmptyError();
        void showPasswordEmptyError();
        void showLoginFailed();
        void navigateToMain();
        void showAlert();
    }

    interface ILoginPresenter extends BasePresenter {
        boolean login(String username, String password);
        void onDestroy();
    }
}
```

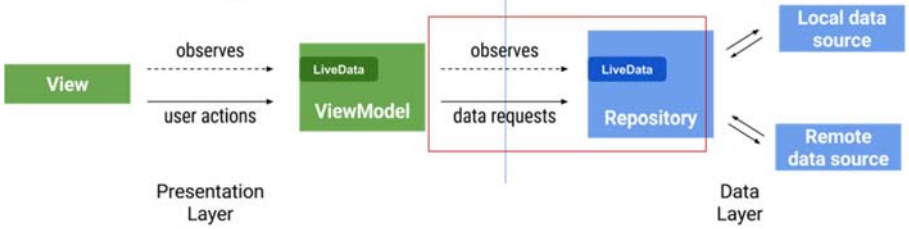

Problems of MVP

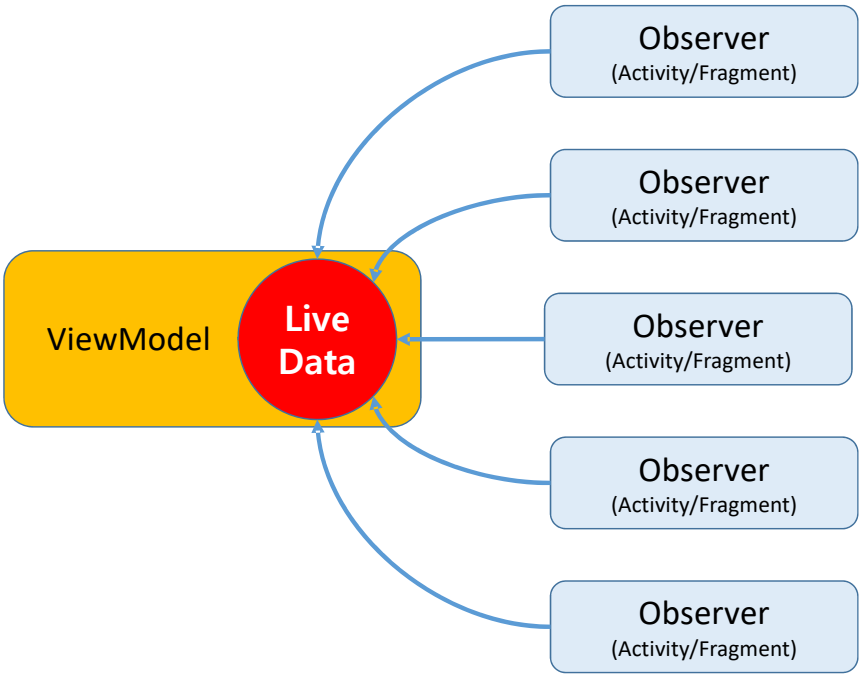
- Presenter is not life-cycle aware.
  - can have obsolete View references, which may cause memory leaks and app crash.
- Bloated Presenter
- Cyclic dependency?

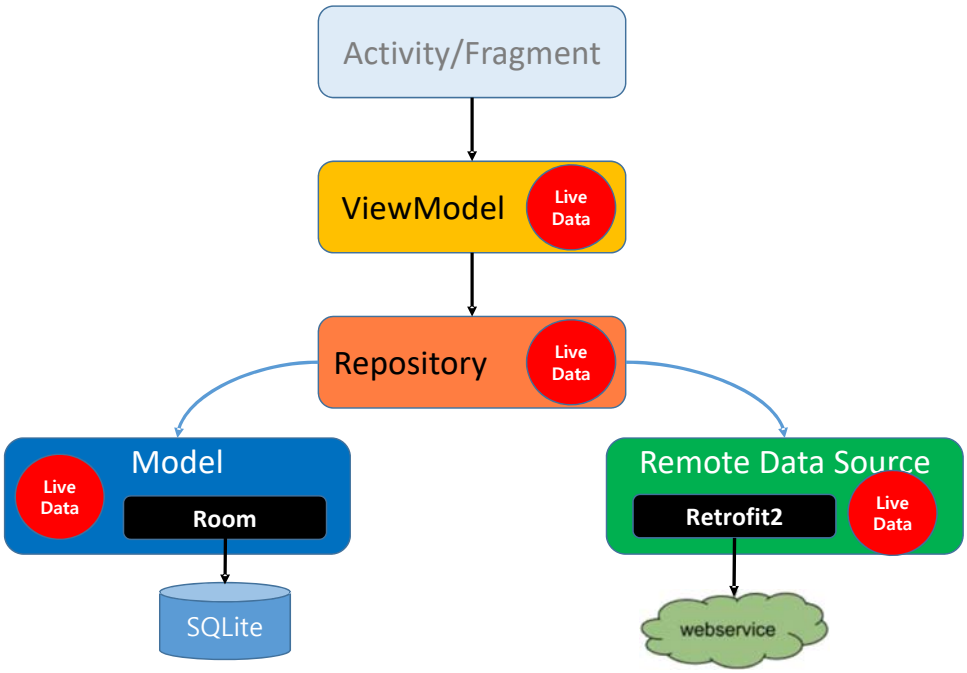
MVVM

Google
Android Architecture Component Library

- **View** - informs the **ViewModel** about the user's actions
- **ViewModel** - *presentation layer, having UI logic and exposes streams of data relevant to the* **View**
  - Everything should go through the view model.
