Chapter 5: Building the Domain of an Android Application

In this chapter, we will analyze what the architecture of an Android application typically looks like and its three main layers (**presentation**, **domain**, and **data**). Then, we will learn how we can translate it into clean architecture and focus on the domain layer, which sits at the center of the architecture. Next, we will look at the role it plays in the architecture of an application and what its entities and use cases are. Finally, we will look at an exercise, in which we are going to see how we can set up an Android Studio project with multiple modules and use them to structure the domain layer.

In this chapter, we will cover the following topics:

- Introducing the app's architecture
- Creating the domain layer

By the end of this chapter, you will be familiar with the domain layer of an application, domain entities, and use cases.

Technical requirements

These are the hardware and software requirements:

- Android Studio – Arctic Fox | 2020.3.1 Patch 3

The code files for this chapter can be found here: https://github.com/PacktPublishing/Clean-Android-Architecture/tree/main/Chapter5 ↗️.

Check out the following video to see the Code in Action:
https://bit.ly/3826FH6 ↗️

Introducing the app's architecture

In this section, we will discuss the most common architecture that can be applied to an Android application and how it can be combined with **clean architecture** principles, and see how we should ideally structure our code base.

In the exercises from the previous chapters, we saw how, for an application that requires the integration of multiple data sources for networking and persistence, we had to put a lot of logic inside the `ViewModel` class. In those examples, `ViewModel` had multiple responsibilities, including fetching the data from the internet, persisting it locally, and holding the required information in the user interface. On top of these extra responsibilities, `ViewModel` also had many dependencies on the different data sources; this means that a change in the networking or persistence libraries would require a change in `ViewModel`. To solve this problem, our code would need to be split into separate layers with different responsibilities. Typically, the layers would look like the following figure:

Figure 5.1 – An app architecture diagram

In *Figure 5.1*, we can see that there are three layers with different responsibilities:

- **Presentation layer**: This layer is responsible for displaying data on the screen (also known as the **UI layer**). This usually contains classes required for managing the user interface and classes that will perform logic related to the user interface, such as `ViewModels`.
- **Domain layer**: This layer is responsible for fetching data from the data layer and performing business logic that can be reused across an app.
- **Data layer**: This layer is responsible for handling the business logic of an application that deals with the managing of data.

We can apply the *clean architecture* principles on top of the layered architecture by placing the domain layer at the center, as shown in *Figure 5.2*, and making it the place to store our *entities* and *use cases*. At the outer layers are the presentation and data layers, which are represented by the **interface adapter layer** (represented by `ViewModels` and `Repositories`) and the **framework layer** (represented by the user interface and persistence and networking frameworks):

Figure 5.2 – An app layer dependency diagram

In the preceding figure, we can see that the dependencies between the domain layer and the data layer are inverted. The domain layer will still draw data from the data layer, but because it has inverted dependencies, it will be less impacted by any changes to that layer, just as if any changes occur to the presentation layer, they will not impact the domain layer. If the app suffers from any changes to the use cases, then it will drive the changes in both the presentation and data layer.

To separate the layers, we can use **Android modules**. This will help us impose further rigor on the project by avoiding unwanted dependencies between the layers. This also helps improve build times in large applications because of Gradle build caching, which will only rebuild modules that had code changes. This will look something like the following figure:

Figure 5.3 – An app module diagram

We can see that there isn't a limited number of modules we need for each layer or that we should have corresponding modules between the three layers. The expansion of each layer can be driven by different factors such as the data sources, the uses of the app, the technologies, and protocols used in those data sources (using REST APIs for certain data and Bluetooth or Near-field communication for other data types). The usage of the use cases might be another factor (such as having a certain set of use cases for use with multiple applications). We might want to expand the presentation layer because of how certain screens are grouped to form certain isolated features and flows inside an application (such as a settings section of the application, or a login/sign-up flow). One interesting aspect to note is the **`:app`** module, which has the role of combining all of the dependencies and assembling them together. Here, we will gather all the required dependencies and initialize them.

An important thing to note here is that the modules aren't equivalent to the layers themselves; data modules can have dependencies to lower-level data modules. In fact, this situation will occur in scenarios when a module from a layer will need to have a dependency on another module from the same layer. If we were to create a dependency between the two, we might end up with a cyclical dependency, which is not wanted. In that

situation, we will need to create a common module between the two that will hold the required dependencies. For example, if we want to navigate from a screen in `:presentation1` to a screen in `:presentation2` or any of the other ones, we will need to create a new module on which all of the presentation modules will depend and which will store the data or logic required to handle the navigation. We will look at this issue in more detail when we discuss the presentation layer.

