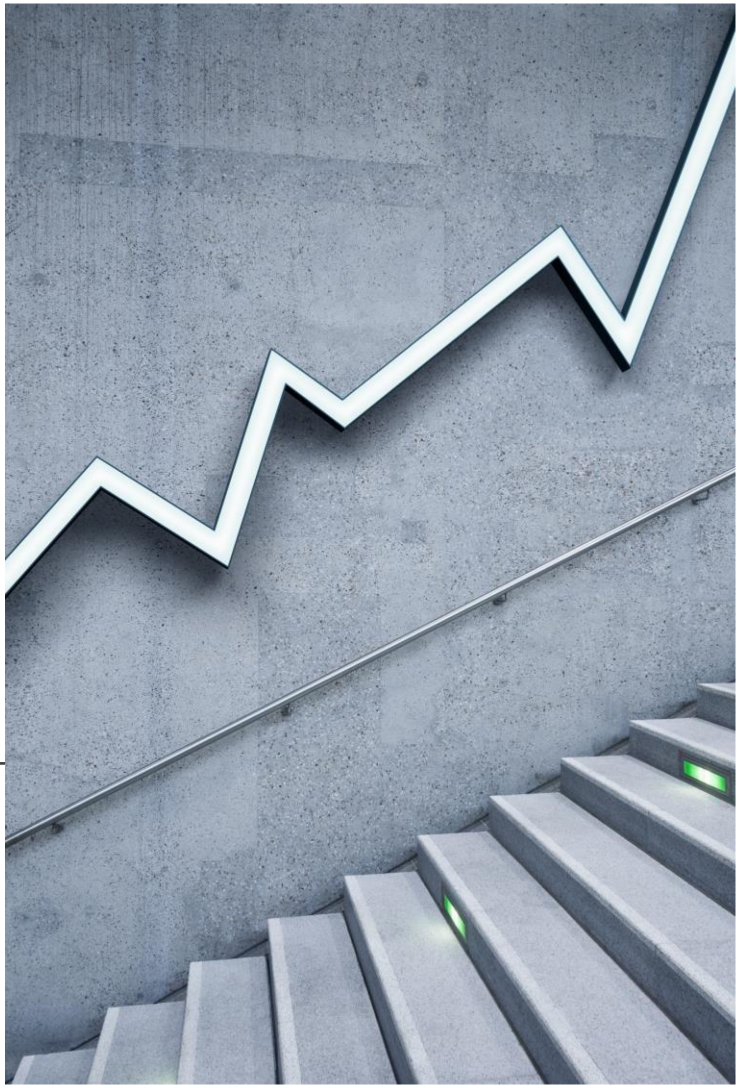


Mobile Architecture

MVC AND MORE

Slides by Tami Meredith

Components

All applications have the same components:

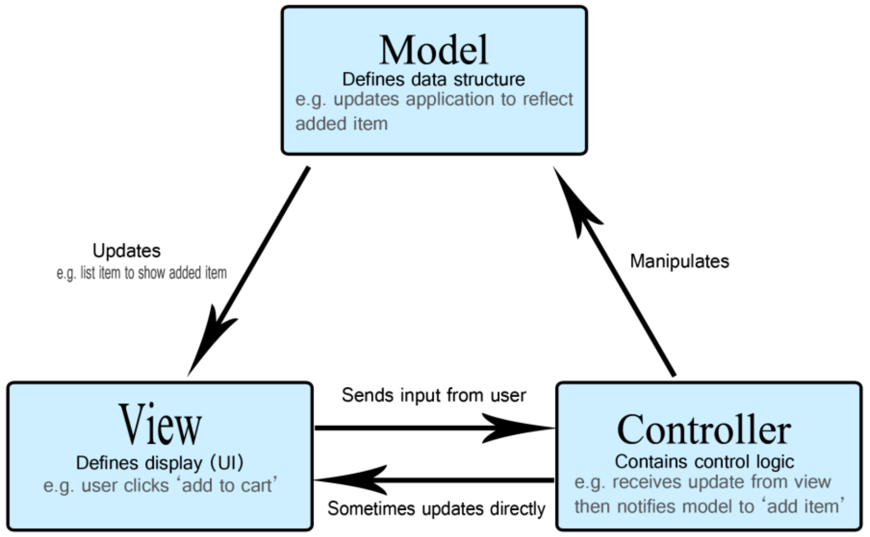
- **User**: Someone to use the application.
- **Data**: The stuff that the user sees.
- **UI**: Something to show the data to the user.
- **UX**: Something to collect user input.
- **Logic**: Techniques to manipulate the data based on input.

