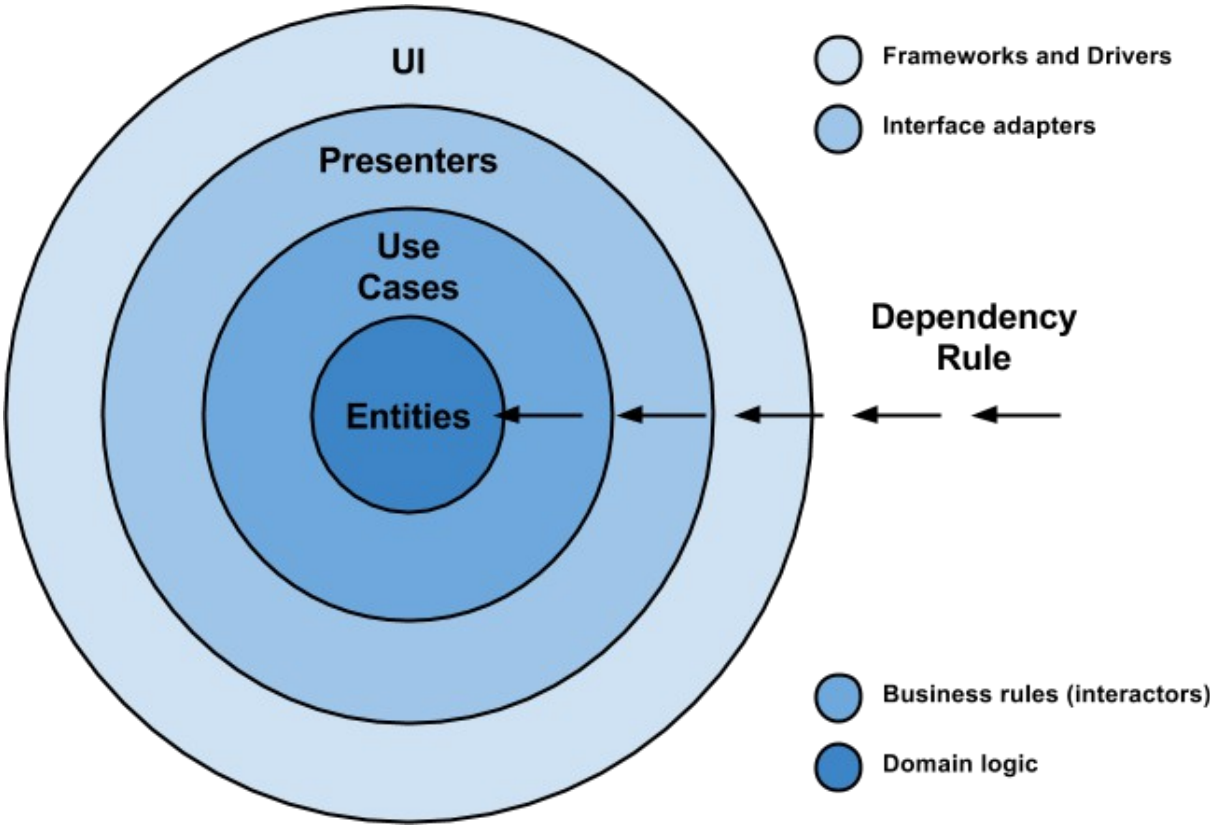
The idea is simple: clean architecture stands for a group of practices that produce systems that are:

Independent of Frameworks.
Testable.
Independent of UI.
Independent of Database.
Independent of any external agency.



It is not a must to use only 4 circles (as you can see in the picture), because they are only schematic but you should take into consideration the Dependency Rule: source code dependencies can only point inwards and nothing in an inner circle can know anything at all about something in an outer circle.