- **DataModel** - abstracts the data source. The **ViewModel** works with the **DataModel** to get and save the data.







Benefits of MVVM and LiveData

1. LiveData ensures your UI matches your data state.
   - LiveData follows the observer pattern.
   - LiveData notify observer objects when the lifecycle state changes.
   - LiveData can leads to cleaner code structure

2. You won't get memory leaks, well .. Less likely :-)
   - Observers are bound to lifecycle objects.
   - Lifecycle objects are cleaned up after their associated lifecycle is destroyed.

Benefits of MVVM and LiveData (Cont'd)

3. No crashes due to stopped activities
   - If the observers lifecycle is inactive, such as in the case of an activity in the backstack, then it does not receive any LiveData events.

4. No more manual lifecycle handling
   - UI components just observe relevant data and don't stop or resume observation.
   - LiveData automatically manages all of this since it's aware of the relevant lifecycle status changes while observing.

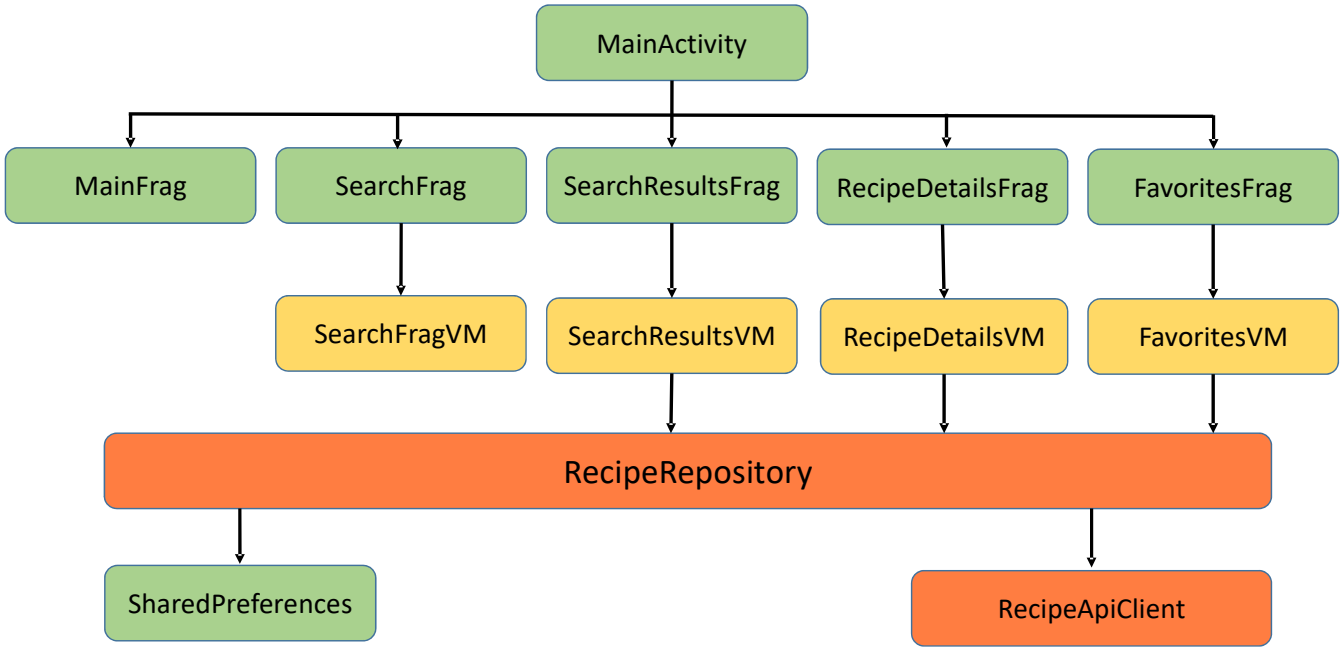
Benefits of MVVM and LiveData (Cont'd)

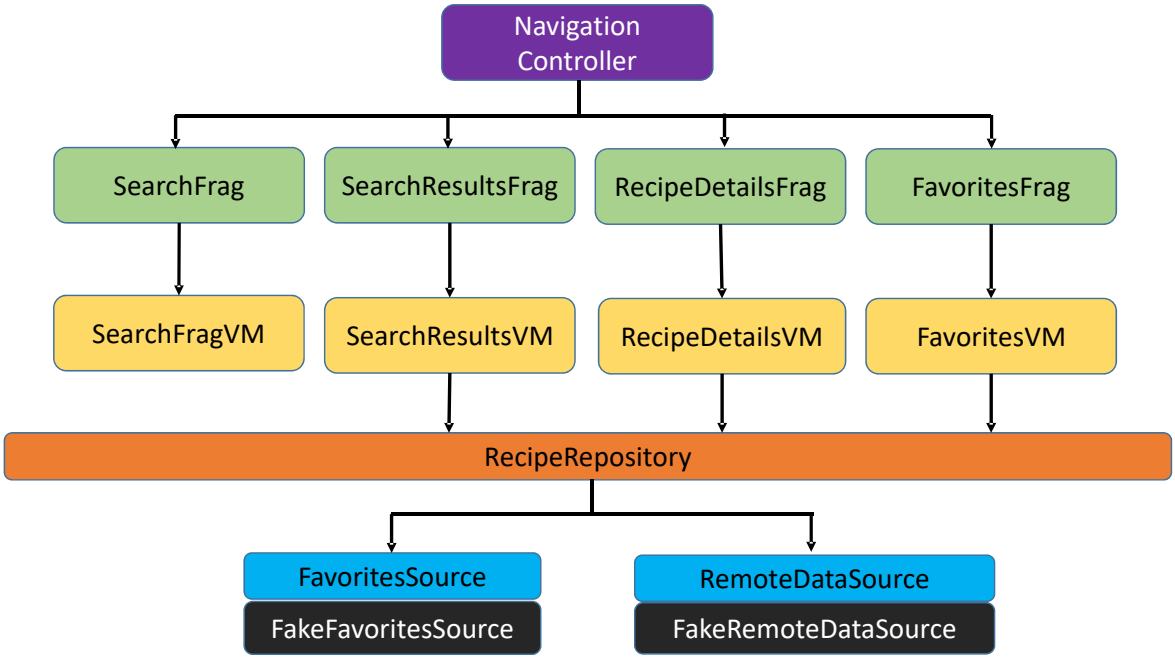
5. Your data is always up to date.
   - What happens if a lifecycle becomes inactive?
   - It would receive the latest data on becoming active again.

6. Proper configuration changes
   - On configuration changes, like device rotation, immediately latest available data is received.
   - Prevents redundant request to network or DB.

MVP/MVVM and Testability

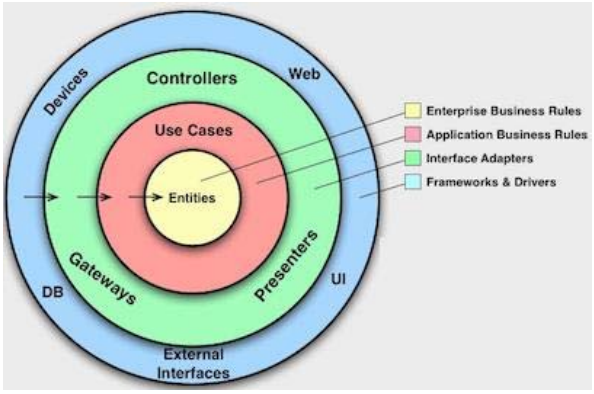
- **Views** should be dumb and implement Passive View.
  - Passive View pattern is a concept that UI logic should be minimum.