To create a new Android Studio module, you need to right-click the project in Android Studio, select **New**, and then **Module**, as shown in the following figure:

Figure 5.4 – Creating a new Android Studio module

You will then be prompted to select the type of module, and depending on the functionality, you can select **Android Library** if the module doesn't contain code from the Android framework or **Java or Kotlin Library** if

the module doesn't have any dependencies to the Android framework. In the exercises that follow, we will be using Android libraries. Once the module is created, it will already contain a set of generated files and folders. One of the most relevant ones will be the **`build.gradle`** file. The plugin section in the file will indicate that an Android library was created:

```
plugins {
    id 'com.android.library'
    …
}
```

If we want to add a dependency to the newly created module, we can use the following in the **`app`** module:

```
dependencies {
    implementation(project(path: ":my-new-module"))
    …
}
```

The syntax to add a dependency to a module is similar to the syntax to add an external dependency and it's through the Gradle **`implementation`** method. The rest indicates that the **`app`** module will depend on another module inside the same project.

In this section, we have analyzed the layers of Android app architecture and how we can apply clean architecture principles to these layers. In the following section, we will look at how we can build a domain layer.

Creating the domain layer

In this section, we will discuss how to build the domain layer and what goes into it through certain examples. Finally, we will look at an exercise in which a domain layer is created.

Because the domain layer sits at the center of the application, it will need to have a minimal number of dependencies. This means that the Gradle modules that form the domain layer will need to be the most stable modules in the project. This is to avoid causing other modules to change because of a change that occurred in a dependency that the domain modules use. The domain should be responsible for defining the entities and use cases for the application.

Entities are represented by objects that hold data and are mainly immutable. Let's assume we want to represent a user as an entity. We might end up with something like the following:

```
data class User(
    val id: String,
    val firstName: String,
    val lastName: String,
    val email: String
) {
    fun getFullName() = "$firstName $lastName"
}
```

Here, we use a simple **`data class`**, and we declare all our fields immutable with the **`val`** keyword. We also have a business logic function for this object, which will return the full name of the user.

Next, we need to define our use cases. Because the use cases will need to get data from the data layer, we will first need to create an abstraction for our repository, and we will end up with the following:

```
interface UserRepository {
    fun getUser(id: String): User
}
```

Here, we just have a simple method that will return a user based on **`id`**. We can now create a use case for retrieving the user:

```
class GetUserUseCase(private val userRepository:
UserRepository) {
    fun getUser(id: String) =
userRepository.getUser(id)
}
```

In the preceding example, we define a use case to retrieve the user, which will have a dependency on the **`UserRepository`**, which will be used to retrieve the user information. If we look at the preceding example, we can see a bit of redundancy because the use case doesn't have any extra logic and just returns the value of the repository. The benefit of use cases comes when we want to combine multiple results of multiple repositories.

Let's assume that we want to associate the user with a particular location, defined as follows:

```
data class Location(
    val id: String,
    val userId: String,
    val lat: Double,
    val long: Double
)
```

Here, we just keep the latitude and longitude associated with a particular user. Now, let's assume that we would have a repository for the different locations:

```
interface LocationRepository {
    fun getLocation(userId: String): Location
}
```

Here, we again have an abstraction of a repository with a method to get a specific location based on **`userId`**. If we want to get a user and an associated location, we will need to create a specific use case for this:

```
class GetUserWithLocationUseCase(
    private val userRepository: UserRepository,
```

```
    private val locationRepository:
LocationRepository
) {
    fun getUser(id: String) =
        UserWithLocation(userRepository.getUser(id),
locationRepository.getLocation(id))
}
data class UserWithLocation(
    val user: User,
    val location: Location
)
```

In the preceding example, we create a new entity called **`UserWithLocation`**, which will store **`User`** and **`Location`**. **`UserWithLocation`** will then be used as a result for the **`getUser`** method in **`GetUserWithLocationUseCase`**. This will depend on both **`UserRepository`** and **`LocationRepository`** to fetch the relevant data.

We can further improve the use cases by handling the threading as well. Because use cases will mainly deal with retrieving and managing data, which needs to be asynchronous, we should handle this on a separate thread. We can use **Kotlin flows** to manage this, and we might end up with something like this for the repositories:

```
interface UserRepository {
    fun getUser(id: String): Flow<User>
}
interface LocationRepository {
    fun getLocation(id: String): Flow<Location>
}
```

Here, we change the return types of the methods to a Kotlin flow, which might emit a stream of data or a single item. Now, we can combine the different flows in the stream in the use case:

```
class GetUserWithLocationUseCase(
    private val userRepository: UserRepository,
```

```
    private val locationRepository:
LocationRepository
) {
    fun getUser(id: String) = combine(
        userRepository.getUser(id),
        locationRepository.getLocation(id)
    ) { user, location ->
        UserWithLocation(user, location)
    }.flowOn(Dispatchers.IO)
}
```

Here, we combine the **`User`** and **`Location`** flows into a **`UserWithLocation`** flow, and we will execute the data fetching on the **`IO`** dispatcher.