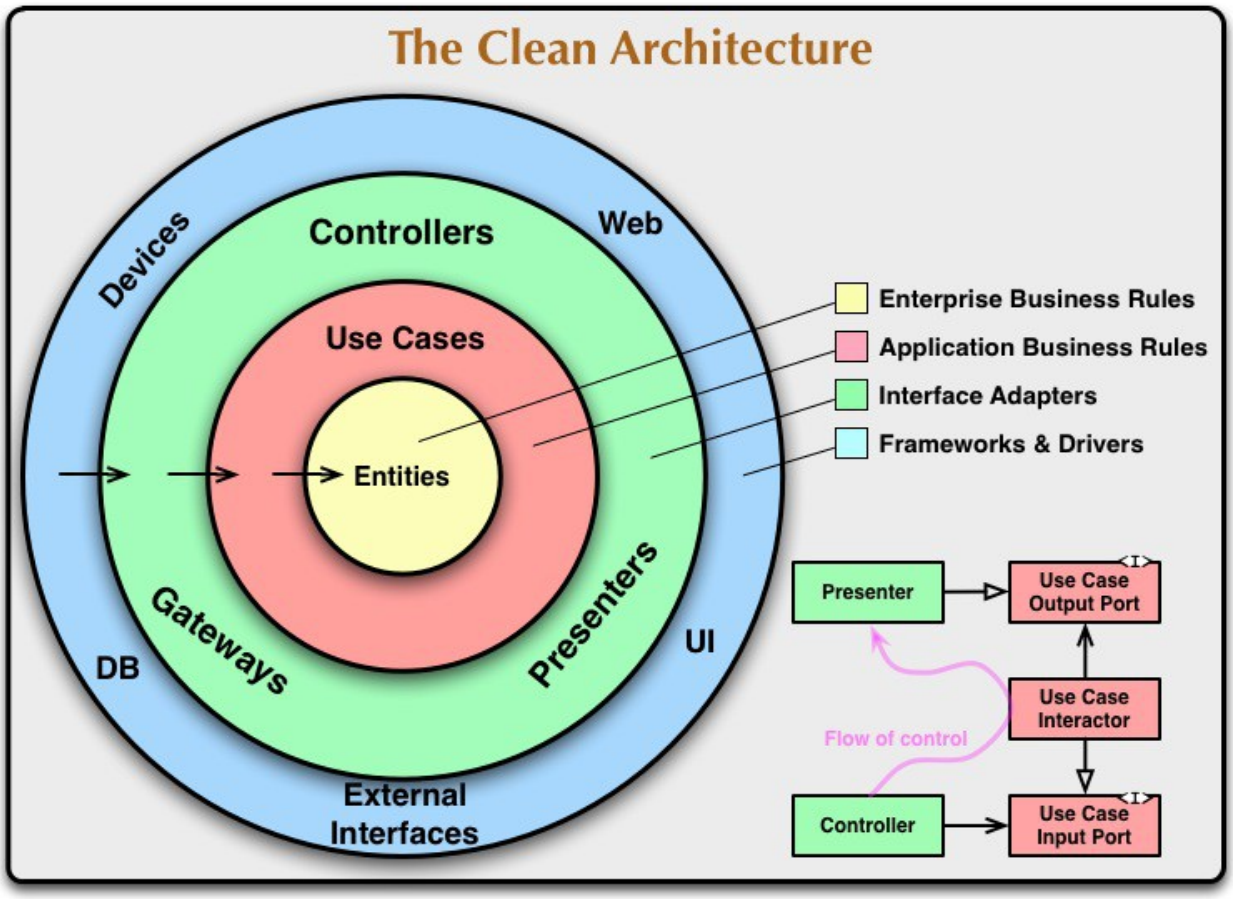
The overriding rule that makes this architecture work is The Dependency Rule. This rule says that source code dependencies can only point inwards- Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the an inner circle. That includes, functions, classes. variables, or any other named software entity.

By the same token, data formats used in an outer circle should not be used by an inner circle, especially if those formats are generate by a framework in an outer circle. We don't want anything in an outer circle to impact the inner circles.

Entities:
These are the business objects of the application.

Use Cases:
These use cases orchestrate the flow of data to and from the entities. Are also called Interactors.

Interface Adapters: This set of adapters convert data from the format most convenient for the use cases and entities. Presenters and Controllers belong here. Frameworks and Drivers: This is where all the details go: UI, tools, frameworks, etc.



Entities :

Entities encapsulate Enterprise wide business rules. An entity can be an object with methods, or it can be a set of data structures and functions. It doesn't matter so long as the entities could be used by many different applications in the enterprise.

If you don't have an enterprise, and are just writing a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes. For example, you would not expect these objects to be affected by a change to page navigation, or security. No operational change to any particular application should affect the entity layer.

Use Cases :

The software in this layer contains application specific business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their enterprise wide business rules to achieve the goals of the use case.

