Huy Bui

ANDROID APPLICATION FOR STUDENTS’ PERSONAL FINANCES

Bachelor’s thesis

Information Technology

Bachelor of Engineering

2020

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland
University of Applied Sciences

Author (authors) Huy Bui	**Degree title** Bachelor of Engineering	**Time** May 2020
Thesis title Android Application for Students' Personal Finances		58 pages 0 pages of appendices
Commissioned by Not given		
Supervisor Timo Mynttinen		

Abstract

College students often deal with multiple challenges at the same time, from academic exercises to personal finances. To help students to manage their personal finances, this thesis aims to create a simple Android application that records and analyses students' daily expenses. Unlike commercial finance application on the Google Play Store, the application created in this thesis is totally free. This is reasonable because students have considerable need of a high-quality finance management tool, but their budget is too limited to afford commercial application.

Throughout the theoretical, basic Android components, the Model-View-ViewModel architecture, dependency injection, and Room library were explained. The outcome indicated that the Android development is a time-consuming and frustrated process. This thesis also how a simple Android application is developed with good architecture practices.

Keywords

android, dependency injection, mvvm, kotlin, dagger

CONTENTS

1 INTRODUCTION

College students often have to spend a fortune for tuition costs. This results in tight monthly budget which should be enough to pay for good food, study materials, and other expenses. Choosing what, when and how to buy a product is a complicated decision. The primary objective of this project is to address the students' difficulty in making good financial decisions, by creating an Android application that tracks students' daily purchases, analyses their spending habits, and proposes good financial decisions.

The motivation for this project is derived from the fact that smart phones, especially Android, are the must have of college students everywhere. Apparently, using a mobile application dedicated to finance management is much more convenient than Microsoft Excel, which is commonly used by students to record their daily transactions.

A finance application on smart phones is not a new idea. There are already many applications on the market that serve the same purpose as this project. However, such applications require monthly subscriptions or in-app purchases to function properly, which is not affordable for students. This project aims to imitate the functionalities of finance applications on the market on a single mobile application and provide it for free.

The rest of this thesis is divided into two parts: Theoretical part and Practical part. The former part explains the background knowledge of Android development. This part starts with the concepts of activity and fragment, fundamental Android components. Followed by these basics are advanced Android libraries such as Room, Dagger and Live Data. The second part, on the other hand, focuses on the process of Android development. There are seven sections in this part corresponding to seven phases of the development process. In each section, the meaning of each block of code and the application of the theory mentioned in the Theoretical part will be explained thoroughly.

2 THEORECTICAL PART

This chapter introduces the concepts of mobile user experiences, activities, fragments, the Dagger library, View models, and the Room library. These concepts are crucial parts of the modern Android development process. Therefore understanding them will lead to deep and easy comprehension of this thesis.

2.1 Mobile user experience

It is a common habit that mobile users often interact with multiple applications in a short period of time. For example, suppose a user wants to take a photo and share it on a social network such as Facebook. The following task flow usually happens in such a situation:

1. The user is texting his/her friends on the Messenger application and clicks “take a photo” button. The Messenger application triggers the default camera application in the phone and navigates the user to that application.
2. The user takes a photo with the camera application and moves that photo to a photo editing application, let’s say Adobe, to blur the background of the photo.
3. Suddenly, a phone call interrupts the process. The user answers the phone and continues editing the photo.
4. Eventually, the user navigates back to the Messenger application and shares the photo.

Every application mentioned in the task flow above experiences an interruption, either expected or unexpected, from another application. The data in those applications has to be managed carefully so that the user can resume the task flow he/she has left at any time. This is the reason why the deep understanding of Android components such as activities and fragments is important. The deeper understanding of the Android system, the easier developers can optimize the application. The content of Section 2.2 and 2.3 serves that need where theory, lifecycle and the implementation of Android activities and fragments are explained.

According to the Android documentation (2020e), a mobile phone is a resource-constraint machine. This condition of the environment along with the multi-tasking habit of the users produces a high-pressure resource competition among applications. Therefore, memory leaks and heavy computing operation will make the application to crash frequently. To avoid such a terrible situation, good architecture and programming technique should be taken into account. There are many approaches for such purpose, but Model-View-ViewModel (Model-View-ViewModel) and dependency injection are recommended by the Android community. Overviews of these two concepts will be demonstrated in Section 2.4 and 2.6.

2.2 Activity

A typical Android application is a complex mixture of multiple components, including activities, fragments, content providers, and broadcast receivers. This section will give an overview of an Android activity.

According to Android Documentation (2019a), an Android activity displays a single screen with user interface. It’s likely that the user’s navigation between screens involves switching from an activity to another. Take Facebook Messenger as an example. The first screen of the application is a list of past conversations, which is, behind the scene, rendered by an activity, called Conversation List Activity. A click on any item of the conversation list would take the user to a list of messages. This magic is done by replacing Conversation List Activity (to save memory) with a new activity, called Message List Activity, to handle message listing and sending.

An Android activity does not stay in the system forever. In fact, it will be removed from the system, if the user closes the application. According to the Android documentation (2019a), all processes related to the Android activity that takes place after the activity is activated and before its termination is called *activity lifecycle*. Figure 1 demonstrates all details of the lifecycle of an activity in a diagram.

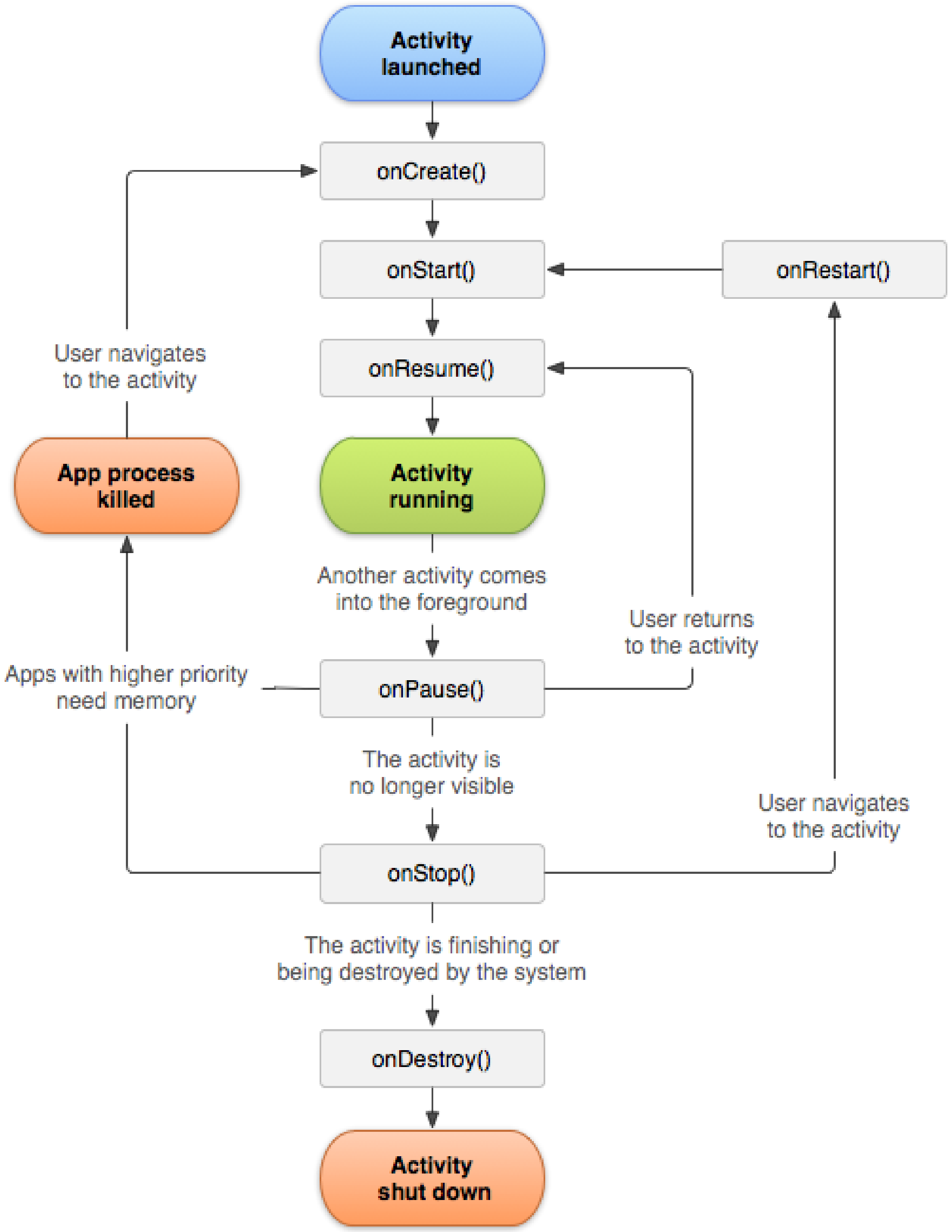
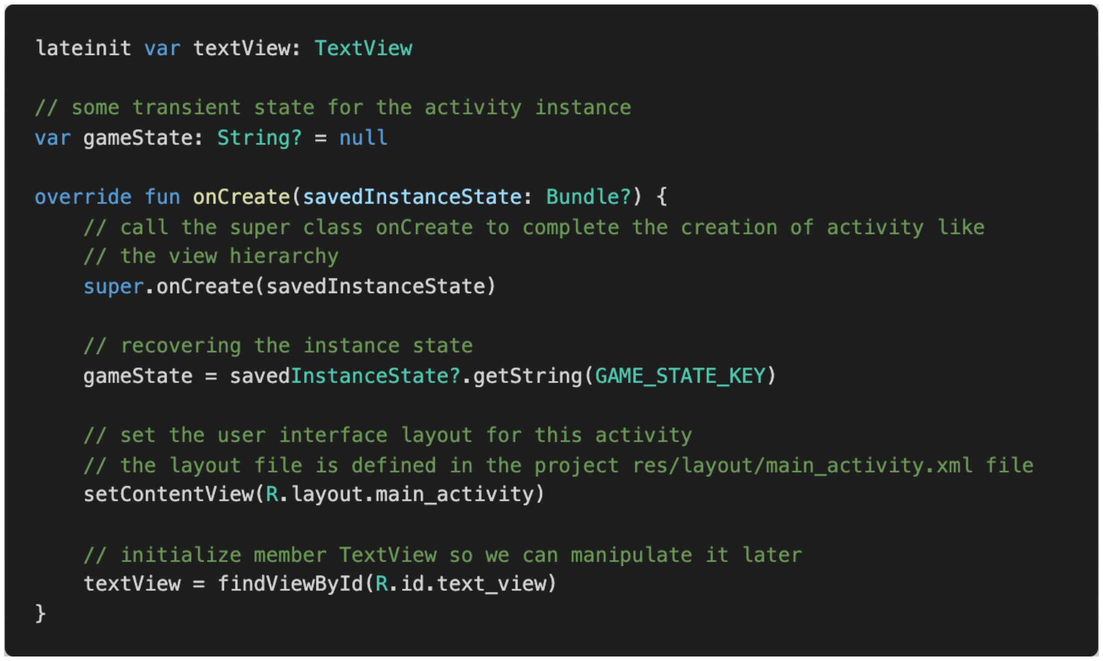

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 1. A simplified illustration of the activity lifecycle (Android Documentation 2019a)

Looking at Figure 1, we can see that the activity will invoke **onCreate()** method on its first launch. At this point, the activity has just woken up and does not display anything on the screen. The process then continues with **onStart()** method to render all view components whose functionality and animation are not available until **onResume()** is called. After these methods finish, the activity becomes active and ready to serve users' interaction.

From a programmer's perspective, the methods mentioned above are called *lifecycle call-backs* and they are responsible for running all the code of an application. In other words, not a single line of code will be executed unless it is located in one of the lifecycle call-backs. Due to these crucial reasons, it is important for programmers to understand the conceptual and implementation information of each lifecycle call-back. The detailed description of these call-backs, based on Android Documentation (2019a), will be shown in the following paragraphs.

onCreate() method: This method is called only once on activity creation. Basic start-up logic, such as declaring and binding data to the user interface, initializing variables, must reside in this method. Figure 2 shows an example of **onCreate()** implementation.



```
lateinit var textView: TextView

// some transient state for the activity instance
var gameState: String? = null

override fun onCreate(savedInstanceState: Bundle?) {
    // call the super class onCreate to complete the creation of activity like
    // the view hierarchy
    super.onCreate(savedInstanceState)

    // recovering the instance state
    gameState = savedInstanceState?.getString(GAME_STATE_KEY)

    // set the user interface layout for this activity
    // the layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity)

    // initialize member TextView so we can manipulate it later
    textView = findViewById(R.id.text_view)
}
```

Figure 2. An example of **onCreate()** implementation (Android Documentation 2019a)

onStart() method: At the end of **onCreate()**, the activity enters the Started state and executes the **onStart()** method very quickly. This method renders user interface components on the screen and prepares for the activity to enter the foreground. Unlike **onCreate()** method, which contains the majority of development code, **onStart()** is barely used by programmers. The reason lies on the short execution time of this call-back. **onStart()** keeps the activity to stay in the Started state in which the application is unresponsive. Therefore, the Android OS attempts to get out of the Started state quickly.

onResume() method: After **onStart()**, the Android activity continues with **onResume()** method, and enters the Resumed state. Having view components prepared by **onStart()** method, the activity is now responsive to every user's interaction. In other words, at the Resumed state, the start-up process of the activity finally ends. Unlike **onStart()** callback with the short longevity, **onResume()** never ends until an interruptive event occurs. For instance, **onResume()** will stop if the user presses on the Back button, or launches a new activity, or makes a phone call. If the activity returns from these events, it will enter the Resumed state and execute **onResume()**. For this reason, programmers should implement **onResume()** to assign resources (such as camera and text files) or to initialize variables.

onPause() method: Whenever an event interrupts a running activity, the **onPause()** method is called. This method swipes the activity from the foreground. Similar to the **onPause()** method, Android OS executes **onPause()** briefly. This is the reason why heavy-load operation, such as network calls and database transactions, should not occur during **onPause()**.

onStop() method: The system invokes **onStop()** call-back on the activity's termination, or the user's navigation to another activity. **onStop()** is the recommended place for time-consuming operations. Such operations might be releasing resources, pausing animations, shutting down CPU-intensive performance, or saving data to database.

onDestroy(): The system invokes this call-back during the activity's termination. All remaining resources that haven't been released on **onStop()** should be finally released here.

Knowing the concept of the Android activity and its call-backs is only one part of understanding the Android development process. The next concept that we will focus on is the Android fragment.

2.3 Fragment

This section will introduce the concept of the Android fragment. The information of this section derives from the Android Documentation (2019b), which is an overview provided by Google about Android fragments and its applications.

To make it simple, a fragment is a *sub-activity*. In other words, a fragment has all methods and behaviours of an activity. The only difference is that the operation of an activity is managed by the application itself, while a fragment is hosted by an activity. As a result, the fragment's lifecycle is directly affected by the host activity. Meaning that, when the activity enters the Paused state, so do all fragments in it; when the activity is destroyed, so are all fragments in it.

As mentioned before, a fragment contains methods similar to an activity, including lifecycle call-backs such as **onCreate()**, **onStart()**, **onPause()** and **onStop()**. However, there are also lifecycle methods that are specific to the fragment only, namely **onAttach()**, **onDetach()**, **onCreateView()**, **onDestroyView()**, and **onActivityCreated()** (see also Figure 3).

