
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

12: Clean Architecture on Android

- You'll need to have Android Studio 3.0.1 or later installed.
- You'll need Kotlin Plugin 1.2.21 or later installed in Android Studio.
- You'll need Auto Import enabled in Android Studio preferences, by going to "Editor->General->Auto Import" and setting "Insert imports on paste" to All and checking "Add unambiguous imports on the fly".

12: Clean Architecture on Android

As mobile software development has matured in the past decade, developers have sought out ways to improve app architecture in order to increase app quality and maintainability. On Android, the MVP and MVVM presentation architecture patterns have had significant popularity.

In the past few years, a number of examples of Clean Architecture on Android have also been presented within the Android community. This session will discuss some of the history and theory behind Clean Architecture, show an example app use case from the "outside-in", and then demonstrate the development of a new app use case from the "inside-out".

34 Clean Architecture on Android: Demo 1

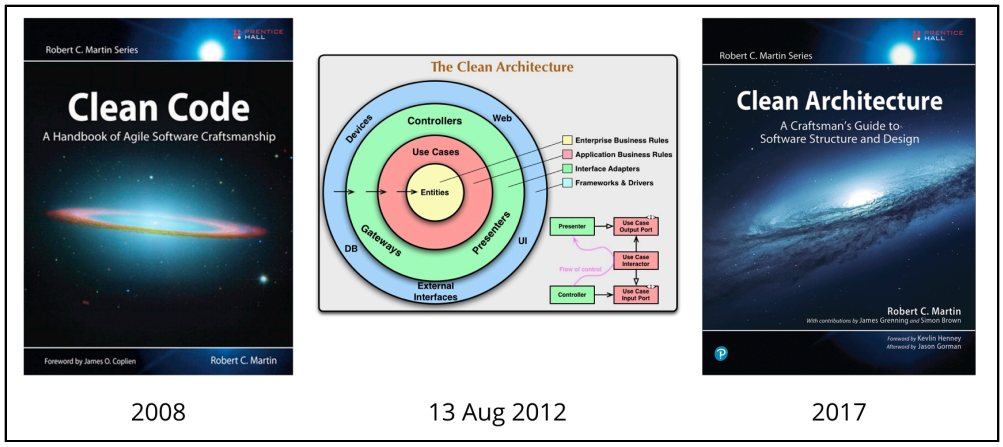
By Joe Howard

In this demo, you'll see a bit of the history and theory of Clean Architecture, and walk through an example app use case in the starter project from the "outside-in", that is, from the outer layer towards the inner layers.

1) The Clean Architecture

The history of the Clean Architecure parallels the history of software development over the past few decades. There are many ingredients within Clean Architecure, from Object-Oriented Design principles, software components, software testing, abstraction, use cases, frameworks, etc.

We don't have time to review the full history of Clean Architecture and related patterns, such as Ports and Adapters and the Hexagonal Architecure, but we'll take a quick look at some of the direct influences on Clean Architecture and then give a brief discussion on the history of Clean Architecture on Android.



2008 13 Aug 2012 2017

In 2008, Robert C. "Uncle Bob" Martin released his book "Clean Code," which talks about concepts from code formatting, to how to write functions and name variables,

to testing and refactoring.

In 2012, Uncle Bob wrote a blog post (http://bit.ly/2a1px8t) entitled "The Clean Architecture", a follow-up to an earlier blog post called "Screaming Architecture". In the latter blog post, he defines the Clean Archictecture and described the structuring of software systems into layers that allow for improved testability and maintainability.

In September 2017, Uncle Bob released a new book entitled "Clean Architecture", where he collects the software principles within Clean Architecture and details how to build software using the approach.

Looking at the diagram from the blog post, in Clean Architecture, code is structured into layers, with higher level or more abstract code at the center, and lower level code going out through the layers. The center are core business rules, the next layer is use cases and application specfic logic, then presentation logic, and finally, app details such as the delivery mechanism and database. A distinguishing feature of Clean Architecture is that source code dependencies point from the outside in, with inner layers independent of outer layers.

2) Goals and Principles of Clean Architecure

I've summarized here what I see as the main guiding goals and principles of Clean Architecture.

