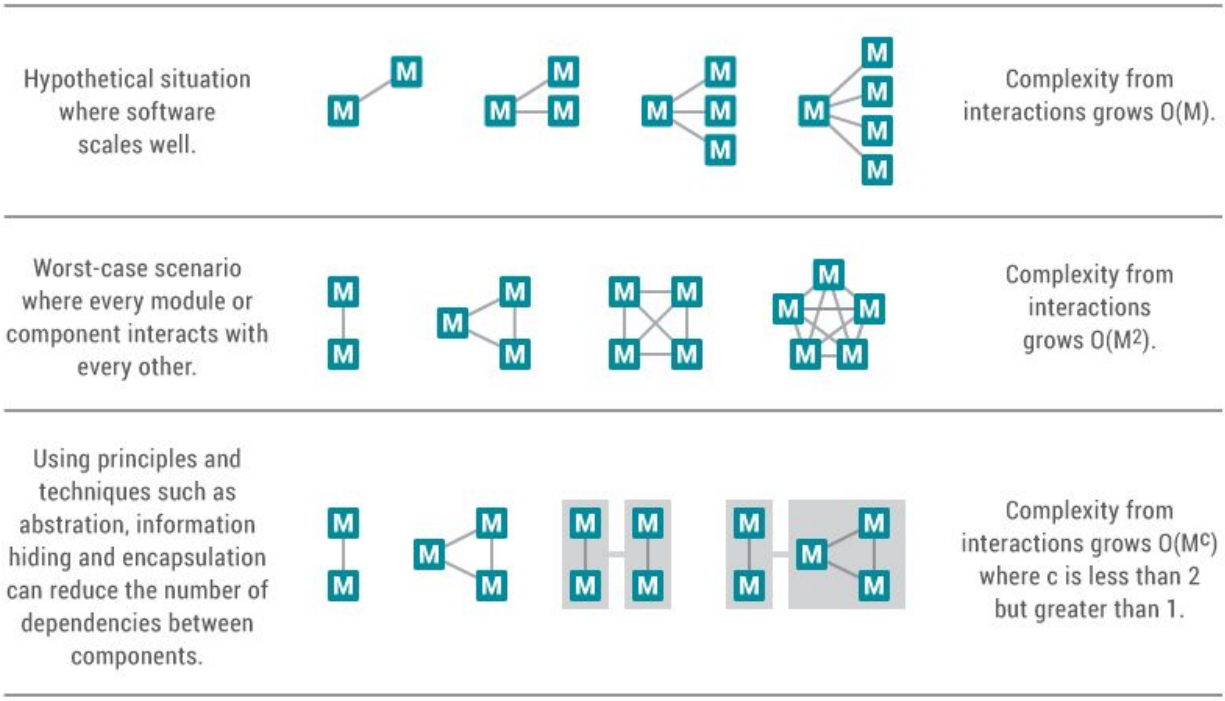
Mobile Software Architectures

Adam Geber, PhD
Computer Science @ UChicago

Complexity

Software systems grow in complexity exponentially. Anyone who has taken an introductory algorithms class will immediately understand the figure below. The theoretical best-case scenario for the growth in complexity of a software system is O(M) where M is the number of modules. At the most atomic level, a module may be considered a source file or a configuration file. The best-case linear-growth scenario O(M) is a theoretical minimum, and in practical terms can never be achieved due the inevitability of coupling. The worst-case exponential-growth scenario of $O(M^2)$ is likewise a theoretical maximum where every module is coupled to every other module. (See figure below). Thus, the rate of growth in complexity of any single software system can be expressed as $O(M^c)$ where the exponent c is a floating point number greater than 1 and less than 2.

[Figure 27 showing how software systems grow in complexity[1]]



From a software engineering standpoint, the best way to reduce M (number of Modules) is to follow sound principles of object oriented software engineering which promote code reuse, including: inheritance, composition, polymorphism, and programming to interfaces. The most effective way to reduce (c) is to compartmentalize the software modules into layers, which is the goal of any good architecture.