At a basic level, only **onCreate()**, **onCreateView()** and **onPause()** should be taken into account due to their significant impacts to the performance of a fragment. The following paragraphs demonstrate more details about the implementation and programming recommendations of these methods.

onCreate() method: As the name implied, **onCreate()** method is where the initialization of a fragment occurs. Therefore, this method is a good starting point for heavy and non-UI-related operations such as loading data from database, creating a rich-feature object or fetching user information. It is also recommended that any component which must be persisted during the entire life of the fragment should be initialized at **onCreate()** method.

onCreateView() method: Following **onCreate()** is the second-important call-back, **onCreateView()** upon which the system renders all UI components. It is common among Android community that variables referencing UI components are initialized in **onCreateView()**. Other features of the fragment that requires UI components should also take place in this method.

onPause() method: This method is an ideal place to save any data and clean up resources, as the fragment only call this method during its termination.

According to Android Documentation (2019b), besides generic fragments, there are types of fragments that serve specific needs of users: Dialog Fragment, List Fragment, and Preference Fragment Compat. Detailed explanation of these special types of fragments will be shown in the following paragraphs.

Dialog fragment: In an Android application, it is a common user experience that sometimes a small window appears to issue error messages or asking for the user's decision on some tasks. The mechanism behind such windows is the Dialog fragment. This is a helper class that has all behaviours of a fragment plus fancy animations such as “floating” in the middle of the screen or the ambient blur around a dialog. A Dialog fragment is recommended for displaying error or success signal of a process. It is a bad practice to use Dialog fragments for displaying a progress bar, because the application’s user interface would be unresponsive, if the operation takes too long.

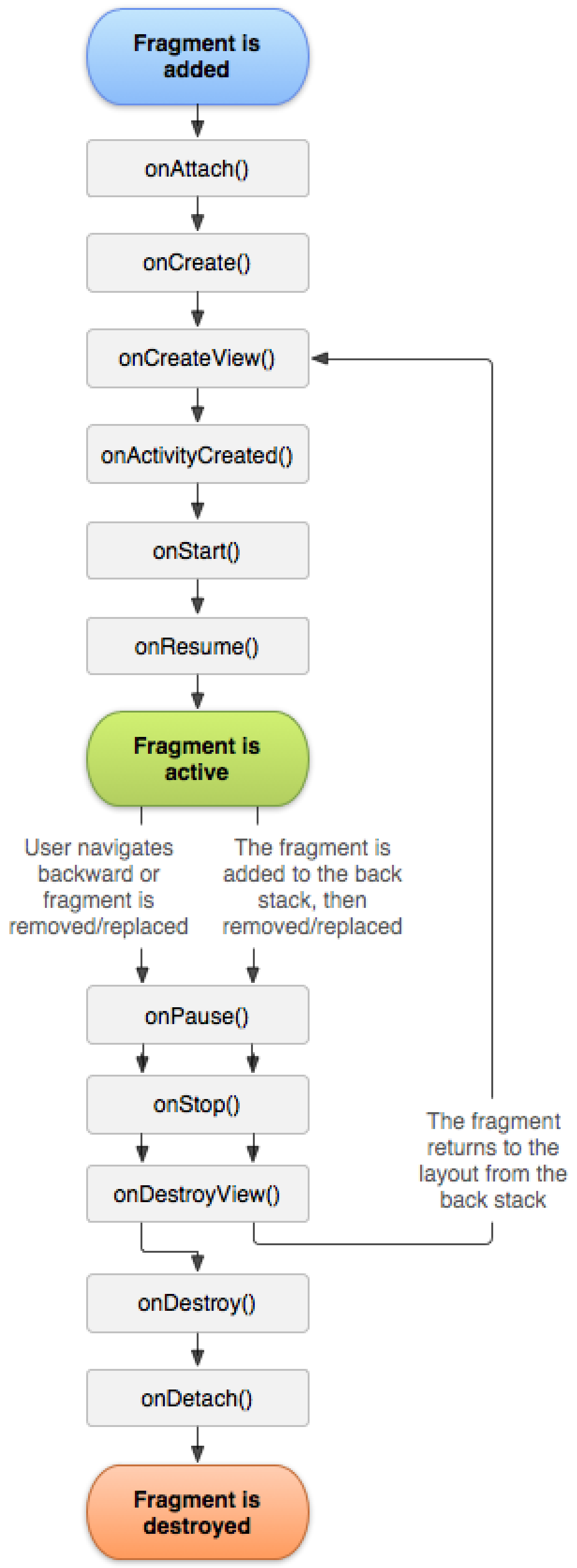


Figure 3. The lifecycle of a fragment while its activity is running (Android Documentation 2019b)

List fragment: It is common to see an Android screen with a list of items. A List fragment is a candidate that supports such behaviour. It displays a list of items that are managed by an adapter and provides several methods for managing a list view, such as the **onListItemClick()** call-back to handle click events. However, this fragment has become deprecated lately with the birth of Recycler views. It is a preferred method for displaying a list using a Recycler view instead of List view. In such cases, a fragment that includes a Recycler view in its layout should be created.

Preference Fragment Compat: This kind of fragment displays a hierarchy of Preference objects as a list. This is used to create a settings screen for your application.

It is the end of Section 2.3 where the basics of the Android development, meaning the concept of activities and fragments, should be covered. Knowing the basics, a developer should continue the learning journey with a clean architecture. One of the most popular architectures used in Android development is Model-View-ViewModel, or Model-View-ViewModel. The next part of this thesis will demonstrate the benefits and usage of the Model-View-ViewModel architecture.

2.4 The Model-View-ViewModel architecture

2.4.1 An overview

According to Google I/O (2017), in the past, Android developers used to create large activities and fragments. These components are responsible for both business and UI logic. This reduces the scalability of the application and complicates testing process.

To be aware of such problems, the Android community has developed many patterns that divide activities' and fragments' implementation into smaller and dedicated components. One of the most popular patterns is the Model-View-View Model architecture.

Looking at Figure 4, we can see the overall relationship among components of the Model-View-ViewModel architecture. It is obvious that the architecture establishes a layer hierarchy where View is the outside layer, View Model is the intermediary and Model is the inner layer. (Chugh 2019.) Each layer can only communicate with the layer next to them. For example, the View component can transfer data to View Model, but it is not allowed do the same thing to the Model and vice versa.

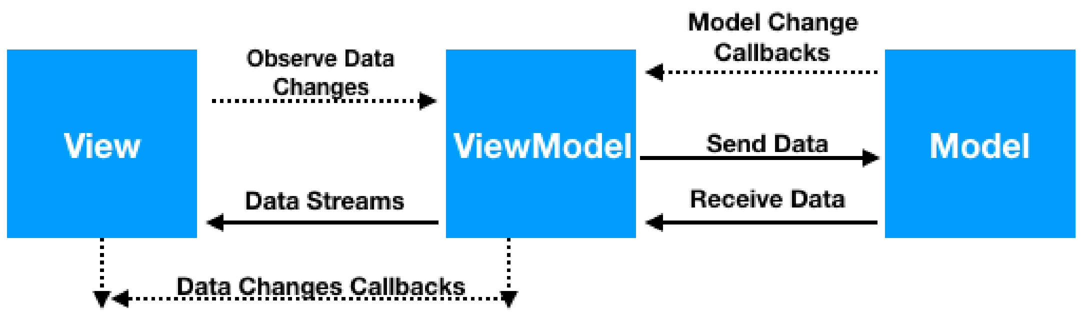


Figure 4. Model-View-ViewModel implementation diagram (Chugh 2019)

The following paragraphs explain in detail the operation and implementation of each component of the Model-View-ViewModel architecture, based on the information provided by Android Documentation (2019c).

Model components: As the name implied, components of this type define the data structure across the application. Model components also hold data both in persistent database and run-time environment.

View components: All layout files are considered View components. They are often named after the activity or fragment that host it. For example, the layout file of Conversation List Activity would be **activity_conversation_list**. View components convert data from View Models into human-readable and beautiful graphical artefacts.

View Model components: All data operations should take place in View Model components. These components either fetch data from the Internet and retrieve it from the database, and then convert it into user-desired data. The result data is finally inflated by View components. In other words, View Model components act as links between the Model and the View components. In some cases, View Model components also use hooks or call-backs to update the View.

On the whole, the components of the Model-View-ViewModel architecture should work together in the layer-hierarchy manner to provide not only a reactive but also high-maintenance application. To optimize these benefits of the Model-View-ViewModel architecture, each Model-View-ViewModel component should have only one responsibility. Because Model components are supposed to hold data, they should not be responsible for rendering graphics, or dealing with business logic. Likewise, View Model components should fulfil their only responsibility which is collecting and transforming data. View components, on the other hand, should do nothing but generate the user interface.

It is this end of section 2.4 where we have covered the overview of the Model-View-ViewModel architecture. It is important to keep in mind the responsibility of each Model-View-ViewModel component: Model, View Model and View, and how they work together to enhance the scalability and ease the maintenance process.

2.4.2 The lifecycle of a View Model component

To understand more about Model-View-ViewModel, we need a much more in-depth explanation for the View Model components. Let's start with the lifecycle of a View Model component. As we are already familiar with the Android activity, it is reasonable to compare the lifecycle of a View Model component to that of an activity (Figure 5).

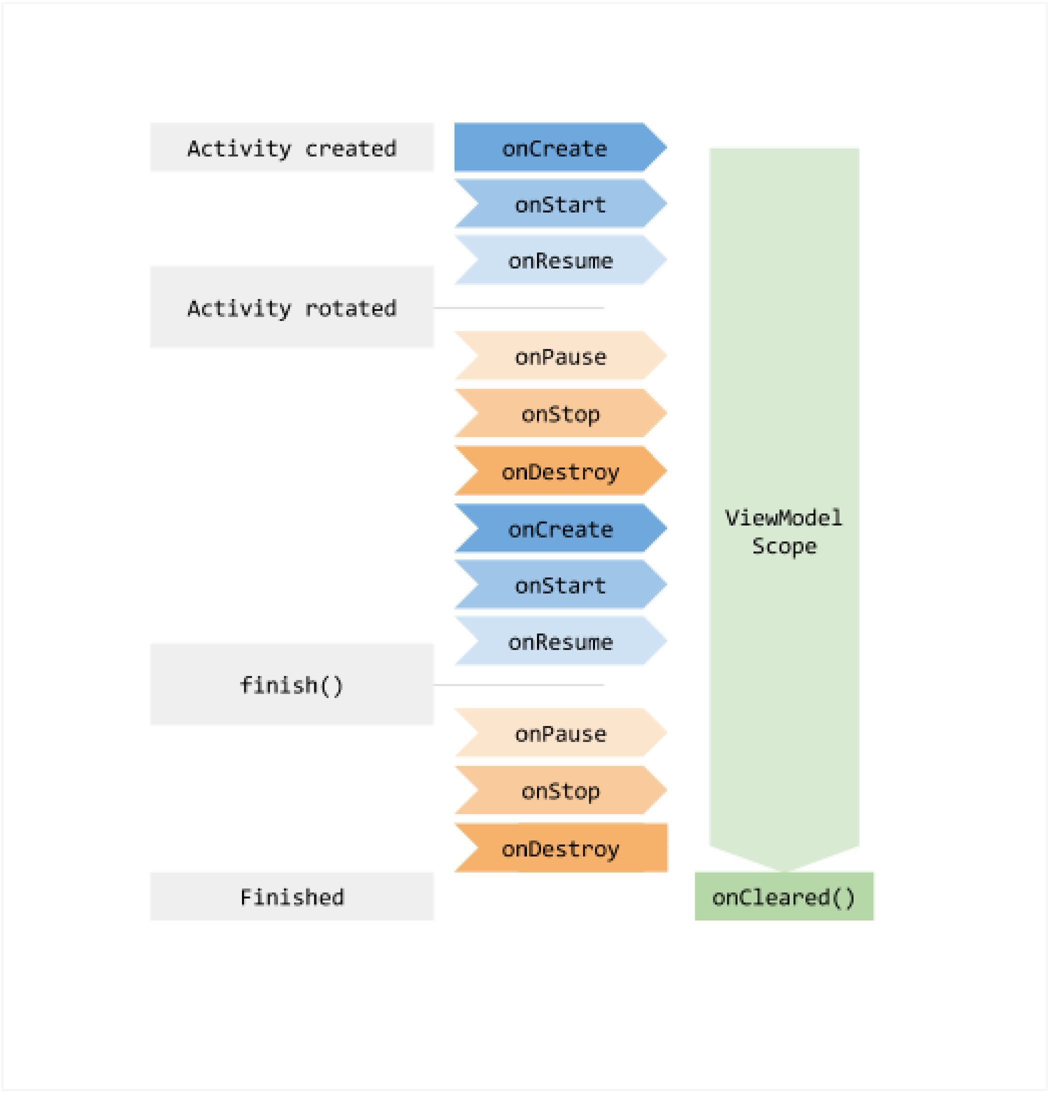


Figure 5. The lifecycle diagram of a View Model component compared to the lifecycle of the activity hosting it (Android Documentation 2019c)

As we saw in Figure 5, the View Model component is created when its host activity is launched. However, when the device is rotated, meaning that the host activity is restarted by the Android operating system, the View Model component still persists in the memory. This ensures that when the host activity is brought back (after the rotation) to the exact same state before the rotation. In other words, any data held by the View Model component will not be lost after device rotation.

This feature of the View Model has significant impact on the application's performance. Suppose that our application frequently makes network calls to fetch heavy data from a web server. Without the View Model, whenever the device is rotated or a phone call arrives, the data you have just downloaded will be swiped away. It is a waste of resources since the application may have to reissue the network calls it has already made. On the contrary, with View Model and Model-View-ViewModel pattern, no additional network calls are required after device rotation or configuration changes. (Android documentation 2019c.)

The View Model does not live forever, of course. According to the Android Documentation (2019c), a View Model component remains in memory until the lifecycle it's scoped to goes away permanently. In other words, if the View Model is scoped to an activity, it will be destroyed when the activity is finished. In case of a fragment, the View Model will remain in the memory until the fragment is detached.

During the View Model's termination, **onClear()** method is called. To prevent memory leaks, all network calls and resources binding should be cleaned up in the **onClear()** method.

2.4.3 An example implementation

There is an example on Model-View-ViewModel implementation in Android Documentation (2019c) that we should pay attention to. This section simply explains that example in more detail using the concept of the View Model mentioned in Section 2.4.2.

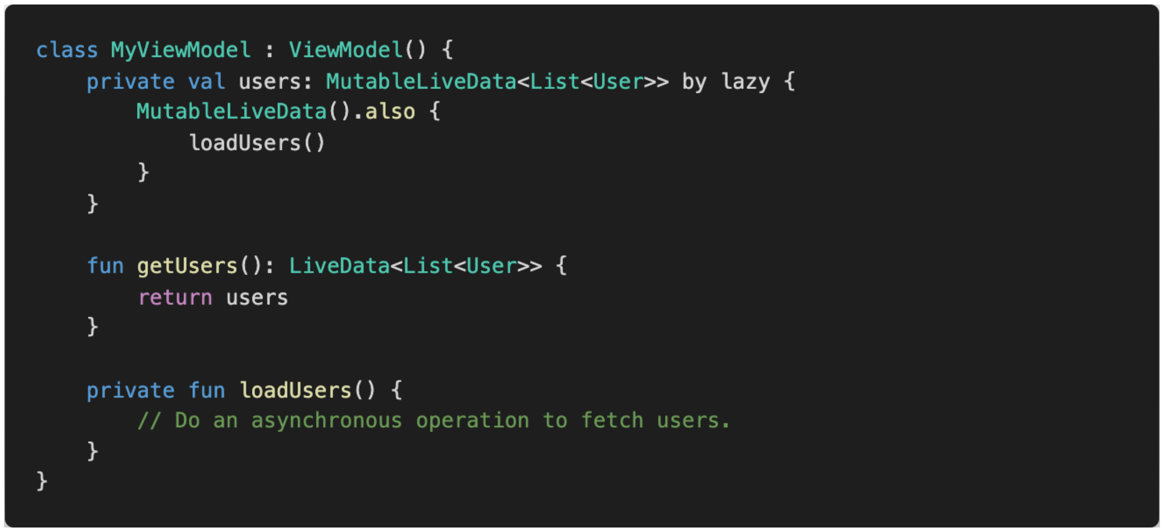
The context of the example is that we are supposed to build a fragment that displays a list of users. The first thing to do when implementing the Model-View-ViewModel architecture is to create components, meaning that there should be at least one file for the View, the View Model, and the Model, as follows:

- **UserProfile.java:** This is the Model component responsible for defining data structure of each user's information in database and run-time environment. The View Model utilizes this file to retrieve data from the database and to store that data in the memory.