For the goals, we're looking for:

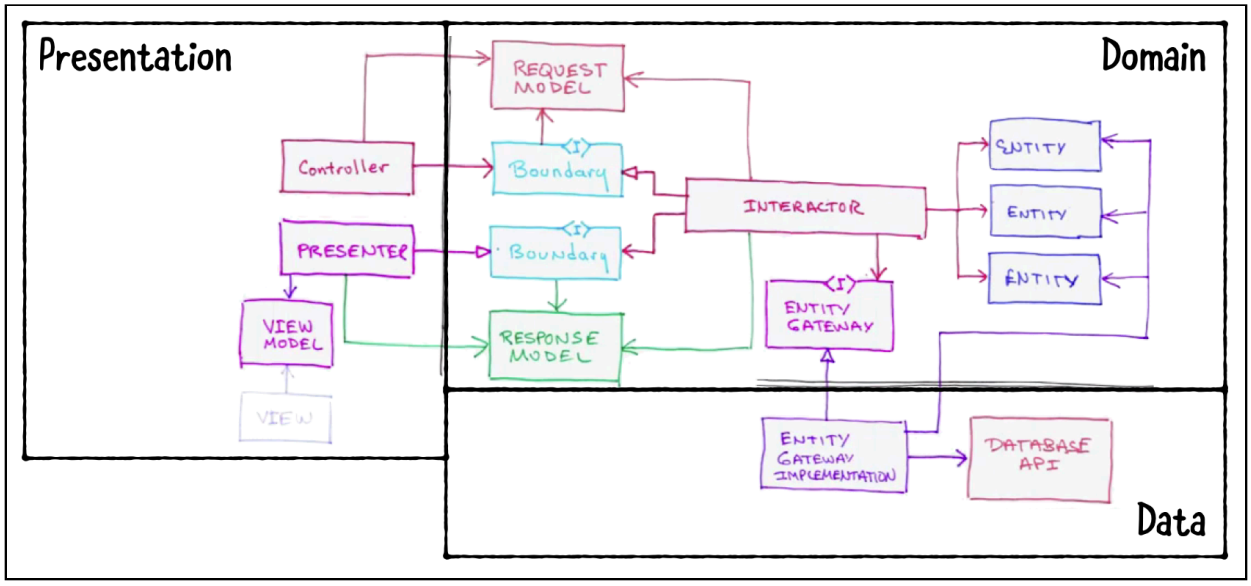
- develop-ability - to improve the ability of developers to add new features with high quality;
- deployability - to ease the manner in which software is delivered to users and testers;
- maintainability - once deployed, to reduce risk in making changes and increase the ease with which new features are added to software
- testability - to make software easier to test and ideally be built via test-driven development
- deferability - to allow software teams to defer decisions on certain details of their software until such decisions are necessary

To meet these goals, the Clean Architecture includes the following software principles, among many others:

- Single Responsibility Principle - code is grouped into items that change for the same reasons, separating concerns,
- components - the structures in which related software items are grouped,

- boundaries - there is a clear boundary between components,
- Dependency Inversion Principle - source code dependencies across boundaries are developed in the directions of abstractions. That is, lower level code depends on higher level code, and higher level code is independent of lower level code.

