Chapter 8: Getting Started with Clean Architecture in Android

In this chapter, we're continuing our journey of improving the architectural design of the Restaurants application.

More specifically, we will try to adopt some design decisions from the well-known Clean Architecture. **Clean Architecture** is a software design philosophy that tries to create projects with the best level of the following:

- Separation of concerns
- Testability
- Independence of frameworks or libraries used in peripheral layers, such as the UI or Model layer

By doing so, Clean Architecture tries to allow the business parts of our applications to adapt to changing technologies and interfaces.

Clean Architecture is a very broad and complex topic, so, in this chapter, we will try to focus only on establishing a better separation of concerns by separating existing layers even further, but more importantly, by defining a new layer called the **Domain layer**.

In this chapter, we will on one hand borrow some architectural decisions from Clean Architecture through the *Defining the Domain layer with Use Cases* section and the *Separating the Domain model from Data models* section. On the other hand, we will try to improve project architecture with other techniques through the *Creating a package structure* section and the *Decoupling the Compose-based UI layer from ViewModel* section.

Another essential principle of Clean Architecture is the **Dependency Rule** that we will briefly cover in the *Further reading* section where you will find proper resources to follow up with.

We will cover the following topics in this section:

- Defining the Domain layer with Use Cases
- Separating the Domain model from Data models
- Creating a package structure
- Decoupling the Compose-based UI layer from ViewModel

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Building Compose-based Android projects for this chapter usually requires your day-to-day tools. However, to follow along smoothly, make sure you have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds, but note that the IDE interface and other generated code files might differ from the ones used throughout this book.
- Kotlin 1.6.10 or a newer plugin installed in Android Studio
- The Restaurants app code from the previous chapter

The starting point for this chapter is represented by the Restaurants application developed in the previous chapter. If you haven't followed the implementation from the previous chapter, access the starting point for this chapter by navigating to the `Chapter_07` directory of the repository and importing the Android project titled `chapter_7_restaurants_app`.

To access the solution code for this chapter, navigate to the `Chapter_08` directory:

https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_08/chapter_8_restaurants_app.

Defining the Domain layer with Use Cases

So far, we've talked about the Presentation layer (with UI and presentation logic) and the Model layer (with data logic). Yet, apart from these two layers, most of the time, applications also encapsulate a different type of logic, different from UI, presentation, or data logic.

To identify this type of logic, we must first acknowledge that most applications have a dedicated business scope – for example, a food delivery application could have the business scope of taking orders and generating revenue for the stakeholder. The **stakeholder** is the entity interested in the business, such as the company that owns the restaurant chain.

Such applications can contain business rules imposed by the stakeholders that can vary from minimum order amounts, custom availability ranges for certain restaurants, or predefined time frames for different delivery charges; the list could go on. We can refer to such business rules that are dictated by stakeholders as **business logic**.

For our Restaurants app, let's imagine that the stakeholder (for example, the company we would be building the application for) asked us to always show the restaurants alphabetically, no matter what. This shouldn't be something that the user would know about; instead, it should be a predefined business rule that we must implement.

Now, sorting restaurants alphabetically isn't that big of a deal, so the natural question that arises is, where should we apply this sorting logic?

To figure this out, let's recap the current layering of the project. Right now, the Presentation layer is connected to the Model layer.

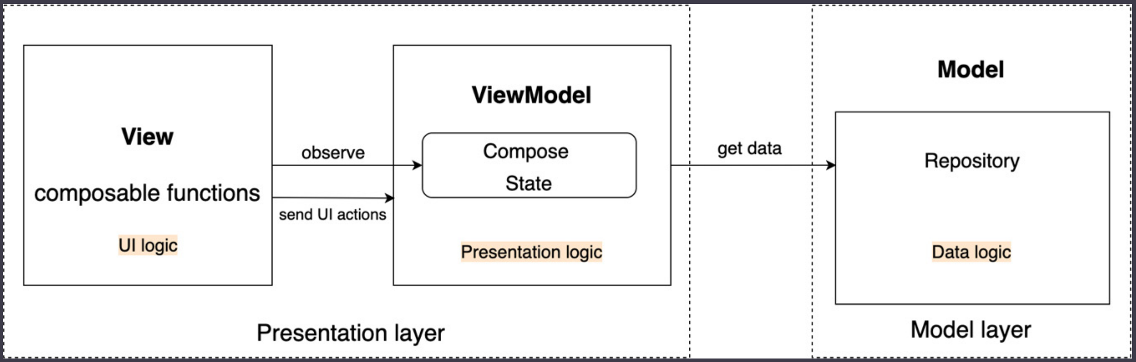


Figure 8.1 – Layering of responsibilities in the Restaurants app

With our existing layer structuring, we could sort the restaurants in the following:

- **UI level (composables)**: Since this sorting logic is business logic, we should try to avoid adding it here.
- **Inside the ViewModel**: If this sorting was a presentation option (so the user could sort restaurants in different ways from the UI by selecting a picker or button), we would have considered this to be presentation logic that can be held inside a `ViewModel` class; yet, remember that this rule is part of the business requirements, and the user shouldn't know about it, so it's probably not a good idea to implement it here.
- **Inside the Repository**: Here, we store data logic (such as caching), which is different from business logic.

None of the options are ideal, and we will see in a moment why this is the case. Until then, let's have a compromise and add this business rule inside the Model layer:

1. Inside `RestaurantsRepository`, refactor the `getAllRestaurants()` method to sort the restaurants by `title` by calling the `sortedBy { }` extension function on the restaurants that are returned:

```
suspend fun getAllRestaurants(): List<Restaurant> {
    return withContext(Dispatchers.IO) {
        try {
            refreshCache()
        } catch (e: Exception) {…}
        return@withContext restaurantsDao.getAll()
            .sortedBy { it.title }
    }
}
```

2. Build and run the application.

The restaurants are now correctly sorted by their title, yet if you toggle a restaurant as a favorite, you might notice an initial flicker and a re-ordering effect on the list. Can you guess why this has happened?

The issue here is that in **`RestaurantsRepository`**, the **`toggleFavoriteRestaurant()`** method returns the unsorted version of the restaurants from the **Data Access Object** (**DAO**):

```
suspend fun toggleFavoriteRestaurant(…)=
withContext(…){
    …
    restaurantsDao.getAll()
}
```

To fix this, we could repeat the same sorting logic from the **`getAllRestaurants()`** method.

Yet, this approach is problematic because we would be repeating or duplicating the sorting business rule. Worse than that, since we are in the Model layer, we're mixing data logic with business logic. We shouldn't be mixing data caching logic with business rules.

It's clear that for us to correctly encapsulate business logic and to be able to reuse it, we should extract it to a separate layer. Just like whenever we wanted to prevent any changes impacting the Presentation layer from affecting other layers, such as the UI or Model layers, we want to separate the business logic inside a separate layer so that any changes to the business logic shouldn't impact other layers and their corresponding logic.

According to Clean Architecture concepts, the layer that encapsulates business rules and business logic is referred to as the Domain layer. This layer sits between the Presentation layer and the Model layer. It should process the data from the Model layer by applying the business rules that it incorporates, and then feed the Presentation layer with the business-compliant content.

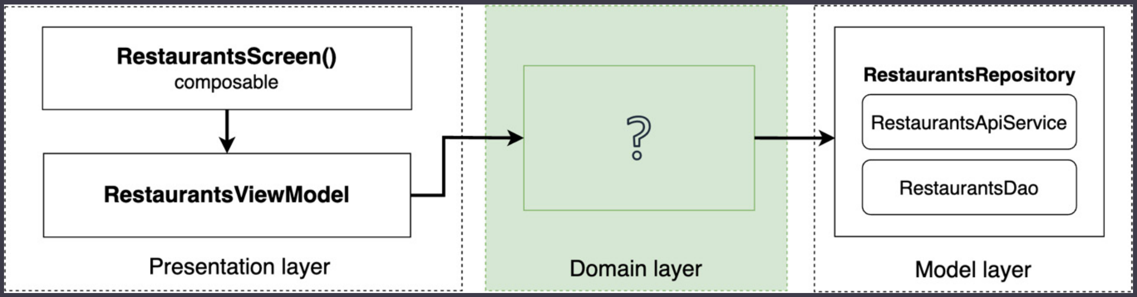


Figure 8.2 – Layering of responsibilities in the Restaurants app, including the Domain layer

In other words, in a particular flow (as in the screen with the list of restaurants), the Presentation layer through the `ViewModel` would connect to the Domain layer instead of the Model layer. In turn, the Domain layer would get the data from the Model layer.

