

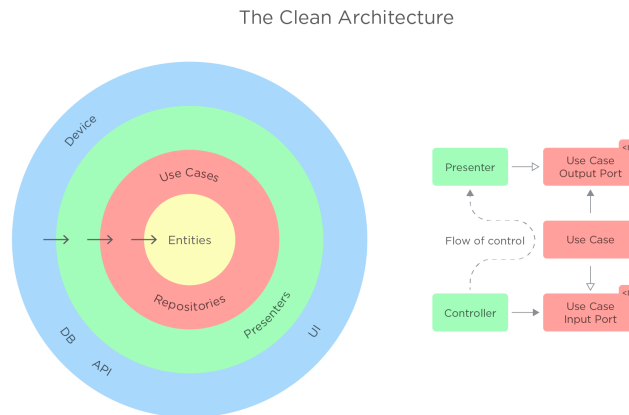
Home  
About  
Company  
Our process  
Leadership  
Careers  
Projects  
Blog  
Contact

# Android Architecture: Part 3 – Applying clean architecture on Android

November 17, 2016 [Tomislav Homan](#)

So far in this series, we've covered some [beginner's mistakes](#) and gone through the [clean architecture](#). In this last part, we will cover **the last piece of the puzzle**: labels, or more precisely, components.

First, I'll remove the stuff we don't use on Android projects and I'll add some stuff that we do use but that isn't found in the original Uncle Bob's diagram. It looks like this:



Source: 8thlight.com

I'll go from the most abstract center to the edges.

## Entities

Entities, aka **domain objects** or business objects, are the core of the app. They represent the main functionality of the app, and you should be able to tell what the app is about just by looking at them. They contain business logic, but it's constrained to them only—validation and stuff like that. They don't interact with the gritty outside-world details, and they also don't handle

persistence. If you had a news app, the entities would be Category, Article, and Commercial, for example.

## Use cases

Use cases, aka interactors, aka business services, are an extension of the entities, an extension of the business logic, that is. They contain the logic that isn't constrained only to one entity but handle more of them. An indicator of a good use case is that you can describe what it does in a simple sentence using common language—for example, “Transfer money from one account to another.” You can even use such nomenclature to name the class, e.g., `TransferMoneyUseCase`.

## Repositories

Repositories serve to persist the entities. It's as simple as that. They are defined as interfaces and serve as output ports for the use cases that want to perform CRUD operations on entities. Additionally, they can expose some more complex operations

related to persistence such as filtering, aggregating, and so on. Concrete persistence strategies, e.g., database or the Internet, are implemented in the outer layers. You could name the interface `AccountRepository`, for instance.

## Presenters

Presenters do what you would expect them to do if you are familiar with the MVP pattern. They handle user interactions, invoke appropriate business logic, and send the data to the UI for rendering. There is usually some mapping between various types of models here. Some would use controllers here, which is fine. The presenter we use is officially called the supervising controller. We usually define one or two presenters per screen, depending on the screen orientation, and our presenters' lifecycles are tied to that of a view. One piece of advice: try to name your methods on a presenter to be technology agnostic. Pretend that you don't know what technology the view is implemented in. So, if you have methods named `onSubmitOrderButtonClicked` and

onUserListItemSelected in the view, the corresponding presenter methods that handle those events could be named submitOrder and selectUser.

## Device

This component has already been teased before in the notifications (again) example. It contains the implementations of the gritty Android stuff such as sensors, alarms, notifications, players, all kinds of \*Managers, and so on. It is a two-part component. The first part is the interfaces defined in the inner circles that business logic uses as the output port for communication with the outer world. The second part, and that's drawn in the diagram, are implementations of those interfaces. So, you can define, for example, interfaces named Gyroscope, Alarm, Notifications, and Player. Notice that the names are abstract and technology agnostic. Business logic doesn't care how the notification will be shown, how the player will play the sound, or where the gyroscope data comes from. You can make an implementation that writes the notifications to the terminal, write the sound

data to the log, or gather the gyroscope data from the predefined file. Such implementations are useful for debugging or creating a deterministic environment for you to program in. But you will, of course, have to make implementations such as `AndroidAlarm`, `NativePlayer`, etc. In most cases, those implementations will just be wrappers around Android's manager classes.