3) Uncle Bob Clean Architecture Design

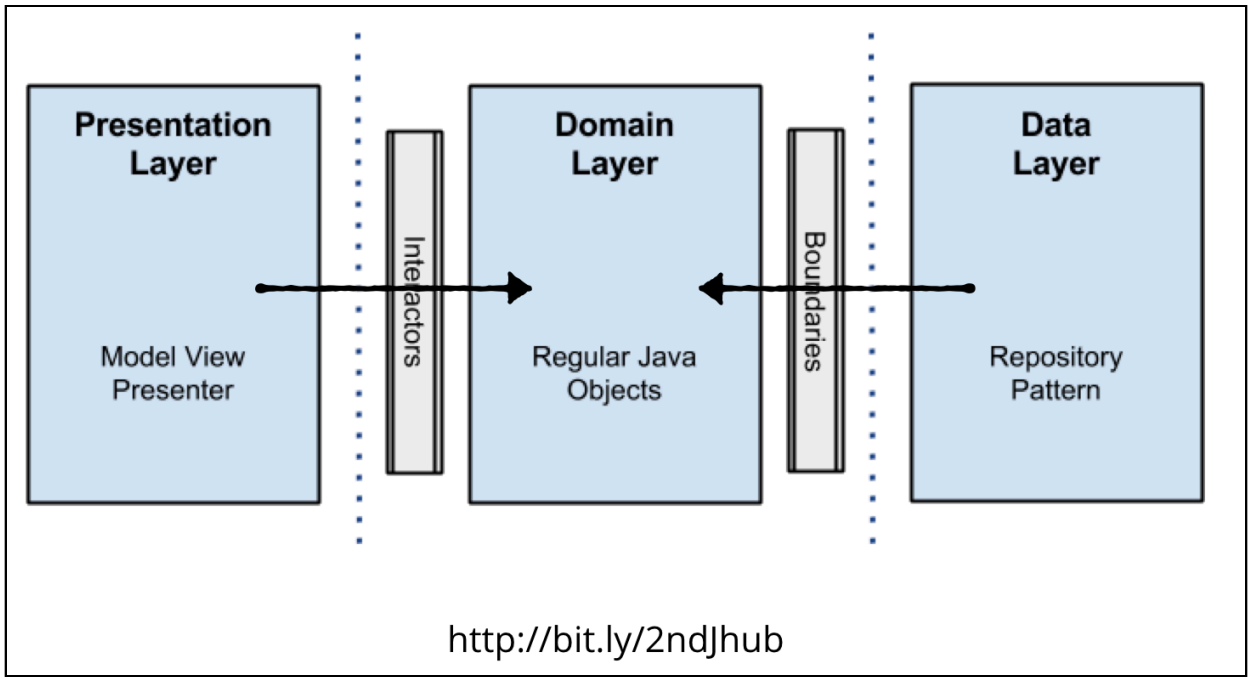


This diagram from Uncle Bob, represents a bit more detail on the Clean Architecture structure. You have entities in the core, use cases or "interactors" that depend on those entities, and boundaries that separate presentation and persistence from the use cases. Interfaces at the boundaries ensure that the dependency rules are being followed.

If we group code within the boundaries, we have the domain at the core, and presentation and data across the boundaries.

4) The Android Way

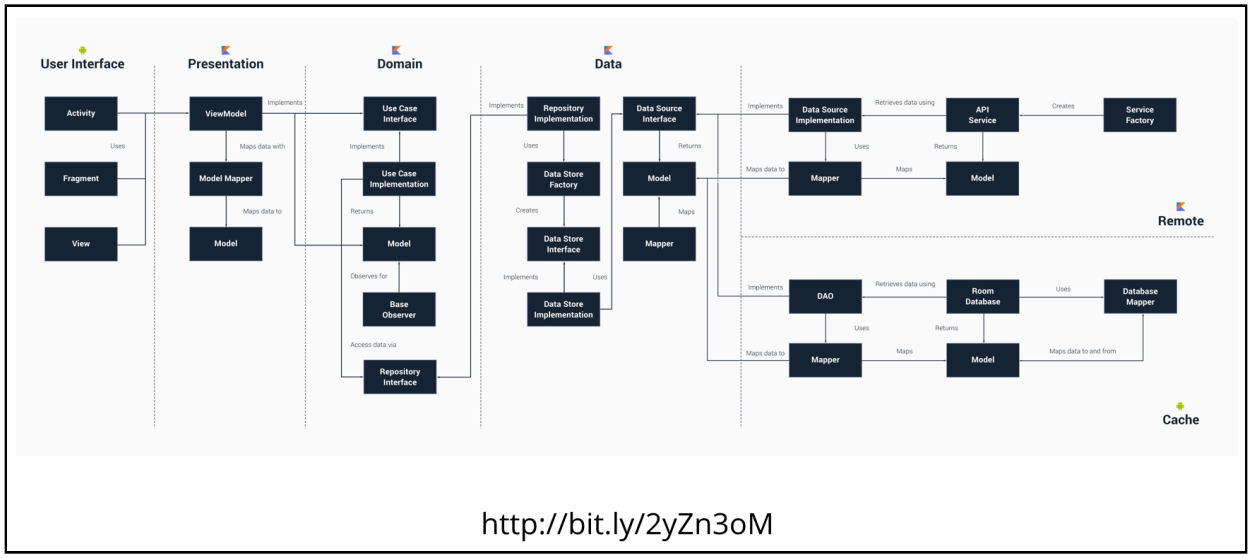
In fact, one of the first blog posts (http://bit.ly/2ndJhub) examining Clean Architecture on Android defined the various software layers in exactly this way.



http://bit.ly/2ndJhub

Source code dependencies in the presentation and data layers, point inwards to the domain layer. The presentation layer uses the MVP pattern, and the data layer the repository pattern. The domain layer consists purely of Java code, and so can be unit tested using only JUnit.