- **fragment_user_profile.xml:** This is the View component which decides graphical demonstration of data from the View Model component.

- **UserProfileFragment.java:** This is the fragment, or UI controller that glues the View and the View Model components together.

- **UserProfileViewModel.java:** This is the View Model component whose job is acquiring and keeping the list of users. Figure 6 illustrates the implementation of this component.

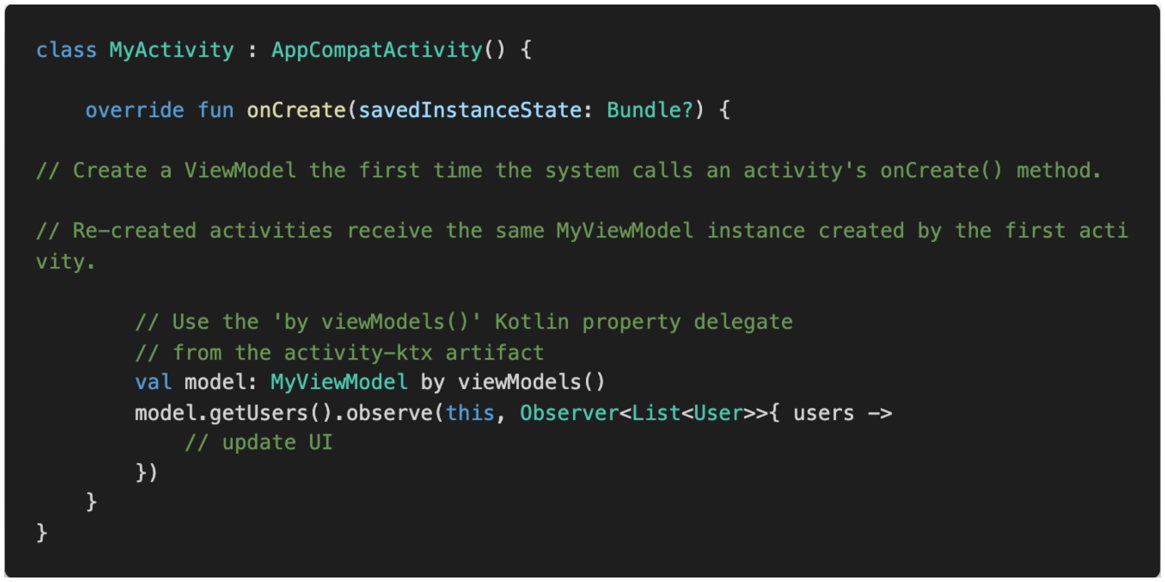


```
class MyViewModel : ViewModel() {
    private val users: MutableLiveData<List<User>> by lazy {
        MutableLiveData().also {
            loadUsers()
        }
    }

    fun getUsers(): LiveData<List<User>> {
        return users
    }

    private fun loadUsers() {
        // Do an asynchronous operation to fetch users.
    }
}
```

Figure 6. Implementation of UserProfileViewModel.java (Android Documentation 2019c)

When building a View Model component, programmers need to keep in mind that "A View Model must never reference a view, lifecycle, or any class that may hold a reference to the activity context", according to Android Documentation (2019c). The reason is that any references to the UI controller, meaning the host activity or fragment, will be lost on configuration changes due to interruption, such as device rotation or a phone call. In such cases, the Android operating system creates a new instance of the host activity (or fragment) with new views, lifecycles, and context. This means that all references to the old UI controller in the View Model will be invalid. Consequently, the application will crash because of Null Pointer Exception.

Last but not least, the UI controller needs to know about its View Model component. The association between the UI controller and its View Model is created inside **onCreate()** method using **ViewModelProviders** (Figure 7).



```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {

// Create a ViewModel the first time the system calls an activity's onCreate() method.

// Re-created activities receive the same MyViewModel instance created by the first acti
vity.

        // Use the 'by viewModels()' Kotlin property delegate
        // from the activity-ktx artifact
        val model: MyViewModel by viewModels()
        model.getUsers().observe(this, Observer<List<User>>{ users ->
            // update UI
        })
    }
}
```

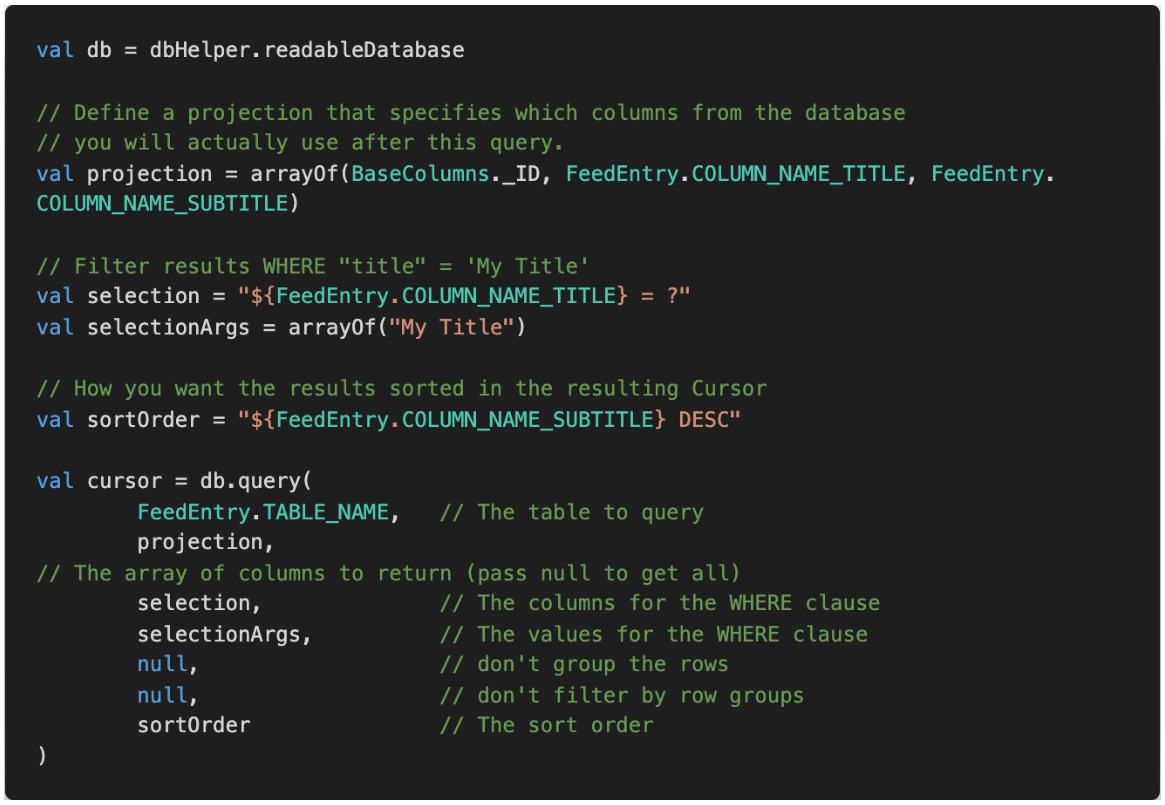
Figure 7. Creating the association between the UI controller and its View Model component (Android Documentation 2019c)

To summarize, this chapter described the Model-View-ViewModel (Model-View-ViewModel) architecture, which is one of the most popular patterns used in Android development. The end of the chapter provided basic knowledge of the Model-View-ViewModel components and how to implement those components in a code project.

2.5 Room library

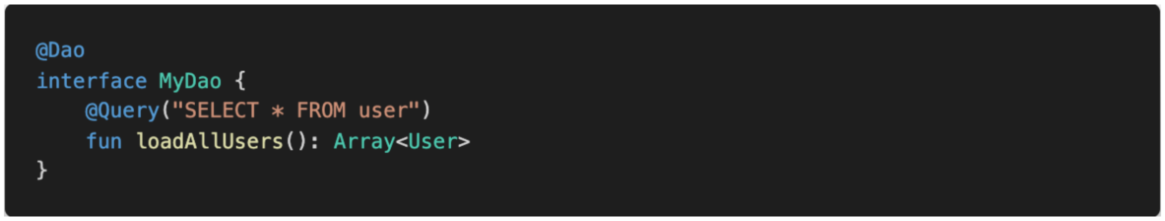
We will use Room library (or Room for short) to store the users' data in a structured database. This section gives an overview of Room and how it fits into the overall Android system. The knowledge in this section derives from the Android Documentation (2020a).

Room is built on top of SQLite, which is the traditional database structure of Android applications. To be more detailed, Room provides an abstraction layer over SQLite to allow fluent, short and efficient database query. Before the release of Room, programmers have to write long database query as shown in Figure 8. With the help of Room, on the other hand, that query can be re-written as shown in Figure 9.



```
val db = dbHelper.readableDatabase

// Define a projection that specifies which columns from the database
// you will actually use after this query.
val projection = arrayOf(BaseColumns._ID, FeedEntry.COLUMN_NAME_TITLE, FeedEntry.
COLUMN_NAME_SUBTITLE)

// Filter results WHERE "title" = 'My Title'
val selection = "${FeedEntry.COLUMN_NAME_TITLE} = ?"
val selectionArgs = arrayOf("My Title")

// How you want the results sorted in the resulting Cursor
val sortOrder = "${FeedEntry.COLUMN_NAME_SUBTITLE} DESC"

val cursor = db.query(
        FeedEntry.TABLE_NAME,   // The table to query
        projection,
// The array of columns to return (pass null to get all)
        selection,              // The columns for the WHERE clause
        selectionArgs,          // The values for the WHERE clause
        null,                   // don't group the rows
        null,                   // don't filter by row groups
        sortOrder               // The sort order
)
```

Figure 8. Old-fashioned SQLite database query (Android Documentation 2020a)



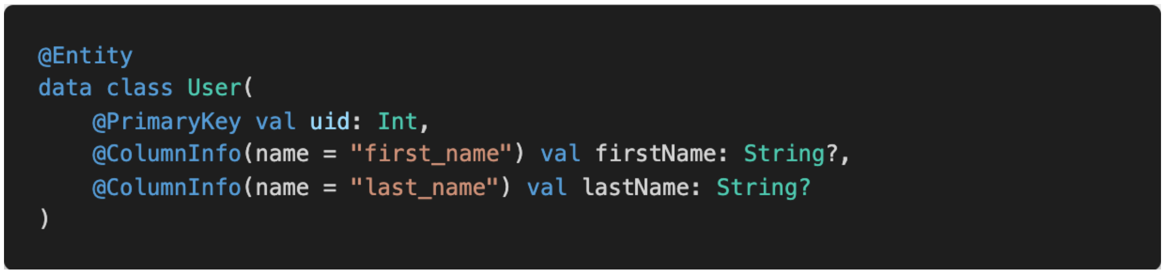
```
@Dao
interface MyDao {
    @Query("SELECT * FROM user")
    fun loadAllUsers(): Array<User>
}
```

Figure 9. Writing SQLite database query with the help of Room (Android Documentation 2020a)

There are three major components in Room: Database, Entity and DAO. The following paragraphs will introduce more details about these components and how they work together.

Database: This is the main interface that interacts directly to the underlying SQLite database. Programmers should create a Database component using an abstract class that inherits from the **RoomDatabase** class and has the **@Database** annotation above the its declaration.

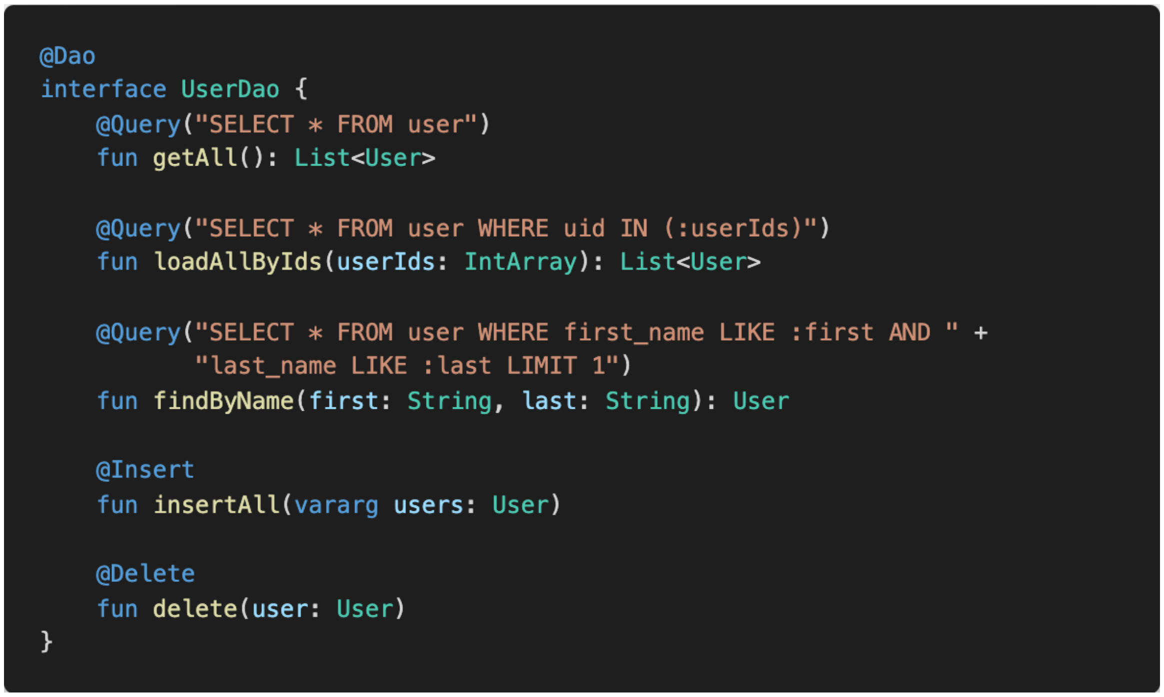
Entity: An entity is a row in SQL language. However, in Room library, an entity is understood as a single data type. Each instance of a data type is stored as a row in the table that has the same name as that of the data type it serves. All properties on the entity class are automatically defined as fields (or columns) in the database. An entity is created by simply annotating a class with **@Entity**, as shown in Figure 10. Furthermore, one or more properties of the entity class has to be annotated with **@PrimaryKey** to become the primary key of the table.



```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Figure 10. An example of creating an entity (Android Documentation 2020a)

DAO: Data Access Object, or DAO, is where we define database queries. To create a DAO, programmers create either an interface or an abstract class, and annotates it with **@DAO**, as shown in Figure 11. Any queries inside a DAO, should be annotated according to its purpose, such as **@Insert** for adding data, **@Update** for modifying data, **@Delete** for deleting a row, **@Query** for retrieving data. It is noted by Google that at least one abstract method that returns a DAO should be placed in the Database component.



```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

Figure 11. An example of creating a DAO (Android Documentation 2020a)

As stated in the Android Documentation 2020a, Room database is created on the start-up of the Android application. Because this initialization is expensive, it is recommended that only one instance of Room database be created and that one should live as long as the application is running. When the Room database is created, as we can see in Figure 12, the application uses a DAO to query desired entities from the database. It is then the job of DAO to utilizes the entity to either get or set values corresponding to the table columns within the database.

