

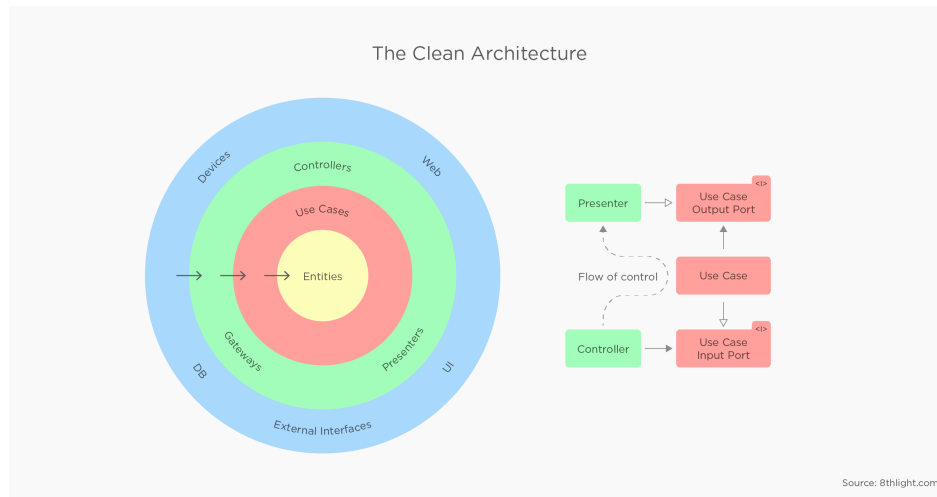
Home
About
Company
Our process
Leadership
Careers
Projects
Blog
Contact

Android Architecture: Part 2 – the clean architecture

November 10, 2016 Tomislav Homan

In the [first part of the series](#), we covered the mistakes we had made on our path to finding the architecture that works. In this part, we will present the so-called **Clean Architecture**.

The first image you come across when you google “clean architecture” is this:



It is also known as **onion architecture**, because the diagram looks like an onion (and it makes you cry when you realize how much boilerplate you have to write); or ports and adapters, because there are, as you can see, some ports in the bottom right corner. **Hexagonal architecture** is yet another similar architecture.

Clean architecture is the brainchild of previously-mentioned Uncle Bob, who also wrote books on Clean Code and Clean Coder. The main point of this approach is that the business logic, also known as domain, is at the center of the universe.

Master of your domain

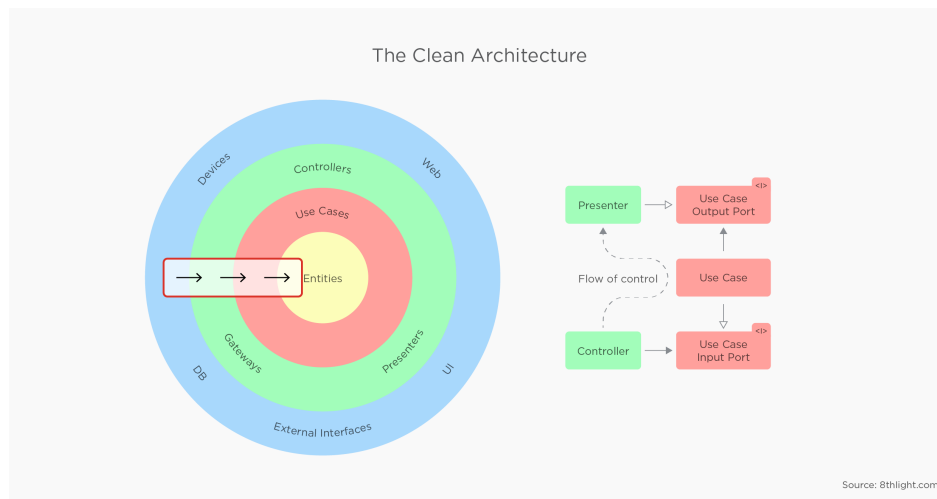
When you open your project, you should already know what this app is all about, regardless of the technology. Everything else is an implementation detail. For example, persistence—it's a detail. Define an interface, make in-memory a quick and dirty implementation, and don't think about it until the business is done. Then you can decide how you really want to persist the data. Database, internet, combination, file system—maybe leave them in-memory, maybe it turns out you don't have to persist them at all. In a sentence: inner layers contain business logic, outer layers contain implementation details.

That being said, there are a couple of features of clean architecture that enable that:

1. Dependency rule
2. Abstraction
3. Communication between layers

I. Dependency rule

Dependency rule can be explained by the following diagram:



Outer layers should depend on inner layers. Those three arrows in the red square represent dependencies. Instead of “depends on,” maybe it’s better to use terms like “sees,” “knows about,” or “is aware of.” In these terms, outer layers see, know about, and are aware of inner layers, but inner layers neither see nor know about, nor are aware of, outer layers. As we said previously, inner layers contain business logic and outer layers contain implementation details. Combined with the dependency rule, it follows that business logic neither sees, nor knows, nor is aware of, implementation details. And that’s exactly what we are trying to accomplish.