The choice of architecture will play a determinant role in mitigating complexity. Architectures that provide the following: (1) good separation of concerns (2) well defined interfaces between layers, and (3) exclusively inward-pointing dependencies are less complex than those that do not deliver these benefits.

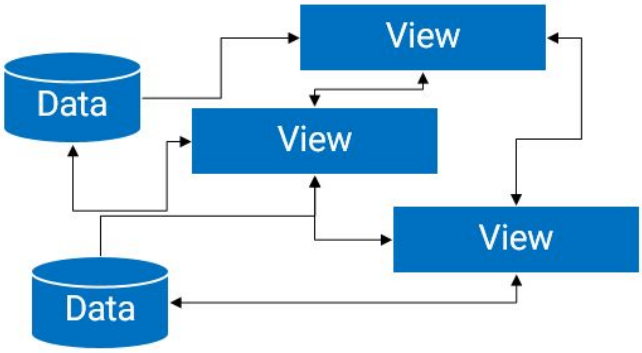
Let's review the following four architectures: MV (Model-View), MVC (Model-View-Controller), MVVM (Model-View-ViewModel), and MVP (Model-View-Presenter)[2].

Model-View or MV consists of two layers, Model and View. MV works well for simple applications. Indeed, most of the apps one finds in online articles or in reference projects on github use the MV architecture because MV is so straightforward, rapid to develop (initially), and easy to understand. MV provides little separation between the Model and the View. The

[1] http://sce2.umkc.edu/BIT/burrise/pl/introduction/

[2] https://www.linkedin.com/pulse/difference-between-mvc-mvp-mvvm-swapneel-salunkhe

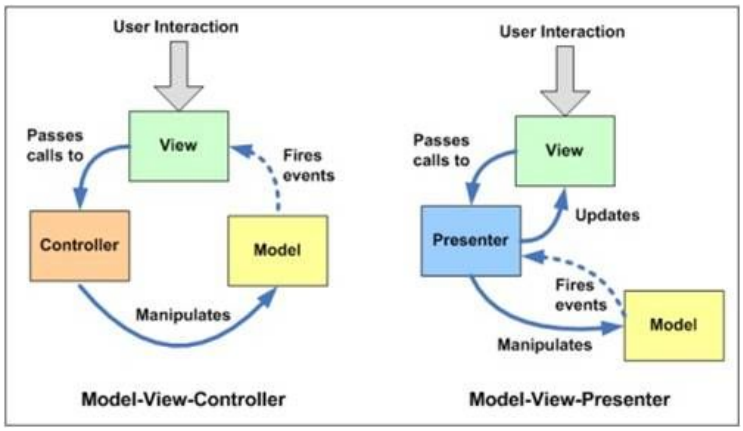
dependencies in MV flow in both directions so that the Model has dependencies in the View and vice versa. This cross-dependency is known as coupling and it renders the code-base inflexible in the long-term. As the application becomes more complex, changes to either the View or the Model will require corresponding changes to the other.