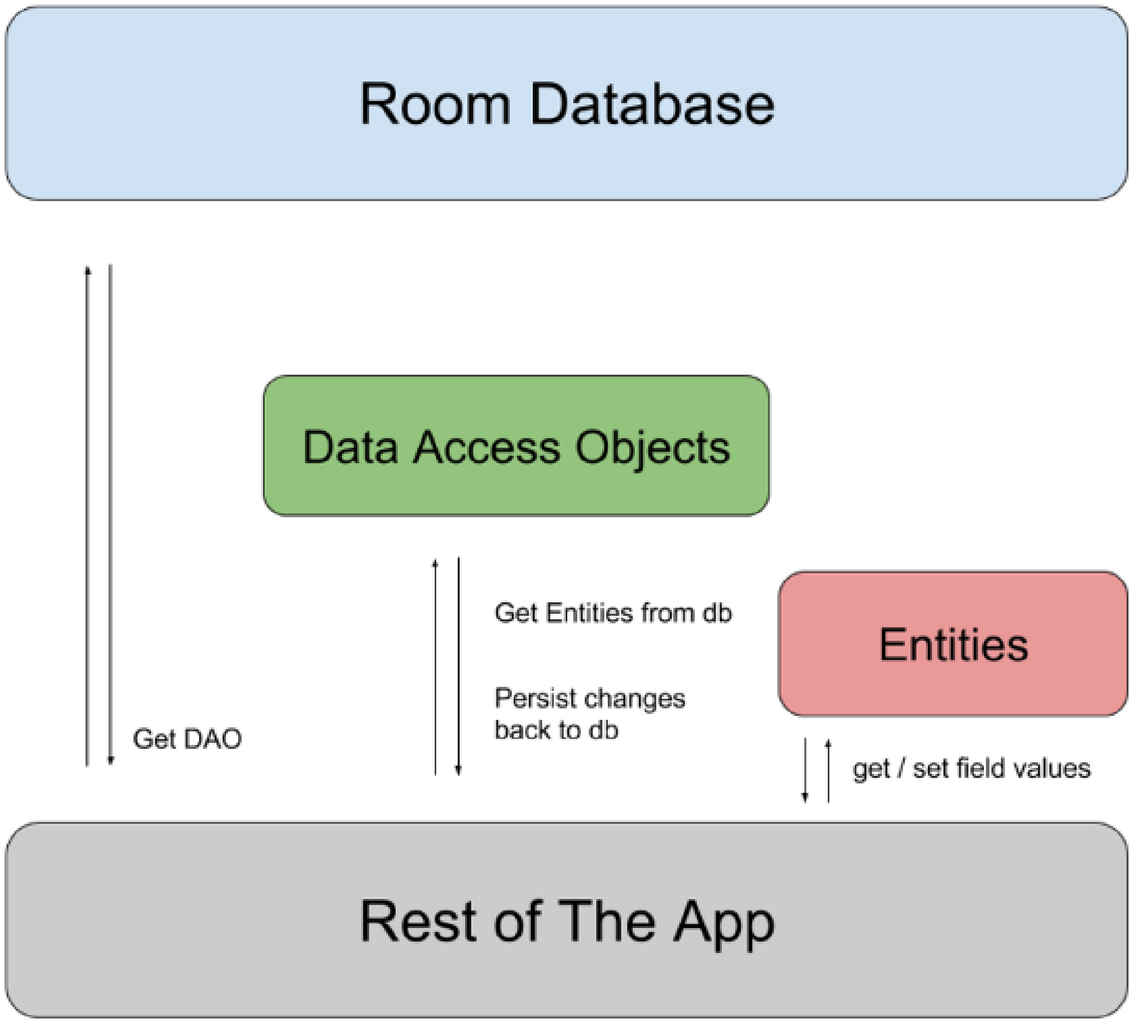


Figure 12. Room architecture diagram (Android Documentation 2020a)

Room, in short, is awesome and has made Android development more efficient in the past few years. Having read this section, readers are now familiar with all concepts and basic usages of this fascinating library. In the practical part of this thesis, more advanced techniques with Room will be used and explained later. So that it is important that concepts of this section will be kept in mind for later comprehension of this thesis.

2.6 Dependency injection

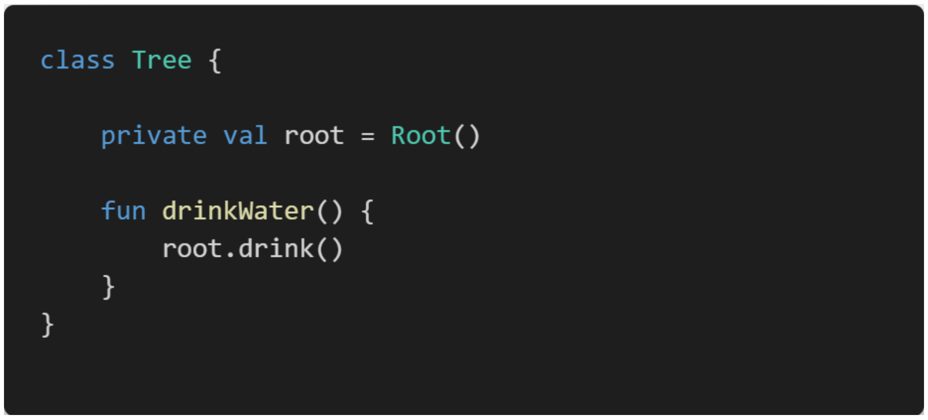
2.6.1 An overview

Dependency injection (DI) is a popular technique in programming that removes boilerplate code and increases the scalability and performance. The principle of DI has become an indispensable criterion of reliable applications. Based on the explanation in the Android documentation (2020f), this section provides an overview of how dependency injection works and how to use it.

In terms of programming, an application is an orchestration of classes and objects. In other words, we cannot avoid the fact that a class requires references to other classes. For example, a **Tree** class might need references to the **Root**, or **Trunk**, or **Leaf** classes. These references are called *dependencies*.

There are three ways for a class to get its dependencies, as follows:

1. The class creates the dependency it needs. In the example above, the implementation of class **Tree** would include the initialization of a **Root** or a **Trunk** object (Figure 13).
2. All dependencies of the application are created somewhere, and the class just grabs whatever dependencies it needs. In the example above, a **Root** object may be initialized at the start-up process and kept in the memory, and whenever a **Tree** is created, it would grab the instance of **Root** in the memory to use.
3. The dependencies are supplied as parameters. In the example above, suppose that a **Tree** object requires a **Root** and a **Trunk** object, then the **Tree**'s constructor requires one parameter of type **Root** and one of type **Trunk**.


```
class Tree {

    private val root = Root()

    fun drinkWater() {
        root.drink()
    }
}
```
Figure 13. The **Tree** class creating dependencies it needs.

The first option is a bad practice because of the following reasons:

- It is difficult to write a unit test for the **Tree** class because the real instances of **Root** or **Trunk** cannot be mocked.
- Even small changes in the constructor signature of the **Root** or **Trunk** could force the implementation of the **Tree** class to change as well. Imagine if this technique is used across the whole application, small change in one class results in massive modification of other classes,

meaning a small bug can cost a significant amount of time and money to fix.

The second option is better than the first one, because small changes in the constructors of any dependencies only result in the modification of a single class. However, initializing all dependencies at once allocates a significant amount of hardware resources. This is dangerous, because once the system runs out of hardware resources, the whole application will crash.

The third option is called *dependency injection* and is the best technique of all. This approach allows us to mock all dependencies in testing. In the example of the **Tree** class, different implementations of **Root** can be passed into an instance of **Tree** without any further changes. Moreover, this approach only initializes dependencies when they are needed, and swaps those from the memory that are not in use anymore. In other words, dependency injection makes the code testable, scalable and reduces the amount of resource allocation.

There are two types of dependency injections: *manual* dependency injection and *automatic* dependency injection. In Android development, manual dependency injection uses containers as a way to share instances of classes in different parts of the application (Android Documentation, 2020b). This approach results in boilerplate and error-prone code. This is where automatic dependency injection comes into place. In this approach, libraries such as Dagger and Koin automatically generate containers for dependencies. It is developers' job to annotate which classes are dependencies of which.

2.6.2 Dagger library

In section 2.6.2, we cover the definition and motivation of dependency injection. In this section, we will learn a specific library called *Dagger* that handles dependency injection automatically.

According to Android Documentation 2020c, Dagger provides the following benefits:

- Dagger frees you from writing boilerplate and error-prone code by automatically generating containers for dependencies.
- Dagger allows developers to decide the longevity of dependencies, so that unused dependencies will be discarded as soon as possible to save memory space.
- Dagger generates code at the compile time, so that it is traceable and performant.

Dagger is powerful but very easy to use. Let's use Dagger to create dependencies according to Figure 14. In the figure, the **UserRepository** class requires two dependencies, one of type **UserLocalDataSource** and the other is **UserRemoteDataSource**. As stated in section **Error! Reference source not found.**, dependencies should be "injected" as parameters to the constructor. With that in mind, we define **UserRepository** as shown in Figure 15.

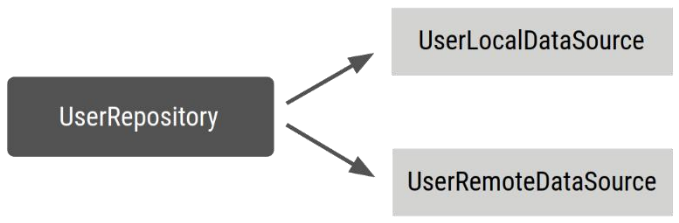
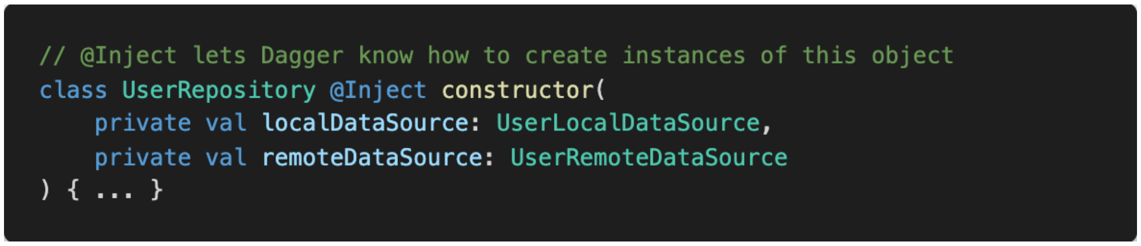


Figure 14. Dependency diagram of the **UserRepository** class (Android Documentation 2020c)

Notice that there is an **@Inject** annotation to the constructor of the **UserRepository** class. This annotation tells Dagger to inject dependencies that match the data type of each parameter. In this case, Dagger inject instances of **UserLocalDataSource** and **UserRemoteDataSource**.

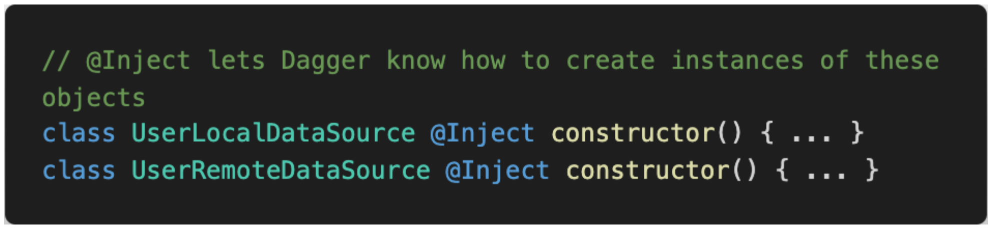


```
// @Inject lets Dagger know how to create instances of this object
class UserRepository @Inject constructor(
    private val localDataSource: UserLocalDataSource,
    private val remoteDataSource: UserRemoteDataSource
) { ... }
```

Figure 15. The constructor signature of the **UserRepository** class

With **@Inject** annotation, Dagger knows how to initialize a **UserRepository** instance, but it has no ideas how to create the **UserLocalDataSource** and

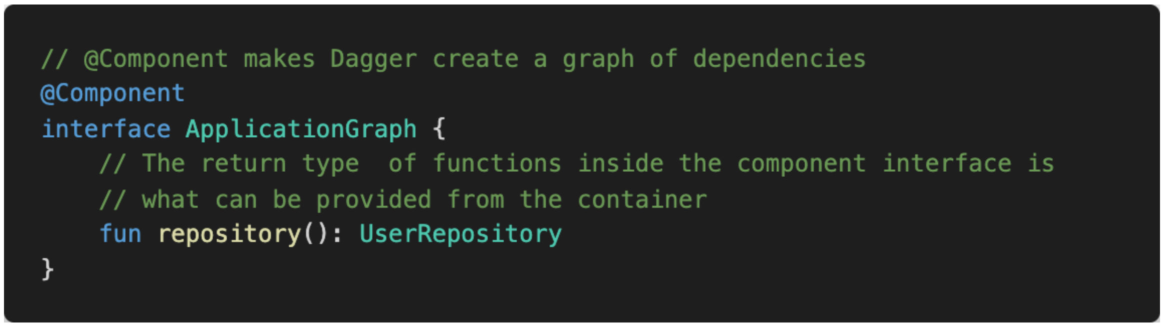
UserRemoteDataSource dependencies. Similarly, **@Inject** annotations should be placed at the constructor of these classes**,** as shown in Figure 16.



```
// @Inject lets Dagger know how to create instances of these
objects
class UserLocalDataSource @Inject constructor() { ... }
class UserRemoteDataSource @Inject constructor() { ... }
```

Figure 16. The constructor signature of the **UserLocalDataSource** class and the **UserRemoteDataSource** class

When all dependencies already has **@Inject** annotations, it is required to have a container responsible for creating and injecting these dependencies at run time. In Dagger, such containers are called graphs. To create graphs, Dagger needs an interface with **@Component** annotation as a guidance. It is developers' job to create the signature of this interface, as shown in Figure 17.

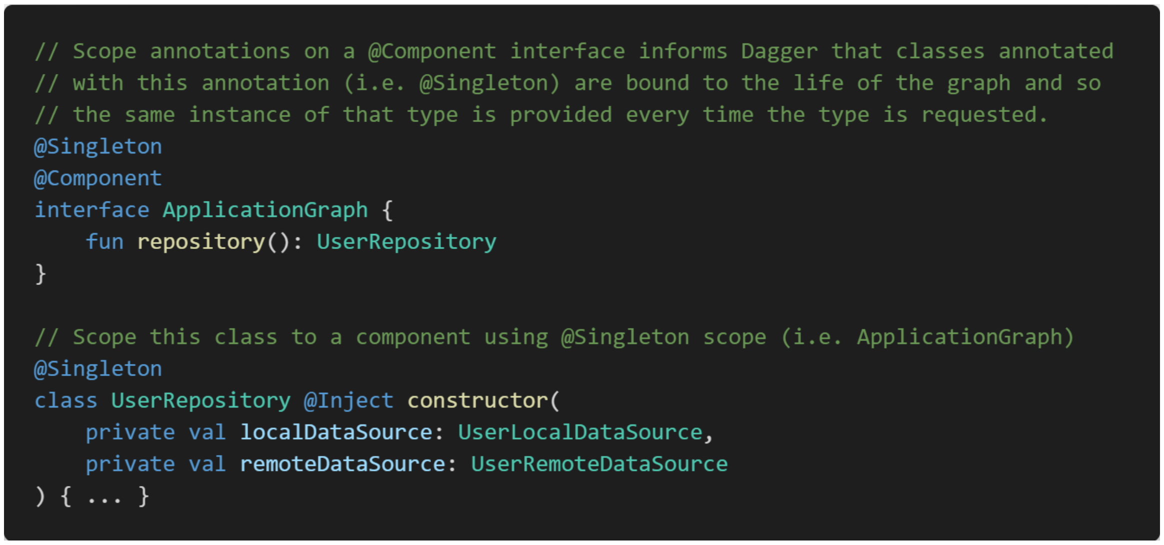


```
// @Component makes Dagger create a graph of dependencies
@Component
interface ApplicationGraph {
    // The return type  of functions inside the component interface is
    // what can be provided from the container
    fun repository(): UserRepository
}
```

Figure 17. A sample signature of the interface responsible for creating and injecting dependencies

When generating the graph from the **ApplicationGraph** interface, Dagger finds ways to create the dependency that matches the return type of each method in the interface. For example, in Figure 17, Dagger browses through the entire code base to find one and only one class with the name of **UserRepository**, or at least a subclass of it. Dagger then searches for all dependencies of this class based on the class' constructor. Finally, an implementation of **repository()** abstract method is created with all required dependencies.