We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI, or any of the common frameworks. This layer is isolated from such concerns.

We do, however, expect that changes to the operation of the application will affect the use-cases and therefore the software in this layer. If the details of a use-case change, then some code in this layer will certainly be affected.

Interface Adapters :

The software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the Database or the Web. It is this layer, for example, that will wholly contain the MVC architecture of a GUI. The Presenters, Views, and Controllers all belong in here. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views.

Similarly, data is converted, in this layer, from the form most convenient for entities and use cases, into the form most convenient for whatever persistence framework is being used. i.e. The Database. No code inward of this circle should know anything at all about the database. If the database is a SQL database, then all the SQL should be restricted to this layer, and in particular to the parts of this layer that have to do with the database.

Also in this layer is any other adapter necessary to convert data from some external form, such as an external service, to the internal form used by the use cases and entities.

Frameworks and Drivers :

The outermost layer is generally composed of frameworks and tools such as the Database, the Web Framework, etc. Generally you don’t write much code in this layer other than glue code that communicates to the next circle inwards.

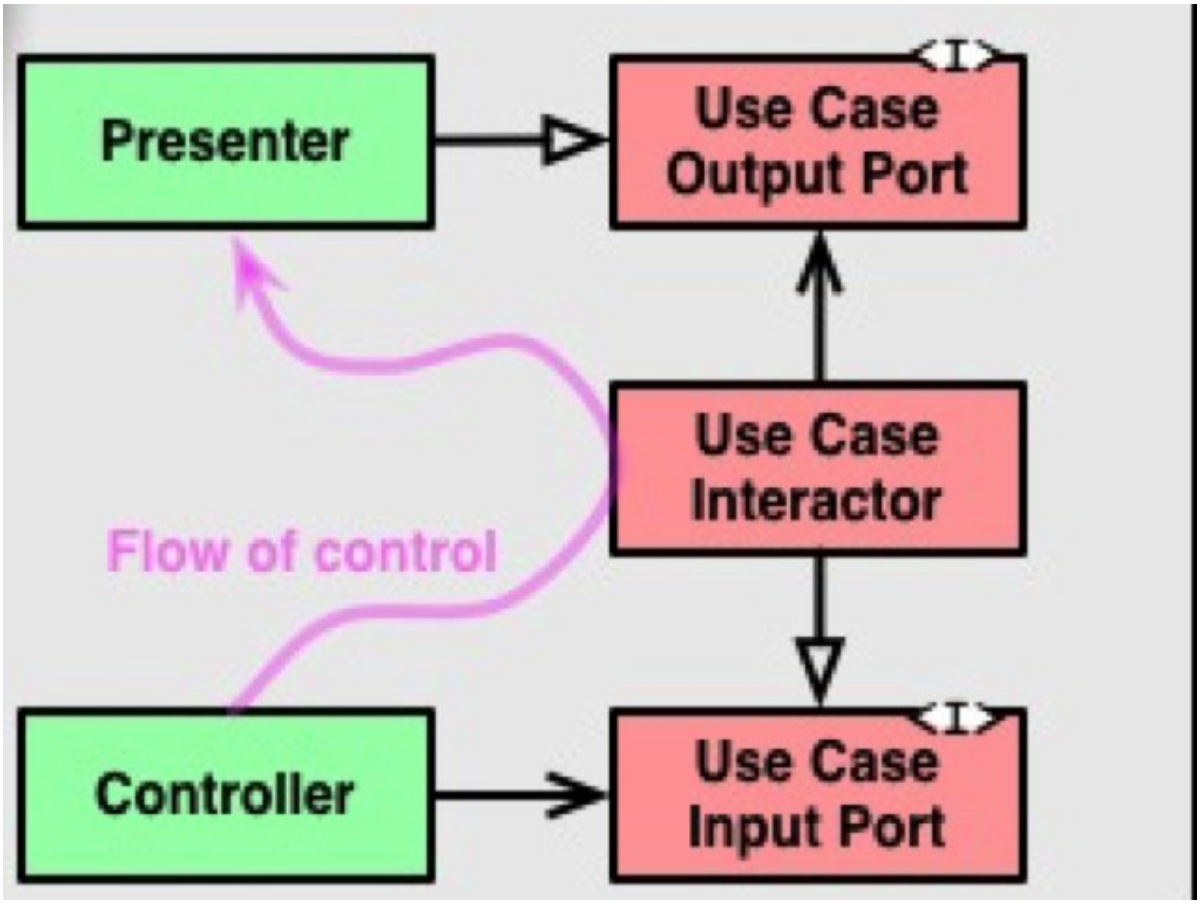
This layer is where all the details go. The Web is a detail. The database is a detail. We keep these things on the outside where they can do little harm.