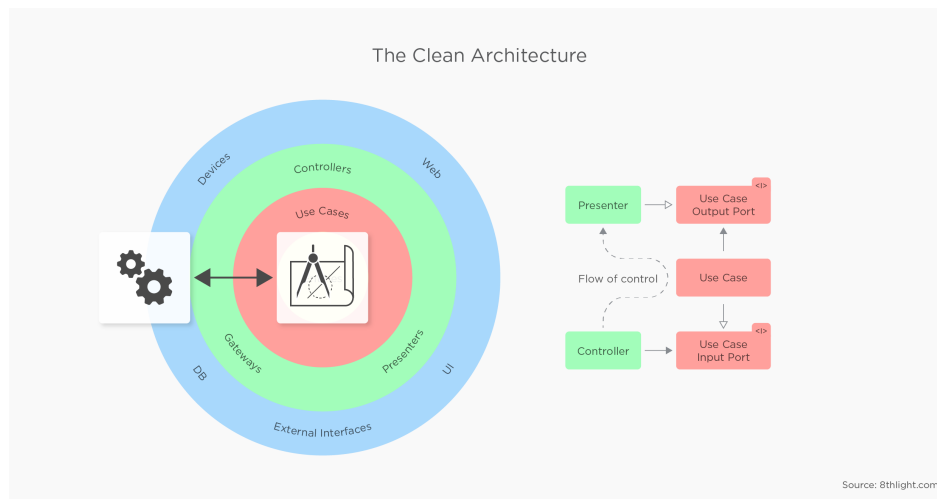
How you implement the dependency rule is up to you. You can put it in different packages, but be careful not to use “outer”

packages from “inner” packages. However, if somebody is not aware of the dependency principle, nothing would stop them from breaking it. A better approach would be to separate the layers into different Android modules, for example, and adjust dependencies in the build file so that the inner layer simply cannot use the outer layer. At Five, we use something in between.

One more thing worth mentioning is that, although nobody can stop you from skipping layers—for example, using some red layer component from the blue layer component—I strongly encourage you to access only components from the layer next to yours.

II. Abstraction

Abstraction principle has already been hinted at before. It says that, as you are moving towards the middle of the diagram, stuff becomes more abstract. That makes sense: as we said that the inner circle contains business logic and the outer circle contains implementation details.

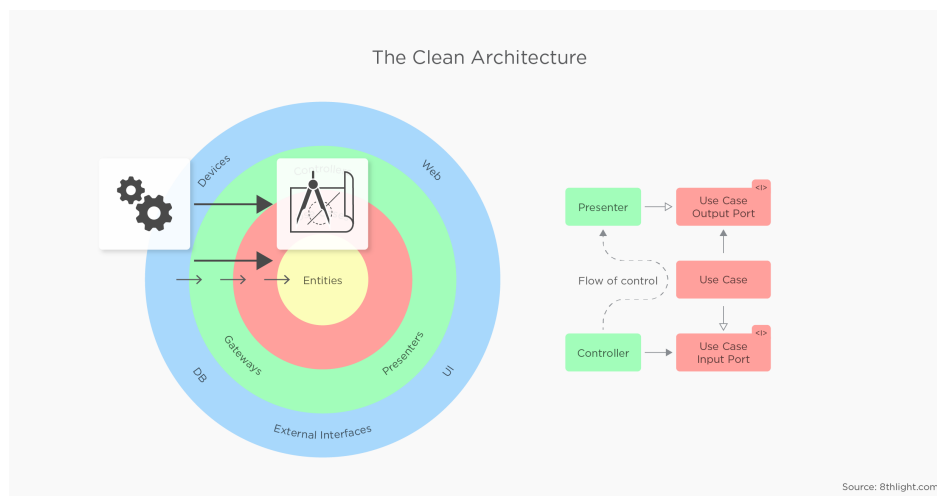


You can even have the same logical component divided between multiple layers, as shown in the diagram. The more abstract part can be defined in the inner layer, and the more concrete part in the outer layer.

An example will make it clear. We can define an abstract interface as “Notifications” and put it in the inner layer, that way your business logic can use it to show the notification to the user when it wants to. On the other hand, we can implement that interface in such a way that the implementation uses Android notification manager to show the notifications, and then put that implementation in the outer layer.