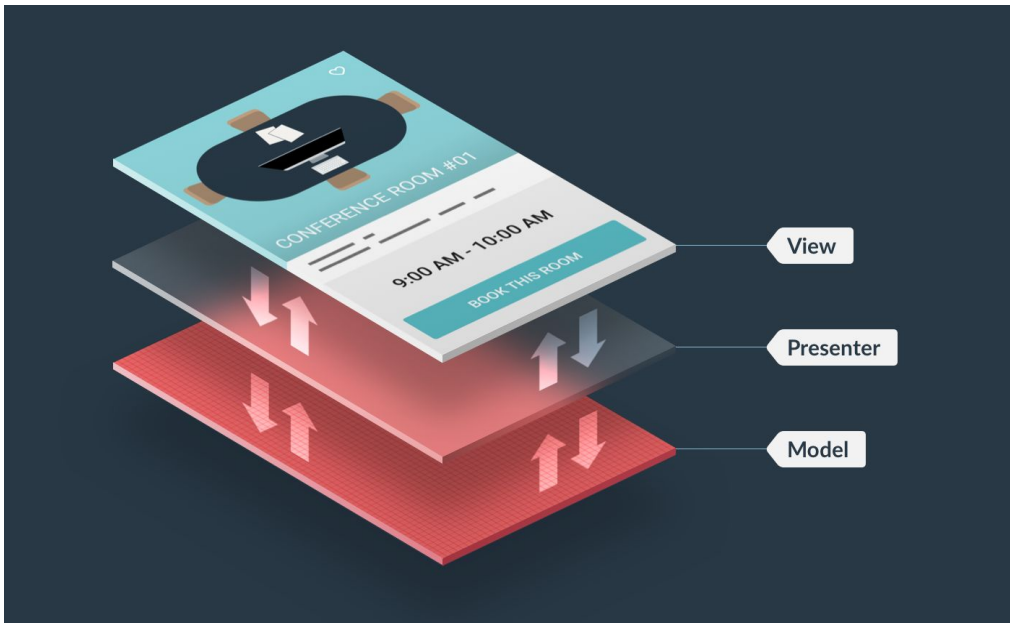
[Figure 1 showing the MV architectural pattern]

MVC consists of three layers, Model, View, and Controller. While MVC provides moderate separation of concerns, the Model and View may still communicate directly which increases coupling. If the software environment provides support for MVC, it is often a great choice. For example, Spring uses MVC as does Ruby-on-Rails. MVC provides good support for test-driven development (TDD) and is often used in Web applications where rapid application development (RAD) is paramount.

MVP consists of three layers, Model, View, and Presenter. The principal difference between MVP and MVC is that in MVP, the Model never touches the View directly. Rather, all communication between the View and the Model is routed through a Presenter. One big advantage of MVP is its use of interfaces at the outward-facing frontiers between layers. With MVP, unit testing is easier because interfaces can be easily stubbed-out. Applications developed with MVP require more architectural planning, but they tend to be more flexible and maintainable than apps developed with less extensible architectural patterns.

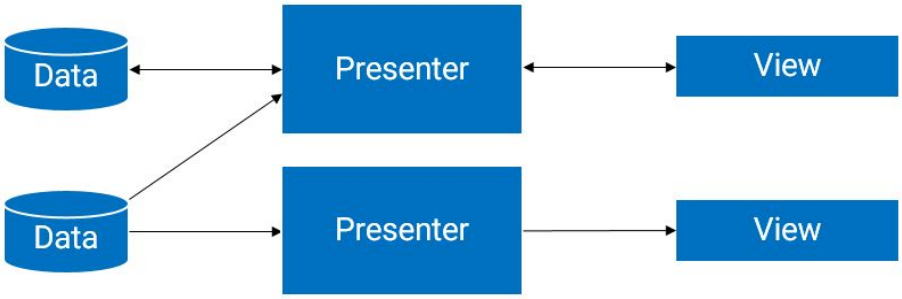


[Figure 2 showing the MVC and MVP architectural patterns[3]]



[Figure 3 showing the MVP architectural pattern]