Only Four Circles?

No, the circles are schematic. You may find that you need more than just these four. There's no rule that says you must always have just these four. However, The Dependency Rule always applies. Source code dependencies always point inwards. As you move inwards the level of abstraction increases. The outermost circle is low level concrete detail. As you move inwards the software grows more abstract, and encapsulates higher level policies. The inner most circle is the most general.

Crossing boundaries :

At the lower right of the diagram is an example of how we cross the circle boundaries.



It shows the Controllers and Presenters communicating with the Use Cases in the next layer. Note the flow of control. It begins in the controller, moves through the use case, and then winds up executing in the presenter. Note also the source code dependencies. Each one of them points inwards towards the use cases.

We usually resolve this apparent contradiction by using the Dependency Inversion Principle. In a language like Java, for example, we would arrange interfaces and inheritance relationships such that the source code dependencies oppose the flow of control at just the right points across the boundary.

For example, consider that the use case needs to call the presenter. However, this call must not be direct because that would violate The Dependency Rule: No name in an outer circle can be mentioned by an inner circle. So we have the use case call an interface (Shown here as Use Case Output Port) in the inner circle, and have the presenter in the outer circle implement it.

The same technique is used to cross all the boundaries in the architectures. We take advantage of dynamic polymorphism to create source code dependencies that oppose the flow of control so that we can conform to The Dependency Rule no matter what direction the flow of control is going in.

What data crosses the boundaries :

Typically the data that crosses the boundaries is simple data structures. You can use basic structs or simple Data Transfer objects if you like. Or the data can simply be arguments in function calls. Or you can pack it into a hashmap, or construct it into an object. The important thing is that isolated, simple, data structures are passed across the boundaries. We don't want to cheat and pass Entities or Database rows. We don't want the data structures to have any kind of dependency that violates The Dependency Rule.

For example, many database frameworks return a convenient data format in response to a query. We might call this a RowStructure. We don't want to pass that row structure inwards across a boundary. That would violate The Dependency Rule because it would force an inner circle to know something about an outer circle.

So when we pass data across a boundary, it is always in the form that is most convenient for the inner circle.

Conclusion :

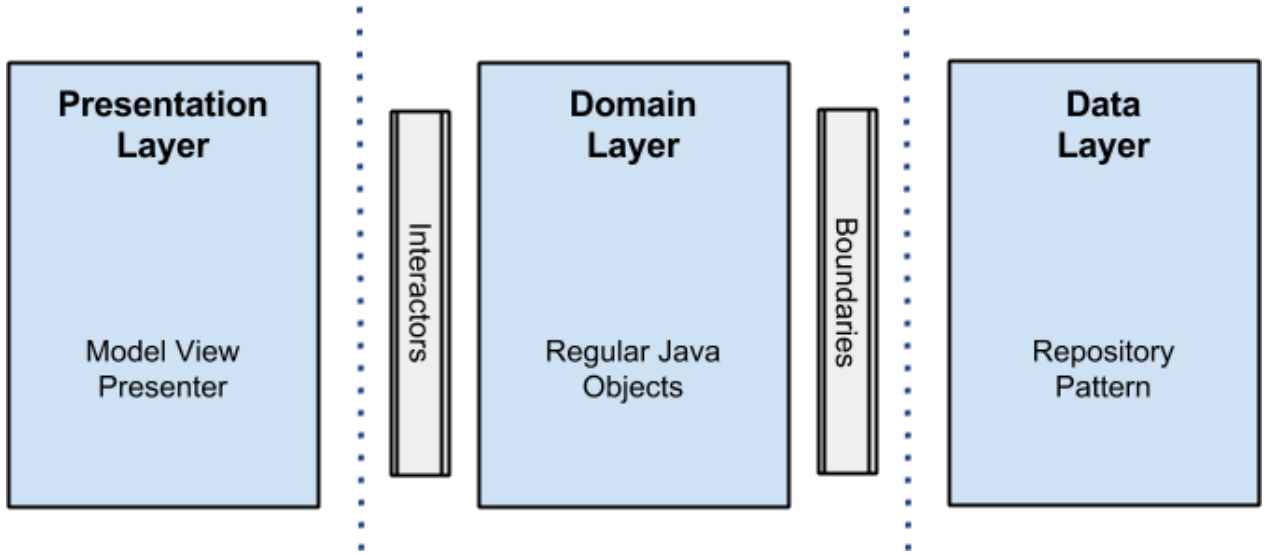
Conforming to these simple rules is not hard, and will save you a lot of headaches going forward. By separating the software into layers, and conforming to The Dependency Rule, you will create a system that is intrinsically testable, with all the benefits that implies. When any of the external parts of the system become obsolete, like the database, or the web framework, you can replace those obsolete elements with a minimum of fuss.