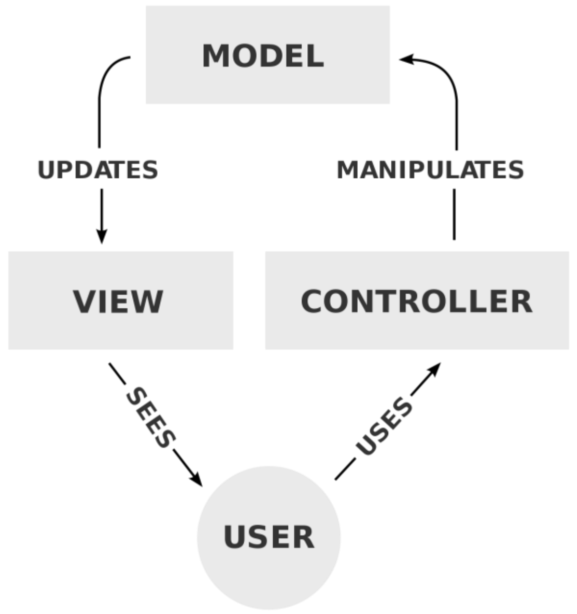
How we combine these parts defines the application’s architecture.

Mobile Architectural Patterns

- Model View Controller
- Model View Presenter
- Model View ViewModel
- Model View Intent
- Clean Architecture

Model View Controller


Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

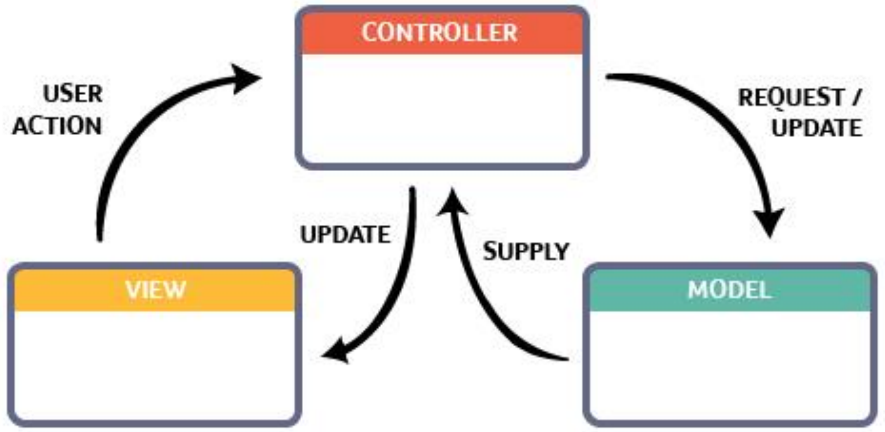


Foundational pattern for UI-based applications.

Model:	Responsible for handling the application's data.
View:	The component that presents the data to the user.
Controller:	Manipulates the data in response to user input.