## DB & API

No philosophy here. Put the implementations of repositories in this component. All the under-the-hood persistence stuff should be here: DAOs, ORM stuff, Retrofit (or something else) stuff, JSON parsing, etc. You can also implement a caching strategy here or simply use in-memory persistence until you are done with the rest of the app. We had an interesting discussion in the team recently. The question was this: Should the repository expose methods such as `fetchUsersOffline` (`fetchUsersFromCache`) and `fetchUsersOnline` (`fetchUsersFromInternet`)? In other words, should business logic know where the data comes from? Having read

everything from this post, the answer is simple: no. But there is a catch. If the decision about the data source is part of the business logic—if the user can choose it, for example, or if you have an app with explicit offline mode—then you CAN add such a distinction. But I wouldn't define two methods for every fetch. I would maybe expose methods such as `enterOfflineMode` and `exitOfflineMode` on the repository. Or if it applies to all the repositories, we could define an `OfflineMode` interface with `enter` and `exit` methods and use it on the business logic side, leaving repositories to query it for the mode and decide it internally.

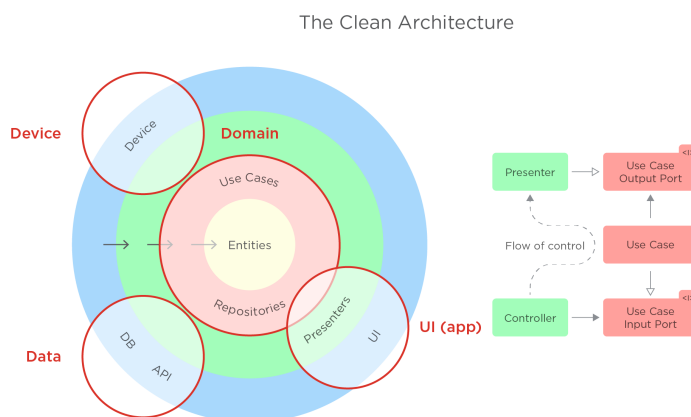
## UI

Even less philosophy here. Put everything related to the Android UI here. Activities, fragments, views, adapters, etc. Done.

## Modules

The following diagram shows how have we divided all these components into Android

Studio modules. You might find another division more appropriate.



Source: 8thlight.com

We group entities, use cases, repositories, and device interfaces into the domain module. If you want an extra challenge with a reward of eternal glory and a totally clean design, you can make that module a pure Java module. It will prevent you from taking shortcuts and putting something related to the Android here.

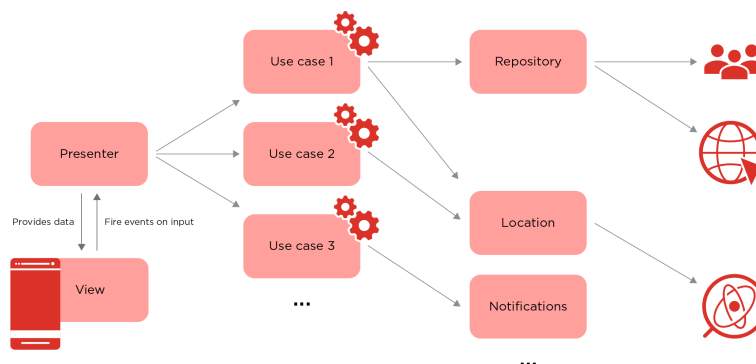
The device module should have everything related to Android that's not data persistence and UI. The data module should hold everything related to data persistence, as we've already said. You cannot make those two into Java modules because they need access to various Android stuff. You can make them into Android library.