Often, when dealing with data loading and management, especially from the internet, we can encounter different errors, which we will have to factor into our use cases. To solve this, we can define error entities. There are many possibilities to define them, including extending the **`Throwable`** class, defining a particular data class, a combination of the two, or combining them with sealed classes:

```
sealed class UseCaseException(override val cause:
Throwable?) : Throwable(cause) {
    class UserException(cause: Throwable) :
        UseCaseException(cause)

    class LocationException(cause: Throwable) :
        UseCaseException(cause)

    class UnknownException(cause: Throwable) :
        UseCaseException(cause)
    companion object {
        fun extractException(throwable: Throwable):
            UseCaseException {
            return if (throwable is UseCaseException)
```

```
                    throwable else
    UnknownException(throwable)
            }
        }
    }
```

Here, we have created a sealed class that will have as subclasses a dedicated error for each entity, plus an unknown error that will deal with errors we haven't accounted for, and a companion method that will check a **`Throwable`** object and return **`UnknownException`** for any **`Throwable`** that isn't **`UseCaseException`**. We will need to make sure that the error is propagated through the flow stream, but first, we can combine the entity for success with the entity for error to ensure that the consumer of the use case will not need to check the type of **`Throwable`** again and make a cast. We can do this with the following approach:

```
sealed class Result<out T : Any> {
    data class Success<out T : Any>(val data: T) :
        Result<T>()
    class Error(val exception: UseCaseException) :
        Result<Nothing>()
}
```

Here, we defined a **`Result`** sealed class, which will have two subclasses for success and error. The **`Success`** class will hold the relevant data for the use case, and the **`Error`** class will contain the exceptions defined before. The **`Error`** class can be further expanded if needed to hold data as well as the error if we want to display the cached or persisted data as a placeholder. We can now modify the use case to incorporate the **`Result`** class and the error state:

```
class GetUserWithLocationUseCase(
    private val userRepository: UserRepository,
    private val locationRepository:
LocationRepository
) {
    fun getUser(id: String) = combine(
```

```
            userRepository.getUser(id),
            locationRepository.getLocation(id)
        ) { user, location ->
            Result.Success(UserWithLocation(user,
    location)) as
                Result<UserWithLocation>
        }.flowOn(Dispatchers.IO)
            .catch {
                emit(Result.Error(UseCaseException.
                    extractException(it)))
            }
    }
```

Here, we return **`Result.Success`**, which will hold the **`UserWithLocation`** object if no errors occur, and add use the **`catch`** operator to emit **`Result.Error`** with **`UseCaseException`** that occurred while fetching the data. Because these operations will repeat for multiple use cases, we can use abstraction to create a template for how each use case behaves and let the implementations deal with only processing the necessary data. An example might look like the following:

```
abstract class UseCase<T : Any, R : Any>(private val
dispatcher: CoroutineDispatcher) {
    fun execute(input: T): Flow<Result<R>> =
        executeData(input)
        .map {
            Result.Success(it) as Result<R>
        }
        .flowOn(dispatcher)
        .catch {
            emit(Result.Error(UseCaseException.
                extractException(it)))
        }
    internal abstract fun executeData(input: T):
Flow<R>
}
```

In the preceding example, we have defined an abstract class that will contain the **`execute`** method, which will invoke the abstract **`executeData`** method and then map the result of that method into a **`Result`** object, followed by setting the flow on a **`CoroutineDispatcher`**, and finally, handling the errors in the **`catch`** operator. The implementation of this will look like the following. Note that the **`internal`** keyword for the **`executeData`** method will only make the method accessible in the current module. This is because we only want the **`execute`** method to be called by the users of this use case:

```
class GetUserWithLocationUseCase(
    dispatcher: CoroutineDispatcher,
    private val userRepository: UserRepository,
    private val locationRepository:
LocationRepository
) : UseCase<String, UserWithLocation>(dispatcher) {
    override fun executeData(input: String):
        Flow<UserWithLocation> {
        return combine(
            userRepository.getUser(input),
            locationRepository.getLocation(input)
        ) { user, location ->
            UserWithLocation(user, location)
        }
    }
}
```

In this example, **`GetUserWithLocationUseCase`** will only have to deal with returning the necessary data relevant to the use case in the **`executeData`** method. We can use generics to bind the types of data we want the use case to process by introducing further abstractions for the required input and output of it:

```
abstract class UseCase<T : UseCase.Request, R :
UseCase.Response>(private val dispatcher:
CoroutineDispatcher) {
```

```
        …
        interface Request
        interface Response
    }
```