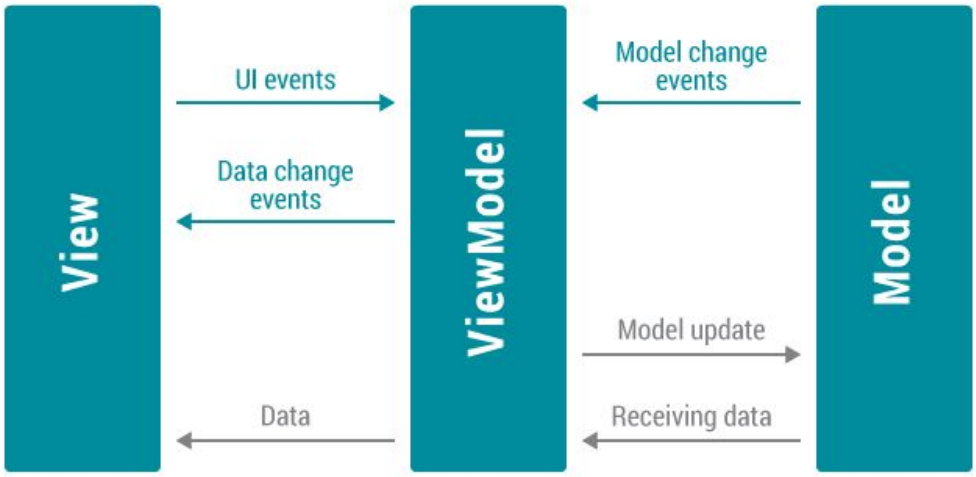
[3] https://www.linkedin.com/pulse/difference-between-mvc-mvp-mvvm-swapneel-salunkhe



[Figure 4 showing the MVP architectural pattern with the one-to-one relationship between Presenter and View]

MVVM is comprised of three layers, Model, ViewModel, and View. MVVM is used widely for Windows Presentation Foundation, Windows phone, Silverlight, and Javascript frameworks such as Knockout. As with MVP, MVVM provides clear separation of concerns between the View and the Model. The MVVM architecture is ideal for situations where two-way data-binding is built into the framework because MVVM requires less code to develop a full-featured application. Also, due to the two-way data binding, MVVM is perfect for apps that require real time up-to-date data like stock ticker apps.

In Android, despite the availability of two-way data binding components such as CursorLoader (local database) and cloud services such as Firebase (two-way bound cloud database managed by Google), no reliable enterprise-strength, two-way data binding standard has emerged, as it has, for example, with the javascript frameworks.



[Figure 5 showing the MVVM architectural pattern with two-way data-binding]

	RAD - Easy to Learn	Framework_ Opinionated	Data Binding	Easy to Test	Good SoC
MV	HIGH	LOW	N/A	LOW	LOW
MVC	MED	HIGH	MED	MED	MED
MVVM	MED	HIGH	HIGH	MED	HIGH
MVP	LOW	LOW	N/A	HIGH	HIGH

[Figure 6 comparing Architectures. RAD stands for Rapid Application Development, and SoC stands for Separation of Concerns]

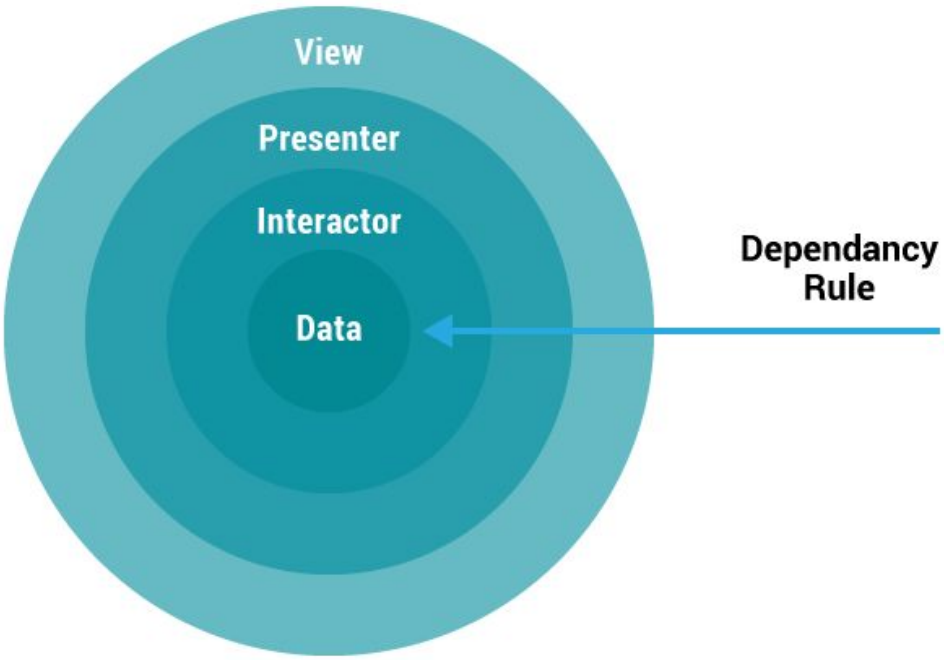
Choosing the appropriate mobile architecture will depend on two principal factors: (1) the anticipated complexity of the application, and (2) the need for real-time data. Let's discuss both in turn.