5) Buffer Clean Architecture Boilerplate



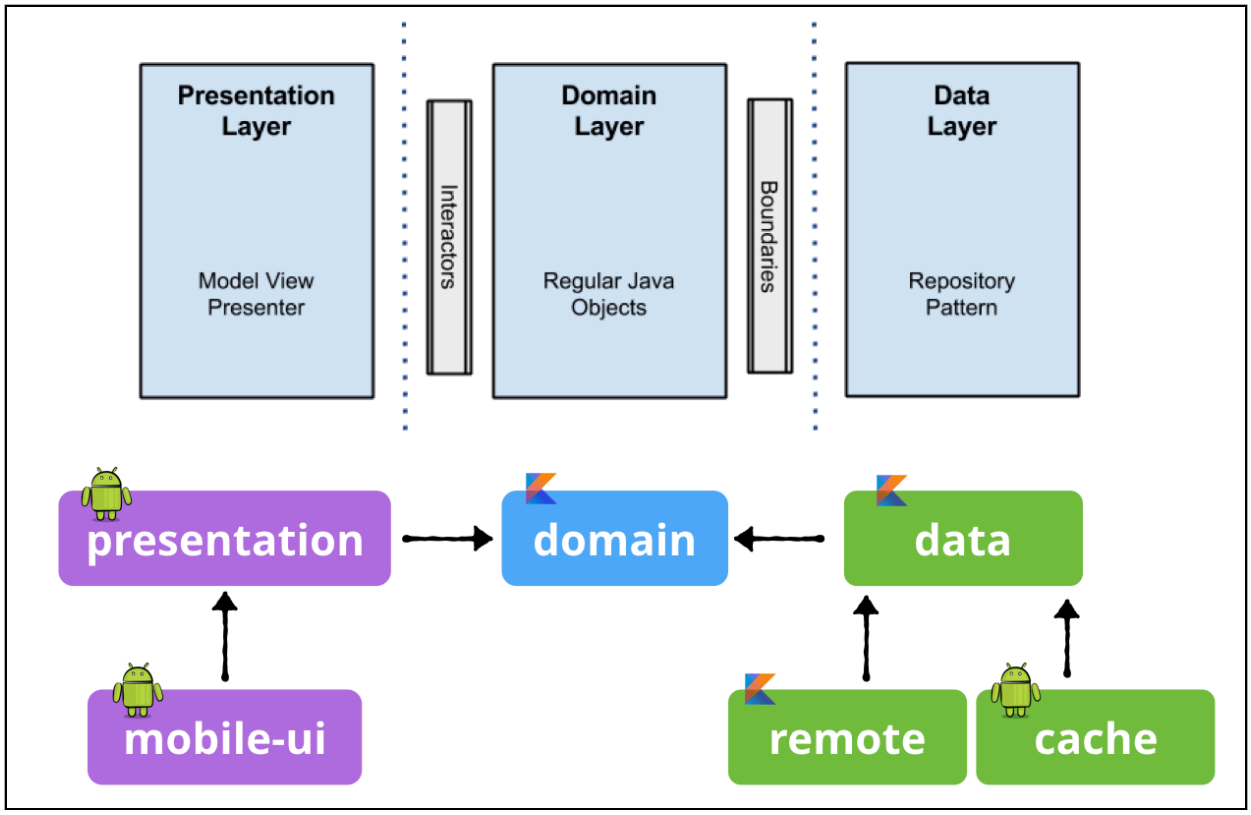
http://bit.ly/2yZn3oM

There have a number of examples of Clean Architecture on Android since that 2014 post. One of the more recent is this example (http://bit.ly/2yZn3oM) from Buffer, where they're using Kotlin, RxJava, and the recently announced Android

Architecture Components from Google.

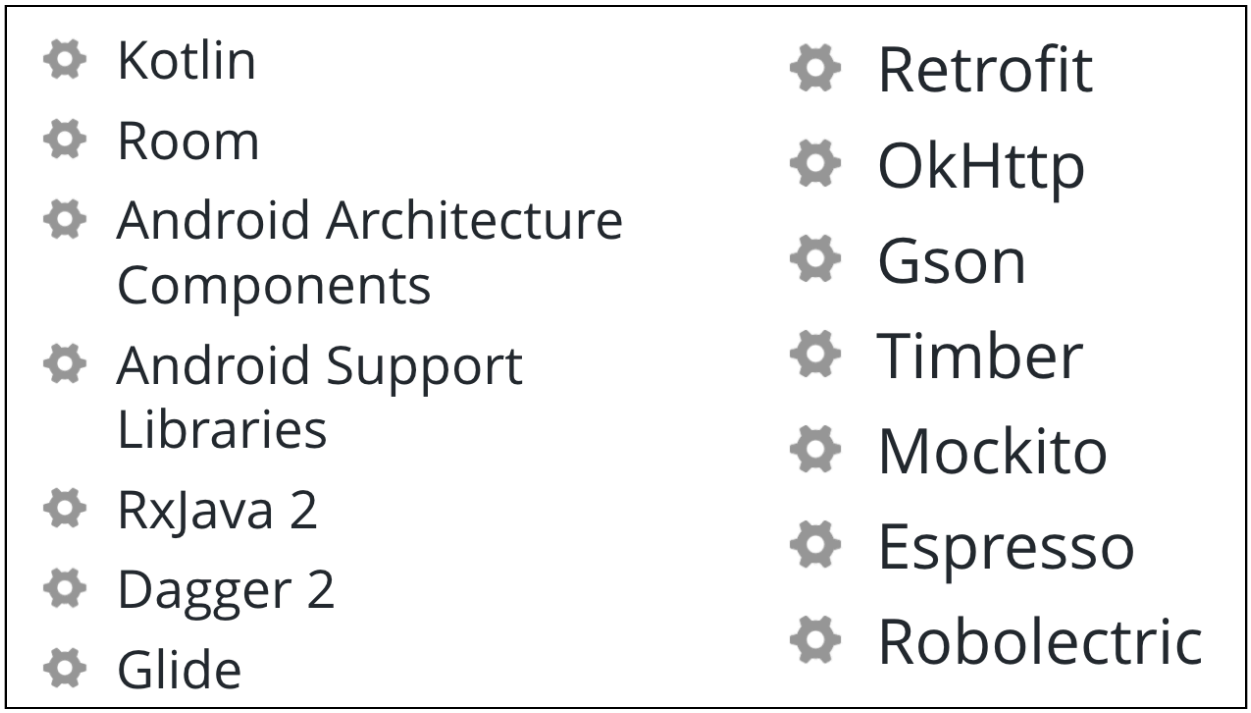
The sample app for this tutorial is a variation of that boilerplate example from Buffer. We'll examine it in detail and then augment it with a new use case. There's a lot going on in this diagram, but we'll see it's really not so bad.