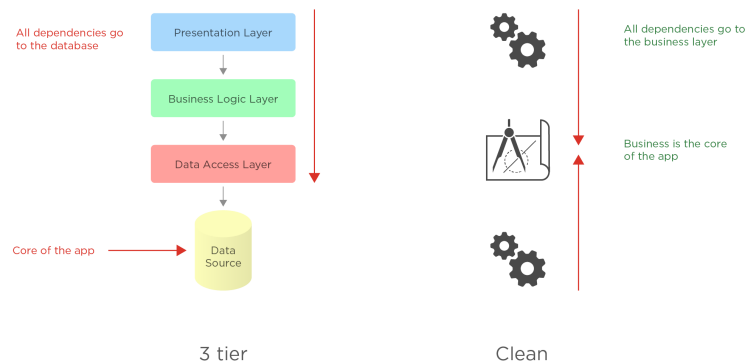
In this way, business logic can use a feature —notifications in our example—but it doesn’t know anything about implementation

details: how the actual notifications are implemented. Moreover, business logic doesn't even know that implementation details exist. Look at the following image:



When you combine abstraction with the dependency rule, it turns out that our abstract business logic using notifications neither sees nor knows about, nor is aware of, concrete implementation that uses the Android notification manager. That is good, because we can switch that concrete implementation and business logic won't even notice it.

Let's just briefly compare this to how abstraction and dependencies look and work when using standard three-tier architecture.



You can see in the diagram that all dependencies in the standard three-tier architecture go to the database. That means that abstraction and dependency don't match. Logically, the business layer should be the center of the app, but it isn't, as dependencies go towards the database.

The business layer knows about the database and it shouldn't. It should be the other way around. Whereas in clean architecture, dependencies go to the business (inner) layer and abstraction also rises towards the business layer, so they match nicely.

This is important because abstraction is the theory and dependencies are the practice. Abstraction is the logical layout of the app, and dependencies are how it is actually

composed together. In clean architecture these two match up, while in the standard three-tier architecture they don't; and this can quickly lead to all kinds of logical inconsistencies and mess if you aren't careful.

III. Communication between layers

Now that we have divided our app into the modules, nicely separated everything, put business logic in the center of our app and implementation details in the outskirts, everything looks great. But you've probably quickly run into an interesting problem.

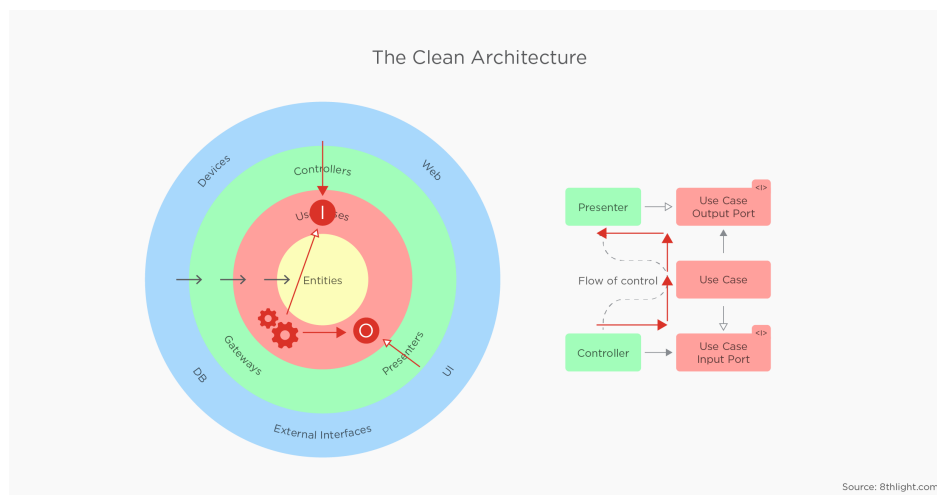
If your UI is an implementation detail, the internet is an implementation detail, and business logic is in between, **how the heck can we fetch the data from the internet, pass it through the business logic and then send it to the screen?**

Business logic is in the middle and should mediate between the internet and the UI, but it doesn't even know that those two

guys exist. This is a question of communication and data flow.

We would like the data to be able to flow from the outer layers to the inner ones and vice versa, but the dependency rule doesn't allow that. Let's strip that down to the simplest example.

We have only two layers, the green one and the red one. The green one is outer and knows about the red one, and the red one is inner and knows only about itself. We want the data to flow from the green one to the red one and back to the green one. The solution has already been hinted at before and is shown in the following diagram:



The part of the diagram at the bottom right side shows the data flow. Data goes from the controller, through the use case (or replace use case with the component of your choice) input port, then through the use case itself, and after that through the use case output port back to the presenter.