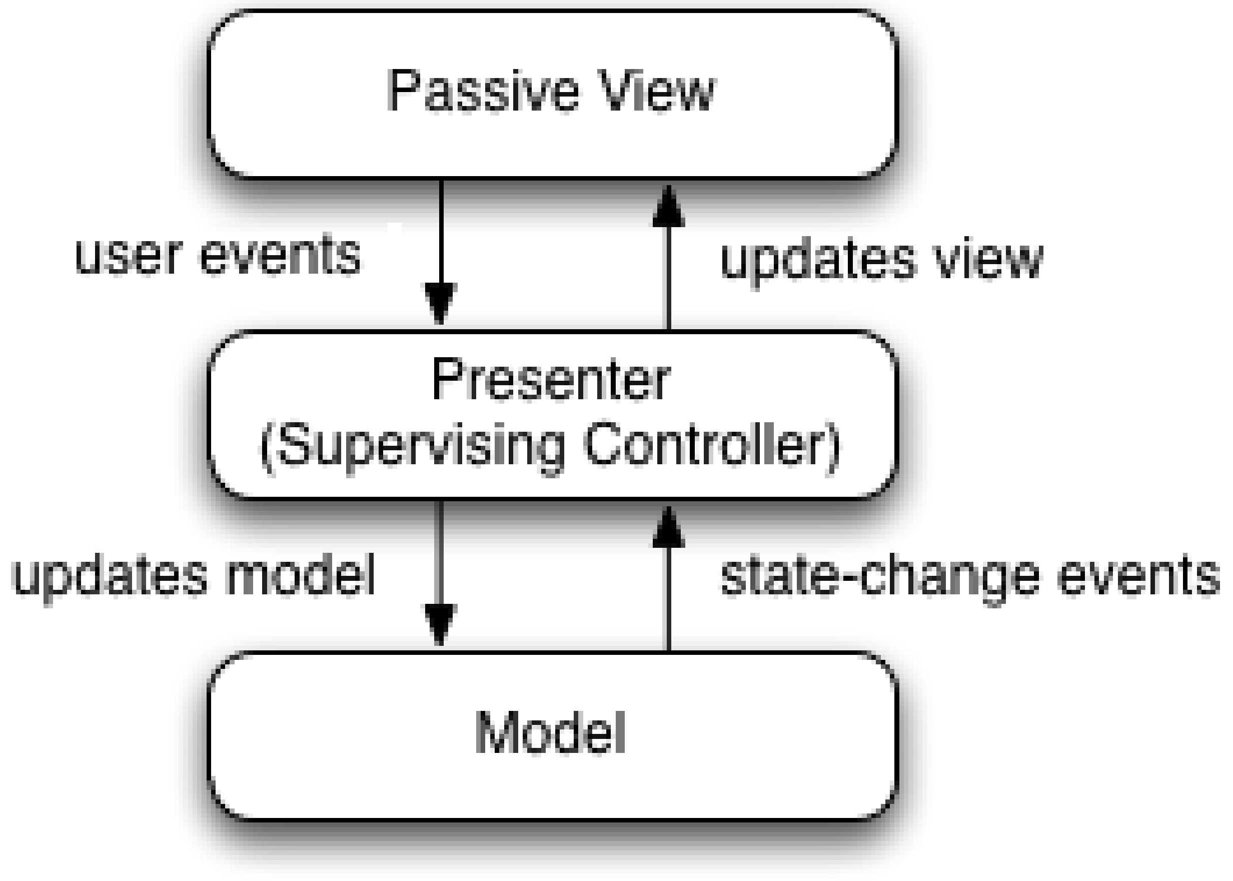
An Alternative Implementation of MVC

- Model updates view by going through the controller.
- Updates to view now come from only one place.
- Potentially less efficient for performing view updates.
- Creates looser coupling and makes it easier to isolate the model or the view (separation of concerns).
- Makes testing easier.



Model View Presenter

- It is a derivation of MVC.
- We rename the controller to be a presenter.
- Explicitly prevents the model from accessing the view.
- This is really the revised MVC that we had in the last slide.