NOTE

Not all applications, screens, or flows contain business logic. For these cases, the Domain layer is optional. The Domain layer should hold business logic, but, if there is no such logic, there should be no such layer.

But what should the Domain layer contain?

According to Clean Architecture concepts, repeatable business logic that is related to a specific application action or flow should be encapsulated in a Use Case. In other words, **Use Cases** are classes that extract repeatable business rules related to a single functionality of your application as a single unit of business logic.

For example, an online ordering app can have business logic related to displaying only stores in the near proximity of the user. To encapsulate this business rule, we could create a **`GetStoresInProximityUseCase`** class. Or, maybe there is some business logic associated with the logout action triggered by the user (such as executing some user benefits or points calculation behind the scenes); then, we could implement **`LogOutUserUseCase`**.

So, in our Restaurants app, any business logic must be encapsulated in a Use Case that sits between the Presentation layer and the Model layer:

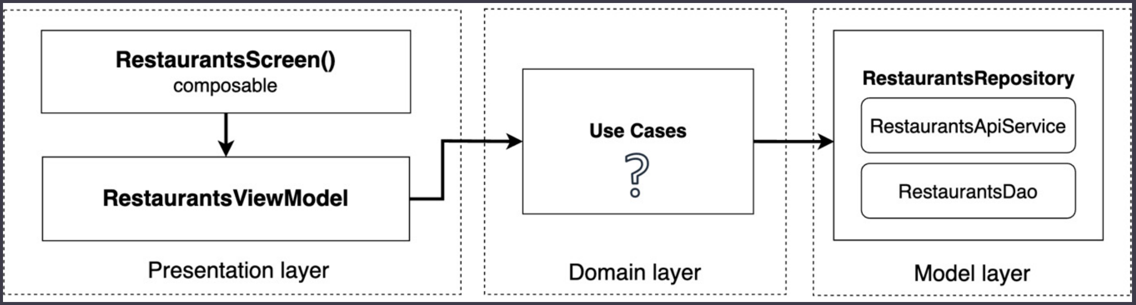


Figure 8.3 – Layering of responsibilities where the Domain layer contains Use Cases

A separated Domain layer brings the following benefits:

- Improves the testability of the app by separating business logic into its own classes. This way, business responsibilities are separated from other components and their logic can be tested separately without having to care about components from other layers.
- By separating business logic inside Use Cases, we avoid code duplication, and we improve the re-usability of business rules and their corresponding logic.
- Improves the readability of the classes that contain Use Cases dependencies. This is because each unit of business is now extracted separately and provides developers with valuable insights into the business actions each screen or flow executes.

Before jumping into a practical example, let's briefly cover a few important aspects of Use Cases:

- They can use (or depend on) other Use Cases. Since Use Cases define a single unit of reusable business logic, then Use Cases can use other Use Cases to define complex business logic.
- They usually obtain their data from the Model layer but are not conditioned to only one `Repository` class – in other words, you can access multiple repositories from within your Use Case.
- They usually have only one public method, mostly because Use Cases encapsulate business rules related to a single functionality of your app (like `LogOutUserUseCase` does).
- They should follow a naming convention. A popular convention for the Use Case class is a verb in the present tense that defines the action, usually followed by a few words that express the *what*, and that ends with the UseCase suffix. Some examples could be `GetStoresInProximityUseCase` or `CalculateOrderTotalUseCase`.

It's time to see what a Use Case class looks like. In our Restaurants app, the business logic of sorting restaurants alphabetically is a good match for it being extracted to a Use Case because of the following:

- It's a business rule dictated by the stakeholder.
- It's repeated twice.
- It's part of a specific action of the app (getting the restaurants).

Let's define our first Use Case class!

1. Click on the application package, select **New**, and then **Kotlin Class/File**. Enter `GetRestaurantsUseCase` as the name, select **Class**, and add this code:

```
class GetRestaurantsUseCase {
    private val repository: RestaurantsRepository =
        RestaurantsRepository()
    suspend operator fun invoke(): List<Restaurant>
{
        return repository.getAllRestaurants()
                        .sortedBy { it.title }
        }
    }
```

Functionally, this Use Case class gets the restaurants from **`RestaurantsRepository`**, applies the business rule of sorting the restaurants alphabetically, just like **`RestarauntsViewModel`** did, and then returns the list. In other words, **`GetRestaurantsUseCase`** is now the one responsible for applying business rules.

This Use Case does that with only one public method, which is also a **`suspend`** function because the **`repository.getAllRestaurants()`** call is a suspending function call. But, more importantly, why did we name the function of the Use Case as **`invoke()`** while also specifying the **`operator`** keyword?

We did that because Kotlin allows us to define an **`invoke`** operator on a class so we can call it on any instances of the class without a method name. This is how we will call the **`invoke()`** operator of **`GetRestaurantsUseCase`**:

```
val useCase = GetRestaurantsUseCase()
val result = useCase()
```

This syntax is especially useful for us because our Use Case classes have only one method, and the name of the class is already suggestive enough, so we don't need a named function.

2. Make sure to remove the sorting logic that we initially added in the **`getAllRestaurants()`** method in **`RestaurantsRepository`**. The returned data of the method should look like this:

```
    suspend fun getAllRestaurants(): List<Restaurant> {
        return withContext(Dispatchers.IO) {
            try { … } catch (e: Exception) {…}
            return@withContext restaurantsDao.getAll()
        }
    }
```

3. Inside **RestaurantsViewModel**, add a new dependency to the **GetRestaurantsUseCase** class:

```
class RestaurantsViewModel() : ViewModel() {
  private val repository = RestaurantsRepository()
  private val getRestaurantsUseCase =
GetRestaurantsUseCase()
  […]
}
```

4. Then, inside the **getRestaurants()** method of the **ViewModel**, remove the call for restaurants to the **repository** variable, and instead, call the **invoke()** operator for the **getRestaurantsUseCase** variable:

```
private fun getRestaurants() {
    viewModelScope.launch(errorHandler) {
        val restaurants = getRestaurantsUseCase()
        _state.value = _state.value.copy(
            restaurants = restaurants, […])
    }
}
```

Before building and running the app, let's try to identify any other business rules for this particular flow of the app.

If we have a look inside **RestaurantsRepository**, the **toggleFavoriteRestaurant()** method takes in an **oldValue: Boolean** parameter, and negates it before passing it to **PartialRestaurant**:

```
suspend fun toggleFavoriteRestaurant(
    id: Int,
    oldValue: Boolean
) =
    withContext(Dispatchers.IO) {
        restaurantsDao.update(
            PartialRestaurant(
                id = id,
                isFavorite = !oldValue
            )
        )
        restaurantsDao.getAll()
    }
```

This happens every time we mark a restaurant as a favorite or not favorite. The rule of negating **`oldValue`** of the favorite status of the restaurant (by passing **`!oldValue`**) can be considered a business rule imposed by the stakeholder: *whenever a user presses on the heart icon of a restaurant, we must toggle its favorite status to the opposite value*.

To be able to reuse this business logic and not have it done by **`RestaurantsRepository`** , let's also extract this rule to a Use Case.