Android Architecture APP (CLEAN ARCHITECTURE APP)

To achieve this, my proposal is about breaking up the project into 3 different layers, in which each one has its own purpose and works separately from the others.

It is worth mentioning that each layer uses its own data model so this independence can be reached (you will see in code that a data mapper is needed in order to accomplish data transformation, a price to be paid if you do not want to cross the use of your models over the entire application).

Here is an schema so you can see how it looks like:



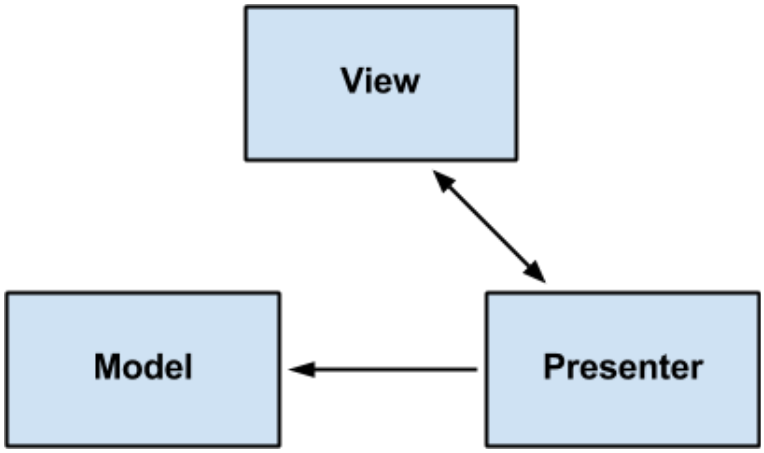
NOTE: I did not use any external library (except gson for parsing json data and junit, mockito, robolectric and espresso for testing). The reason is that it makes the example clearer. Anyway do not hesitate to add ORMs for storing disk data or any dependency injection framework or whatever tool or library you are familiar with, that could make your life easier.

REMEMBER: reinventing the wheel is not a good practice.

Presentation Layer

Is here, where the logic related with views and animations happens. It uses no more than a Model View Presenter (MVP from now on), but you can use any other pattern like MVC or MVVM. I will not get into details on it, but here **fragments and activities are only views**, there is no logic inside them other than UI logic, and this is where all the rendering stuff takes place.

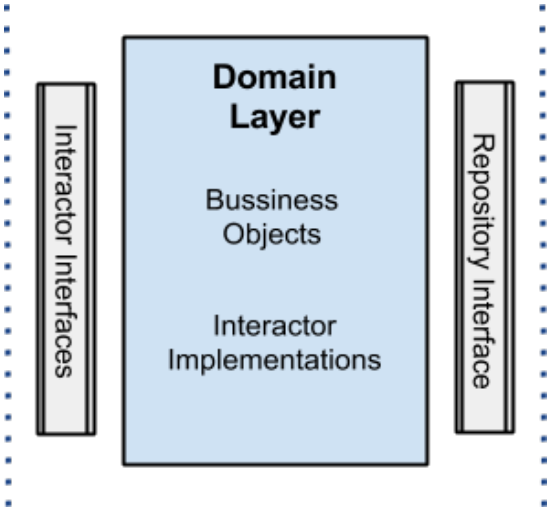
Presenters in this layer are composed with interactors (use cases) that perform the job in a new thread outside the main android UI thread, and come back using a callback with the data that will be rendered in the view.