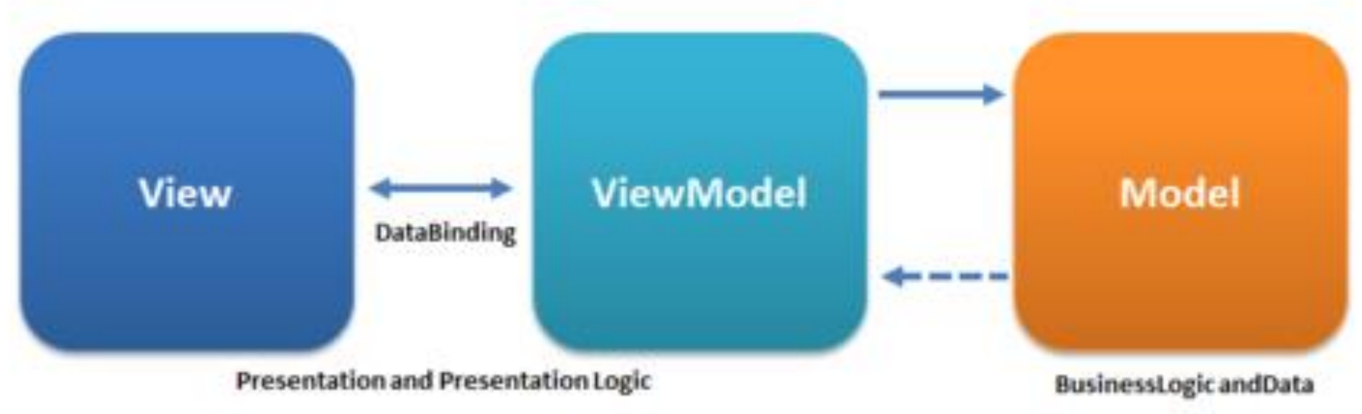


Notes

- While we think of “the” controller as a single entity, it is common to have multiple controllers (e.g., one for each specific view).
- Mixing MVC and MVP is easy, as we can replace some controllers with presenters.
- Many designs place domain logic (e.g., business rules) within the model, separating them from the programming logic of the presenter/controller.
- The Model is more than just a database. It can have data-structures, neural networks, decision trees, APIs, and other functional components.

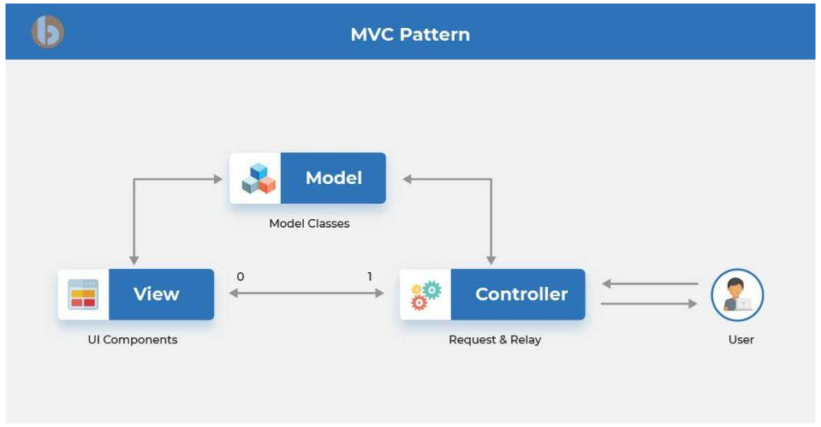
Model View ViewModel

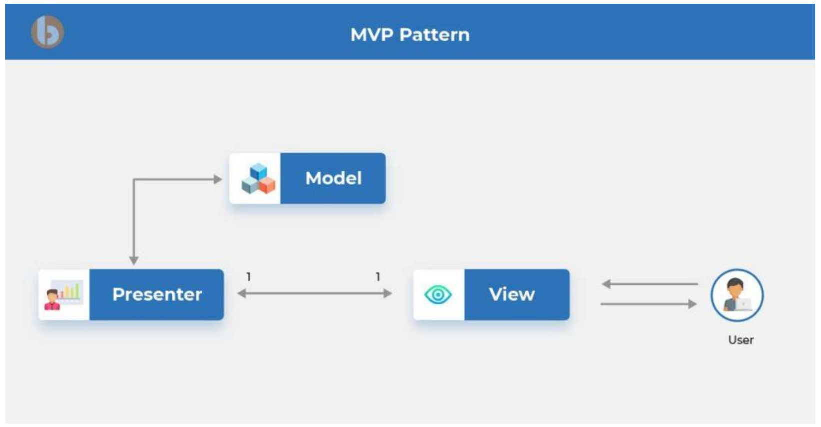
- Sometimes called Model View Binder
- The view is no longer considered an active entity allowing it to be implemented in a passive manner (e.g., using XML) – this approach helps UI/UX developers.
- The ViewModel is bound to the view and becomes the active part of the view. It contains all the display logic.
- Binding is done via properties and event callbacks.
- The controller is merged into the model.
- Intended to simplify event-driven programming.
- Considered inefficient for very large applications.