6) Buffer Android Modules



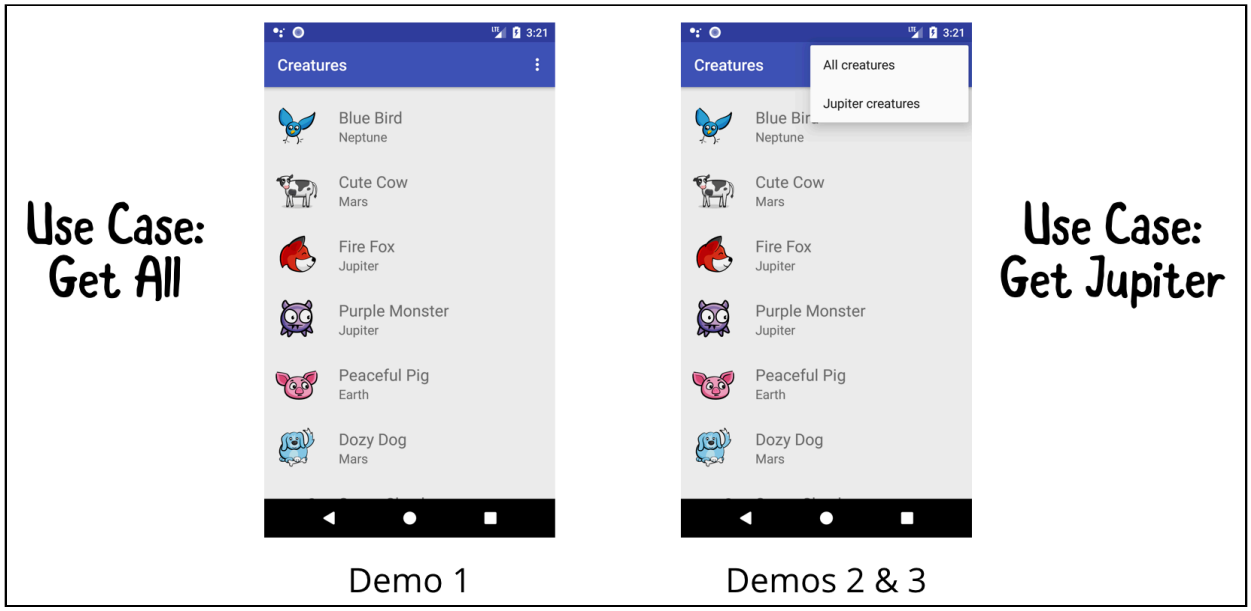
The Buffer clean architecture boilerplate project uses six software modules, domain, data, remote, cache, presentation, and ui. Here you can see it's really just a small expansion of the earlier example into a few more modules for convenience. But the code dependencies still flow inward to the domain module.

7) Tools Used



Here is a list for completeness of the tools used in the sample project. We're using Kotlin, Room for persistence, ViewModel and LiveData from Android Archtecture components, RxJava for data flow, and a number of other common Android libraries for networking and testing.

8) Demo Overview



The screenshot on the left is our sample app, written using Clean Architecture. It

shows a list of Creatures, each one having an avatar, name, and home planet.