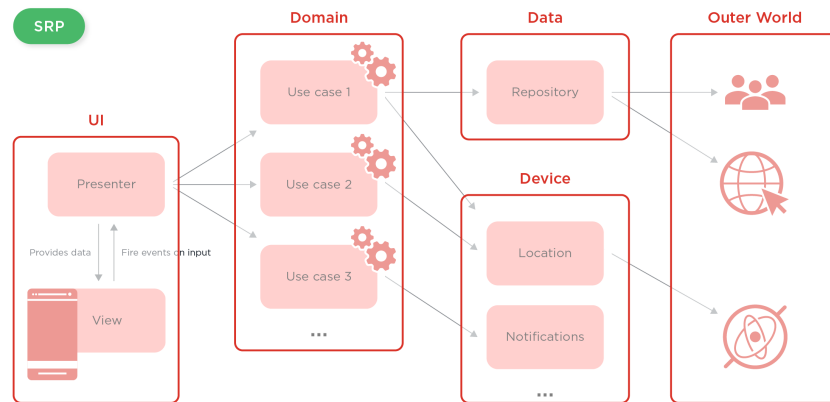
Finally, we group everything related to the UI (including presenters) into the UI module. You can explicitly name it UI but because of all the Android stuff here, we leave it named “app,” just as Android Studio named it during the creation of the project.

## Is it better?

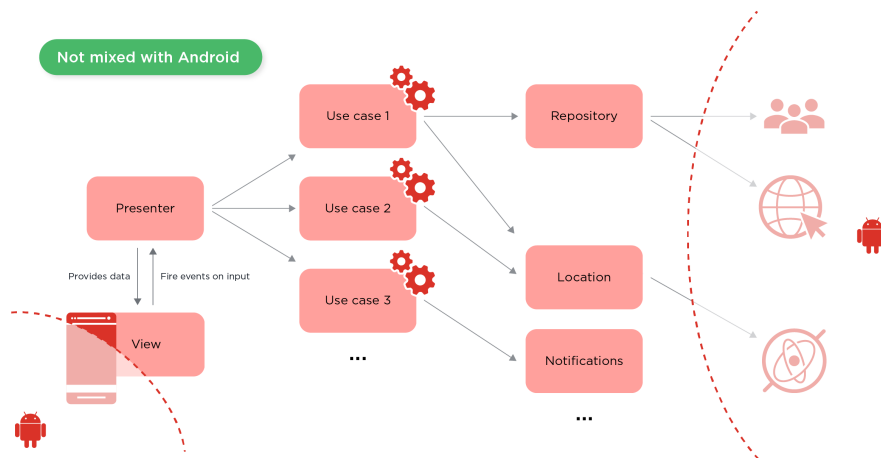
To answer that question, I’ll ditch Uncle Bob’s diagram and sprawl the components described before into a diagram like those we have used to rate previous types of architectures. After doing that, we get this:



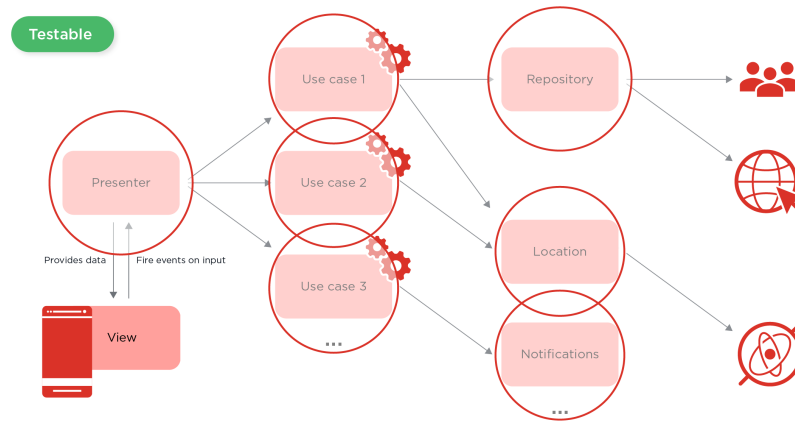
Now let’s apply the same criteria that we have used on previous architectures.