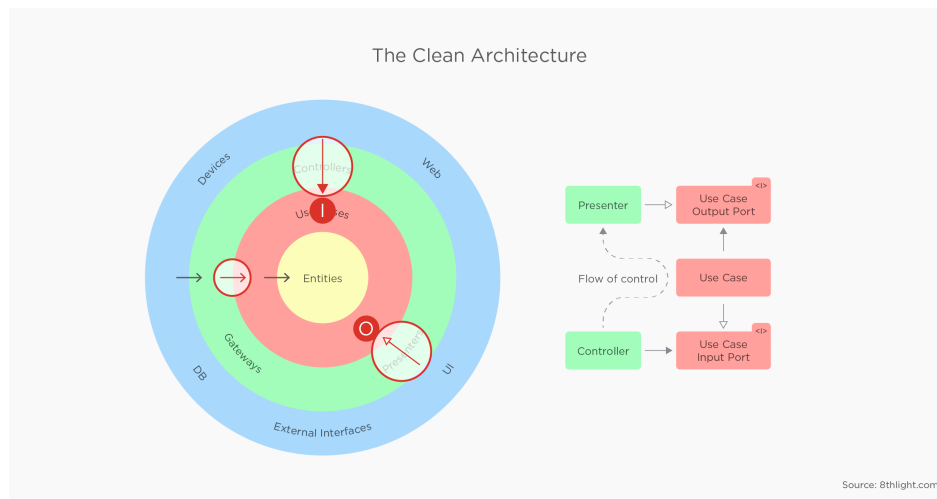
Arrows on the main part of the diagram denote composition and inheritance—composition indicated by a filled arrowhead, inheritance an empty arrowhead.

Composition is also known as **has-a relationship**, and inheritance **is-a relationship**. “I” and “O” in the circles represent input and output ports. It can be seen that a controller defined in the green layer has an input port defined in the red layer. Use case (gears, business logic, whatever—not important now) is (or implements) that input port and has an output port. And finally, the presenter defined in the green layer is actually an output port defined in the red layer.

We can match that to the dataflow now. The controller has an input port—it literally has a reference to it. It calls a method on it, so that data goes from the controller to the input

port. But the input port is an interface, and the actual implementation is the use case: so it has called a method on a use case and the data flows to the use case. Use case does something and wants to send the data back. It has a reference to the output port—as the output port is defined in the same layer—so it can call the method on it. Hence, data goes to the output port. And finally, the presenter is, or implements, the output port; that's the magic part. As it implements the output port, the data actually flows into it.

The trick is that the use case knows only its output port; the world ends at this output port. It's up to the presenter to implement it. It could've been implemented by anything, as use case doesn't know or care, and is aware only of its little world inside its layer. We can see that, by combining composition and inheritance, we can make the data flow in both directions, although inner layers aren't aware that they are communicating with the outside world. Take a quick glance at the following diagram:



You can see that both has-a and is-a arrows point to the middle—the same as dependency arrows. Well, it's logical. According to the dependency rule, it's the only possible way. The outer layer can see the inner layer, but not the other way around. The only tricky part is that is-a relationship, although it points to the middle, reverses the dataflow.

Notice that it's the inner layer's responsibility to define its input and output ports, so that the outer layers can use them to establish communication with it. I've said that this solution has already been hinted at before, and it has. The notifications example that illustrated abstraction is also an example of this kind of communication. We have a notifications interface defined in the inner layer that business logic can use to show


notifications to the user, but we also have an implementation defined in the outer layer. In that case, the notifications interface is business logic's output port, which it uses to communicate to the outer world—to the concrete implementation in this example. You don't have to name your classes `FooOutputPort` or `BarInputPort`; we name the ports just to explain the theory.

Conclusion

So, is this overly complicated, overly obscured over-engineering? Well, it's simple when you get used to it. **And it's necessary.** It allows us to make that nice abstraction / dependency match actually communicate and work in the real world. Maybe all of this reminds you of the string theory: beautiful, theoretically elegant, but overly complicated and we still don't know whether it works, but in our case – it does. :)

So that's it for the second part of the series. [The last, third part](#), after all that we have learned about the theory and architecture, will cover all you need to know about **labels** on those diagrams, or in other words

- separate components. We will show you a real life clean architecture applied on Android.

Comments**Community****1 Login** ▼ **Recommend** 9 **Share****Sort by Best** ▼

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

**AbdElraouf Sabri** • 5 months ago

Great article, I actually think it's one of the best articles that illustrate clean architecture. Thanks you.

^ | v • Reply • Share ›

**Asim Qasimzade** • 7 months ago

Wonderful article. Almost scientific. Now all this Android Architecture mess in my head is organized into some logical diagram that actually makes sense.

^ | v • Reply • Share ›

Related

BLOG

20 Tools to Consider When Doing Qualitative Remote

How to Decide on The Right Tool in Less Than 5 Minutes? User research is

Gabrijela Šitum

SEARCH ENGINE
OPTIMIZATION, SEO

Complete SEO Guide for Web Developers

Complete SEO guide for meta tags, URLs, robots, sitemaps, social tags,

Ozren Lapčević

BLOG

Android , Part 5: How Clean Arc

Why should about test Programm

David Geček