(1) As there is a high correlation between complexity and coupling, simple apps mitigate in favor of simple architectures like MV which are easy to learn and rapid (initially) to implement, while complex apps mitigate in favor of robust architectures like MVVM and MVP. (2) Business requirements will play a big role in determining the appropriate architecture. Consider ride-sharing applications like Uber which display and update the locations of nearby Uber cabs in real-time. If the business rules require real-time data, then a two-way data-binding architecture like MVVM is preferable. If real-time data is not a business requirement, then using an architecture like MVP is preferable for reasons we shall discuss below.

The Dependency Rule

The dependency rule is widely agreed among software architects to be the cornerstone of any clean architecture[4]. As one moves into the software sphere from the View towards the Data core, each successive layer must be ignorant of the previous layer and the dependencies point inward. For example, the Presenter layer has a dependency on the Interactor layer, but the Interactor is completely ignorant of the Presenter. To pass data out towards the periphery, the Interactor uses interfaces. The same is true of the relationship between Data and Interactor. See Figure 7 below.

[4] https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html



[Figure 7 showing the Dependency Rule[5]]

MVPI

Figure 7 above uses four concentric rings, whereas all the architectures we've discussed thus far have only two or three layers. The addition of a fourth layer, called the Interactor, allows for even greater separation of concerns than the three-tier architectures already discussed. From here on out, we will simply refer to this architecture as MVPI whose acronym stands for Model, View, Presenter, Interactor. Let's describe each layer of MVPI starting the data layer and moving outward in the reverse order of dependencies.

Data

The data layer is where the models reside. In addition, the data layer is where the interfaces to external repositories are defined. The external repositories contain business entities. The models are defined to be useful to the client, thus they may be supersets or subsets of the business entities. In some cases, there will be a simple one-to-one mapping between the fields in the entities, and the fields in the models. In other cases, especially when differentiated back-end systems are used, the models need not mirror the business entities exactly, rather they should be defined as a superset to accommodate the full range of closely related entities anticipated.