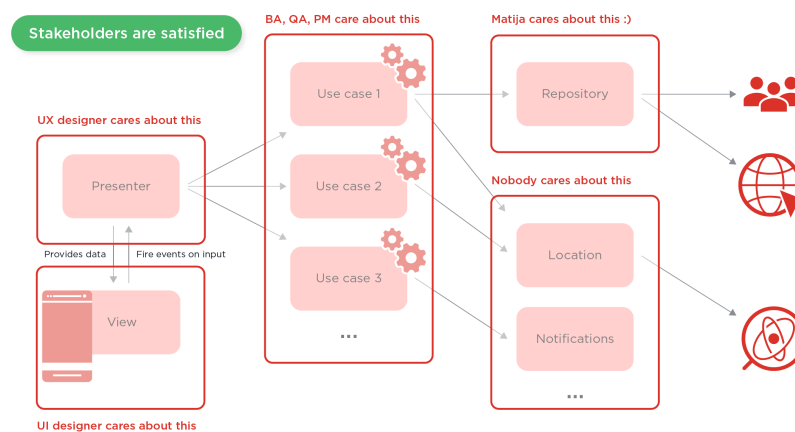
It's all neatly separated by module level, package level, and class level. So SRP should be satisfied.



We have pushed Android and the real-world stuff as far out on the outskirts as we can. Business logic doesn't touch the Android directly anymore.



We have nicely separated classes that are easy to test. Classes touching the world can be tested using Android test cases; the one not touching it can be tested using JUnit. Someone malevolent would maybe call that class explosion. I call it testable. :)



It may be complicated – but  
it's worth it

I hope that my carefully chosen criteria, which was not made up just to favor the clean architecture, will convince you to give it a try. It seems complicated, and there are lots of details here, **but it's worth it**. Once you wire it all up, testing is much easier, bugs are easier to isolate, new features are easy to add, code is more readable and maintainable, everything works perfectly, the universe is satisfied.

So, this is it. If you still haven't done so, look at the previous articles in the series: [Mistakes](#) and [Clean Architecture](#). If you have any comments or questions, leave feedback below. We are always interested in hearing what you think.

Read the [4th part](#) of our Android Architecture Series.


---

Comments

Community

 Login ▾

 Recommend 24

 Share

Sort by Best ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISOUS 

**Waleed sarwar** • a year ago

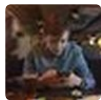
I think i have read every blog post related to Clean Architecture for Android on Internet. But So far this is one of best explanation of Clean Architecture in Android.

5 ^ | v • Reply • Share ›

**Tomo** → Waleed sarwar • 10 months ago

Thanks :)

^ | v • Reply • Share ›

**Nicklas Nilsson** • 7 months ago

Hi, great tutorial! Do you know any example code using these principles for a "package by feature" structure instead of "package by layer"?

1 ^ | v • Reply • Share ›

**Tomo** → Nicklas Nilsson • 7 months ago

I did a quick search now on repos that I know of, but all of them are packaged by layer. Everybody is talking that it would be also nice to package by feature, but apparently nobody is doing it :)

^ | v • Reply • Share ›

**Team Tiger** • 8 months agoHi **@Tomo**,

Thanks for your sharing!

I have some concerns that I got on my own project.

I would like to store the current user's logged in state. Some where in my code (Ex: activity, fragment, presenters,...), I would like to check this state then process the next step like:

```
if (userSession.isLoggedIn()) {
    // Navigate to home
} else {
    // Open Login page
}
```

I have no idea which layer should it be and how to access this. Please give your advices!

Thanks!

1 ^ | v • Reply • Share ›

**Tomo** → Team Tiger • 8 months agoHi **@Team Tiger**

I would store it in data layer. Now which specific technology to use is up to you. The simplest way is probably to store the logged in user id in the preferences. If your session has some other data besides the logged in user, such as session length stored for analytics, then you can store serialized session object in the preferences. Either way you can have a SessionRepository with getCurrent() method that gives you back the current session - it should have loaded user id (or whole user object if you need it) in it. After that you can call isLoggedIn() method that simply checks is current user null, or current user id empty string.