5. First, inside **`RestaurantsRepository`**, rename the **`oldValue`** parameter to **`value`** and make sure to not negate it anymore when passing it to the **`isFavorite`** field of **`PartialRestaurant`**:

```
suspend fun toggleFavoriteRestaurant(id: Int,
value: Boolean)=
    withContext(Dispatchers.IO) {
        restaurantsDao.update(
           PartialRestaurant(id = id, isFavorite =
value)
        )
        restaurantsDao.getAll()
    }
```

6. Click on the application package, select **New**, and then **Kotlin Class/File**. Enter **`ToggleRestaurantUseCase`** as the name, select **Class**, and add this code:

```
class ToggleRestaurantUseCase {
    private val repository: RestaurantsRepository =
        RestaurantsRepository()
    suspend operator fun invoke(
        id: Int,
        oldValue: Boolean
    ): List<Restaurant> {
        val newFav = oldValue.not()
        return repository
            .toggleFavoriteRestaurant(id, newFav)
    }
}
```

This Use Case now encapsulates the business rule of negating the favorite flag of a restaurant with the **`val newFav = oldValue.not()`** line. While the business logic here is rather slim, in production apps, things tend to get more complex. This Use Case should be called whenever we mark a restaurant as a favorite or not favorite.

7. Inside **`RestaurantsViewModel`**, add a new dependency to the **`ToggleRestaurantUseCase`** class:

```
class RestaurantsViewModel() : ViewModel() {
    private val getRestaurantsUseCase =
GetRestaurantsUseCase()
    private val toggleRestaurantsUseCase =
ToggleRestaurantUseCase()
    […]
}
```

At this step, you can also safely remove the **`RestaurantsViewModel`** class's dependency to the **`RestaurantsRepository`** class by removing the **`reposi-tory`** variable.

8. Then, inside the **`toggleFavorite()`** method of the **`ViewModel`**, remove the call for toggling the restaurant on the **`repository`** variable, and instead, call the **`invoke()`** operator for the **`toggleRestaurantUseCase`** variable:

```
fun toggleFavorite(id: Int, oldValue: Boolean) {
    viewModelScope.launch(errorHandler) {
        val updatedRestaurants =
            toggleRestaurantsUseCase(id, oldValue)
        _state.value = _state.value.copy(…)
        }
    }
```

Now, the business rule of toggling a restaurant as a favorite or not is done inside **ToggleRestaurantUseCase**.

9. Now that we have extracted business logic into Use Case classes, build the app and run it. The application should behave the same.

Yet, if you try toggling a restaurant as a favorite, the list of restaurants still flickers, and their order seems to change. Can you think of why this happens?

Let's circle back to **RestaurantsRepository** and check out the **toggleFavoriteRestaurant** method:

```
suspend fun toggleFavoriteRestaurant(…)=
withContext(…) {
    restaurantsDao.update(
        PartialRestaurant(id = id, isFavorite =
value)
    )
    restaurantsDao.getAll()
}
```

The problem with this method is that it returns the restaurants obtained from the Room DAO by calling **restaurantsDao.getAll()**. These restaurants are not sorted alphabetically, as our business rules now indicate. So, every time we toggle a restaurant as favorite, we update the UI with the unsorted list of restaurants.

We need to somehow reuse the sorting logic from **GetRestaurantsUseCase**:

1. First, from within **RestaurantsRepository**, remove the **restaurantsDao.getAll()** call from the **toggleFavoriteRestaurant** method:

```
suspend fun toggleFavoriteRestaurant(…)=
withContext(…) {
    restaurantsDao.update(
        PartialRestaurant(id = id, isFavorite =
value)
    )
}
```

This way, this method no longer returns a list of restaurants; it just updates a specific restaurant. As of now, the **toggleFavoriteRestaurant** method doesn't return anything anymore.

2. Then, inside the **ToggleRestaurantUseCase** class, remove the return statement for the **repository.toggleFavoriteRestaurant()** line, and instead return the sorted list of restaurants by directly instantiating and calling the **invoke()** operator on the **GetRestaurantsUseCase** class:

```
class ToggleRestaurantUseCase {
    private val repository: … =
RestaurantsRepository()
    suspend operator fun invoke(…):
List<Restaurant> {
        val newFav = oldValue.not()
        repository.toggleFavoriteRestaurant(id,
newFav)
        return GetRestaurantsUseCase().invoke()
    }
}
```

This approach fixes our issue – whenever we toggle a restaurant as a favorite or not, the UI no longer flickers because the UI is updated with the correctly sorted list – yet this happens with a lengthy delay.

Unfortunately, this functionality is not efficient at all because whenever we toggle a restaurant as a favorite or not, the **GetRestaurantsUseCase** calls the **RestaurantsRepository** class's **getAllRestaurants()** method that, in turn, triggers a request to get the restaurants again from the Web API, attempts to cache them into **Room**, and only then provides us with a list, hence the delay we've just experienced.

In a good application architecture, a network request that gets the new list of items shouldn't be done after every UI interaction with an item. Let's fix this by refactoring our code and by creating a new Use Case that only retrieves the cached restaurants, sorts them, and returns them:

1. First, inside **RestaurantsRepository**, add a new method called **getRestaurants()** that only retrieves the restaurants from our Room DAO:

```
suspend fun getRestaurants() : List<Restaurant> {
    return withContext(Dispatchers.IO) {
        return@withContext restaurantsDao.getAll()
    }
}
```

2. Click on the application package, select **New**, and then **Kotlin Class/File**. Enter **GetSortedRestaurantsUseCase** as the name, select **Class**, and add this code:

```
class GetSortedRestaurantsUseCase {
    private val repository: RestaurantsRepository =
        RestaurantsRepository()
    suspend operator fun invoke(): List<Restaurant>
{
        return repository.getRestaurants()
            .sortedBy { it.title }
    }
}
```

The **GetSortedRestaurantsUseCase** class now retrieves the restaurants from the **RestaurantsRepository** by calling the previously created **getRestaurants()** method (without triggering any network request or caching), applies the sorting business rule, and finally, returns the list of restaurants.

3. Use the newly created **`GetSortedRestaurantsUseCase`** class inside **`ToggleRestaurantUseCase`** so that we only get the cached restaurants every time we toggle a restaurant as a favorite or not:

```
class ToggleRestaurantUseCase {
    private val repository: … =
RestaurantsRepository()
    private val getSortedRestaurantsUseCase =
        GetSortedRestaurantsUseCase()
    suspend operator fun invoke(…):
List<Restaurant> {
        val newFav = oldValue.not()
        repository.toggleFavoriteRestaurant(id,
newFav)
        return getSortedRestaurantsUseCase()
    }
}
```

Now, we must refactor **`GetRestaurantsUseCase`** to reuse the sorting business logic from within **`GetSortedRestaurantsUseCase`** because the alphabetical sorting logic is now duplicated in both Use Cases:

1. First, inside **`RestaurantsRepository`**, update the **`getAllRestaurants`** method to no longer return the restaurants by no longer returning **`restaurantsDao.getAll()`**, while also removing the function's return type:

```
suspend fun getAllRestaurants() {
    return withContext(Dispatchers.IO) {
        try { … } catch (e: Exception) { … }
    }
}
```

2. Rename the **`getAllRestaurants`** method to **`loadRestaurants()`** to better reflect its responsibility:

```
suspend fun loadRestaurants() {
    return withContext(Dispatchers.IO) {
        try { … } catch (e: Exception) { … }
    }
}
```

3. Inside **GetRestaurantsUseCase**, add a new dependency to the **GetSortedRestaurantUseCase** class and refactor the class as follows:

```
class GetRestaurantsUseCase {
    private val repository: … =
RestaurantsRepository()
    private val getSortedRestaurantsUseCase =
        GetSortedRestaurantsUseCase()
    suspend operator fun invoke(): List<Restaurant>
{
        repository.loadRestaurants()
        return getSortedRestaurantsUseCase()
    }
}
```

Inside the **invoke()** function, we made sure to first call the newly renamed **loadRestaurants()** method of the **RestaurantsRepository** and then, in addition, to invoke **GetSortedRestaurantsUseCase**, which is now also returned.