In Demo 1, we'll look at the structure of the app into modules, and examine the only use case present in the app, which is simply to retrieve the list of creatures and show them to the user.

In Demos 2 and 3, we'll build a new use case in the app, which is to respond to a click on the app menu to filter the creatures shown to only those from the planet Jupiter.

9) Demo 1

We'll start by opening the project in Android Studio 3.0.1 or above. We see each of our modules: cache, data, domain, mobile-ui, presentation, and remote, along with the some of the build files for the app.

Gradle is the build tool used for Android. It's used both for building the software as well as dependency management. You see that each module has a "build.gradle" file, and there is a project level "build.gradle" also. We've defined a bunch of dependency values in "dependencies.gradle".

We'll start in the mobile-ui module, which kind of serves as the "main" function for the app, which Uncle Bob describes as the outermost layer of the structure. `MainActivity` is the screen that shows the list of creatures in a `RecyclerView`.

`onStart()` is one of the Android Activity lifecycle methods. Open **MainActivity.kt** and find the following:

```
undefined
```

Here you see `MainActivity` has a view model, `browseCreaturesViewModel`, and it's asking the view model for the list of creatures. The ViewModel is from the presentation layer.

ViewModel is one of the Architecture Components, and `getCreatures()` returns another Architecture Component, a `LiveData` item. We observe the `LiveData` to determine what state to show based on the `Resource`: loading, empty, or success.

Moving to the presentation layer to look at `BrowseCreaturesViewModel`. `getCreatures()` is implemented as shown below:

```
undefined
```

In `fetchCreatures()`, we're calling `execute(...)` on the `getCreatures` property, which is a `GetCreatures` instance from the domain layer. The definition of `GetCreatures` is below:

```
undefined
```

You can see `GetCreatures` is a `FlowableUseCase`. `Flowable` is an RxJava object, it's the RxJava 2 analog of an `Observable`, and `buildUseCaseObservable` returns a `Flowable` list of creatures.

Within `buildUseCaseObservable`, `GetCreatures` is asking the `creatureRepository` for creatures. This is a place in the code where the Dependency Inversion principle is critical. We're in the domain module, which is the highest level component, but we're making a call on a repository, which is in the data module.

By dependency inversion, we're making the call on an interface that's defined in the domain module itself, but implemented in the data module. That way, domain does not have a source code dependency on data, even though it can pass the flow of control into the data module.

`CreatureDataRepository` in the data module is the concrete class that implements the `CreatureRepository` interface from the domain module. `getCreatures()` is implemented below:

```
undefined
```

We're geting a data store from a factory and getting the creatures, and eventually saving creatures into the cache.

`retrieveDataStore(...)` returns either a cache store or remote data store depending on whether the data is cached and expired:

```
open fun retrieveDataStore(isCached: Boolean):
  CreatureDataStore {
    if (isCached && !creatureCache.isExpired()) {
      return retrieveCacheDataStore()
    }
    return retrieveRemoteDataStore()
}
```

The cache and remote data stores are defined in the cache and remote modules, respectively. We're using the Room library for persistence in the cache, and the Retrofit library to make remote data calls.

10) That's it!

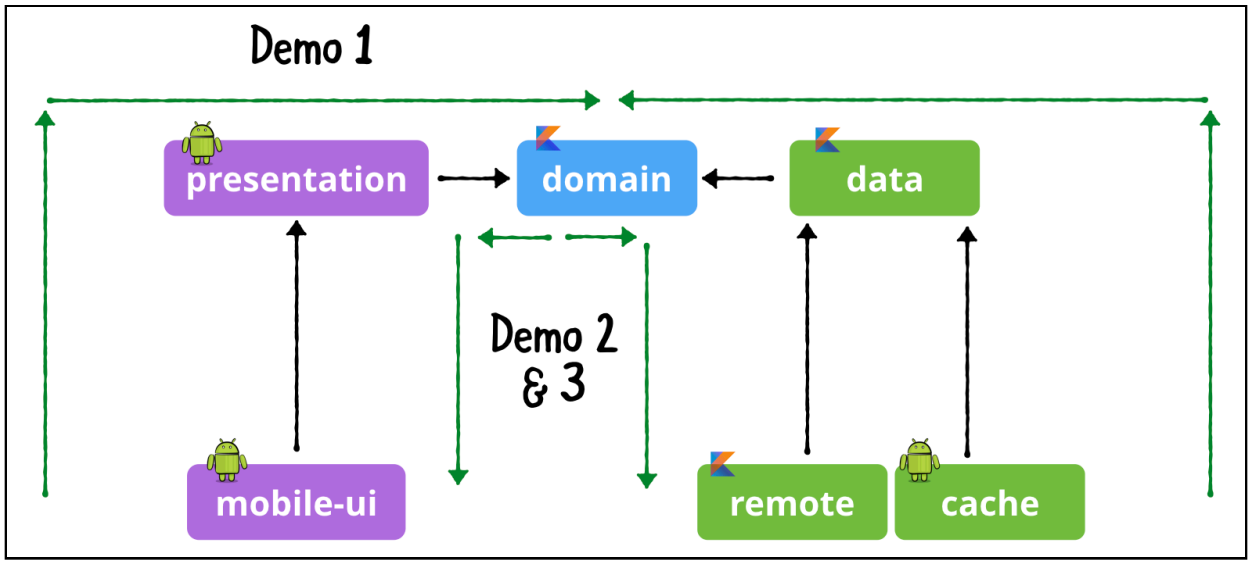
Congrats, at this time you should have a good understanding of the layers that go into a clean architecture and what an Android use case looks like at the code level! It's time to move onto building a new use case from scratch.