^ | v • Reply • Share ›



**Joao Sousa** • 9 months ago

Nice article! Please do share some code :)

1 ^ | v • Reply • Share ›



**Tomo** → Joao Sousa • 9 months ago

Thanks, there will be some, but I cannot exactly say when, as I'm not doing it.

^ | v • Reply • Share ›



**Luna Vulpo** • a month ago

how in such architecture places a external triggers like FCM notifications or LocationFence. In my app I need to show screen (start activity) when I get notification from FCM and when user enter to zone (base on location fences)?

^ | v • Reply • Share ›



**mohammed youser sawwas** • 2 months ago

Really Nice!

I suggest an awesome name for this awesome architecture which is :

"3D app" architecture,

obviously stands for the modules names:

Domain,Data,Device and app.

^ | v • Reply • Share ›



**Tomo** → mohammed youser sawwas • 2 months ago

Hehe, good suggestion with 3D, it didn't run across my mind :)

^ | v • Reply • Share ›



**Antony Gunawan** • 2 months ago

Hi, I want to ask something

Where do we store the view state variable in this architecture?

Thank you

^ | v • Reply • Share ›



**Tomo** → Antony Gunawan • 2 months ago

We didn't rely on it so much. It's a nice thing to let the framework handle, to keep some minor stuff like the text that you started writing into edit text survive the screen being recreated and you can use regular Android framework receipts to achieve that no matter are you using clean approach or not. On the other hand the major stuff, the core app state should always be able to be recreated in any moment from the 'single source of truth' like the local DB.

^ | v • Reply • Share ›



**Khanh Nguyen Xuan** • 2 months ago

Hi, it's actually the best post about the clean architecture.

I have some open questions:

1. Will we have "requestPermissionUseCase" for requesting permissions?
2. Using RX inside makes use cases have only one mission is to call the repository. There's no business logic inside.

^ | v • Reply • Share ›



**nijkanada** → Khanh Nguyen Xuan • 2 months ago

"In software and systems engineering, a use case is a list of actions or event steps typically defining the interactions between a role(actor) and a system to achieve a goal." - Wikipedia.

Requesting permissions is not a goal of actor. and I recommend you to read a book to understand what is usecase.

^ | v • Reply • Share ›



**Khanh Nguyen Xuan** → nijkanada  
• 2 months ago

Thanks for replying,

It is not a usecase, so what I have to do to request permissions?

^ | v • Reply • Share ›



**nijkanada** → Khanh Nguyen Xuan  
• 2 months ago

I can't write english. Sorry about that.  
Think why should user request a permission for "what", "What to do".  
Requesting permission is not a main concern of application. It is just low-level detail(just needed mechanism to do something). So, the detail should be hided from use case.

^ | v • Reply • Share ›



**Tomo** → nijkanada • 2 months ago

I think permission asking could be hidden in the utility class in the device module. You can define Permissions interface in the domain and PermissionsImpl in the device module. Then a use case can refer to that use helper class if it has to. It same thing as with notifications. For example you can have

CheckOpenedStoresNearbyUseCase and inside:

```
public Observable<result> execute() {
    if
    (permissions.hasLocationPremissoin())
    {
        val nearestStore =
        storeRepository.list().filterByLocation(loc
        if (isStoreOpened(nearestStore)) {
            notifications. show("You have one
            opened store near");
        }
    }
}
```

You can see that one 'high-level' class like

CheckOpenedStoresNearbyUseCase uses a lot of little helper classes (hidden behind the interface) like permissions, storeRepository, locations, notifications, etc...

^ | v • Reply • Share ›



**nijkanada** → Tomo • 2 months ago

Also, there is AOP. Can use aspect



Also, there is AOP. CUZ, use case doesn't want to know permission checking. You can implement that using Decorator pattern. So, there is no permission checking code in use case source code. It can be hidden perfectly.