4. To better reflect its purpose, rename the **GetRestaurantsUseCase** class to **GetInitialRestaurantsUseCase**:

```
class GetInitialRestaurantsUseCase {
    private val repository: … =
RestaurantsRepository()
    private val getSortedRestaurantsUseCase =
        GetSortedRestaurantsUseCase()
    suspend operator fun invoke(): List<Restaurant>
{...}
}
```

5. As a consequence, inside **RestaurantsViewModel**, update the type for the **getRestaurantsUseCase** variable:

```
class RestaurantsViewModel() : ViewModel() {
  private val repository = RestaurantsRepository()
```

```
    private val getRestaurantsUseCase =
        GetInitialRestaurantsUseCase()
    …
}
```

6. Build the app and run it. The application should now behave correctly when marking a restaurant as a favorite or not; the restaurants remain sorted alphabetically.

Let's now move on to another way of improving the architecture of our app.

Separating the Domain model from Data models

Inside the Domain layer, apart from Use Cases, another essential business component in our app is the **Domain model component**. The Domain model components are those classes that represent core business data or concepts used throughout the application.

NOTE

Since the Domain models reside inside the Domain layer, they should be agnostic of any third-party library or dependency – ideally, they should be pure Java or Kotlin classes.

For example, in our Restaurants app, the core entity used throughout the app (retrieved, updated, and displayed) is the `Restaurant` data class, which contains data such as `title` and `description`.

If we think about it, our Restaurants app's core business entity is represented by the restaurant itself: that's what the application is about, so it's only natural that we would consider the `Restaurant` class as a business entity.

NOTE

In Clean Architecture, Domain model classes are often referred to as Entity classes. However, it's important to mention that the Room database **`@Entity`** *annotation has nothing to do with Clean Architecture; any class annotated with the Room* **`@Entity`** *annotation doesn't automatically become an entity. In fact, as per Clean Architecture, Entity classes should have no library dependencies such as database annotations.*

If we have a look at our **`Restaurant`** data class though, we can identify a serious issue:

```
import androidx.room.ColumnInfo
      …
import com.google.gson.annotations.SerializedName
@Entity(tableName = "restaurants")
data class Restaurant(
    @PrimaryKey()
    @ColumnInfo(name = "r_id")
    @SerializedName("r_id")
    val id: Int,
    @ColumnInfo(name = "r_title")
    @SerializedName("r_title")
    val title: String,
      …
)
```

Can you spot the problem?

While, in the beginning, the **`Restaurant`** data class was a pure Kotlin data class with some fields, in time, it grew to something more than that.

We first added Retrofit to our app so we could get the restaurants from a Web API, and had to mark the fields we obtained with **`@SerializedName`** annotations so that the GSON (Google Gson) deserialization would work. Then we added Room to the mix because we wanted to cache the restaurants, so we had to add an `@Entity` annotation to the class, and other annotations, such as `@PrimaryKey` and `@ColumnInfo`, to its fields.

While it was convenient for us to use only one Data model class throughout the app, we have now coupled a Domain model class (`Restaurant.kt`) to library dependencies, such as GSON or Room. This means that our Domain model is coupled to the Data or Model layer that is responsible for obtaining data.

According to Clean Architecture, the Domain model classes should reside inside the Domain layer and be agnostic of any libraries tightly related to the way we retrieve or cache data from several sources.

In other words, we need to make a separation between Domain models and **Data Transfer Objects** (**DTOs**) by creating separate classes for both types. While Domain models are plain Kotlin classes, DTOs are classes that contain both the fields needed for a specific data operation, such as caching items to a local source, but also dependencies such as library annotations.

With such a separation, the Domain model is now a business entity that doesn't care about implementation details (such as libraries), so every time we might have to replace a library (such as Retrofit or Room) with another library, we must only update the DTOs (hence, the Model layer) and not classes within the Domain model.

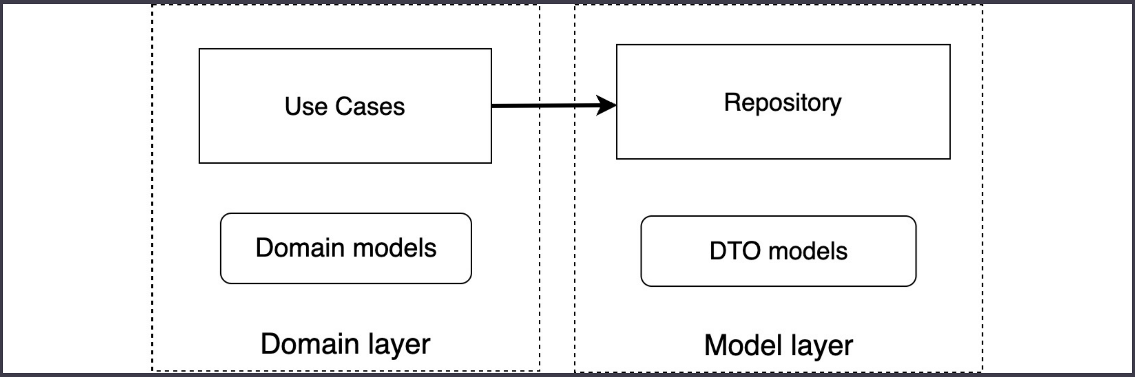


Figure 8.4 – Separating Domain models from DTO models

To achieve such a separation in our Restaurants app, we must split our **`Restaurant`** class into three classes. We must do the following:

- Create two DTOs as **`data class`** classes that will be used for transferring data:
  - A **`RemoteRestaurant`** class that will contain the fields received from the Web API. These fields will also be annotated with GSON serialization annotations required by Retrofit to parse the response.
  - A **`LocalRestaurant`** class that will contain the fields and their corresponding annotations required by Room to cache restaurants.
- Refactor the **`Restaurant`** data class to be a plain Kotlin data class, without any third-party dependencies. This way, the **`Restaurant`** data class will be a proper Domain model class, independent of the Model layer that is tightly coupled to third-party libraries.

Let's begin!

1. Click on the application package, select **New**, and then **Kotlin Class/File**. Enter **`RemoteRestaurant`** as the name, select **Class**, and add this code to define the DTO for our remote source (Firebase remote database):

```
data class RemoteRestaurant(
    @SerializedName("r_id")
    val id: Int,
    @SerializedName("r_title")
    val title: String,
    @SerializedName("r_description")
    val description: String)
```

Inside this class, we have added all the fields received from the Web API, along with their corresponding serialization fields. You can get these annotations and their imports from the **`Restaurant`** class.

Another advantage of having a separate DTO class is that it now contains only the necessary fields – for instance, unlike **`Restaurant`**,

RemoteRestaurant no longer contains an **isFavorite** field because we don't receive it from the REST API of our Firebase Database.

2. Click on the application package and create a new file called **LocalRestaurant**. Add this code to define the DTO for our local source (Room local database):

```
@Entity(tableName = "restaurants")
data class LocalRestaurant(
    @PrimaryKey()
    @ColumnInfo(name = "r_id")
    val id: Int,
    @ColumnInfo(name = "r_title")
    val title: String,
    @ColumnInfo(name = "r_description")
    val description: String,
    @ColumnInfo(name = "is_favorite")
    val isFavorite: Boolean = false)
```

You can get the fields, annotations, and their imports from the **Restaurant** class.

3. Now, navigate to the **Restaurant** class. It's time to remove all its third-party dependencies to Room and GSON and keep it as a simple Domain model class containing the fields that define our restaurant entity. It should now look like this:

```
data class Restaurant(
    val id: Int,
    val title: String,
    val description: String,
    val isFavorite: Boolean = false)
```

Make sure to also remove any imports for the GSON and Room annotations.