That being said, creating an object with complicated dependency relationship is resource consuming. It is worse if such objects are reused multiple times across the application. *Singleton*, “a software pattern that restricts the instantiation of a class to one single instance” (Wikipedia, 2020), is an approach to solve such issue. Any classes with **@Singleton** annotation will be initialized once and kept in the memory through entire application (Figure 18). Any usage of such classes does not require initialization anymore but referencing to memory space instead.

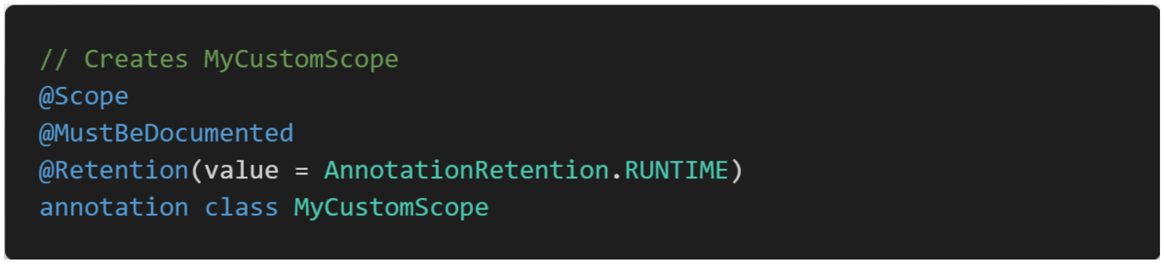


```
// Scope annotations on a @Component interface informs Dagger that classes annotated
// with this annotation (i.e. @Singleton) are bound to the life of the graph and so
// the same instance of that type is provided every time the type is requested.
@Singleton
@Component
interface ApplicationGraph {
    fun repository(): UserRepository
}

// Scope this class to a component using @Singleton scope (i.e. ApplicationGraph)
@Singleton
class UserRepository @Inject constructor(
    private val localDataSource: UserLocalDataSource,
    private val remoteDataSource: UserRemoteDataSource
) { ... }
```

Figure 18. Creating a singleton with the **@Singleton** annotation

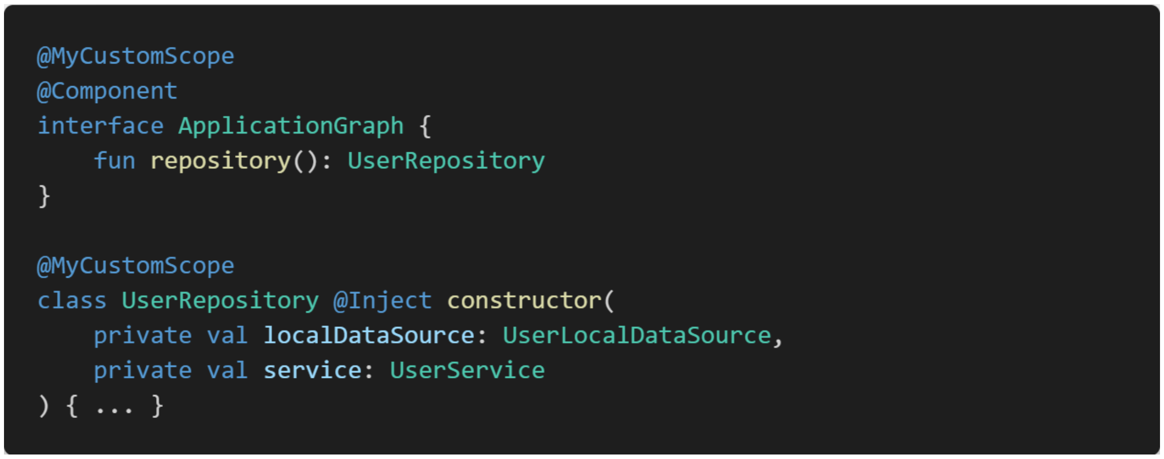
Annotation **@Singleton** actually represents a Dagger scope which is a longevity of an object (in memory). In Figure 18, the **UserRepository** class has the same **@Singleton** annotation as **ApplicationGraph**. This means that the lifetime of the **UserRepository** instance is bound to that of the **ApplicationGraph** instance.

Besides **@Singleton** annotation, custom scope annotations can be created in form of a class, as shown in Figure 19. In Dagger, every scope annotation has the same functionality no matter how it is named. The **@MyCustomScope** annotation in Figure 19 has the same functionality as the **@Singleton** annotation mentioned above. The final output of the application is still the same if the **@Singleton** annotation is completely replaced by **@MyCustomScope**.



```
// Creates MyCustomScope
@Scope
@MustBeDocumented
@Retention(value = AnnotationRetention.RUNTIME)
annotation class MyCustomScope
```

Figure 19. A custom scope annotation created in form of a class (Android documentation, 2020d)



```
@MyCustomScope
@Component
interface ApplicationGraph {
    fun repository(): UserRepository
}

@MyCustomScope
class UserRepository @Inject constructor(
    private val localDataSource: UserLocalDataSource,
    private val service: UserService
) { ... }
```

Figure 20. Replacing the **@Singleton** annotation by the **@MyCustomScope** annotation (Android documentation, 2020d)

In this section, basics of Dagger including Dagger's benefits in application development, graphs and annotations are covered briefly. Dagger is the core library that constructs the architecture of the mobile application in this project, so it is important that readers get an overview of Dagger. To make the theory part easy to read, more advanced topic of Dagger such as multi-binding and sub-component will be explained later in the practical part of this thesis, where more complicated situation occurs.

3 PRACTICAL PART

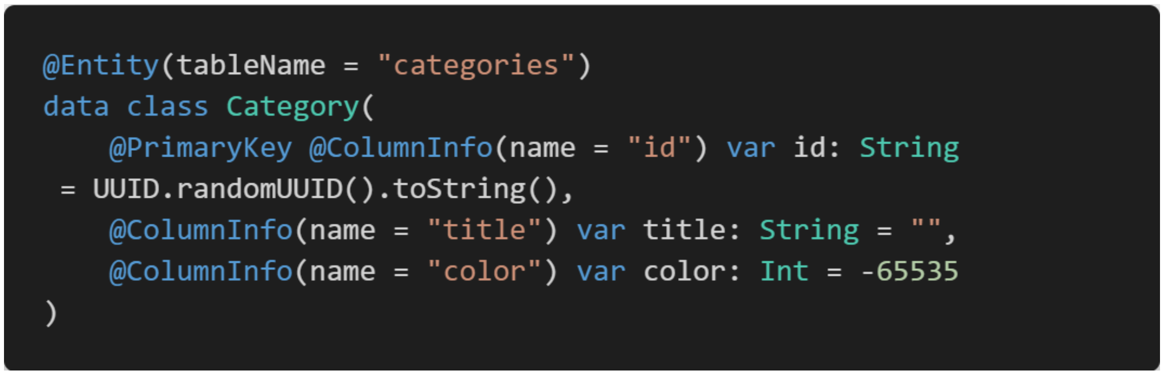
This chapter describes the structure and development process of the Android application called *Fitsu* which helps student manage their monthly finances. All concepts involved in this chapter base on or extend those explained in Chapter **Error! Reference source not found.**. Therefore, there will be a significant n umber of references to chapter 2 for brevity.

3.1 Design data model

In terms of programming, an application is an orchestration of classes. Classes are responsible for either holding data (data class) or performing an action on data. This section describes classes that store data in the Fitsu application, following the instructions stated in Android documentation 2020e.

There are three data classes in the Fitsu application: **Category**, **Transaction** and **Budget**. Each class contains an “ID” field, so that an object can be identified when stored on the local database. The value of this field is generated randomly by the **UUID** class. Room annotations such as **@PrimaryKey**, **@Entity** and **@ColumnInfo** are used to inform Room library about the properties of the class and how each property should be treated.

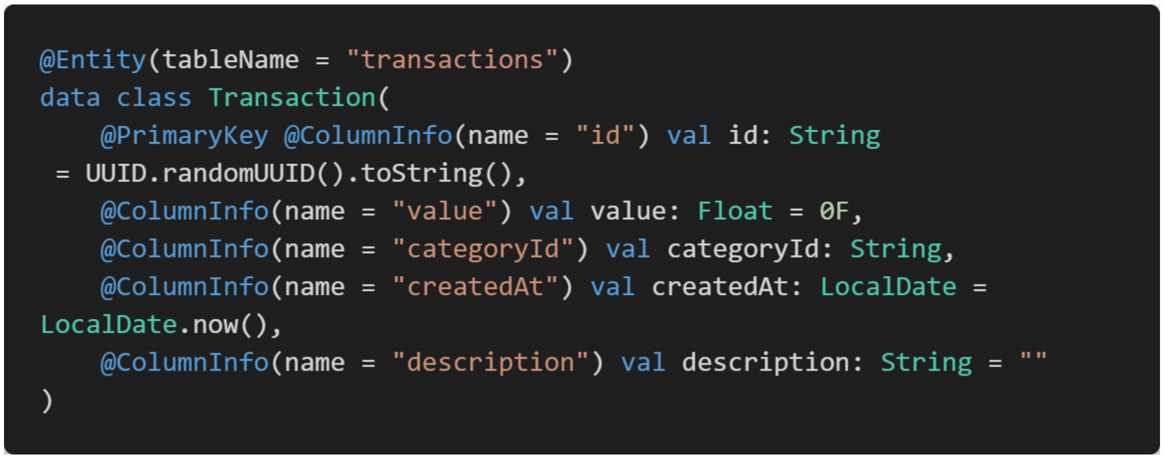
The **Category** class describes the type of purchased goods such as food, transportation, health care, etc. This class has three properties: **id**, **title** and **color**, as shown in Figure 21**.** As the name implies, the **title** property is the name of the category, while **color** is chosen by the user to graphicallly differentiate one category from the others. All data of type **Category** is stored in the database under the table **catagories**.



```
@Entity(tableName = "categories")
data class Category(
    @PrimaryKey @ColumnInfo(name = "id") var id: String
 = UUID.randomUUID().toString(),
    @ColumnInfo(name = "title") var title: String = "",
    @ColumnInfo(name = "color") var color: Int = -65535
)
```

Figure 21. The **Category** model describing details of the purchased goods

As the name implies, **Transaction** describes the details of a transaction including purchase price, date of purchase, category of the purchased goods and optional description. This class has five properties: **id**, **value**, **createdAt**, **categoryId** and **description.** These properties are stored as columns of the table **transactions** in the local database. Taking each property detail, the **value** property holds the transaction cost; the **createdAt** property reports the conducted date of the transaction; the **categoryId** links the transaction to the category (type of the purchased goods) it belongs to; the **description** specifies additional information in the form of text.

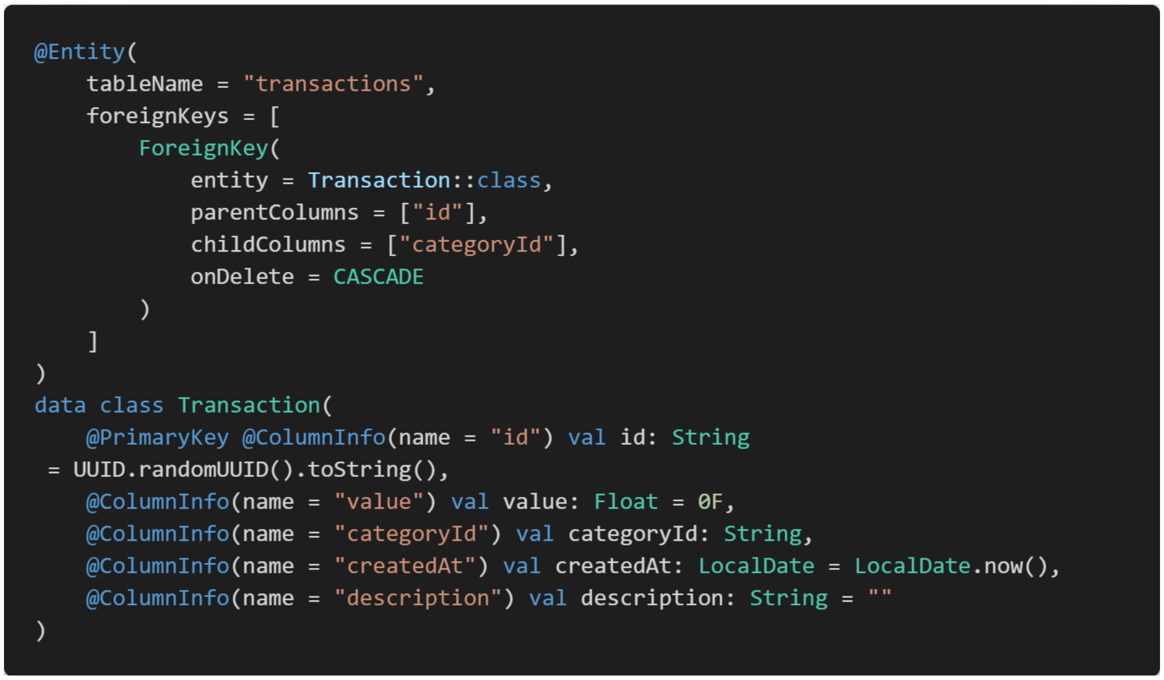


```kotlin
@Entity(tableName = "transactions")
data class Transaction(
    @PrimaryKey @ColumnInfo(name = "id") val id: String
 = UUID.randomUUID().toString(),
    @ColumnInfo(name = "value") val value: Float = 0F,
    @ColumnInfo(name = "categoryId") val categoryId: String,
    @ColumnInfo(name = "createdAt") val createdAt: LocalDate =
LocalDate.now(),
    @ColumnInfo(name = "description") val description: String = ""
)
```

Figure 22. Transaction model describing the details of a transaction

It is essential that the **categoryId** property is the foreign key that forms the one-to-many relationship between the **transactions** table and the **categories** table. This means that users can bind as many transactions as they want to a single category, but each transaction can only refer to exactly one category.

