Fernando Cejas

Welcome! I'm Fernando Cejas, Core Engineer working at @SoundCloud.
In this blog, you will find content related to software engineering.
All views, posts and opinions shared are my own.

GitHub Twitter Speaker Deck LinkedIn

Architecting Android...The clean way?

03 Sep 2014

Over the last months and after having a few android discussions at Tuenti with colleagues like @pedro_g_s and @flipper83, I have decided that was a good time to write an article about **architecting android applications.**

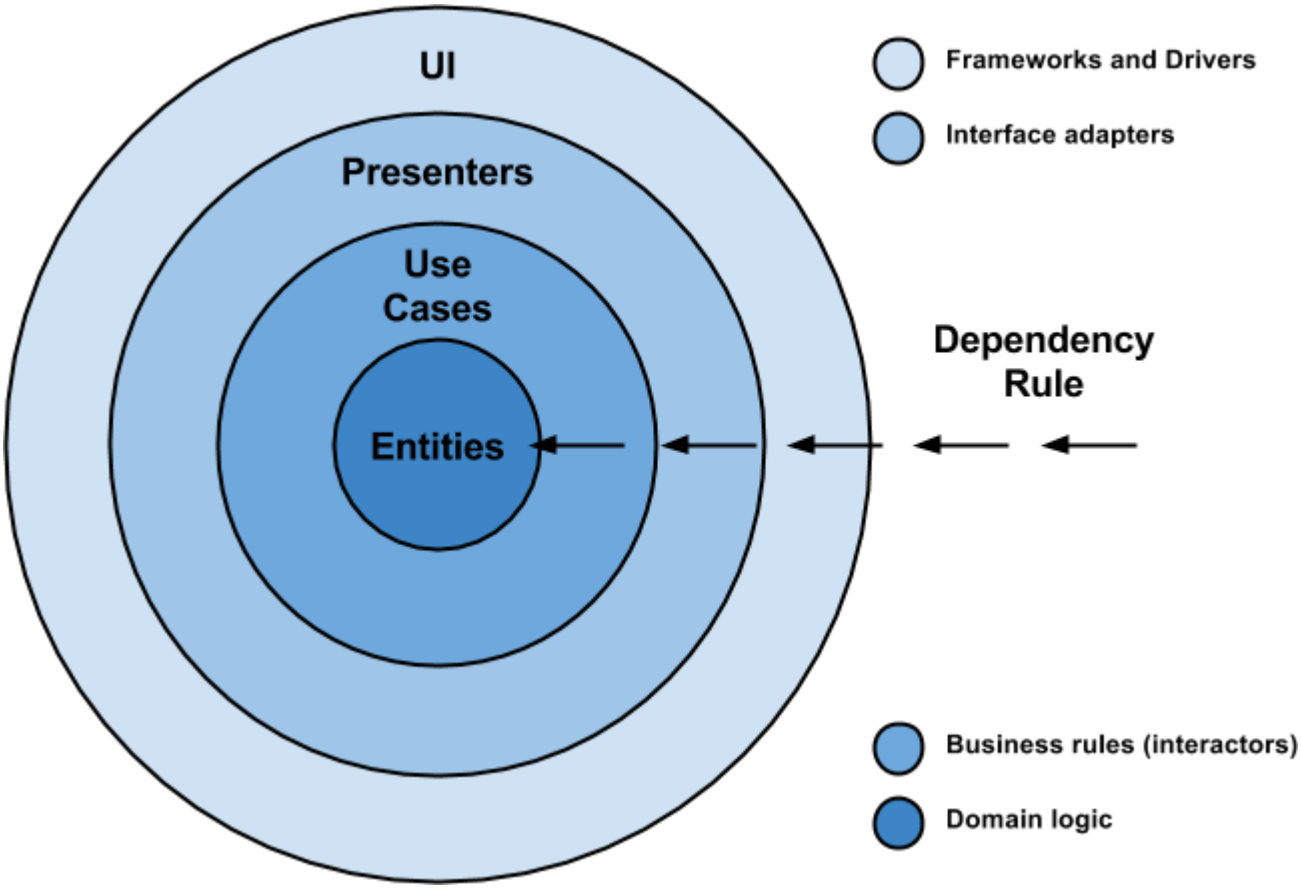
The purpose of it is to show you a little approach I had in mind in the last few months plus all the stuff I have learnt from investigating and implementing it.