Here, we have bound the generics in the **`UseCase`** class to two interfaces – **`Request`** and **`Response`**. The former is represented by the input data required by the use case, and the latter is represented by the output of the use case. The implementation will now look like this:

```
    class GetUserWithLocationUseCase(
        dispatcher: CoroutineDispatcher,
        private val userRepository: UserRepository,
        private val locationRepository:
    LocationRepository
        ) : UseCase<GetUserWithLocationUseCase.Request,
        GetUserWithLocationUseCase.Response>(dispatcher)
    {
        override fun executeData(input: Request): Flow
            <Response> {
            return combine(
                userRepository.getUser(input.userId),
                locationRepository.getLocation(input.user
    Id)
            ) { user, location ->
                Response(UserWithLocation(user,
    location))
            }
        }
        data class Request(val userId: String) : UseCase.
            Request
        data class Response(val userWithLocation:
            UserWithLocation) : UseCase.Response
    }
```

Here, we provided the implementations for the **`Request`** and **`Response`** classes and used them when extending from the base class. In this case,

the `Request` and `Response` classes represent **data transport objects**. When we create templates for use cases, it is important to observe their evolution because as the complexity increases, the template may become inadequate.

Often, we will have the opportunity to build a new use case from existing smaller use cases. Let's assume that for retrieving a user and retrieving a location, we have two separate use cases:

```
class GetUserUseCase(
    dispatcher: CoroutineDispatcher,
    private val userRepository: UserRepository
    ) : UseCase<GetUserUseCase.Request,
    GetUserUseCase.Response>(dispatcher) {
    override fun executeData(input: Request): Flow
        <Response> {
        return userRepository.getUser(input.userId)
            .map {
                Response(it)
            }
    }
    data class Request(val userId: String) : UseCase.
        Request
    data class Response(val user: User) :
UseCase.Response
}
class GetLocationUseCase(
    dispatcher: CoroutineDispatcher,
    private val locationRepository:
LocationRepository
    ) : UseCase<GetLocationUseCase.Request,
    GetLocationUseCase.Response>(dispatcher) {
    override fun executeData(input: Request): Flow
        <Response> {
```

```
        return
locationRepository.getLocation(input.userId)
            .map {
                Response(it)
            }
    }
    data class Request(val userId: String) : UseCase
        .Request
    data class Response(val location: Location) :
UseCase.
        Response
}
```

In the preceding examples, we have two classes for each use case to retrieve a user and a location.

We can modify **`GetUserWithLocationUseCase`** to instead use the existing use cases, like this:

```
class GetUserWithLocationUseCase(
    dispatcher: CoroutineDispatcher,
    private val getUserUseCase: GetUserUseCase,
    private val getLocationUseCase:
GetLocationUseCase
    ) : UseCase<GetUserWithLocationUseCase.Request,
        GetUserWithLocationUseCase.Response>
(dispatcher) {
    override fun executeData(input: Request): Flow
        <Response> {
        return combine( getUserUseCase.executeData
                (GetUserUseCase.Request(input.userId)
),
            getLocationUseCase.executeData
                (GetLocationUseCase.Request(input.use
rId))
        ) { userResponse, locationResponse ->
```

```
                Response(UserWithLocation(userResponse.us
er,
                    locationResponse.location))
            }
        }
        data class Request(val userId: String) : UseCase
            .Request
        data class Response(val userWithLocation:
             UserWithLocation) : UseCase.Response
    }
```

Here, we changed the dependencies to instead use two existing use cases instead of the repositories, invoked the **`executeData`** method from each one, and then built a new **`Response`** using the responses from both use cases.

In this section, we looked at how to build a domain layer with entities, use cases, and abstractions for repositories. In the section that follows, we will look at an exercise related to building a domain layer.

Exercise 05.01 – Building a domain layer

In this exercise, we will create a new project in **Android Studio** in which a new Gradle module named **`domain`** will be created. This module will contain entities containing the following data:

- **`User`**: This will have an ID of the **`Long`** type and a name, username, and email.
- **`Post`**: This will have an ID and a user ID as a **`Long`** type, a title, and body.
- **`Interaction`**: This will contain the total number of interactions with the app.
- **`Errors`**: This is for when posts or users cannot be loaded.

The following use cases will need to be defined for the application:

- Retrieving a list containing posts with user information, grouped with the interaction data
- Retrieving information about a particular user based on the ID
- Retrieving information about a particular post based on the ID
- Updating the interaction data

To complete this exercise, you will need to do the following:

- Create a new project in Android Studio.
- Create a mapping of all the library dependencies and the versions in the root **`build.gradle`** file.
- Create the **`domain`** module in Android Studio.
- Create the required entities for the data and errors.
- Create the **`Result`** class, which will hold the success and error scenario.
- Create the repository abstractions to obtain the user, post, and interaction information.
- Create the four required use cases.