Domain Layer

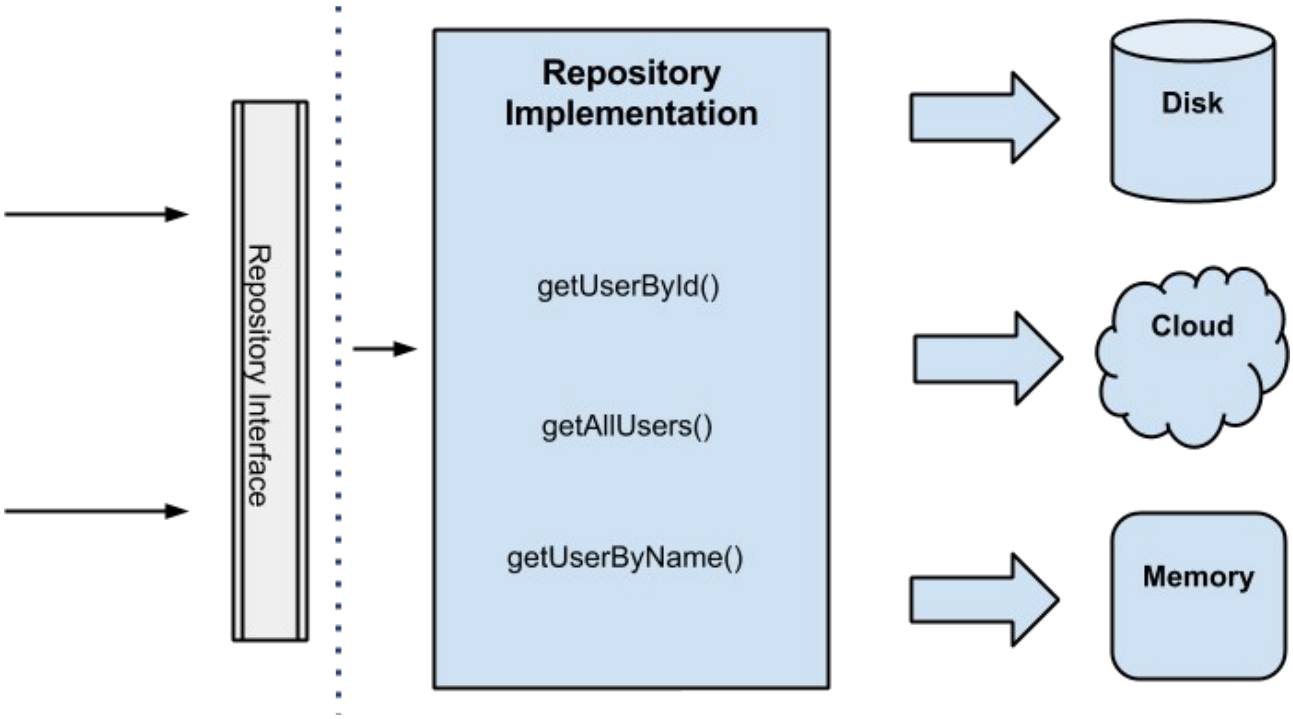
Business rules here: all the logic happens in this layer. Regarding the android project, you will see all the interactors (use cases) implementations here as well.

This layer is a pure java module without any android dependencies. All the external components use interfaces when connecting to the business objects.



Data Layer

All data needed for the application comes from this layer through a **UserRepository** implementation (the interface is in the domain layer) that uses a Repository Pattern with a strategy that, through a factory, picks different data sources depending on certain conditions.
For instance, when getting a user by id, the disk cache data source will be selected if the user already exists in cache, otherwise the cloud will be queried to retrieve the data and later save it to the disk cache.
The idea behind all this is that **the origin of the data is transparent to the client**, which does not care if the data is coming from memory, disk or the cloud, the only truth is that the information will arrive and will be gotten.



NOTE: In terms of code I have implemented a very simple and primitive disk cache using the file system and android preferences, it was for learning purpose. Remember again that you **SHOULD NOT REINVENT THE WHEEL** if there are existing libraries that perform these jobs in a better way.