  - As for SRP, **views have only the responsibility of rendering.**
- The **Presentation layer** should be responsible of having business logic (like input validation), communication to data layer and provide data for UI rendering.
- **Presentation layer** (**ViewModel** or **Presenter**) **does not use android framework it should be agnostic,** so you can encapsulate all logic and that makes it testable. **It should not have any direct dependency of view.**
  - We can use JVM unit testing here.

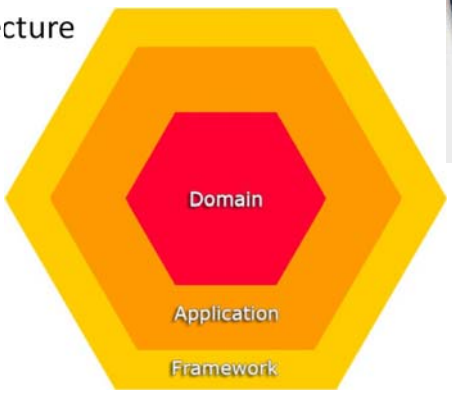


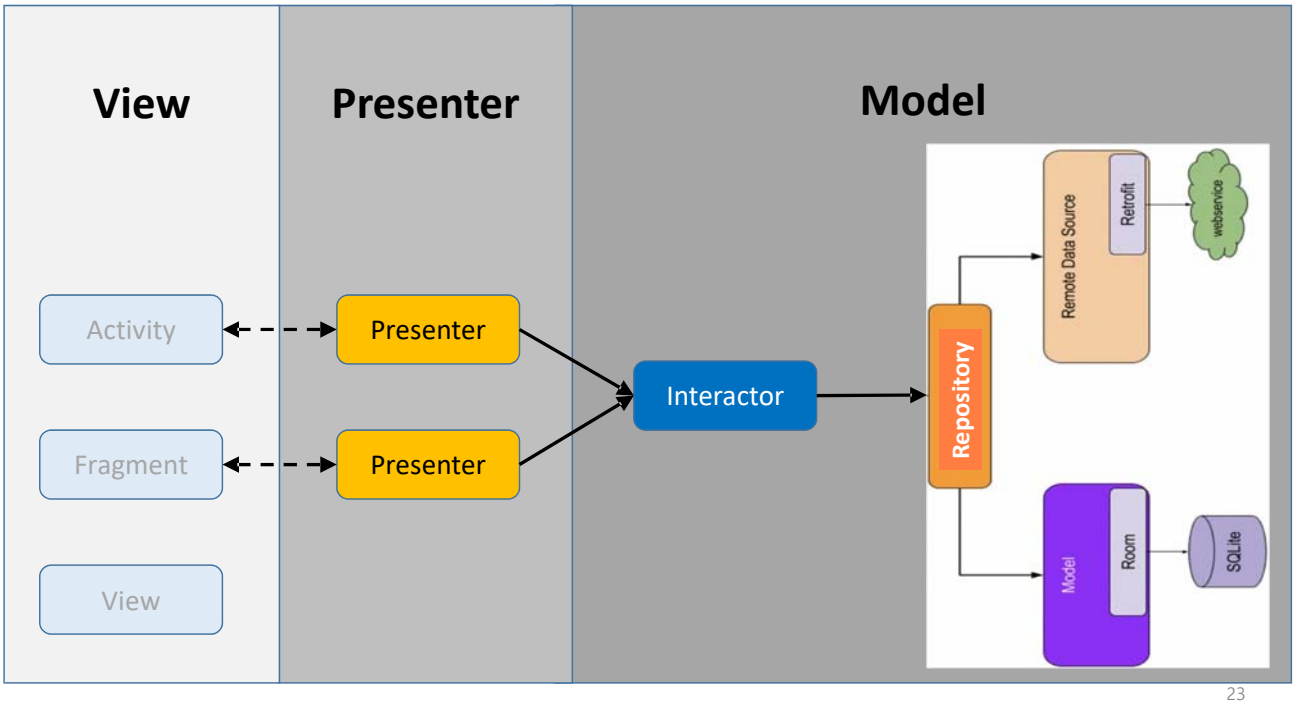


What about Use Cases?

- Hexagonal Architecture
  - aka Ports and Adapter)
- Clean Architecture
- Onion Architecture






View
Presenter
Model
Activity
Fragment
View
Presenter
Presenter
Interactor
Repository
Remote Data Source
Retrofit
webservice
Model
Room
SQLite