Follow these steps to complete the exercise:

1. Create a new project in Android Studio and select **Empty Compose Activity**.
2. In the root **`build.gradle`** file, add the following configurations that will be used for all the modules in the project:

```
buildscript {
    ext {
        javaCompileVersion =
JavaVersion.VERSION_1_8
        jvmTarget = "1.8"
        defaultCompileSdkVersion = 31
        defaultTargetSdkVersion = 31
        defaultMinSdkVersion = 21
        …
}
```

3. In the same file, add the versions of the libraries that will be used by the Gradle modules:

```
buildscript {
    ext {
        …
        versions = [
                androidGradlePlugin: "7.0.4",
                kotlin             : "1.5.31",
                hilt               : "2.40.5",
                coreKtx            : "1.7.0",
                appCompat          : "1.4.1",
                compose            : "1.0.5",
                lifecycleRuntimeKtx: "2.4.0",
                activityCompose    : "1.4.0",
                material           : "1.5.0",
                coroutines         : "1.5.2",
                junit              : "4.13.2",
                mockito            : "4.0.0",
                espressoJunit      : "1.1.3",
                espressoCore       : "3.4.0"
        ]
        …
}
```

4. In the same file, add a mapping for the plugin dependencies that the entire project will use:

```
buildscript {
    ext {
        …
        gradlePlugins = [
                android: "com.android.tools.build:
                    gradle:${versions.
                        androidGradlePlugin}",
                kotlin :
"org.jetbrains.kotlin:kotlin-
```

```
                    gradle-
plugin:${versions.kotlin}",
                hilt   : "com.google.dagger:hilt-
                    android-gradle-plugin:
                        ${versions.hilt}"
        ]
        …
}
```

5. Next, you will need to add the dependencies to the **androidx** libraries:

```
buildscript {
    ext {
        …
        androidx = [
                core                    :
"androidx.core:core-ktx:${versions.coreKtx}",
                appCompat               :
"androidx.appcompat:appcompat:${versions.appCompat}
",
                composeUi               :
"androidx.compose.ui:ui:${versions.compose}",
                composeMaterial         :
"androidx.compose.material:material:${versions.comp
ose}",
                composeUiToolingPreview:
"androidx.compose.ui:ui-tooling-
preview:${versions.compose}",
                lifecycleRuntimeKtx     :
"androidx.lifecycle:lifecycle-runtime-
ktx:${versions.lifecycleRuntimeKtx}",
                composeActivity         :
"androidx.activity:activity-
compose:${versions.activityCompose}"
        ]
        …
}
```

6. Next, add the remaining libraries for material design, dependency injection, and tests:

```
buildscript {
    ext {
        …
        material = [
                material: "com.google.android.
                    material:material:$
                        {versions.material}"
        ]
        coroutines = [
                coroutinesAndroid: "org.jetbrains.
                    kotlinx:kotlinx-coroutines-
                       android:${versions.coroutine
s}"
        ]
        di = [
                hiltAndroid :
"com.google.dagger:hilt-
                    android:${versions.hilt}",
                hiltCompiler:
"com.google.dagger:hilt-
                    compiler:${versions.hilt}"
        ]
        test = [
                junit     :
                    "junit:junit:${versions.junit}"
,
                coroutines: "org.jetbrains.kotlinx:
                    kotlinx-coroutines-test:
                        ${versions.coroutines}",
                mockito   : "org.mockito.kotlin:
                   mockito-
kotlin:${versions.mockito}"
        ]
```

```
        androidTest = [
                junit             :
"androidx.test.ext
                    :junit:${versions.espressoJunit
}",
                espressoCore      : "androidx.test.
                    espresso:espresso-core:$
                        {versions.espressoCore}",
                composeUiTestJunit:
"androidx.compose.
                ui:ui-test-
junit4:${versions.compose}"
        ]
    }
    …
}
```

7. In the same file, you will need to replace the previous mappings as plugin dependencies:

```
buildscript {
     …
    dependencies {
        classpath gradlePlugins.android
        classpath gradlePlugins.kotlin
        classpath gradlePlugins.hilt
    }
}
```