35 Clean Architecture on Android: Demo 2

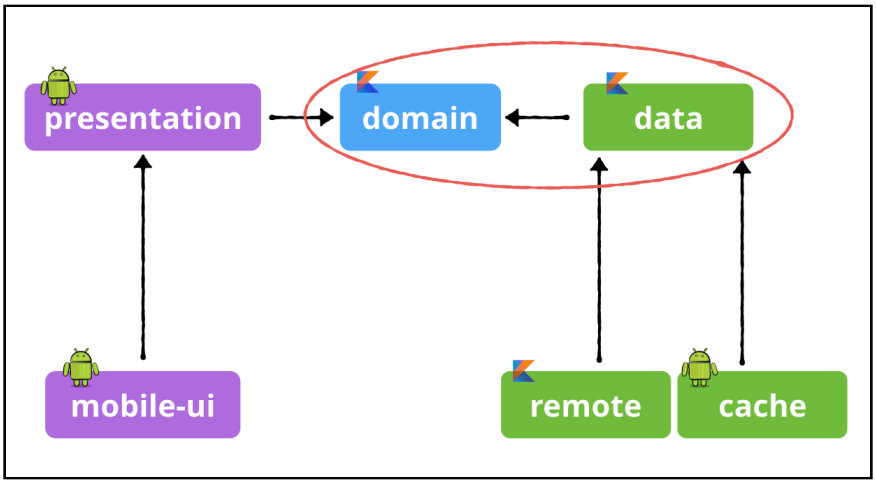
By Joe Howard

In this demo, you'll start building a new use case for the Creatures app. You're going to filter the creatures to only those from Jupiter.

We're going to treat Jupiter as an entirely new use case for the most part. There are other ways to handle this use case, for example, just filtering the creatures already being presented, but in the interest of time we're going to treat Jupiter as a separate use case so we can work with some of the existing code.



In Demo 1, you saw a use case from the "outside-in", that is, inwards towards the domain layer. In building a new use case, you'll start in the domain layer and move outwards.



In Demo 2, you'll work in the domain and data layers. In Demo 3, you'll finish the use case in the presentation and ui layers.

1) Demo 2

Domain Layer

Boundary

Start off in the domain module. First, add a new method into `CreatureRepository`, which forms the boundary with the data layer:

```
undefined
```

FlowableUseCase

Add a new Kotlin class to the interactor.browse package, named `GetJupiterCreatures`. This class is the `FlowableUseCase` for the new use case. Update the file to be:

```
undefined
```

So, we're calling into the `CreatureRepository` interface to `getJupiterCreatures()`.

Domain Layer Tests

Now you'll test the new use case within the domain layer. Add some factory methods into `CreatureFactory` (or uncomment if they exist using Cmd-/ on macOs or Ctrl-/ on Linux and Windows):

```
undefined
```

Now, uncomment the file `GetJupiterCreaturesTest` in the package `usecase.jupitercreatures`. Fix any necessary import statements if needed using Option+Return on macOS or Alt+Enter on Linux and Windows:

```
undefined
```

Run the new tests and you can see they all pass. The most relevant test added is `buildUseCaseObservableReturnsData()`, where the use case is being tested to return creatures from Jupiter.

Data Layer

Ok, with the domain layer taken care of, move to the data layer. While we're treating Jupiter as a new test case, in the interest of time we'll make a compromise in the data layer and re-use some existing code.

CreatureDataRepository

Add the following implementation of `getJupiterCreatures` interface method into the concrete `CreatureDataRepository` class, that meets the domain layer at the boundary:

```
undefined
```

The compromise we're making is, rather than call into the remote and cache layers, we're reusing the `getCreatures()` method to do so, and then filtering the results down to creatures from Jupiter. In a production app, if this were a real use case, you would call the remote to just pull down and cache new Jupiter creatures, and

just call the cache to get creatures from Jupiter.