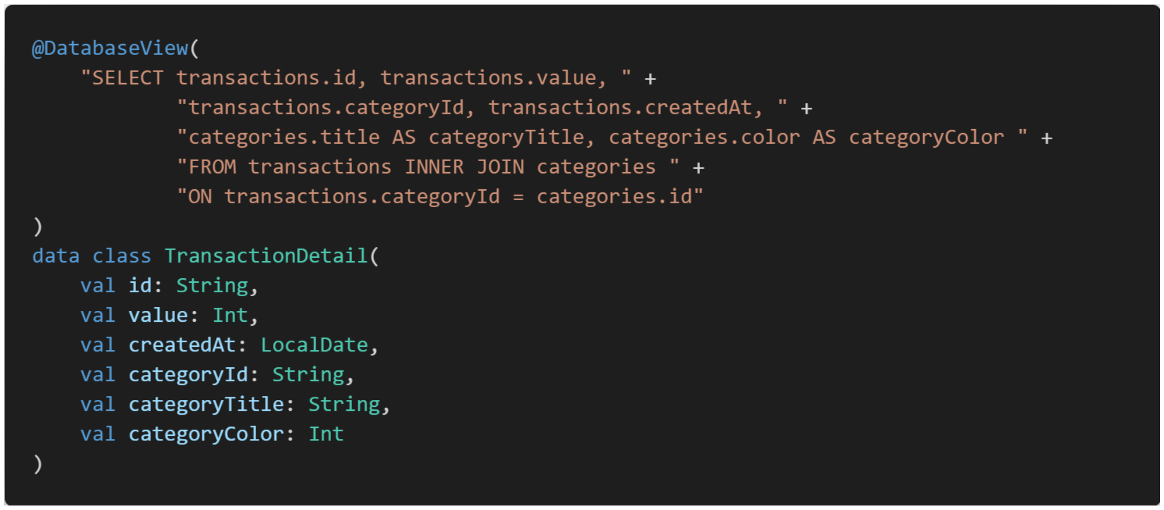
One-to-many relationship is often denoted by the **ForeignKey** constraint, as show in Figure 23. However, to my experience, this approach dramatically raises the compilation time, because the compiler has to generate a large set of rules among properties of the two tables. ***DatabaseView*** is my preferred solution to solve this issue of the **ForeignKey** constraint. **DatabaseView** is a data class with the **@DatabaseView** annotation that collects and stores information from multiple tables from the database to the properties of a class. With **DatabaseView**, no rules between the **categories** table and the **transaction** tables are formed, which reduces the compilation time (Figure 24).



```
@Entity(
    tableName = "transactions",
    foreignKeys = [
        ForeignKey(
            entity = Transaction::class,
            parentColumns = ["id"],
            childColumns = ["categoryId"],
            onDelete = CASCADE
        )
    ]
)
data class Transaction(
    @PrimaryKey @ColumnInfo(name = "id") val id: String
 = UUID.randomUUID().toString(),
    @ColumnInfo(name = "value") val value: Float = 0F,
    @ColumnInfo(name = "categoryId") val categoryId: String,
    @ColumnInfo(name = "createdAt") val createdAt: LocalDate = LocalDate.now(),
    @ColumnInfo(name = "description") val description: String = ""
)
```

Figure 23. The **ForeignKey** constraint establishing the relationship between the **categories** table and the **transactions** table

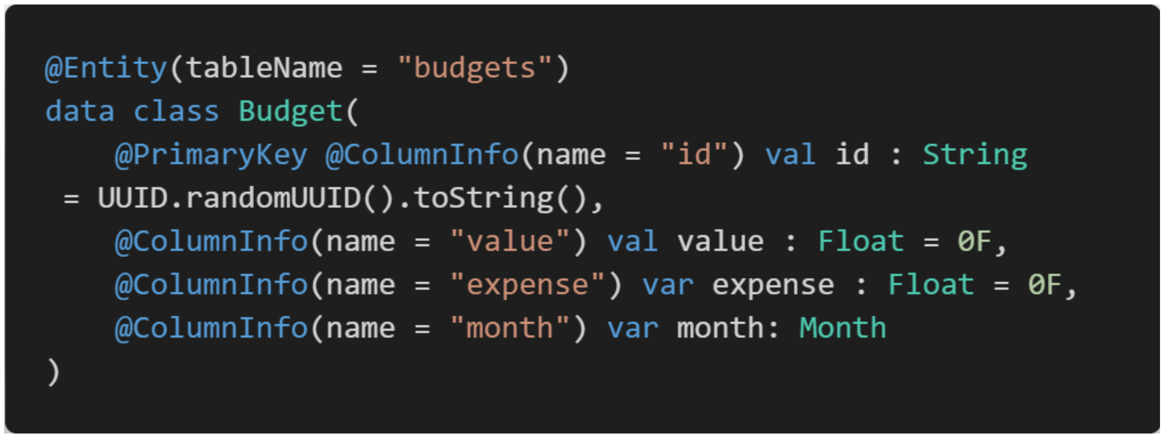
Figure 24 demonstrates how information from both **categories** and **transactions** table is collected in a single class. Methods that conduct the data operation are generated by the Room library based on the query statement inside the **@DatabaseView** annotation. Because the Room library translates text into code, naming properties does matter. It is the rule that the names of the selected columns in the **SELECT** clause must match those of the class properties. For example, in Figure 24, the **categoryTitle** column in the **SELECT** clause has the same name of the **categoryTitle** property of the **TransactionDetail** class.



```
@DatabaseView(
    "SELECT transactions.id, transactions.value, " +
            "transactions.categoryId, transactions.createdAt, " +
            "categories.title AS categoryTitle, categories.color AS categoryColor " +
            "FROM transactions INNER JOIN categories " +
            "ON transactions.categoryId = categories.id"
)
data class TransactionDetail(
    val id: String,
    val value: Int,
    val createdAt: LocalDate,
    val categoryId: String,
    val categoryTitle: String,
    val categoryColor: Int
)
```

Figure 24. Joining two tables using the **@DatabaseView** annotation and an SQL query

With the **Category** and **Transaction** data class, the Fitsu application is capable of storing any details of a transaction. Writing methods to calculating and analysing those transactions is not problematic anymore, because no complicated network or caching policy are involved. However, those actions consume a significant amount of time and CPU power. Storing pre-calculated results is the approach to address that issue in this thesis. To be more accurate, the total transaction cost of a particular month is stored in the database under the **budgets** table.



```
@Entity(tableName = "budgets")
data class Budget(
    @PrimaryKey @ColumnInfo(name = "id") val id : String
 = UUID.randomUUID().toString(),
    @ColumnInfo(name = "value") val value : Float = 0F,
    @ColumnInfo(name = "expense") var expense : Float = 0F,
    @ColumnInfo(name = "month") var month: Month
)
```

Figure 25. The Budget class storing the sum of transactions in a month

In the **budgets** table, the **expense** column holds the sum of all transactions in the month specified by the **month** column. The value of the **expense** column can then be subtracted by the user’s budget stored in the **value** column to find the monthly balance. Since the **expense** value is calculated based on transactions, whenever a transaction is added or modified, the **expense** value gets updated as well.

On the whole, the Fitsu applicatiion has three data models: **Category**, **Transaction** and **Budget**. As for the relationship between these models, the **Transaction** model depends on the **Category** model in the form of one-to-many relationship, while **budget** is the holder of pre-calculated sum of monthly transactions.

Having a design for all the data models, we move to the next step of designing the system structure in Section 3.2.

3.2 Component diagram

This section demonstrates how components communicate to each other in the Fitsu application. The idea of this section derives from Figure 26, an architecture recommended by the Android documentation 2020e. Knowledge of Room library (section 2.5) and Model-View-ViewModel architecture (section 2.4) will be referenced across the section, therefore readers should review these concepts before continue.

In Figure 26, each component stays in its layer and only references to the components one level below it. For example, the **Activity/Fragment** layer only references **ViewModel** components. The **Repository** or **Remote Data Source** components**,** although reside in the lower layer, are not referenced by the **Activity/Fragment** components.

Moreover, each component depends on exactly one component below it. An activity in the **Activity/Fragment** layer must depends on one and only one **ViewModel** component. The same rule applies for each **ViewModel** component, where exactly one repository is referenced. Components in the **Repository** layer are the only exceptions. A repository utilizes both the local database (left side of Figure 26) and remote data source (right side of Figure 26).

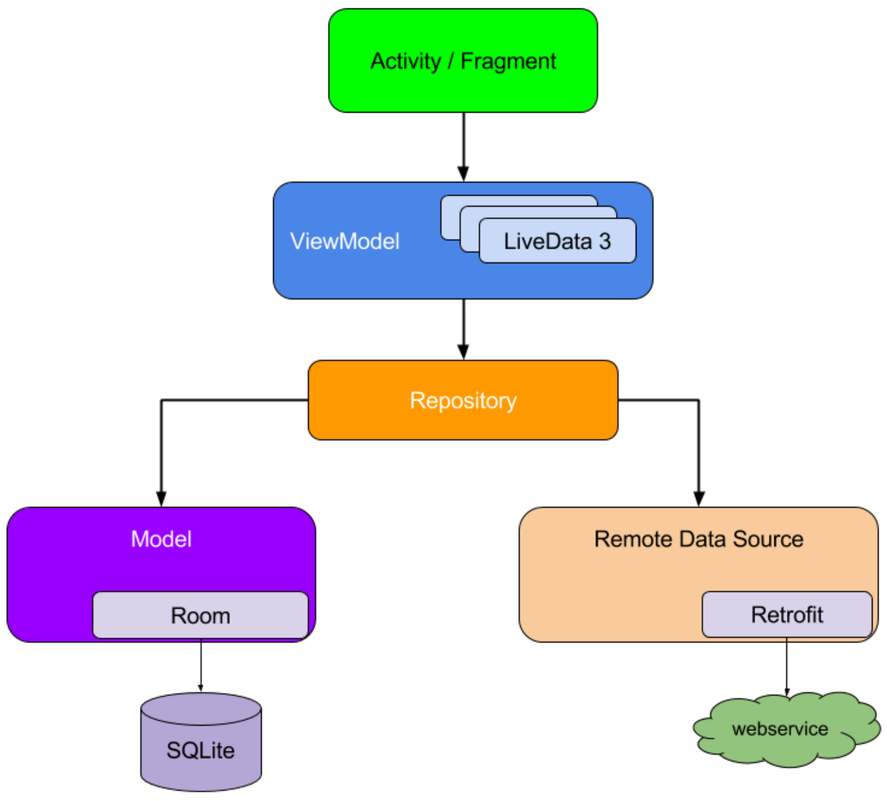


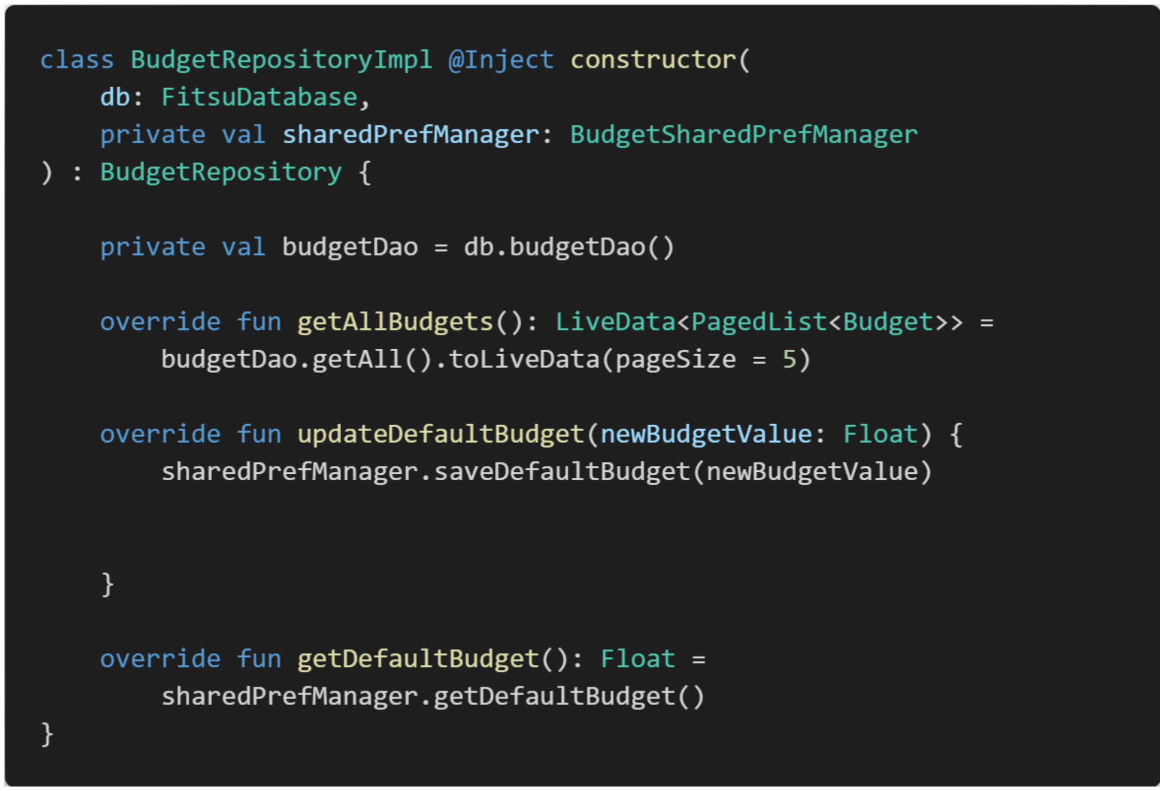
Figure 26. Recommended architecture by Android documentation 2020e

Components in the **Repository** act as data managers. They use data in the local database to display information to the user, while downloading new updates from the remote backend server. These updates result in a refresh of the user interface with the help of **LiveData**. Using this approach, the database serves as the single source of truth, and other parts of the application access it using the **Repository** components.

This architecture has proved its effectiveness by good feedback from users. Despite of unstable internet connection, the application with this architecture still allows users to view information that persists locally. Once the Internet is available, database is updated in background based on data from the remote backend server. It is also convenient for developers to manage data flow because only local database is used to display information on the screen.

Figure 27 illustrates an implementation of the architecture mentioned above in Fitsu application. The **BudgetRepositoryImpl** resides in the **Repository** layer

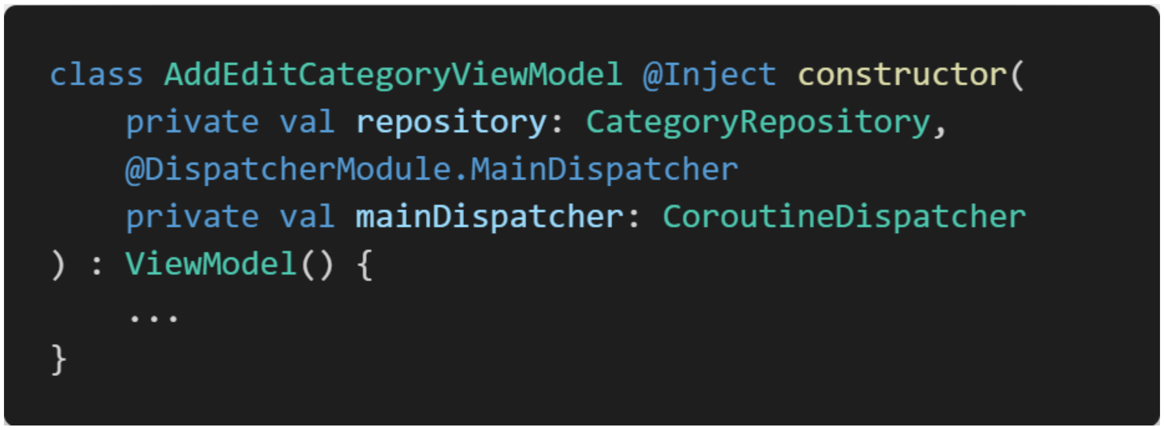
(Figure 26) and references to the database for data. As for remote data source, it is, however, not available yet due to time restriction. This **Repository** component also retrieves data from SharedPrefence, a simple and fast storage for key-value data.



```
class BudgetRepositoryImpl @Inject constructor(
    db: FitsuDatabase,
    private val sharedPrefManager: BudgetSharedPrefManager
) : BudgetRepository {

    private val budgetDao = db.budgetDao()

    override fun getAllBudgets(): LiveData<PagedList<Budget>> =
        budgetDao.getAll().toLiveData(pageSize = 5)

    override fun updateDefaultBudget(newBudgetValue: Float) {
        sharedPrefManager.saveDefaultBudget(newBudgetValue)


    }

    override fun getDefaultBudget(): Float =
        sharedPrefManager.getDefaultBudget()
}
```

Figure 27. The **BudgetRepositoryImpl** class using the local database as the data source