8. Now, you need to switch to the **`build.gradle`** file in the app module and change the existing configurations with the ones defined in the top-level **`build.gradle`**:

```
android {
    compileSdk defaultCompileSdkVersion
    defaultConfig {
        …
        minSdk defaultMinSdkVersion
        targetSdk defaultTargetSdkVersion
```

```
        versionCode 1
        versionName "1.0"
        …
    }
    …
    compileOptions {
        sourceCompatibility javaCompileVersion
        targetCompatibility javaCompileVersion
    }
    kotlinOptions {
        jvmTarget = jvmTarget
        useIR = true
    }
    buildFeatures {
        compose true
    }
    composeOptions {
        kotlinCompilerExtensionVersion
            versions.compose
    }
    …
}
```

9. In the same file, you will need to replace the dependencies with the ones defined in the top-level **`build.gradle`** file:

```
dependencies {
    implementation androidx.core
    implementation androidx.appCompat
    implementation material.material
    implementation androidx.composeUi
    implementation androidx.composeMaterial
    implementation androidx.composeUiToolingPreview
    implementation androidx.lifecycleRuntimeKtx
    implementation androidx.composeActivity
    testImplementation test.junit
}
```

10. In Android Studio, execute **Sync Project with Gradle Files** command and then the **Make Project** command to make sure that the project builds without any errors.
11. Create a new module for the project named **`domain`**, which will be an Android library module.
12. In the **`build.gradle`** file of the **`domain`** module, make sure you have the following plugins:

```
plugins {
    id 'com.android.library'
    id 'kotlin-android'
    id 'kotlin-kapt'
    id 'dagger.hilt.android.plugin'
}
```

13. In the same file, make sure you use the configurations defined in the top-level **`build.gradle`** file:

```
android {
    compileSdk defaultCompileSdkVersion
    defaultConfig {
        minSdk defaultMinSdkVersion
        targetSdk defaultTargetSdkVersion
        …
    }
    …
    compileOptions {
        sourceCompatibility javaCompileVersion
        targetCompatibility javaCompileVersion
    }
    kotlinOptions {
        jvmTarget = jvmTarget
    }
}
```

14. In the same file, you will need to add the following dependencies:

```
dependencies {
    implementation coroutines.coroutinesAndroid
    implementation di.hiltAndroid
```

```
        kapt di.hiltCompiler
        testImplementation test.junit
        testImplementation test.coroutines
        testImplementation test.mockito
    }
```

15. Sync the project with Gradle files and build the project again to make sure that the Gradle configuration is correct.
16. In the **`domain`** module, create a new package named **`entity`**.
17. In the **`entity`** package, create a class named **`Post`**, which will have **`id`**, **`userId`**, **`title`**, and **`body`**:

```
data class Post(
    val id: Long,
    val userId: Long,
    val title: String,
    val body: String
)
```

18. In the same package, create the **`User`** class, which will have **`id`**, **`name`**, **`username`**, and **`email`**:

```
data class User(
    val id: Long,
    val name: String,
    val username: String,
    val email: String
)
```

19. Next, create a class named **`PostWithUser`**, which will contain the **`post`** and **`user`** information:

```
data class PostWithUser(
    val post: Post,
    val user: User
)
```

20. In the same package, create a class called **`Interaction`**, which will contain the total number of clicks:

```
data class Interaction(val totalClicks: Int)
```

21. Now, we need to create the error entities:

```
sealed class UseCaseException(cause: Throwable) :
Throwable(cause) {
    class PostException(cause: Throwable) :
        UseCaseException(cause)
    class UserException(cause: Throwable) :
        UseCaseException(cause)
    class UnknownException(cause: Throwable) :
        UseCaseException(cause)
    companion object {
        fun createFromThrowable(throwable:
Throwable):
            UseCaseException {
            return if (throwable is
UseCaseException)
            throwable else
UnknownException(throwable)
        }
    }
}
```

Here, we have exceptions defined for when there will be an issue with loading the post and user information, and **`UnknownException`**, which will be emitted when something else goes wrong.

22. Next, let's create the **`Result`** class, which will hold the success and error information:

```
sealed class Result<out T : Any> {
    data class Success<out T : Any>(val data: T) :
        Result<T>()
    class Error(val exception: UseCaseException) :
        Result<Nothing>()
}
```

23. Now, we need to move on to defining the abstractions for the repositories, and to do so, we create a new package named **`repository`**.