Getting Started

We know that writing quality software is hard and complex: It is not only about satisfying requirements, also should be robust, maintainable, testable, and flexible enough to adapt to growth and change. This is where **"the clean architecture"** comes up and could be a good approach for using when developing any software application.

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.**

Here is some vocabulary that is relevant for getting familiar and understanding this approach in a better way:

- **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.

For a better and more extensive explanation, refer to this article or this video.

Our Scenario

I will start with a simple scenario to get things going: simply create an small app that shows a list of friends or users retrieved from the cloud and, when clicking any of them, a new screen will be opened and for instance, show more details for that user. Here is a quick video:

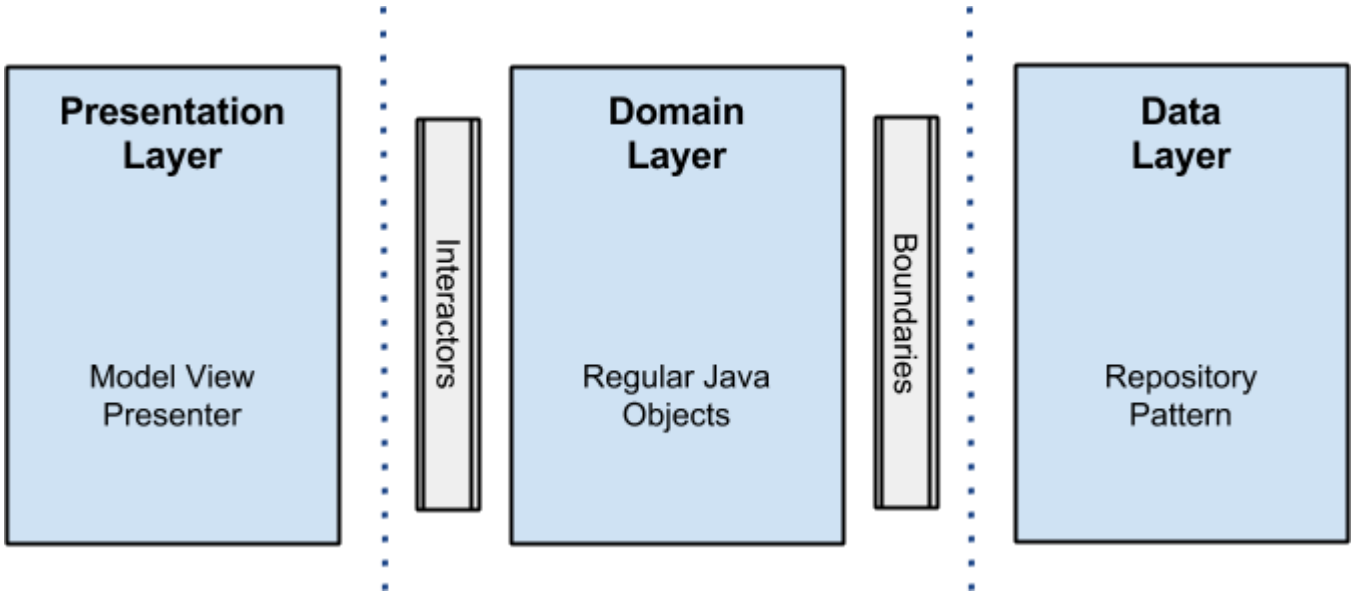
Android Architecture

The purpose is the **separation of concerns** by keeping the business rules not knowing anything at all about the outside world, thus, they can can be tested without any dependency to any external element.