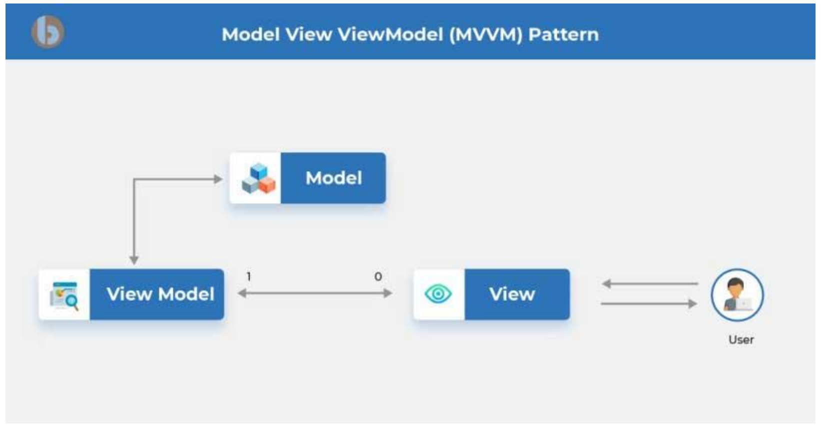


MVC and MVVM

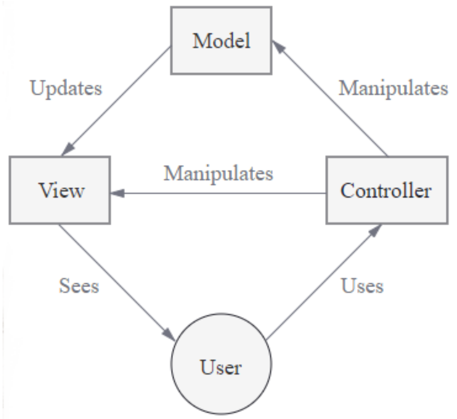
- With MVC, we can view the model as the back-end, and the controller as the client logic (the application’s code).
- Good for structuring systems with a significant back-end.
- Allows app and back-end to be build simultaneously by different teams.
- MVVM is designed for smaller applications, which is why Android treats an application as a collection of activities.
- Android also wants to avoid complex models (e.g., needing SQL) and MVVM does this by reducing the model’s significance.
- In MVVM much of the “Model” is replaced with interfaces to APIs.