1 ^ | v • Reply • Share ›



**Evelio** • 3 months ago

Thank you so much for this post. It was a pleasure read it, by far one of the best I have read about this topic.

^ | v • Reply • Share ›



**Tomo** → Evelio • 2 months ago

Thank you Evelio

^ | v • Reply • Share ›



**Pawel Urban** • 3 months ago

Hey! I'm really amazed by the article. I'm a fan of separation of the interfaces and the plugin architecture too.

However, I went through the attached examples and they seem to be inconsistent with the article. For example, there is a Notifications interface, which by design should hide Android internals. In reality it has a Notification class as an input, straight from the Android API.

Is that correct that considering Clean Architecture this interface should be Android-free and should be put somewhere in the domain layer? Then NotificationsImpl should be in a device layer. Is it ok that it is used in the app module without being proxied by the domain? Thanks!

^ | v • Reply • Share ›



**Tomo** → Pawel Urban • 2 months ago

You are right sir, and we should probably fix the example :)

1 ^ | v • Reply • Share ›



**Hulk Wu** • 4 months ago

It seems that the project is finally divide into modules by different layers: UI, Domain, Device, Data...? But if I want to divide the project into modules by features?

^ | v • Reply • Share ›



**Tomo** → Hulk Wu • 2 months ago



That is the usual uncertainty, divide by features or layers. I guess dividing it by features is also ok. The advantage of dividing it by layers is that you can enforce certain layers to be plain Java only libraries.

^ | v • Reply • Share ›



**narendra techguy** • 7 months ago

still waiting for 4th part with real examples

^ | v • Reply • Share ›



**Tomo** → narendra techguy • 7 months ago

hehe, me too :) I guess guy(s) making sample project are a bit pressed with day-time stuff.

1 ^ | v • Reply • Share ›



**Dmitriy Khaver** • 8 months ago

Hey **@Tomo** !

Thank you and everyone who works for providing these awesome explanations. (:

I would love to look on GitHub repo, tell me please, when it would be ready ?

^ | v • Reply • Share ›



**Tomo** → Dmitriy Khaver • 7 months ago

Hey **@Dmitriy Khaver**, I'm not sure because it's not me making samples, but it'll be published here on blog and on Five Agency's FB page.

^ | v • Reply • Share ›



**vikram singh (Thegreat004)** • 8 months ago

**@Tomo**, excellent way for people to understand it. But it would be nice if you can share code without dependency injection like dagger. Waiting with eager for your positive response.

^ | v • Reply • Share ›



**Tomo** → vikram singh (Thegreat004) • 8 months ago

Hi **@vikram singh (Thegreat004)**, colleague of mine is working on the samples, I'm not sure does he use Dagger. On the other hand some people want to see it with Dagger as that is real world example in the most of the cases. In my opinion, if you are disciplined enough to handle object graph lifecycle properly, you don't need Dagger.

^ | v • Reply • Share ›



**vikram singh (Thegreat004)** → Tomo

• 8 months ago

@Tomo, Basically i am also a new one to MVP and the reason is that if you are new one then you must go for basics that's why i was asking for without Dagger.

^ | v • Reply • Share ›



**Tomo** → vikram singh (Thegreat004)

• 8 months ago

Hmm yeah, I understand. You can try to ignore it when looking into examples, and when doing your app just use the factory instead.

^ | v • Reply • Share ›



**vikram singh (Thegreat004)** → Tomo

• 8 months ago

I don't know the factory methods, that's why I asked for demo app without dependency injection.

^ | v • Reply • Share ›



**Tomo** → vikram singh (Thegreat004)

• 8 months ago

Aha, uh, :) Well ignore factory methods, or abstract factories and such a "fancy" stuff, imagine that you are allowed to use "new" keyword in just one class and that's it :) Create all of your stuff in that class and call it a factory. Yes, it goes more complicated later on, but that's a good approximation for start.

^ | v • Reply • Share ›



**Hoang Lang** • 8 months ago

Thank for the useful post!