4. Inside the **RestaurantsDb** class, update the entity used in Room to our newly created **LocalRestaurant**, while also updating the schema version to **3**, just to be sure that Room will provide a fresh start:

```
@Database(
    entities = [LocalRestaurant::class],
    version = 3,
    exportSchema = false)
abstract class RestaurantsDb : RoomDatabase() {
    abstract val dao: RestaurantsDao
    …
}
```

5. Rename the **PartialRestaurant** class to **PartialLocalRestaurant** to better clarify that this class is used by our local data source, Room:

```
@Entity
class PartialLocalRestaurant(
@ColumnInfo(name = "r_id")
val id: Int,
@ColumnInfo(name = "is_favorite")
val isFavorite: Boolean)
```

6. Inside the **RestaurantsDao** interface, replace the **Restaurant** class usages with **LocalRestaurant**, and the **PartialRestaurant** class usages with **PartialLocalRestaurant**:

```
@Dao
interface RestaurantsDao {
    @Query("SELECT * FROM restaurants")
    suspend fun getAll(): List<LocalRestaurant>
    @Insert(onConflict =
OnConflictStrategy.REPLACE)
    suspend fun addAll(restaurants:
        List<LocalRestaurant>)
    @Update(entity = LocalRestaurant::class)
    suspend fun update(partialRestaurant:
       PartialLocalRestaurant)
    @Update(entity = LocalRestaurant::class)
    suspend fun updateAll(partialRestaurants:
        List<PartialLocalRestaurant>)
    @Query("SELECT * FROM restaurants WHERE
        is_favorite = 1")
    suspend fun getAllFavorited():
List<LocalRestaurant>
}
```

7. Inside **RestaurantsRepository**, navigate to the **toggleFavoriteRestaurant()** method, and replace the **PartialRestaurant** usage with **PartialLocalRestaurant**:

```
  suspend fun toggleFavoriteRestaurant(
        …
  ) = withContext(Dispatchers.IO) {
      restaurantsDao.update(
        PartialLocalRestaurant(id = id, isFavorite =
  value)
      )
  }
```

8. Still inside **RestaurantsRepository**, navigate to the **getRestaurants()** method, and map the **LocalRestaurant** objects (received by the **restaurantsDao.getAll()** method call) to **Restaurant** objects:

```
  suspend fun getRestaurants() : List<Restaurant> {
      return withContext(Dispatchers.IO) {
          return@withContext
  restaurantsDao.getAll().map {
              Restaurant(it.id, it.title,
                  it.description, it.isFavorite)
          }
      }
  }
```

We have mapped **List<LocalRestaurant>** to **List<Restaurant>** by using the **.map { }** extension function. We did that by constructing and returning a **Restaurant** object from **LocalRestaurant**, represented by the **it** implicit variable name.

NOTE

Your Model layer (represented by the **`Repository`** *here), should only return Domain model objects to the Domain entity. In our case,* **`RestaurantsRepository`** *should return* **`Restaurant`** *objects, and not* **`LocalRestaurants`** *objects, simply because the Use Case classes (so, the Domain layer) that use this* **`Repository`** *shouldn't have any knowledge of DTO classes from the Model layer.*

9. Navigate to the **`RestaurantsApiService`** interface (the Retrofit interface) and replace the usages of the **`Restaurant`** class with **`RemoteRestaurant`**:

```
interface RestaurantsApiService {
   @GET("restaurants.json")
    suspend fun getRestaurants():
List<RemoteRestaurant>
   @GET("restaurants.json?orderBy=\"r_id\"")
    suspend fun getRestaurant(…):
        Map<String, RemoteRestaurant>
}
```

10. Going back to **`RestaurantsRepository`**, navigate to the **`refreshCache()`** method and map the **`remoteRestaurants`** list from Retrofit to **`LocalRestaurant`** objects so that **`restaurantsDao`** can cache them:

```
private suspend fun refreshCache() {
    val remoteRestaurants = restInterface
        .getRestaurants()
    val favoriteRestaurants = restaurantsDao
        .getAllFavorited()
    restaurantsDao.addAll(remoteRestaurants.map {
        LocalRestaurant(
            it.id,
            it.title,
            it.description,
            false
        )
    })
    restaurantsDao.updateAll(
        favoriteRestaurants.map {
            PartialLocalRestaurant(
                id = it.id,
                isFavorite = true
            )
        })
}
```

Additionally, make sure to update the usage of **`PartialRestaurant`** to **`PartialLocalRestaurant`** in the **`restaurantsDao.updateAll()`** method call.

11. Navigate to **`RestaurantsDetailsViewModel`** and, inside the **`getRemoteRestaurant()`** method, map the **`RemoteRestaurant`** object received from the Retrofit API to a **`Restaurant`** object by using the **`?.let{ }`** extension function:

```
private suspend fun getRemoteRestaurant(id: Int):
Restaurant {
    return withContext(Dispatchers.IO) {
        val response
=  restInterface.getRestaurant(id)
        return@withContext
response.values.first().let {
            Restaurant(
                id = it.id,
                title = it.title,
                description = it.description
            )
        }
    }
}
```

Remember that in the restaurant details screen, we don't have any business logic or Use Cases, or even a **`Repository`**, so we have directly added a variable for the Retrofit interface inside the `ViewModel` – and that's why we are mapping the Domain model inside the `ViewModel`.

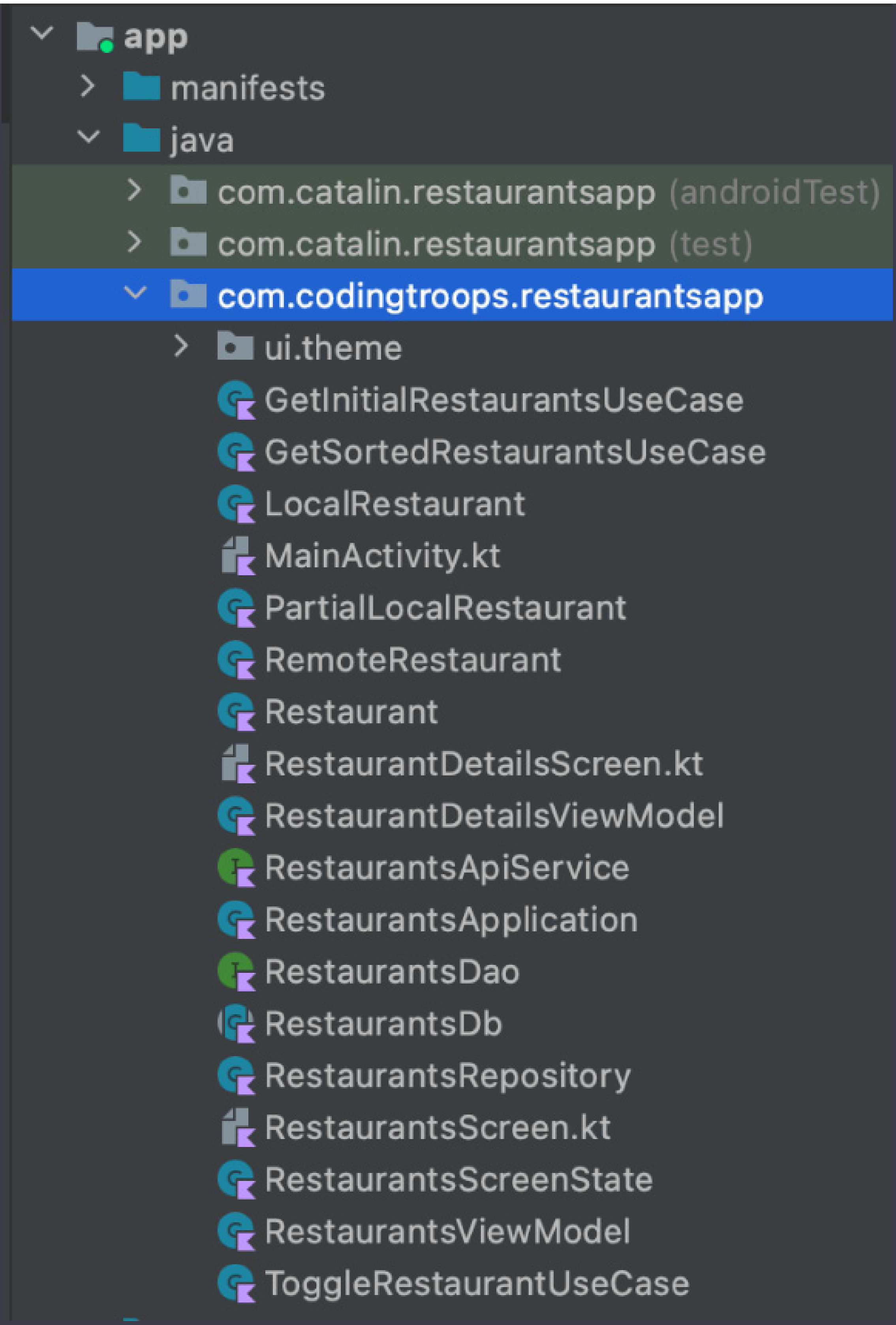
12. Build and run the app. The app should behave the same.