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:

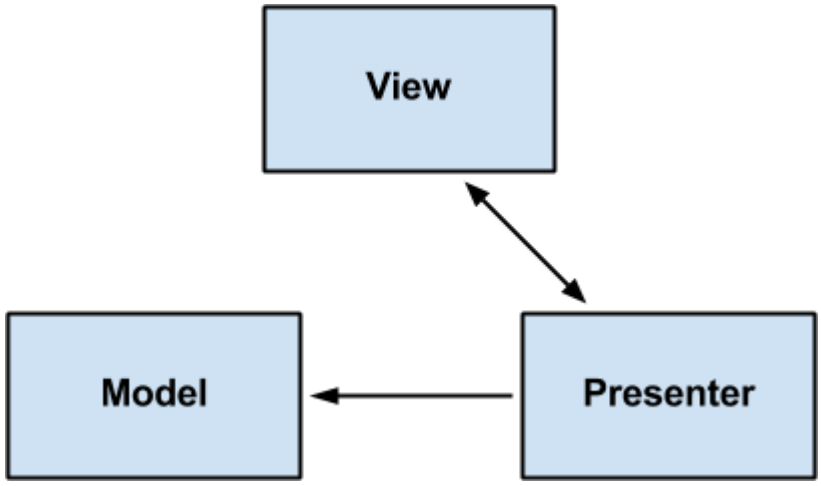


NOTE: I did not use any external library (except gson for parsing json data and junit, mockito, robolectric and espresso for testing). **The reason is that it makes the example clearer.** Anyway do not hesitate to add ORMs for storing disk data or any dependency injection framework or whatever tool or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

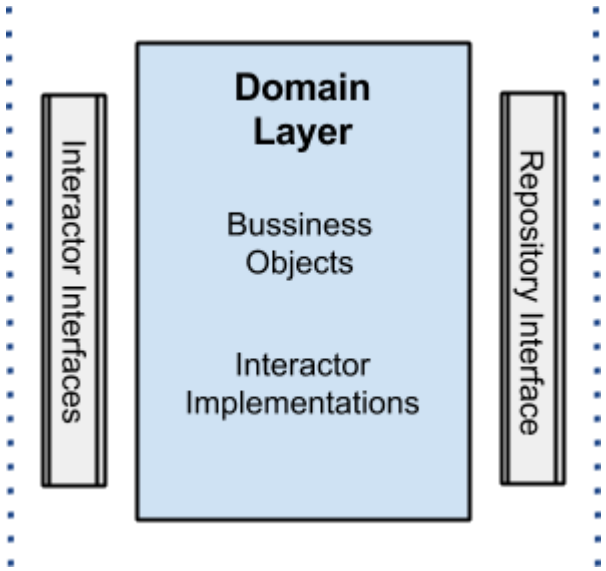


If you want a cool example about Effective Android UI that uses MVP and MVVM, take a look at what my friend Pedro Gómez has done.

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.