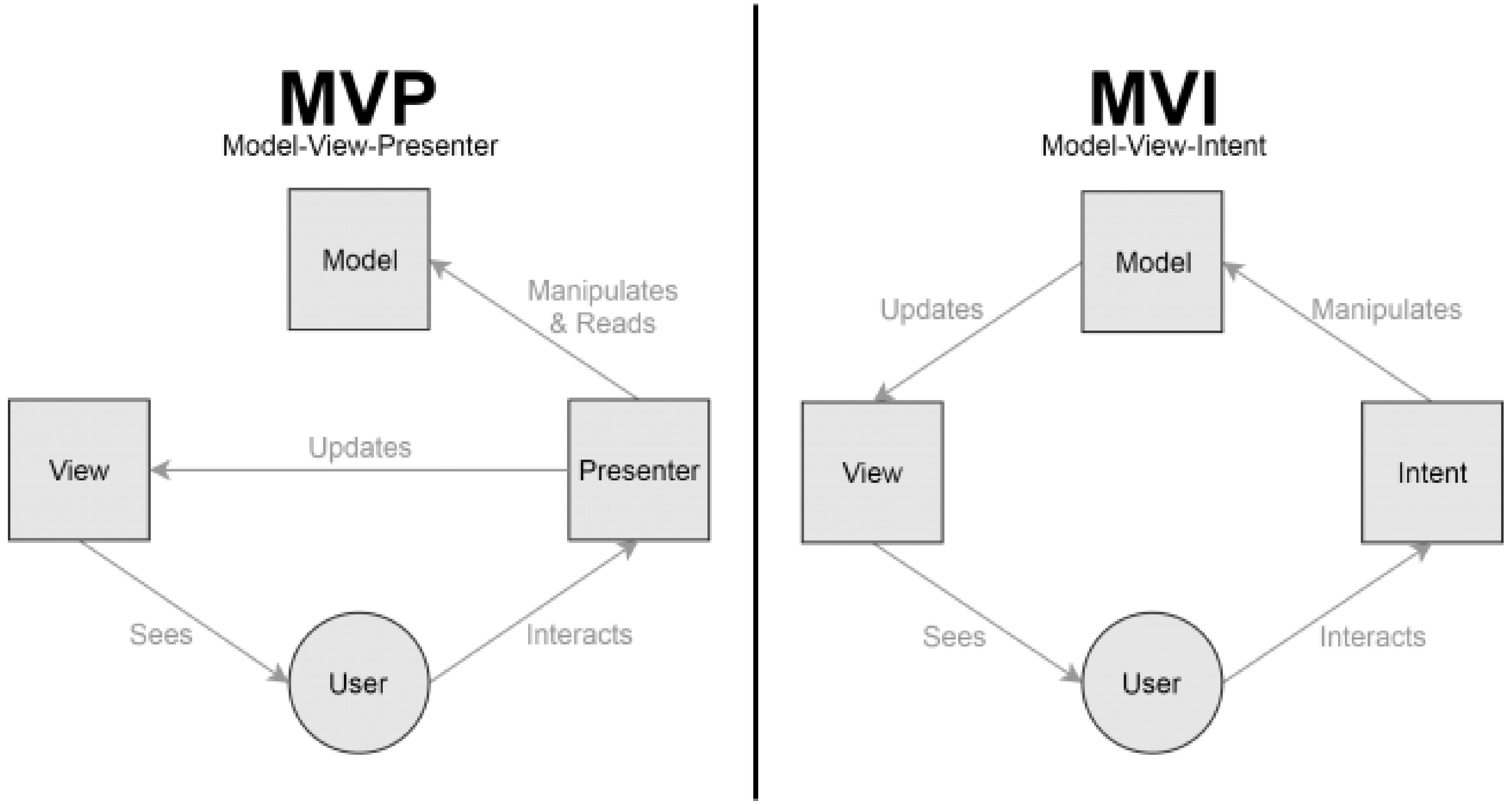
Note: I don't agree fully with the MVC illustration. Users see the view, not the controller.



Comparison of Interaction Patterns

Idea is to create a cyclical methodology for taking user input and producing (UI) output.