Let's now take a break from creating classes and let's organize our project a bit.

Creating a package structure

Our Restaurants app has come a long way. As we tried to separate responsibilities and concerns as much as possible, new classes emerged – quite a few actually.

If we have a look on the left of Android Studio, on the **Project** tab, we have an overview of the classes we've defined in our project.



```
app
  manifests
  java
    com.catalin.restaurantsapp (androidTest)
    com.catalin.restaurantsapp (test)
    com.codingtroops.restaurantsapp
      ui.theme
      GetInitialRestaurantsUseCase
      GetSortedRestaurantsUseCase
      LocalRestaurant
      MainActivity.kt
      PartialLocalRestaurant
      RemoteRestaurant
      Restaurant
      RestaurantDetailsScreen.kt
      RestaurantDetailsViewModel
      RestaurantsApiService
      RestaurantsApplication
      RestaurantsDao
      RestaurantsDb
      RestaurantsRepository
      RestaurantsScreen.kt
      RestaurantsScreenState
      RestaurantsViewModel
      ToggleRestaurantUseCase
```

Figure 8.5 – Project structure without any package structuring strategy

It's clear that our project has no folder structure at all – all files and classes are tossed around inside the **`restaurantsapp`** root package.

NOTE

The name of the root package might differ if you selected a different name for your app.

Because we've opted to throw any new class inside the root package, it's difficult to have clear visibility over the project. Our approach is similar to adding dozens of files and assets on the desktop of our PC – in time, it becomes impossible to find anything on the screen.

To alleviate this issue, we can opt for a **packaging strategy** for our project in which each class belongs to a folder. A clear folder structure allows developers to have good visibility and to gain valuable insight into the application's components, there by allowing easier access and navigation through the project files.

The most common package organizing strategies are as follows:

- **Organize packages by category or layer**: For this strategy, each package contains classes that are of the same type or belong to the same layer. The following are examples:
  - A `presentation` package would contain all the files related to the Presentation layer, regardless of the feature they belong to, such as all the files with composables, and all the `ViewModel` classes.
  - Similarly, a `data` package would contain all files related to the Model layer, regardless of the feature they belong to, such as repositories, Retrofit interfaces, or Room DAO interfaces.
- **Organize packages by feature**: For this strategy, the root packages represent and reflect a specific feature of the app. For example, a `restaurants` package would contain all the classes related to the `restaurants` feature, from UI classes to `ViewModel` classes, Use Cases, and repositories.

Both approaches have their pros and cons, but most notably, the package organization by layer doesn't scale well if the app has a lot of features, as there is no way to differentiate between classes from different features.

On the other hand, the package organization by feature can be problematic if, in each feature package, all classes are thrown around without any distinct categorization.

For our Restaurants app, we will use a mix of these two strategies. More specifically, we will do the following:

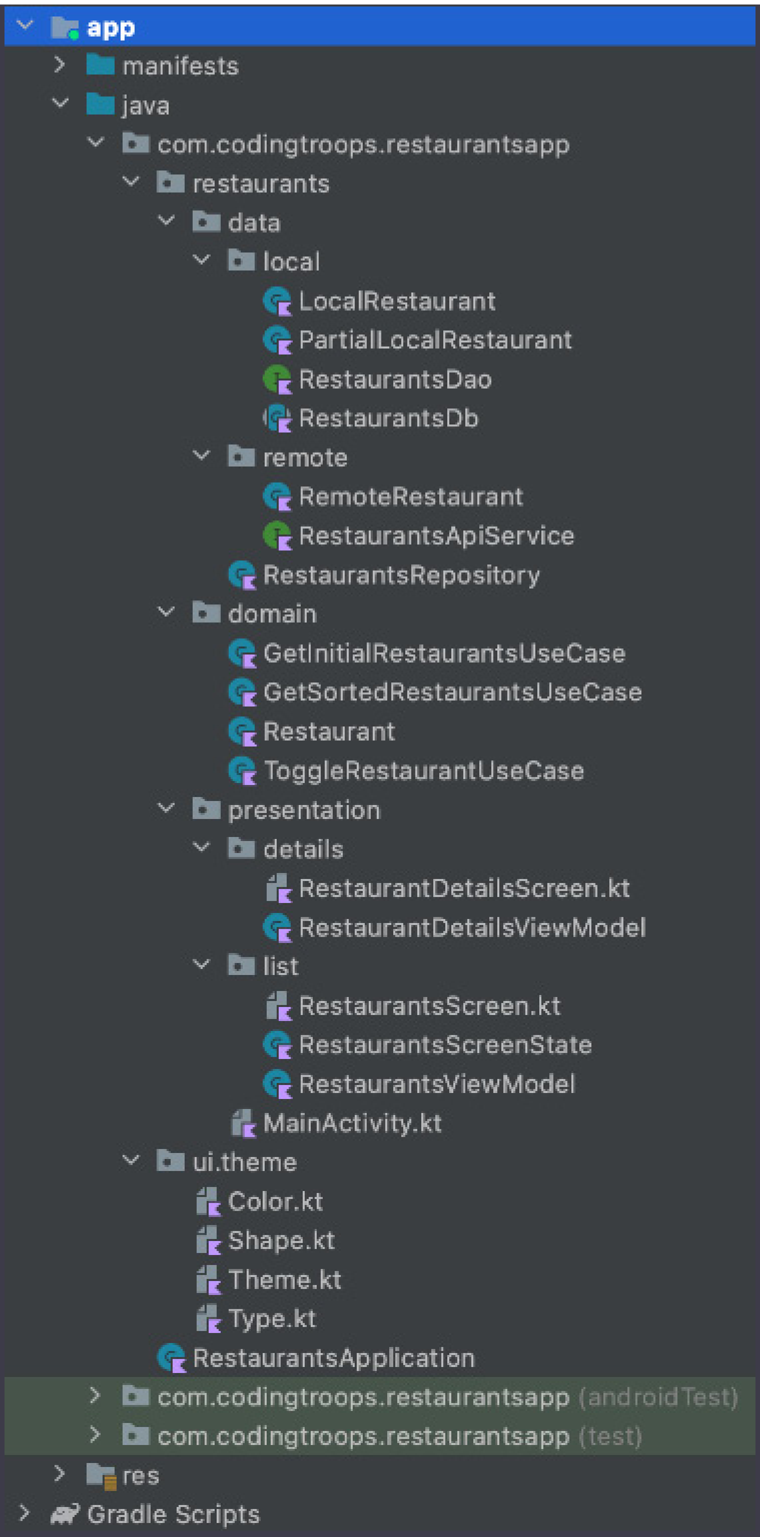
- Keep **`RestaurantsApplication.kt`** inside the root package.
- Create a root package for the only feature our application has, named **`restaurants`**. This package will contain the functionality for displaying both the list of restaurants and the detail screen.
- Create sub-packages inside the **`restaurants`** package for each layer:
  - **`Presentation`**: For composables and **`ViewModel`** classes. Inside this package, we can also break the screens that we have into separate packages: **`list`** for the first screen with the list of restaurants, and **`details`** for the second screen with the details of one restaurant. Additionally, we will keep the **`MainActivity`** class inside the **`presentation`** package since it's the host component for the UI.
  - **`Data`**: For classes within the Model layer. Here, we will not only add **`RestaurantsRepository,`** but we'll also create two sub-packages for the two different data sources: **`local`** (for caching classes such as **`RestaurantsDao`** and **`LocalRestaurant`**), and **`remote`** (for classes related to the remote source such as **`RestaurantsApiService`** and **`RemoteRestaurant`**).
  - **`Domain`**: For business-related classes, the Use Case classes, and also the **`Restaurant.kt`** Domain model class.