Error Handling

This is always a topic for discussion and could be great if you share your solutions here.
My strategy was to use callbacks, thus, if something happens in the data repository for example, the callback has 2 methods `onResponse()` and `onError()`.
The last one encapsulates exceptions in a wrapper class called "**ErrorBundle**": This approach brings some difficulties because there is a chains of callbacks one after the other until the error goes to the presentation layer to be rendered. **Code readability could be a bit compromised.**
On the other side, I could have implemented an event bus system that throws events if something wrong happens, but this kind of solution is like using a GOTO, and, in my opinion, sometimes you can get lost when you're subscribed to several events if you do not control them closely.

Testing

Regarding testing, I opted for several solutions depending on the layer:

- **Presentation Layer:** used android instrumentation and espresso for integration and functional testing.
- **Domain Layer:** JUnit plus mockito for unit tests was used here.
- **Data Layer:** Robolectric (since this layer has android dependencies) plus junit plus mockito for integration and unit tests.

Show me the code

I know that you may be wondering where is the code, right? Well here is the github link where you will find what I have done.
About the folder structure, something to mention, is that the different layers are represented using modules:

- **presentation:** It is an android module that represents the presentation layer.
- **Domain:** A java module without android dependencies.
- **Data:** An android module from where all the data is retrieved.
- **Data-test:** Tests for the data layer. Due to some limitations when using Robolectric I had to use it in a separate java module.

Conclusion

As Uncle Bob says, "**Architecture is About Intent, not Frameworks**" and I totally agree with this statement. Of course there are a lot of different ways of doing things (different implementations) and I'm pretty sure that you (like me) face a lot of challenges every day, but by using this technique, you make sure that your application will be:

- **Easy to maintain.**
- **Easy to test.**
- **Very cohesive.**
- **Decoupled.**

Reactive approach: RxJava

I will point out what makes it interesting in regards of android applications development, and how it has helped me evolve my first approach of clean architecture.

First, I opted for a reactive pattern by converting use cases (called interactors in the clean architecture naming convention) to return `Observables<T>` which means all the lower layers will follow the chain and return `Observables<T>` too.

```
public abstract class UseCase {

  private final ThreadExecutor threadExecutor;
  private final PostExecutionThread postExecutionThread;

  private Subscription subscription = Subscriptions.empty();

  protected UseCase(ThreadExecutor threadExecutor,
      PostExecutionThread postExecutionThread) {
    this.threadExecutor = threadExecutor;
    this.postExecutionThread = postExecutionThread;
  }

  protected abstract Observable buildUseCaseObservable();

  public void execute(Subscriber UseCaseSubscriber) {
    this.subscription = this.buildUseCaseObservable()
        .subscribeOn(Schedulers.from(threadExecutor))
        .observeOn(postExecutionThread.getScheduler())
        .subscribe(UseCaseSubscriber);
  }

  public void unsubscribe() {
    if (!subscription.isUnsubscribed()) {
      subscription.unsubscribe();
    }
  }
}
```

As you can see here, all use cases inherit from this abstract class and implement the abstract method `buildUseCaseObservable()` which will setup an `Observable<T>` that is going to **do the hard job and return the needed data.**

Something to highlight is the fact that on `execute()` method, we make sure our `Observable<T>` executes itself in a separate thread, thus, minimizing how much we block the android main thread. **The result is push back on the Android main thread through the android main thread scheduler.**

So far, we have our `Observable<T>` up and running, but, as you know, someone has to observe the data sequence emitted by it.