Model View Intent



MVI: A Cyclic Approach

What we want is:

```
view( model( intent() ) )
```

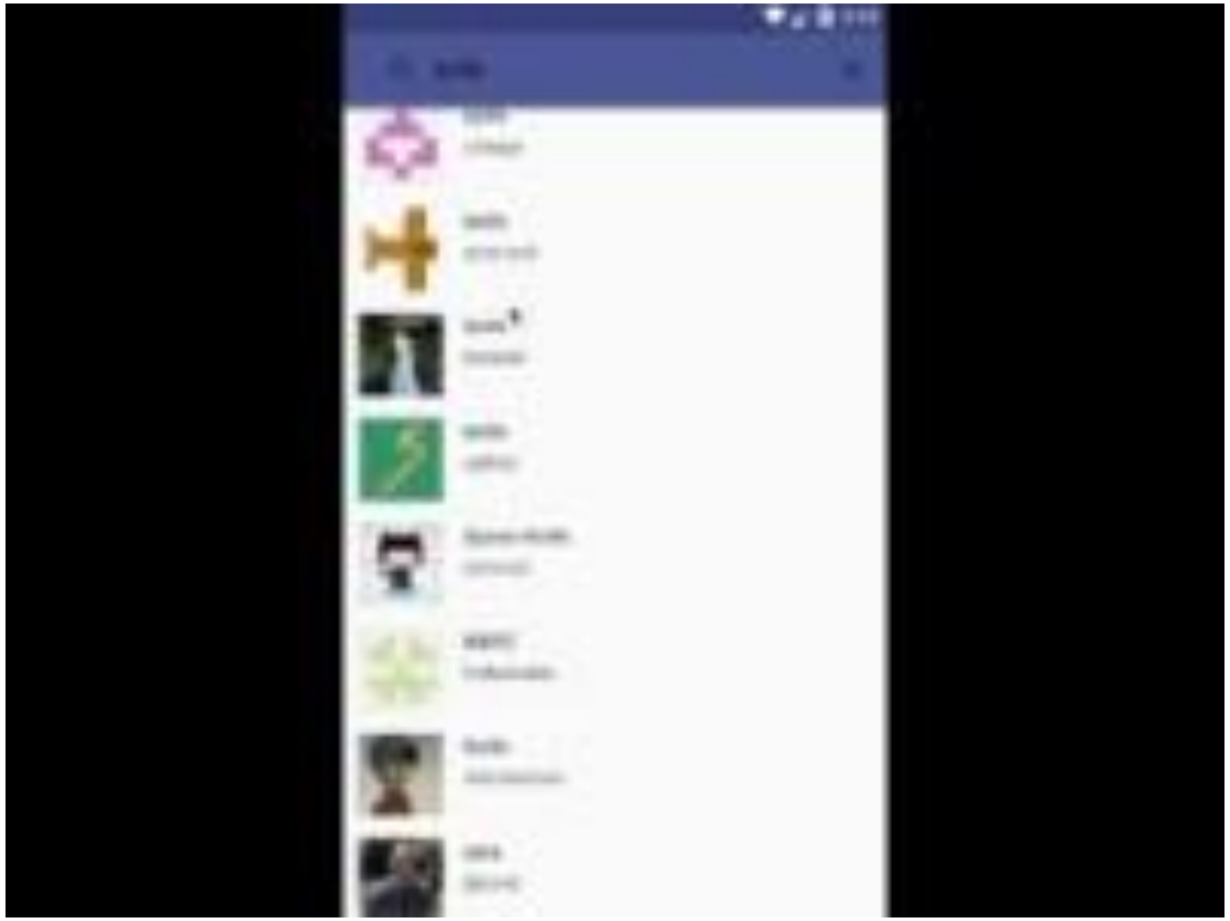
Where:

- `Intent()`: Takes input from the user and creates an object (i.e., encapsulating the user's intent) to pass to the model.
- `Model()`: Takes an intent object and performs computation to create new data. Does all the work and access all the APIs and DBs.
- `View()`: Takes the new data and displays it.

The model copies itself and manipulates the copy, thus causing the old model to be preserved and making it immutable.