With this approach, if we were to add a new feature, maybe related to ordering (which we could call **`ordering`**), the package structure would provide us with immediate information about the features our application contains. When expanding a certain feature package, we can expand the package of the layer we're interested in working with and have a clear overview of the components we need to update or modify.

To achieve such a packaging structure, you will have to perform the following actions a few times:

1. Create a new package. To do that, left-click on a certain existing package (such as the `restaurantsapp` package), select **New**, then **Package**, and finally, enter the name of the package.
2. Move an existing class into an existing package. To do that, simply drag the file and drop it into the desired package.

In the end, the package structure that we described and that we want to achieve is the following:



```
app
  manifests
  java
    com.codingtroops.restaurantsapp
      restaurants
        data
          local
            LocalRestaurant
            PartialLocalRestaurant
            RestaurantsDao
            RestaurantsDb
          remote
            RemoteRestaurant
            RestaurantsApiService
          RestaurantsRepository
        domain
          GetInitialRestaurantsUseCase
          GetSortedRestaurantsUseCase
          Restaurant
          ToggleRestaurantUseCase
        presentation
          details
            RestaurantDetailsScreen.kt
            RestaurantDetailsViewModel
          list
            RestaurantsScreen.kt
            RestaurantsScreenState
            RestaurantsViewModel
          MainActivity.kt
      ui.theme
        Color.kt
        Shape.kt
        Theme.kt
        Type.kt
      RestaurantsApplication
    com.codingtroops.restaurantsapp (androidTest)
    com.codingtroops.restaurantsapp (test)
  res
Gradle Scripts
```

Figure 8.6 – Project structure after applying our package structuring strategy

Keep in mind, however, that when moving the **`MainActivity.kt`** file from its initial location to the **`presentation`** package, you might have to update the **`Manifest.xml`** file to reference the new correct path to the **`MainActivity.kt`** file:

```
<manifest […]>
    […]
    <application
        […]
        <activity
            android:name=".restaurants.presentation.
                MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.RestaurantsAp
p.
                NoActionBar">
            <intent-filter>
                […]
            </intent-filter>
            <intent-filter>
                […]
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Some versions of Android Studio do that out of the box for you; however, if they don't, you might end up with a nasty compilation error because the **`Manifest.xml`** file is no longer detecting our **`Activity`**.

Now that we have refactored the structure of our project, we can say that the packages structure provides us with immediate information about the features of the app (in our case, there is only one feature related to restaurants) and also with a clear overview of the components corresponding to a specific feature.

NOTE

*The autogenerated files for Compose projects (***`Color.kt`***,* **`Shape.kt`***,* **`Theme.kt`***, and* **`Type.kt`***) were left inside the* **`theme`** *package that resides inside the* **`ui`** *package. This is because theming should be consistent across features.*

Let's now move on to another way of improving the decoupling inside the UI layer between the composables and the **`ViewModel`**.

Decoupling the Compose-based UI layer from ViewModel

Our UI layer (represented by the composable functions) is tightly coupled to the **`ViewModel`**. This is natural, since the screen composables instantiate their own **`ViewModel`** to do the following:

- Obtain the UI state and consume it
- Pass events (such as clicking on a UI item) up to the **`ViewModel`**

As an example, we can see how the **`RestaurantsScreen()`** composable uses an instance of **`RestaurantsViewModel`**:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) -> Unit)
{
    val viewModel: RestaurantsViewModel = viewModel()
    val state = viewModel.state.value
    Box(…) { … }
}
```

The problem with our approach is that if we want to later test the UI layer, then, inside the test, the **`RestaurantsScreen`** composable will instantiate **`RestaurantsViewModel`**, which in turn will get data from Use Case classes, which in turn will trigger heavy I/O work in **`RestaurantsRepository`** (like the network request to obtain the restaurants, or the operation of saving them inside the local database).

When we have to test the UI, we should not care whether the **`ViewModel`** obtains the data correctly and translates it into a proper UI state. The effect of separating concerns is to facilitate testing a target class (or composable in this discussion) without having to care about other layers doing their work.

Right now, our screen composables are tied to a library dependency, the **`ViewModel`**, and it's ideal to decouple such dependencies as much as possible to promote reusability and testability.

In order to decouple the **`RestaurantsScreen()`** composable as much as possible from its **`ViewModel`**, we will refactor it so that the following happens:

- It will no longer reference a **`ViewModel`** class (the **`RestaurantsViewModel`** class).
- Instead, it will receive a **`RestaurantsScreenState`** object as a parameter.
- It will also define new function parameters to expose callbacks to its caller – we will see who the caller is in a minute.
  NOTE
  By extracting the **`ViewModel`** *instantiation from screen composables, such as* **`RestaurantsScreen()`**, *we're promoting reusability in the sense that we can much easier replace the ViewModel type that creates the state for this composable. This approach also enables us to port the Compose-based UI layer much easier to* **Kotlin Multiplatform** *(***KMP***) projects.*

Let's begin!

1. Inside the **`RestaurantsScreen`** file, update the **`RestaurantsScreen()`** composable by removing its **`viewModel`** and **`state`** variables, while also making sure it receives a **`RestaurantsScreenState`** object as a **`state`** parameter and an **`onFavoriteClick`** function:

```
@Composable
fun RestaurantsScreen(
    state: RestaurantsScreenState,
    onItemClick: (id: Int) -> Unit,
    onFavoriteClick: (id: Int, oldValue: Boolean) -
> Unit
) {
    Box(…) {
        LazyColumn(…) {
            items(state.restaurants) { restaurant -
>
                RestaurantItem(
                    restaurant,
                    onFavoriteClick = { id,
oldValue ->
                        onFavoriteClick(id,
oldValue)
                    },
                    onItemClick = { id ->
                        onItemClick(id)
                    }
                )
            }
        }
        […]
    }
}
```

Additionally, make sure to remove the **`viewModel.toggleFavorite()`** call and instead, call the newly added **`onFavoriteClick()`** function inside the **`RestaurantItem`** corresponding callback.

2. Since we changed the signature of the **`RestaurantsScreen()`** function, we must also update the **`DefaultPreview()`** composable to correctly call the **`RestaurantsScreen()`** composable:

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    RestaurantsAppTheme {
        RestaurantsScreen(
            RestaurantsScreenState(listOf(), true),
                {},
                { _, _ -> }
            )
        }
    }
```

3. In the **MainActivity** class and inside the **RestaurantsApp()** composable, make the destination composable for **RestaurantsScreen()** responsible for wiring up the screen composable with its **ViewModel**, thereby ensuring good communication between **RestaurantsScreen()** and **RestaurantsViewModel**:

```
@Composable
private fun RestaurantsApp() {
    val navController = rememberNavController()
    NavHost(navController, startDestination =
        "restaurants") {
        composable(route = "restaurants") {
            val viewModel: RestaurantsViewModel =
                viewModel()
            RestaurantsScreen(
                state = viewModel.state.value,
                onItemClick = { id ->
                    navController
                        .navigate("restaurants/$id"
)
                },
                onFavoriteClick = { id, oldValue ->
                   viewModel.toggleFavorite(id,
oldValue)
                })
        }
        composable(
            route = "restaurants/{restaurant_id}",
            […]) { RestaurantDetailsScreen() }
        }
```

With this approach, the destination `composable()` with the initial route of `"restaurants"` is the composable that manages and wires up the `RestaurantsScreen()` composable to its content by doing the following:

- Instantiating `RestaurantsViewModel`
- Getting and passing the state to `RestaurantsScreen()`
- Handling the `onItemClick()` and `onFavoriteClick()` callbacks

4. Build and run the application. The app should behave the same.
5. You will notice that if you rebuild the project and navigate back to the `RestaurantsScreen()` composable, the preview will now function correctly because the `RestaurantsScreen()` composable is no longer tied to a `ViewModel`, and so Compose can very easily preview its content.
   ASSIGNMENT
   In this chapter, we have better decoupled the first screen of the app (the `RestaurantScreen()` *composable) from its* `ViewModel` *to promote reusability and testability. As homework, you can practice doing the same for the* `RestaurantDetailScreen()` *composable.*

Summary

In this chapter, we have dipped our toes into Clean Architecture in Android. We started by understanding a bit about what Clean Architecture means and some of the best ways we can achieve this in our Restaurants app, while also covering the main benefits of following such a software design philosophy.

We started with Clean Architecture in the first section, where we defined the Domain layer with Use Cases, and continued refactoring in the second section, where we separated the Domain model from Data models.

Then, we improved the architecture of the app by creating a package structure and by decoupling the Compose-based UI layer from the `ViewModel` classes even further.

In the next chapter, we will continue our journey of improving the architecture of our application by adopting dependency injection.

Further reading

Clean Architecture is a very complex subject, and one chapter is simply not enough to cover it. However, one of the most important concepts that Clean Architecture brings is the Dependency Rule. The Dependency Rule states that within a project, dependencies can only point inward.

To understand what the Dependency Rule is about, let's visualize the layer dependencies of our Restaurants app through a simplified version of concentric circles. Each concentric circle represents different areas of software with their corresponding layer dependencies (and libraries).

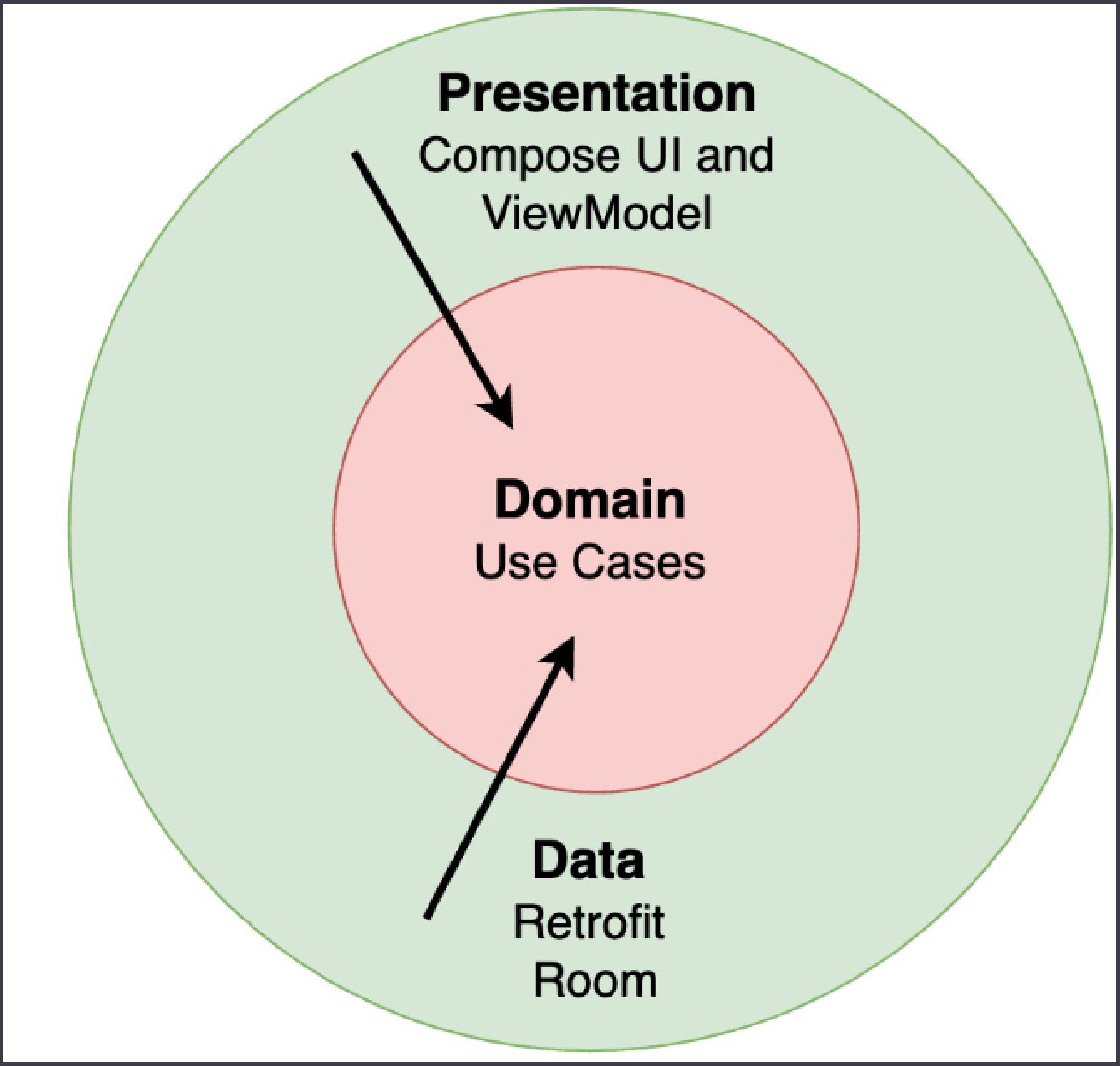


Figure 8.7 – The Dependency Rule with layers and components

This representation dictates that implementation details should be placed in *outer* circles (just as Compose is an implementation detail of the UI layer or Retrofit is an implementation detail for the Data layer), while business policies (Use Cases from the Domain layer) are placed within the *inner* circle.

The purpose of this representation is to enforce the Dependency Rule that states how dependencies should only be pointing inward.

The Dependency Rule (expressed with the inward-pointing arrows) showcases the following:

- The Presentation layer depends inward on the Domain layer (just like the `ViewModel` classes in our app correctly depend on Use Case classes) and how the Data layer should also depend inward on the Domain layer (in our app, Use Cases depend on `Repository` classes, while it should be the other way around – more on this in a second).
- The Domain layer should not depend on an outer layer – in our app, the Use Cases depend on `Repository` classes, which violates the Dependency Rule.

The approach of having the Presentation and Data layers (that contain details of implementation such as the Compose, Room, and Retrofit libraries) depend on the inner Domain layer is beneficial because it allows us to effectively separate the business policies (from within the inner circle, that is, the Domain layer) from outer layers. Outer layers can frequently change their implementation and we don't want these changes to impact the inner Domain layer.

In our Restaurants app though, the Domain layer depends on the Data layer because Use Case classes depend on `Repository` classes. In other words, the Dependency Rule is violated because the inner circle (the Domain layer) depends on an outer circle.

To fix this, we could define an **`interface`** class for the Data layer (for the **`Repository`** classes) and consider it part of the Domain layer (for now, by moving it inside the **`domain`** package).

This way, the Use Cases depend on an interface defined within the Domain layer, so now, the Domain layer has no outer dependencies. On the other hand, the **`Repository`** class (the Data layer) implements an interface provided by the Domain layer, so the Data layer (from the outer circle) now depends on the Domain layer (from the inner circle), thereby correctly adhering to the Dependency Rule.

NOTE

Another way of separating concerns (or layers) and making sure to respect the Dependency Rule is to modularize the app into layers, where each layer is a Gradle module.

I encourage you to study more about the Dependency Rule in Robert C. Martin's blog, while also checking out other strategies for achieving Clean Architecture: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.