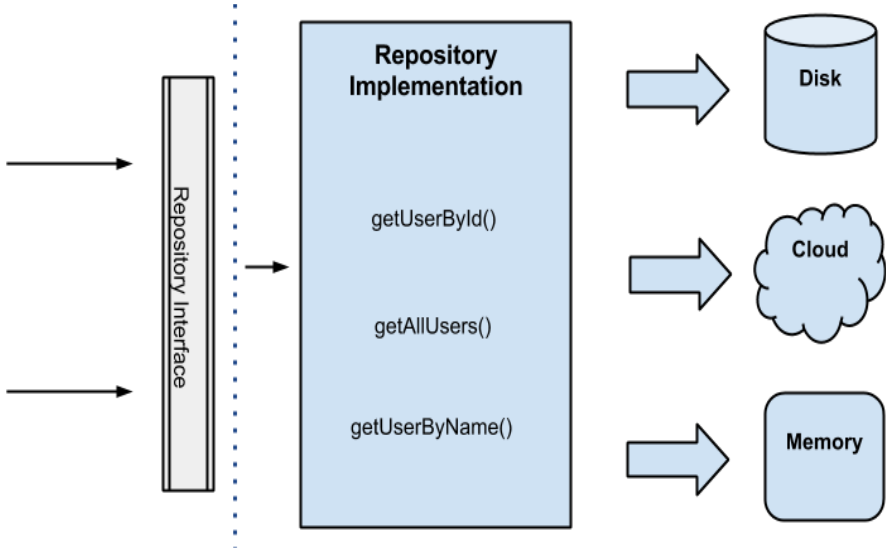
---

[5] https://github.com/android10/Android-CleanArchitecture

Aside from a dependency on the App itself and its configurations, the Data layer has no dependencies whatsoever. It's important that the entities be hidden through well-defined interfaces (See Figure 9 below). These interfaces define the interface methods of the repositories, but leave the implementation details to the repositories themselves. Since, the use cases in the Interactor layer are completely indifferent to the implementation details of the repositories and their entities, changing repository logic or even swapping-out an entire repository is trivial.

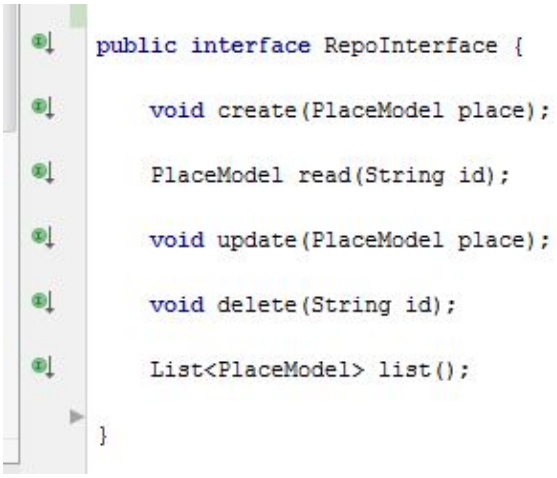
In the proFinal [base code reference] app, we have chosen to store entities in local device databases, though in an enterprise app, these repositories should be remote and their implementation logic completely hidden from the developers. So long as the contract methods of the interfaces are satisfied, the repositories are of no concern to the mobile developers.

In the case of a multinational corporation, differences across markets, such as differing payment systems, can be resolved by defining models and interfaces that accommodate any possible payment resolution system. The country configuration can be sent along with the version, and request that the middleware marshall entities from/to the appropriate payment systems.



[Figure 8 showing the boundary interface to the repositories[6]]

[6] http://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/



```
public interface RepoInterface {

    void create(PlaceModel place);

    PlaceModel read(String id);

    void update(PlaceModel place);

    void delete(String id);

    List<PlaceModel> list();

}
```

[Figure 9 showing the interface to the Repository. The repository implementation details are completely hidden from the App developer]

Interactor

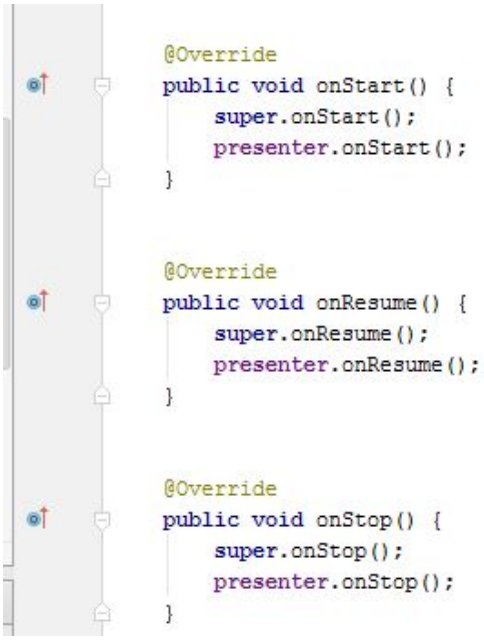
The Interactor is the layer where the business logic resides. Any query strings are built in this layer as well. Business logic is alternatively called a use case. A use case may be defined as any operation that requires a Create, Read, Update, or Delete (CRUD) on a business entity or entities. If the user applies an operation that does not require a CRUD operation on the business entities, then it is not truly business logic. Rather, these are controlling logic which should be promoted to the Presenter discussed next.