Good example on: http://hannesdorfmann.com/android/model-view-intent/

MVI: A Cyclic Approach



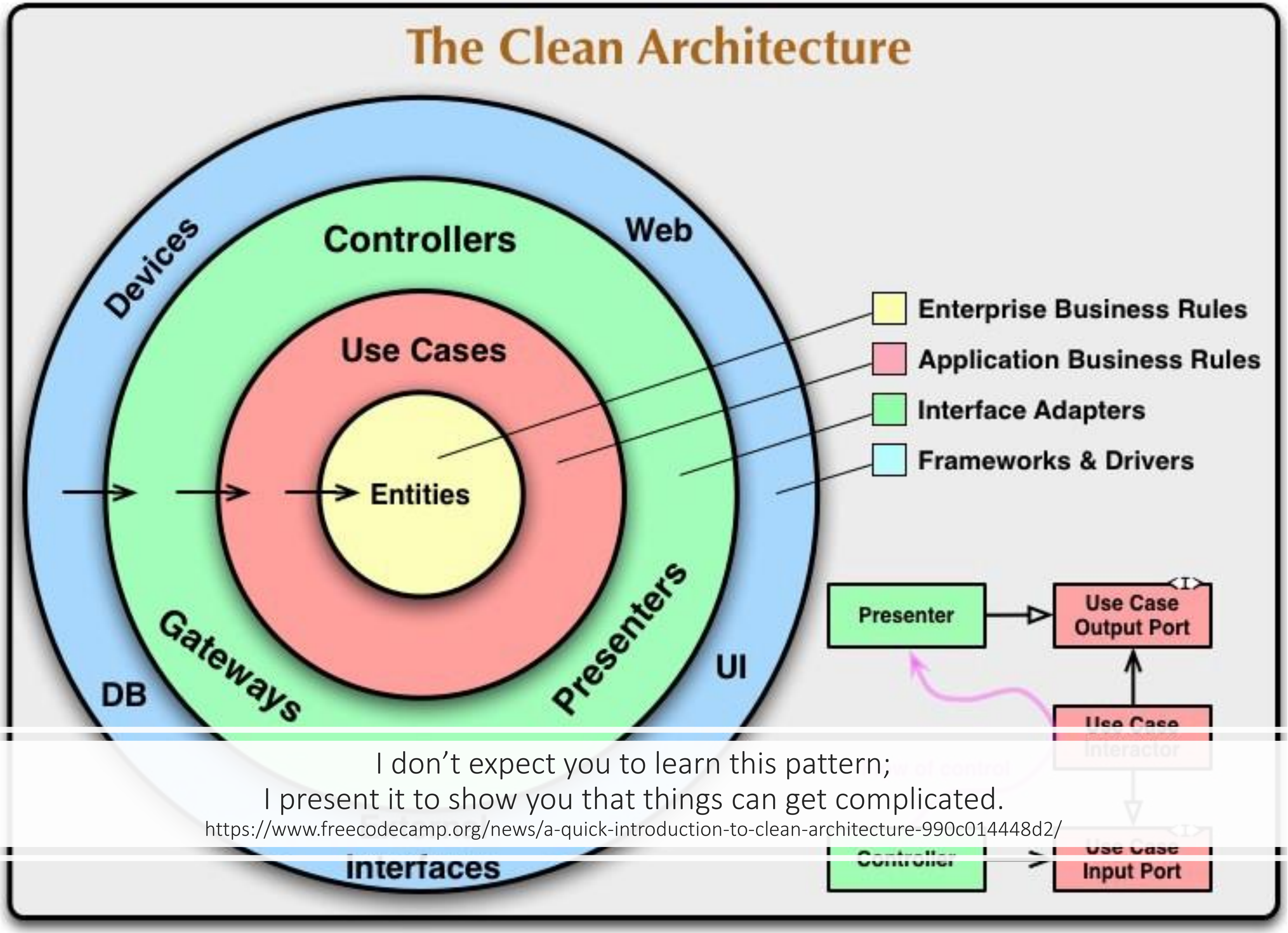
Chained Methods in Kotlin

```
Observable<String> = RxSearchView.queryTextChangeEvents(editSearch)
            .filter { it.queryText().count() >= 3 }
            .debounce(500, TimeUnit.MILLISECONDS)
```

Is the same as …

```
result1 = RxSearchView.queryTextChangeEvents(editSearch)
result2 = result1.filter { it.queryText().count() >= 3 }
Observable<String> = result2.debounce(500, TimeUnit.MILLISECONDS)
```

To chain methods, we have to ensure that each method (except the last) returns an instance of its class.
The syntax saves us from creating temporary variables that are used once only.



I don’t expect you to learn this pattern;
I present it to show you that things can get complicated.
https://www.freecodecamp.org/news/a-quick-introduction-to-clean-architecture-990c014448d2/

Which Architecture?

- Sometimes we don’t get a choice.
  - Android intends you to use MVVM.
  - Some companies use one exclusively.
- Sometimes the situation strongly suggests the use of one.
  - When the back-end is considerable, MVC/MVP helps with simultaneous development.
  - Applications with a lot of highly structured data need a strong model.

No choice is wrong,

but some are better.

Each has advantages and disadvantages.

Architecture

- Software architecture is a complex topic that senior (experienced) developers need to understand.
- Google/Android tries to enable inexperienced developers by making everything work with MVVM and expecting you to use it.
- Most architecture for mobile development is based on concepts developed for web development, including the architectures.
- There are very few independent “mobile applications.” Most apps are members of a complex system with DBs, external services (APIs), etc. The system and the app may have their own architectures.