24. In the **repository** package, create an interface for managing the post data:

```
interface PostRepository {
    fun getPosts(): Flow<List<Post>>
    fun getPost(id: Long): Flow<Post>
}
```

25. In the same package, create an interface for managing the user data:

```
interface UserRepository {
    fun getUsers(): Flow<List<User>>
    fun getUser(id: Long): Flow<User>
}
```

26. In the same package, create an interface for managing the interaction data:

```
interface InteractionRepository {
    fun getInteraction(): Flow<Interaction>
    fun saveInteraction(interaction: Interaction):
        Flow<Interaction>
}
```

27. Now, we move on to the use cases and start by creating a new package named **usecase**.
28. In this package, create the **UseCase** template:

```
abstract class UseCase<I : UseCase.Request, O :
UseCase.Response>(private val configuration:
Configuration) {
    fun execute(request: I) = process(request)
        .map {
            Result.Success(it) as Result<O>
        }
        .flowOn(configuration.dispatcher)
        .catch {
            emit(Result.Error(UseCaseException.
                createFromThrowable(it)))
        }
    internal abstract fun process(request: I):
Flow<O>
```

```
        class Configuration(val dispatcher:
            CoroutineDispatcher)
        interface Request
        interface Response
    }
```

In this template, we have defined the data transfer objects' abstraction, as well as a **`Configuration`** class that holds **`CoroutineDispatcher`**. The reason for this **`Configuration`** class is to be able to add other parameters for the use case without modifying the **`UseCase`** subclasses. We have one **`abstract`** method, which will be implemented by the subclasses to retrieve the data from the repositories, and the **`execute`** method, which will take the data and convert it to **`Result`**, handle the error scenarios, and set the proper **`CoroutineDispatcher`**.

29. In the **`usecase`** package, create the use case to retrieve the list of posts with the user information and the interaction data:

```
class GetPostsWithUsersWithInteractionUseCase
@Inject constructor(
    configuration: Configuration,
    private val postRepository: PostRepository,
    private val userRepository: UserRepository,
    private val interactionRepository:
        InteractionRepository
    ) : GetPostsWithUsersWithInteractionUseCase
    GetPostsWithUsersWithInteractionUseCase {
    override fun process(request: Request):
        Flow<Response> =
        combine(
            postRepository.getPosts(),
            userRepository.getUsers(),
            interactionRepository.getInteraction()
        ) { posts, users, interaction ->
            val postUsers = posts.map { post ->
                val user = users.first {
```

```
                    it.id == post.userId
                }
                PostWithUser(post, user)
            }
            Response(postUsers, interaction)
        }
    object Request : UseCase.Request
    data class Response(
        val posts: List<PostWithUser>,
        val interaction: Interaction
    ) : UseCase.Response
}
```

In this class, we extend the **`UseCase`** class, and in the **`process`** method, we combine the posts, users, and interaction flows. Because there is no input required, the **`Request`** class will have to be empty, and the **`Response`** class will contain a list of combined user and post information as well as the interaction data. The **`@Inject`** annotation will help us inject this use case later in the presentation layer.

30. In the same package, create the use case to retrieve a post by ID:

```
class GetPostUseCase @Inject constructor(
    configuration: Configuration,
    private val postRepository: PostRepository
    ) : UseCase<GetPostUseCase.Request,
    GetPostUseCase.Response>(configuration) {
    override fun process(request: Request): Flow
        <Response> =
        postRepository.getPost(request.postId)
            .map {
                Response(it)
            }
    data class Request(val postId: Long) : UseCase.
        Request
    data class Response(val post: Post) : UseCase.
```

```
            Response
    }
```

31. In the same package, create the use case to retrieve a user by ID:

```
    class GetUserUseCase @Inject constructor(
        configuration: Configuration,
        private val userRepository: UserRepository
        ) : UseCase<GetUserUseCase.Request,
        GetUserUseCase.Response>(configuration) {
        override fun process(request: Request): Flow
            <Response> =
            userRepository.getUser(request.userId)
                .map {
                    Response(it)
                }
        data class Request(val userId: Long) : UseCase.
            Request
        data class Response(val user: User) : UseCase.
            Response
    }
```

32. Now, we move on to the last use case for updating the interaction data:

```
    class UpdateInteractionUseCase @Inject constructor(
        configuration: Configuration,
        private val interactionRepository:
            InteractionRepository
        ) : UseCase<UpdateInteractionUseCase.Request,
        UpdateInteractionUseCase.Response>
    (configuration) {
        override fun process(request: Request): Flow
            <Response> {
            return
    interactionRepository.saveInteraction
                (request.interaction)
                .map {
                    Response
                }
```

```
    }
    data class Request(val interaction:
Interaction) :
         UseCase.Request
    object Response : UseCase.Response
}
```