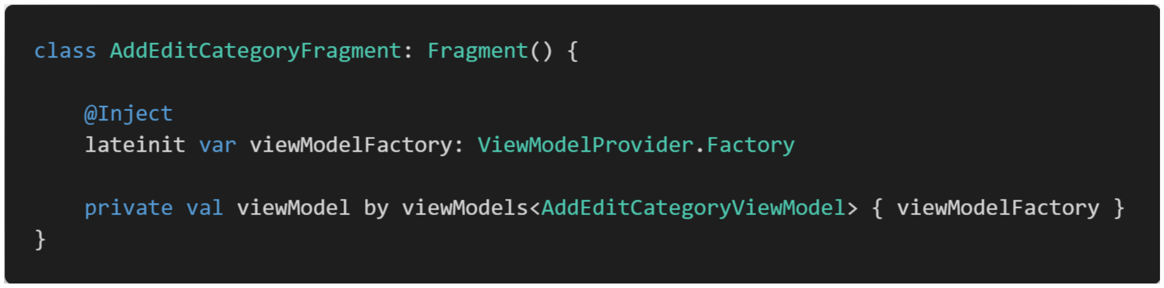
Move to the upper level, the **ViewModel** level, we take the **AddEditCategoryViewModel** class as an example (Figure 28). This class has components in the one-level lower as dependencies, which is **CategoryRepository.** A custom coroutine dispatcher is also involved as a dependency. This dispatcher is a part of the Coroutine library responsible for redirecting database operation to the background thread.



```
class AddEditCategoryViewModel @Inject constructor(
    private val repository: CategoryRepository,
    @DispatcherModule.MainDispatcher
    private val mainDispatcher: CoroutineDispatcher
) : ViewModel() {
    ...
}
```

Figure 28. The **AddEditCategoryViewModel** depending on the **CategoryRepository**

Finally, at the last layer of the architecture, a **ViewModel** is injected by Dagger to the fragment that needs it. In case of Figure 29, the **AddEditCategoryFragment** class has an **AddEditCategoryViewModel** object as a dependency. Unlike a **ViewModel** where dependencies are passed as parameters, dependencies are injected as fields to fragments. Since this section only focuses on the architecture side, more details about injecting dependencies as fields will be discussed further in section 3.3.



```
class AddEditCategoryFragment: Fragment() {

    @Inject
    lateinit var viewModelFactory: ViewModelProvider.Factory

    private val viewModel by viewModels<AddEditCategoryViewModel> { viewModelFactory }
}
```

Figure 29. The **AddEditCategoryFragment** class with **AddEditCategoryViewModel** object as a dependency

On the whole, data is driven from the database to the user interface through four layers: the **Database** layer, the **Repository** layer, the **ViewModel** layer and finally the **Activities/Fragments** layer. Each layer depends on the components one level below it. Design the application this way produces pleasant user experience as data is always available for shown.

Some implementations of this design from the Fitsu application are demonstrated in this section. However, dependency injection, on which this design relies, is not taken into account in this section for cohesion. This topic will be discussed further in section 3.3.

3.3 Dependency injection

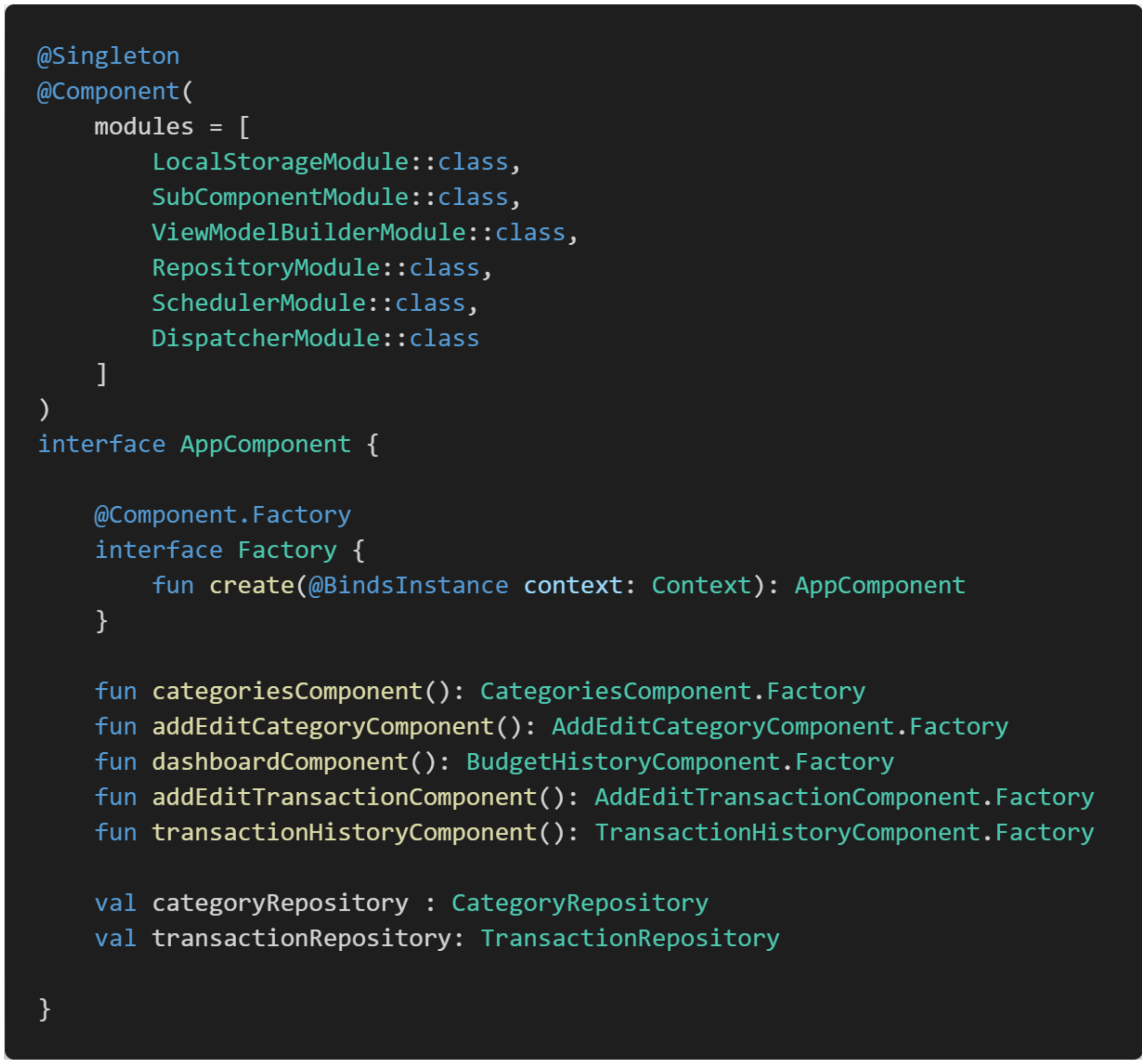
An overview of dependency injection was already introduced in Section 2.6. This section discusses further the dependency injection in practical situations. Moreover, advanced concepts such as modules, field injection, multi-binding and subcomponent will be explained in the real context of the Fitsu application.

3.3.1 Dagger modules

The Dagger module is the first topic taken into account in this section. Figure 30 shows the root Dagger graph of the Fitsu application. The **modules** property of the **@Component** annotation is filled with -Module suffix classes. These classes are called *modules*.

As explained in Section 2.6.2 that any classes' constructors with the **@Inject** annotation will be initialized and injected to appropriate places by Dagger. However, constructors of classes provided by third parties and the Android framework does not have the **@Inject** annotation, meaning that Dagger must be informed of how the objects of these classes are initialized. This is where Dagger modules come in. A Dagger module is a class that provides object initialization methods of un-injectable (unable to have the **@Inject** annotation in the constructor) classes.

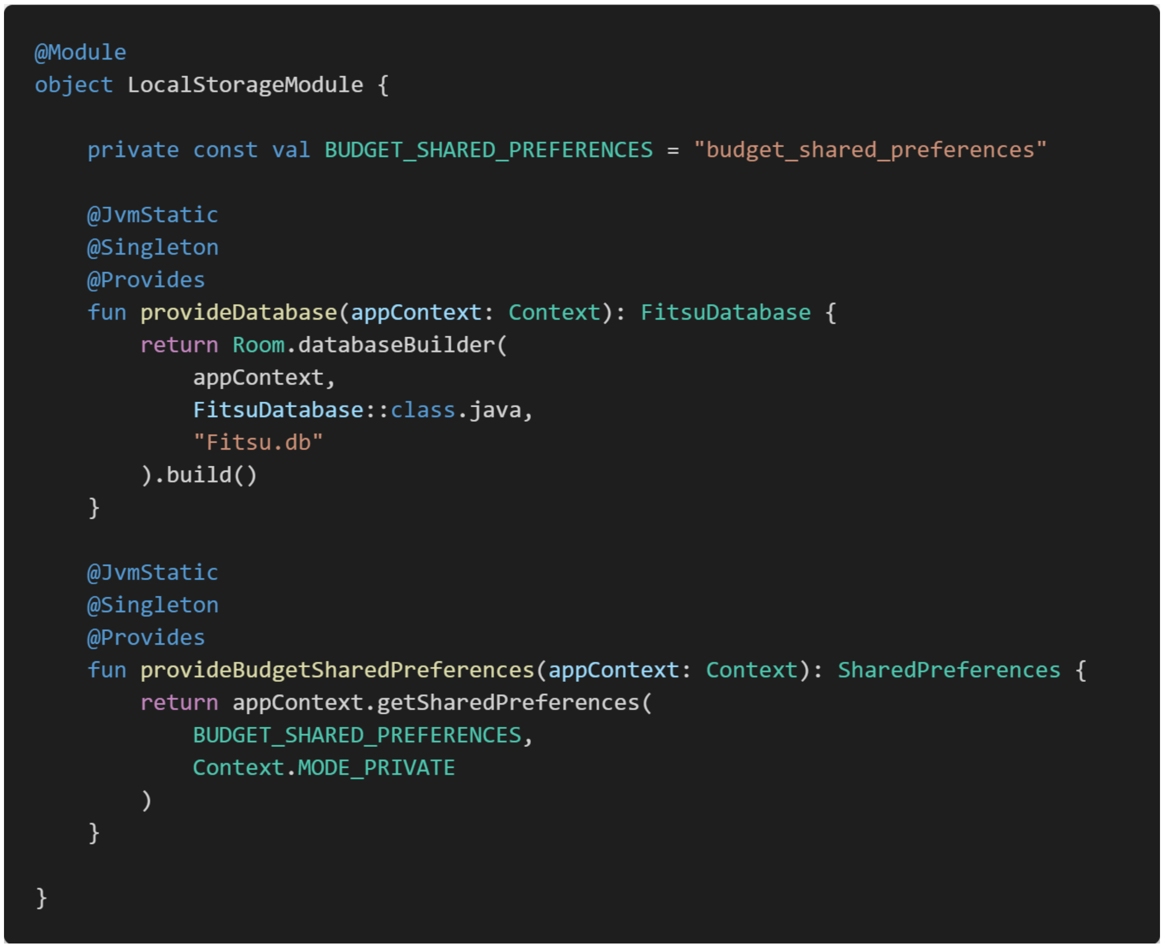
Figure 31 demonstrates an implementation of a Dagger module. Having the **@Module** annotation at the declaration, the **LocalStorageModule** class is treated as a module by Dagger. In its body, any classes with the **@Provides** annotation is considered an initialization method. In this section, such methods are called *provider method* for short. As section 2.6.2 introduced, Dagger only takes the type of a method. The same rule applies for provider methods. When the application is running, Dagger calls a provider method whenever its type matches exactly that of the required dependency.



```kotlin
@Singleton
@Component(
    modules = [
        LocalStorageModule::class,
        SubComponentModule::class,
        ViewModelBuilderModule::class,
        RepositoryModule::class,
        SchedulerModule::class,
        DispatcherModule::class
    ]
)
interface AppComponent {

    @Component.Factory
    interface Factory {
        fun create(@BindsInstance context: Context): AppComponent
    }

    fun categoriesComponent(): CategoriesComponent.Factory
    fun addEditCategoryComponent(): AddEditCategoryComponent.Factory
    fun dashboardComponent(): BudgetHistoryComponent.Factory
    fun addEditTransactionComponent(): AddEditTransactionComponent.Factory
    fun transactionHistoryComponent(): TransactionHistoryComponent.Factory

    val categoryRepository : CategoryRepository
    val transactionRepository: TransactionRepository

}
```

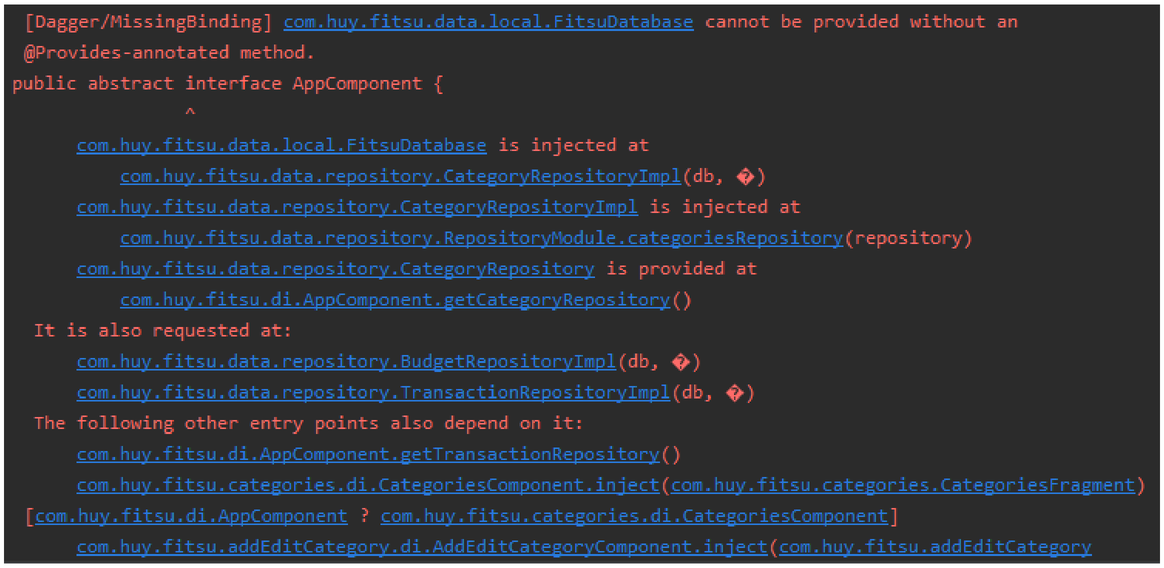
Figure 30. The AppComponent class (Dagger graph) with dependencies for the whole Fitsu application.