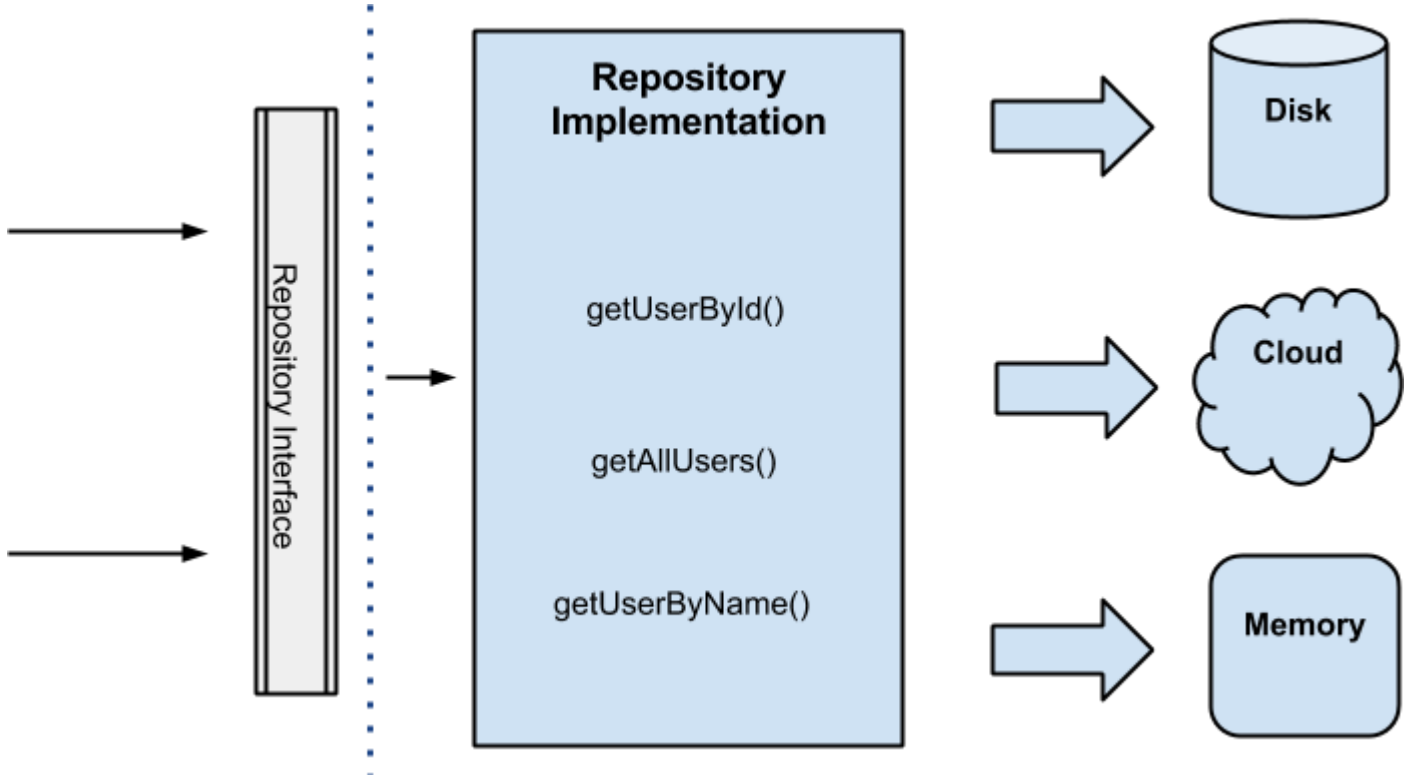


Data Layer

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 data origin is transparent for the client, which does not care if the data is coming from memory, disk or the cloud, the only truth is that the data will arrive and will be got.



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 that 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.**

As a conclusion I strongly recommend you give it a try and see and share your results and experiences, as well as any other approach you've found that works better: we do know that **continuous improvement** is always a very good and positive thing.

I hope you have found this article useful and, as always, any feedback is very welcome.

Source code

- Clean architecture github repository – master branch
- Clean architecture github repository – releases

Further reading:

- Architecting Android..the evolution
- Tasting Dagger 2 on Android
- The Mayans Lost Guide to RxJava on Android
- It is about philosophy: Culture of a good programmer

Links and Resources

- The clean architecture by Uncle Bob
- Architecture is about Intent, not Frameworks
- Model View Presenter
- Repository Pattern by Martin Fowler
- Android Design Patterns Presentation

Kindness is one of the greatest gifts you can bestow upon another.
If someone is in need, lend them a helping hand. Do not wait for a thank you.
True kindness lies within the act of giving without the expectation of something in return.

This site is maintained by android10.