33. To unit-test the code, we need to create a new folder called **`resources`** in the **`test`** folder of the **`domain`** module.
34. Inside the **`resources`** folder, create a subfolder called **`mockito-extensions`**; inside this folder, create a file named **`org.mockito.plugins.MockMaker`**; and inside this file, add the following text – **`mock-maker-inline`**. This allows the Mockito testing library to mock the **`final`** Java class, which in Kotlin means all classes without the **`open`** keyword.
35. Create a new class named **`UseCaseTest`** in the test folder of the **`domain`** module:

```
class UseCaseTest {
    @ExperimentalCoroutinesApi
    private val configuration =
UseCase.Configuration
        (TestCoroutineDispatcher())
    private val request = mock<UseCase.Request>()
    private val response = mock<UseCase.Response>()
    @ExperimentalCoroutinesApi
    private lateinit var useCase:
        UseCase<UseCase.Request, UseCase.Response>
    @ExperimentalCoroutinesApi
    @Before
    fun setUp() {
        useCase = object : UseCase<UseCase.Request,
            UseCase.Response>(configuration) {
            override fun process(request:
Request):
                Flow<Response> {
```

```
                    assertEquals(this@UseCaseTest.reque
st,
                        request)
                    return flowOf(response)
                }
            }
        }
    }
```

Here, we provide an implementation for the **`UseCase`** class, which will return a mocked response.

36. Next, create a test method that will verify the successful scenario for the **`execute`** method:

```
    @ExperimentalCoroutinesApi
    @Test
    fun testExecuteSuccess() = runBlockingTest {
        val result =
useCase.execute(request).first()
        assertEquals(Result.Success(response),
result)
    }
```

Here, we assert that the result of the **`execute`** method is **`Success`** and that it contains the mocked response.

37. Next, create a new test class named **`GetPostsWithUsersWithInteractionUseCaseTest`**:

```
class GetPostsWithUsersWithInteractionUseCaseTest {
    private val postRepository =
mock<PostRepository>()
    private val userRepository =
mock<UserRepository>()
    private val interactionRepository =
mock<InteractionRepository>()
```

```
    private val useCase =
GetPostsWithUsersWithInteractionUseCase(
        mock(),
        postRepository,
        userRepository,
        interactionRepository
    )
}
```

Here, we mock all the repositories and inject the mocks into the class we want to test.

38. Finally, create a test method that will verify the **`process`** method from the use case we are testing:

```
    @ExperimentalCoroutinesApi
    @Test
    fun testProcess() = runBlockingTest {
        val user1 = User(1L, "name1", "username1",
"email1")
        val user2 = User(2L, "name2", "username2",
"email2")
        val post1 = Post(1L, user1.id, "title1",
"body1")
        val post2 = Post(2L, user1.id, "title2",
"body2")
        val post3 = Post(3L, user2.id, "title3",
"body3")
        val post4 = Post(4L, user2.id, "title4",
"body4")
        val interaction = Interaction(10)
        whenever(userRepository.getUsers()).thenRet
urn
            (flowOf(listOf(user1, user2)))
        whenever(postRepository.getPosts()).thenRet
urn
```

```
            (flowOf(listOf(post1, post2, post3,
post4)))whenever(interactionRepository.getInteracti
on
            ()).thenReturn(flowOf(interaction))
        val response = useCase.process
            (GetPostsWithUsersWithInteractionUseCas
e.
                Request).first()
        assertEquals(
            GetPostsWithUsersWithInteractionUseCase
.
                Response(
                listOf(
                    PostWithUser(post1, user1),
                    PostWithUser(post2, user1),
                    PostWithUser(post3, user2),
                    PostWithUser(post4, user2),
                ), interaction
            ),
            response
        )
    }
```

Here, we provide mock lists of users and posts and a mock interaction, then we return these for each of the repository calls, and then we assert that the result is a list of four posts, written by two users and the mock interaction.

If we run the tests for these two methods, they should pass. To test the remaining use cases, the same principles can be applied that we used for **`GetPostsWithUsersWithInteractionUseCaseTest`** – create mock repositories, inject them into the object we wish to test, and then define the mocks for the input of the **`process`** method and the results we should expect, which will give us output as shown in the following screenshot:

Figure 5.5 – Output of the use case unit tests

In this section, we performed an exercise in which we created a simple domain that contained entities, a few simple use cases, and a particular use case that combined multiple data sources. The domain module has dependencies on flows and Hilt. This means that changes to these libraries might cause changes to our domain module. This decision was made because of the benefits that these libraries provide when it comes to reactive programming and dependency injection. Because we considered dependency injection when defining the use cases, this made them more testable, as we could inject mock objects into the tested objects very easily.

Summary

In this chapter, we looked at how the architecture of an Android app is layered and focused on the domain layer, discussing the topics of entities and use cases. We also learned how to use dependency inversion to place use cases and entities at the center of our architecture. We did this by creating repository abstractions that can be implemented in the lower layers. We also learned how to use library modules to enforce separations between layers.

In the chapter's exercise, we created a domain module for an Android application, providing an example of what a domain layer might look like. In the next chapter, we will focus on the data layer, in which we will provide implementations for the repository abstractions we defined in the domain layer, and discuss how we can use these repositories to manage an application's data.