Data Layer Tests

Add some new factory methods to `CreatureEntityFactory` in `test.factory`:

```
undefined
```

In the factory methods, we're adding both Jupiter and Mars creatures, and we'll test that only Jupiter creatures are present in the result from `getJupiterCreatures()`.

Now, use those factory methods in a new test that is added to `CreatureDataRepositoryTest`:

```
undefined
```

Run the test, and you'll see that creatures from Jupiter are coming from the data repository. If you change the string to "Mars" you'll see that the test fails, since the data repository is correctly only returning creatures from Jupiter.

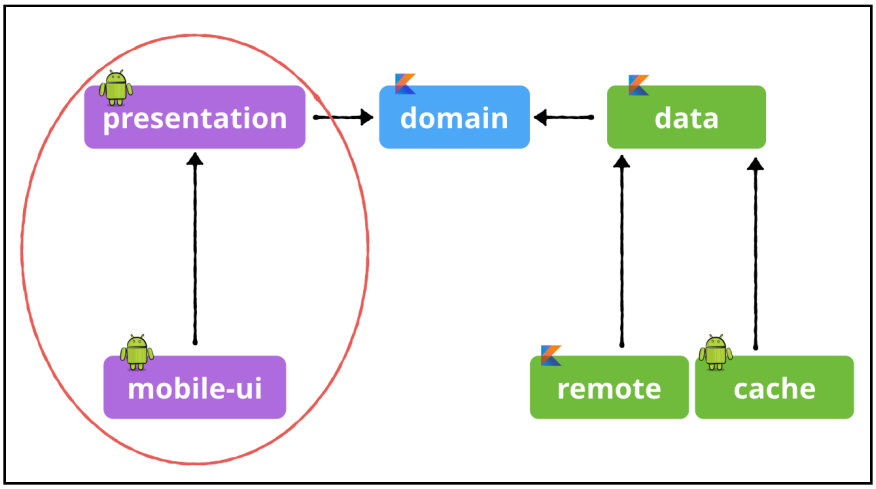
2) That's it!

Congrats, at this time you should have a good understanding of creating a use case in the domain and data layers of your app. In the final demo, you'll finish the use case in the presentation and ui layers.

36 Clean Architecture on Android: Demo 3

By Joe Howard

In this demo, you'll finish building the new use case for the Creatures app. We're filtering the creatures to only those from Jupiter.



In Demo 2, you worked in the domain and data layers. In Demo 3, you'll finish the use case in the presentation and ui layers.

1) Demo 3

Presentation Layer

ViewModel

First, add a `GetJupiterCreatures` parameter into the `BrowseCreaturesViewModel` primary constructor:

```
undefined
```

Next, add the `jupiterLiveData` property:

```
undefined
```

Next, add a new subscriber into `BrowseCreaturesViewModel`:

```
undefined
```

This subscriber posts values to the `jupiterLiveData` object.

Next, add `getJupiter` and `fetchJupiter` methods:

```
undefined
```

You need to update the constructor call in `BrowseCreaturesViewModelFactory`:

```
undefined
```

```
undefined
```

Presentation Layer Tests

Now you'll test the new use case within the presentation layer. Add some factory methods into `CreatureFactory` in the presentation layer:

```
undefined
```

In `BrowseCreaturesViewModelTest`, add a mock for the `GetJupiterCreatures`

```
undefined
```

Add the mock to the setup method and update the call to the view model constructor:

```
undefined
```

Now, you can add the test for the use case:

```
undefined
```

Run the new test and you can see that it passes.

UI Layer

Ok, with the presentation layer taken care of, move to the UI layer. Now the data grabbed by the presentation layer just needs to be displayed to the user.

You first need to update `BrowseActivityModule` for injection of the `GetJupiterCreatures` interactor:

```
undefined
```

```
undefined
```

Now the toasts that are shown when the menu items are chosen need to be replaced with calls to the view model. Open up `MainActivity`, and find `onOptionsItemSelected`. First, handle the case of Jupiter being selected. Replace the toast with:

```
undefined
```

Do similar for the case of all creatures being selected:

```
undefined
```

Build and run the `mobile-ui` configuration in the emulator or on device. Choose the menu item, and we can finally see the new user case in action in the app!

2) That's it!

Congrats, at this time you should have a good understanding of implementating a new use case with Clean Architecture. In Demo 2, you handled adding and testing the use case in the domain and data layers. In Demo 3, you've done the same in the presentation and UI layers.