The Interactor acts as a buffer between the Presenter and the Data layers. The Presenter shuttles data operations through the use cases. Use cases can be owned by the business analysts and because they reside in one place, they are easily updated. In addition to providing additional separation of concerns, use cases promote code reuse and facilitate testing. For example, despite differences among implementation languages, use cases could very well be repurposed to work across platforms such as Web, Android, and iOS. A use case has built-in support for backgrounding long-running tasks as required.

Presenter

The Presenter provides a buffer between the Interactor and the View. The Presenter is where all the controlling logic resides. The life-cycle of the Presenter is bound to the life-cycle of the View (see Figure 10 below). This means that the Presenter is highly responsive to OS and navigation events. For example, if a view needs to be refreshed upon onResume(), the presenter's matching onResume() callback can be invoked, which in turn, executes a UseCase in the Interactor which in turn, fetches an updated model from the data layer. Likewise, if navigating to, or away from, a View requires that a resource connection be established or closed, the presenter's onStart() and onStop() callbacks can be invoked to perform these tasks.

In MVPI, there is a 1-to-1 relationship between a View and its Presenter (see Figure 4 above) so that each View has its own Presenter. The Presenter layer is the most powerful of the layers because it manages navigation controls and is responsive to OS events and potential changes to its environment.



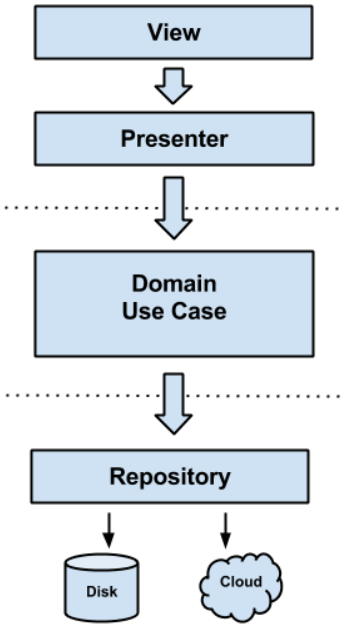
[Figure 10 showing how BaseView is delegating lifecycle callbacks to its presenter]

View

The View is concerned principally with (1) displaying itself and (2) registering interactive components such as buttons and textfields to be responsive to user events. As with most architectures, the View may take advantage of existing mechanisms in Android and iOS to register event sources (such as buttons) with corresponding event listeners. However, the View should not define any presentation logic or business logic; these should be placed in the Presenter and Interactor layers respectively.

The View has a dependency on its presenter, but the View knows nothing about its presenter's implementation details or the presentation logic embedded therein. Furthermore, the View is completely isolated from--and has no dependencies on--the use cases that reside in the Interactor layer. Likewise, the View has no dependencies on the Data layer whatsoever.

A view always has a single presenter, though there may be multiple views for a single presenter. Due to their simplicity, views can be easily swapped out at runtime to suit requirements.



[Figure 11 showing the layers of a 4-tier architecture. Dotted lines are interface frontiers. Data is passed back up through interface frontiers[7]]

Summary of MVPI

When implementing MVPI, it's important that developers understand the distinction between controlling logic and business logic and place those logical units in the Presenter and Interactor layers respectively. To implement correctly, the MVPI architectural pattern requires some architectural forethought. Nevertheless, MVPI provides excellent separation of concerns, isolation of use cases, and a manageable number of layers. One of the advantages of MVPI is that business logic (use cases) may be created or modified without affecting any other layers. Furthermore, use cases may be mocked-out and tested rigorously by intervening at the interactor layer and without ever having to touch the Presenter or View layers; and this is ideal for automated testing frameworks.

[7] https://github.com/android10/Android-CleanArchitecture

MVPI Advantages
Provides excellent separation of concerns
Promotes code reuse
Maintainable / Extensible
Facilitates testing
MVPI Disadvantages
Requires some architectural forethought
Extra learning curve for developers
Redundant in simple apps

[Figure 12 showing Advantages and Disadvantages of MVPI]