I get confuse where those things belong to:

- android service
- utilities, helper class
- customized view class (custom layout, custom imageview...)

Could you share your opinion?!

^ | v • Reply • Share ›



**Tomo** → Hoang Lang • 8 months ago



Hey **@Hoang Lang**. In my opinion:

- android service - I would put it the app module.  
Lately I don't have any logic in the service, I just start it sometimes to prevent the app for dying if I need to do something in the background, but business classes are the real work horses.
- utilities, helper classes - you can make some common java lib that all the rest have ref to. That is if your utils and helpers are being used by all the other modules. If not you can put it where you use it. For example you can have StringUtils that everybody uses and it can go to the common module, on the other hand DatabaseUtils should probably go to data.util package (data module)
- customized view classes - I would but those to the app module.

^ | v • Reply • Share >



**Hoang Lang** → Tomo • 8 months ago

Thank so much for your sharing **@Tomo** :)

^ | v • Reply • Share >



**Tomo** → Hoang Lang • 8 months ago

You're welcome :)

^ | v • Reply • Share >



**Mohammed Alhammouri** • 8 months ago

Great article, I have built two apps using Clean architecture/Rxjava/MVP I am very happy with the results, but I have one question that still looking for an answer to, Do I have to make entity/model/viewModel for each layer and mapper for each entity/viewModel (data/domain/Presentation) ?

I found the benefits of it are real and I am convinced that I should separate each layer with it's own POJOs.


My other question do I really have to make this POJO separation in ANDROID because , it's really annoying that every POJO I add I have to make mapper for it and everytime I change an attribute I change it in 3 classes instead of one ?

^ | v • Reply • Share >



**Tomo** → Mohammed Alhammouri • 8 months ago

Hi **@Mohammed Alhammouri** well yes it's

 **@Mohammed Alammouri**, well, yes, it's tiresome to keep all the POJOS, mapper, entities and (view)models separated. I used to insist on that separation, but if you and your team are disciplined enough I think you might take the pragmatic way and use the same model (domain model) even in outer layer, just always hold in your mind that logical separation and that one class plays domain model role in one layer and for example view model role when it arrives to the UI layer. And of course, if things get more complicated and you notice that you have started to mix the various layers logic in one class split it immediately into multiple models until it becomes too late.

 |  • Reply • Share ›



**Alexey Ershov** • 9 months ago

One of the best posts about Clean Architecture, with attention to details and practical questions.

I have a question, if you don't mind. I have some Web API, and all requests for data go through a Repository, that's no problem. But what about those API calls that actually do something? Like "submit order" or "send message". Should I call them directly from my UseCases, or through some interface like OrderManager, implemented with those API calls? What's your opinion on that? Thanks in advance!

UPD: and in case of OrderManager approach, do I really need UseCases? It seems they will only be forwarding calls to Repositories and SomethingManagers.

 |  • Reply • Share ›



**Tomo** → Alexey Ershov • 8 months ago

Hey Alexey. I would use SubmitOrderUseCase that builds the order and calls save on OrderRepository with that newly created order. OrderManager is ok if there is some more complex logic around orders involved. For example if it needs to be involved in some kind of workflow that puts it through the stages of approval or something, if there are different strategies involved depending on the order type, etc. In your case order just needs to be saved, and for that OrderRepository is enough. But that's a start, if it turns out that you'll need to have more complicated process in the future, it's easy to introduce OrderManager that handles the complicated stuff. Note that in that case OrderManager won't die, it can still be used to persist the order but it will be called from OrderManager

Yes, in most of the situations use cases delegate stuff to repositories. That's because our mobile apps usually don't have lots of business logic, the logic is to fetch or save. Maybe do some filtering, grouping and handle it. And in that case, if you are using Kotlin for

## 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 Arch

Why should about test Programm

David Geček



---

© 2007-2017, Five Agency

[Facebook](#)

[Twitter](#)

[Instagram](#)

[Dribbble](#)