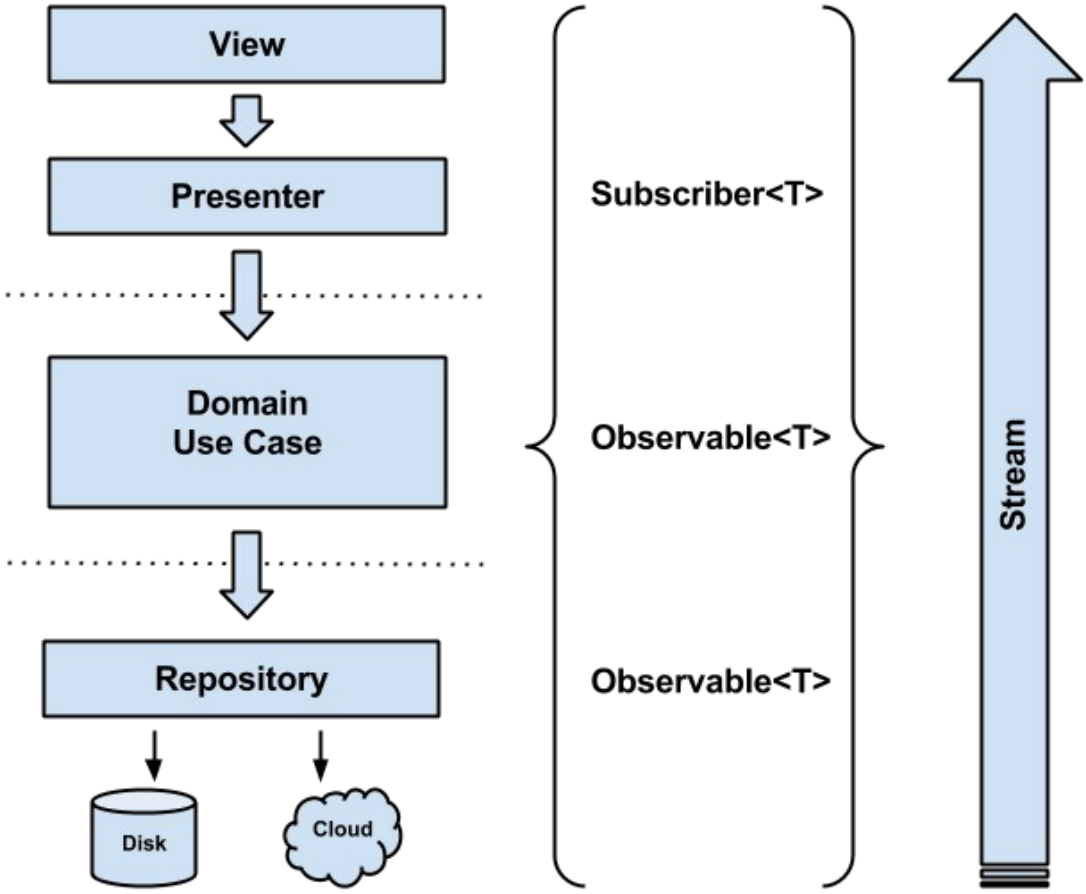
To achieve this, I evolved **Presenters** (part of MVP in the presentation layer) into **Subscribers** which would “react” to these emitted items by use cases in order to update the user interface.

Here is how the Subscriber looks like:

```
private final class UserListSubscriber extends DefaultSubscriber<List<User>> {

  @Override public void onCompleted() {
    UserListPresenter.this.hideViewLoading();
  }

  @Override public void onError(Throwable e) {
    UserListPresenter.this.hideViewLoading();
    UserListPresenter.this.showErrorMessage(new DefaultErrorBundle((Exception)
e));
    UserListPresenter.this.showViewRetry();
  }

  @Override public void onNext(List<User> users) {
    UserListPresenter.this.showUsersCollectionInView(users);
  }
}
```

Every subscriber is an inner class inside each presenter and implements a `DefaultSubscriber<T>` created basically for default error handling.

After putting all pieces in place, you can get the whole idea by having a look at the following picture:



Let's enumerate a bunch of **benefits** we get out of this **RxJava** based approach:

- **Decoupling between Observables and Subscribers:** makes maintainability and testing easier.
- **Simplified asynchronous tasks:** java threads and futures are complex to manipulate and synchronize if more than one single level of asynchronous execution is required, so by using schedulers we can jump between background and main thread in an easy way (with no extra effort), especially when we need to update the UI. We also avoid what we call a "callback hell", which makes our code unreadable and hard to follow up.
- **Data transformation/composition:** we can combine multiple `Observables<T>` without affecting the client, which makes our solution more scalable.
- **Error handling:** a signal is emitted to the consumer when an error has occurred within any `Observable<T>`.

Wrapping up

That is pretty much I have for now, and as a conclusion, keep in mind there are **no silver bullets**. However, a good software architecture will help us keep our **code clean and healthy**, as well as scalable and easy to maintain.
There is a few more things I would like to point out and they have to do with attitudes you should take when facing a software problem:

- **Respect SOLID principles.**
- **Do not over think (do not do over engineering).**
- **Be pragmatic.**
- **Minimize framework (android) dependencies in your project as much as you can.**