It is also important that all modules are included in the Dagger graph. This can be done by assigning the **modules** property with the array of the modules’ class names. In case of missing modules, Dagger will crash the compilation process and print error messages as shown in Figure 32.



```kotlin
@Module
object LocalStorageModule {

    private const val BUDGET_SHARED_PREFERENCES = "budget_shared_preferences"

    @JvmStatic
    @Singleton
    @Provides
    fun provideDatabase(appContext: Context): FitsuDatabase {
        return Room.databaseBuilder(
            appContext,
            FitsuDatabase::class.java,
            "Fitsu.db"
        ).build()
    }

    @JvmStatic
    @Singleton
    @Provides
    fun provideBudgetSharedPreferences(appContext: Context): SharedPreferences {
        return appContext.getSharedPreferences(
            BUDGET_SHARED_PREFERENCES,
            Context.MODE_PRIVATE
        )
    }

}
```

Figure 31. The **LocalStorageModule** class with provision methods of the Room database and the **SharedPreferences** object



```
 [Dagger/MissingBinding] com.huy.fitsu.data.local.FitsuDatabase cannot be provided without an
 @Provides-annotated method.
public abstract interface AppComponent {
                ^
      com.huy.fitsu.data.local.FitsuDatabase is injected at
          com.huy.fitsu.data.repository.CategoryRepositoryImpl(db, �)
      com.huy.fitsu.data.repository.CategoryRepositoryImpl is injected at
          com.huy.fitsu.data.repository.RepositoryModule.categoriesRepository(repository)
      com.huy.fitsu.data.repository.CategoryRepository is provided at
          com.huy.fitsu.di.AppComponent.getCategoryRepository()
  It is also requested at:
      com.huy.fitsu.data.repository.BudgetRepositoryImpl(db, �)
      com.huy.fitsu.data.repository.TransactionRepositoryImpl(db, �)
  The following other entry points also depend on it:
      com.huy.fitsu.di.AppComponent.getTransactionRepository()
      com.huy.fitsu.categories.di.CategoriesComponent.inject(com.huy.fitsu.categories.CategoriesFragment)
 [com.huy.fitsu.di.AppComponent ? com.huy.fitsu.categories.di.CategoriesComponent]
      com.huy.fitsu.addEditCategory.di.AddEditCategoryComponent.inject(com.huy.fitsu.addEditCategory
```

Figure 32. An error message about the missing provider method for the FitsuDatabase

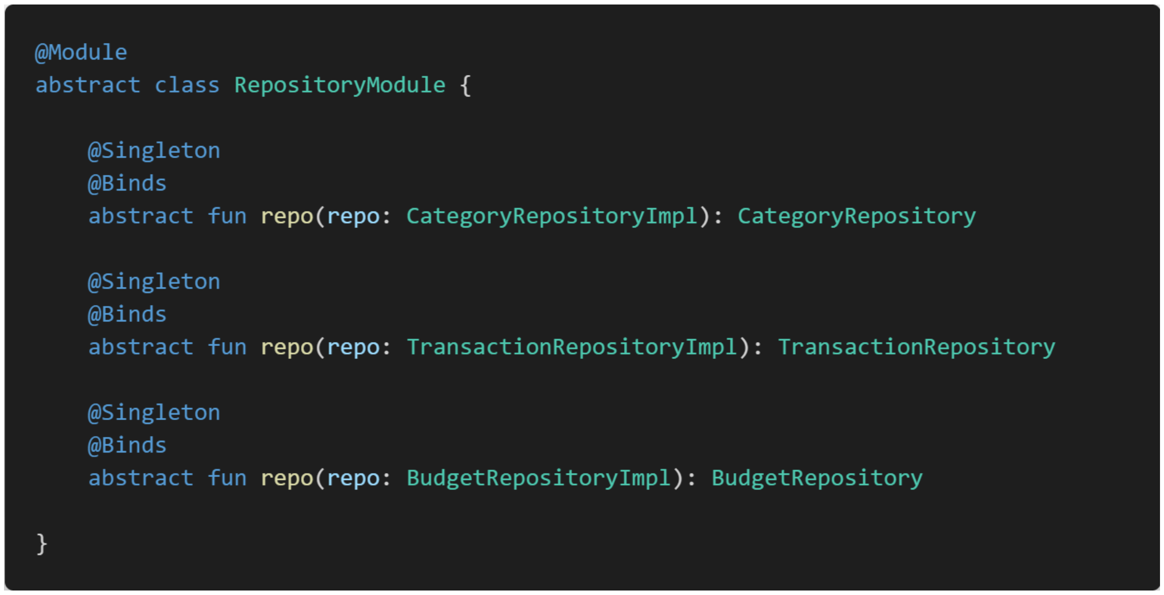
3.3.2 The @Binds annotation

In programming, an interface can be extended by multiple classes or interfaces. Therefore, it is widely used to group a set of related classes or to define obligatory properties and methods of some classes. In the Fitsu application, however, interfaces are used for a more specific reason, that is, creating multiple versions of an object. For example, suppose we want to use different versions of a **CategoryRepository** object: one for the real application and one for testing. To accomplish this, we create a **CategoryRepository** interface with two implementations: the **CategoryRepositoryImpl** class for the real application and the **FakeCategoryRepository** class for testing. The only thing left is to tell Dagger to inject the **CategoryRepositoryImpl** version of the **CategoryRepository** type in the real application and the **FakeCategoryRepository** in the testing environment. This is where the **@Binds** annotation comes in.

According to the Dagger documentation (2020b), the **@Binds** annotation is used for “abstract methods of a Module that delegate bindings”. For example, to create the **CategoryRepositoryImpl** version of a **CategoryRepository** object, a module similar to Figure 33 must be used. It is also vital that the **CategoryRepositoryImpl** class has the **@Inject** annotation in its constructor.

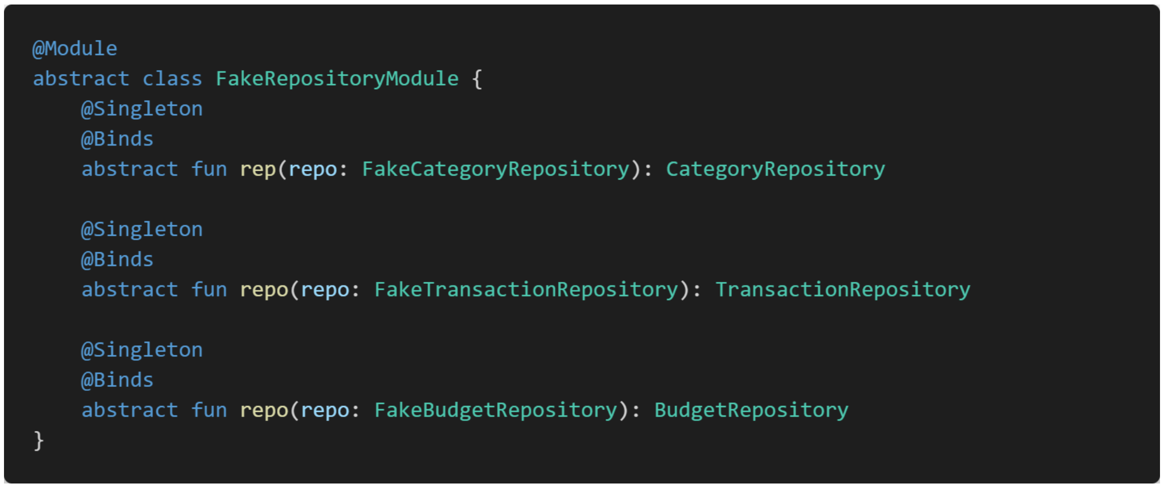
The **@Binds** annotation has many rules that we need to follow. First of all, annotated methods must be abstract. Secondly, one and only one parameter is allowed. This parameter must also satisfy the fact that its type is the child class of the return type of the method. For example, in Figure 33, the type of the **repo** parameter is **CategoryRepositoryImpl** which is a subclass of the **CategoryRepository** class.

Regarding testing, the same template is used to create a testing version of a **CategoryRepository** object, as shown in Figure 34. Even though the syntax is the same, the testing version of a **CategoryRepository** object can only be created if the **FakeRepositoryModule** is included in the testing Dagger graph, as shown in Figure 35.



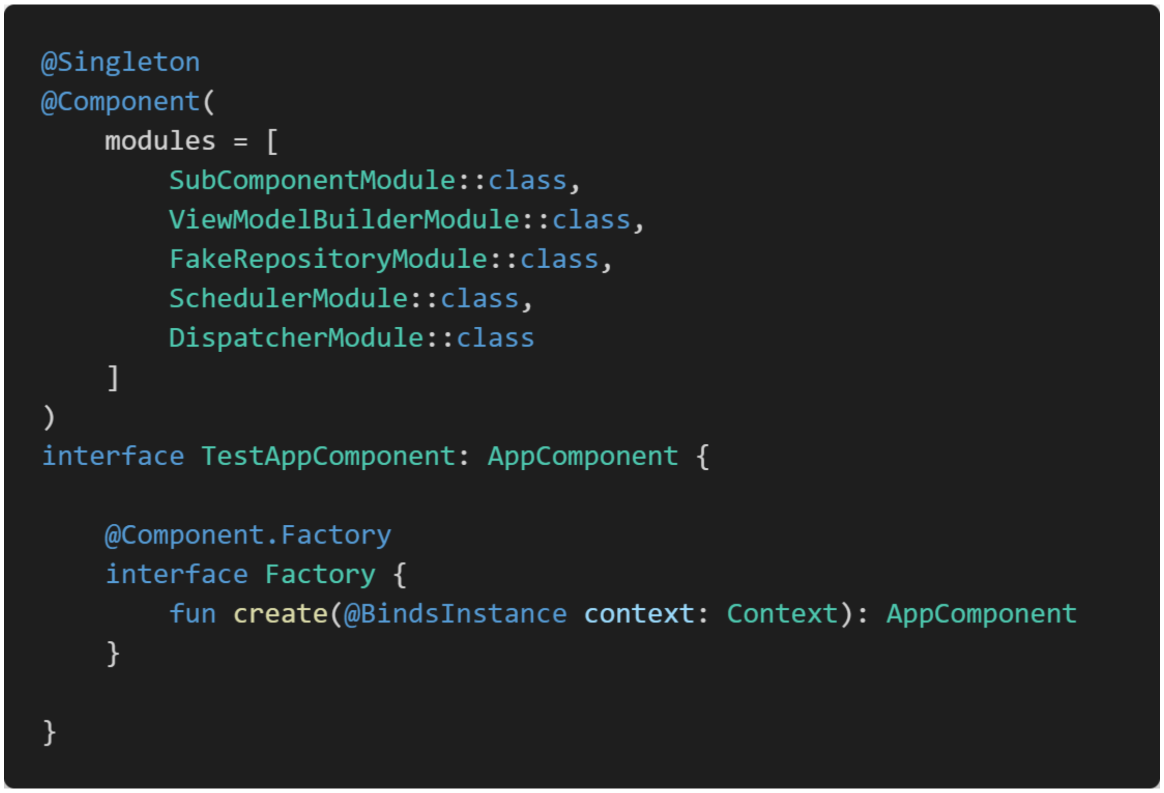
```kotlin
@Module
abstract class RepositoryModule {

    @Singleton
    @Binds
    abstract fun repo(repo: CategoryRepositoryImpl): CategoryRepository

    @Singleton
    @Binds
    abstract fun repo(repo: TransactionRepositoryImpl): TransactionRepository

    @Singleton
    @Binds
    abstract fun repo(repo: BudgetRepositoryImpl): BudgetRepository

}
```

Figure 33. The **RepositoryModule** class instructing Dagger to create the **CategoryRepositoryImpl** version of the **CategoryRepository** object



```kotlin
@Module
abstract class FakeRepositoryModule {
    @Singleton
    @Binds
    abstract fun rep(repo: FakeCategoryRepository): CategoryRepository

    @Singleton
    @Binds
    abstract fun repo(repo: FakeTransactionRepository): TransactionRepository

    @Singleton
    @Binds
    abstract fun repo(repo: FakeBudgetRepository): BudgetRepository
}
```

Figure 34. The **FakeRepositoryModule** class instructing Dagger to create a testing version of the **CategoryRepository** object



```kotlin
@Singleton
@Component(
    modules = [
        SubComponentModule::class,
        ViewModelBuilderModule::class,
        FakeRepositoryModule::class,
        SchedulerModule::class,
        DispatcherModule::class
    ]
)
interface TestAppComponent: AppComponent {

    @Component.Factory
    interface Factory {
        fun create(@BindsInstance context: Context): AppComponent
    }

}
```

Figure 35. The **FakeRepositoryModule** class included in the **TestAppComponent** for testing

In short, the **@Binds** annotation is a powerful tool to instantiate multiple versions of an object in Dagger. A method that has the **@Binds** annotation must be abstract and must have exactly one parameter whose type is a child of the return type. The **@Binds** annotation along with **@Provides** and **@Inject** provides a perfect solution for Dagger to create any objects in any situation.

3.3.3 Dagger multi-binding

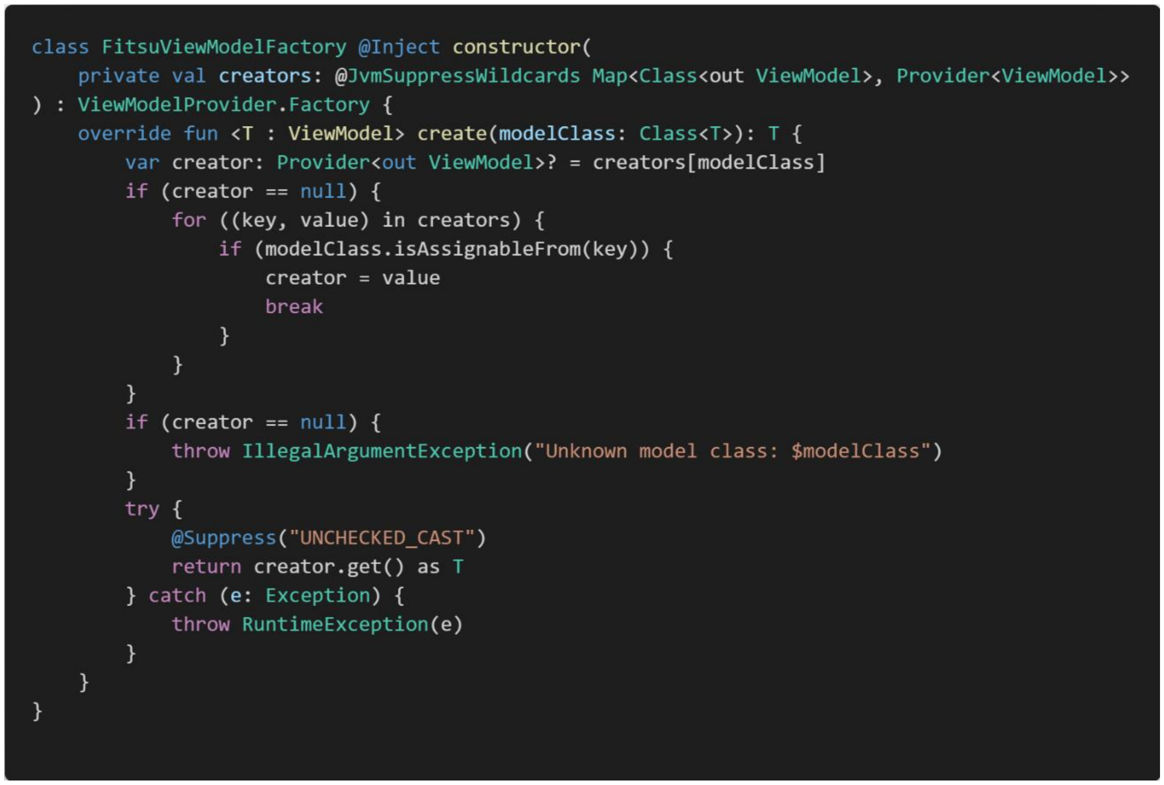
This section focuses on the multi-binding feature of Dagger. It first explains the reason of using multi-binding in the Fitsu application and then the usage of this feature with code base.

Besides removing boilerplate code and increasing the scalability of the application, Dagger introduces a problem regarding the **ViewModel** instantiation. A **ViewModel** object cannot be instantiated directly by calling its constructor. A factory class extending the **ViewModelProvider.Factory** is used instead. It is apparent that Dagger has no idea about such classes. Multi-binding is one way of telling Dagger how to instantiate **ViewModel** objects in this situation.

According to Dagger documentation (2020a), the multi-binding feature binds several objects of the same type to a collection or a map. With this feature, Dagger is capable of creating a map of all **ViewModel** objects where keys are class names and values are the corresponding class providers. This map can then be injected to the **FitsuViewModelFactory** class (Figure 36) that uses the map to create **ViewModel** objects automatically. The implementation of this class is forked from the Android Github (2020).

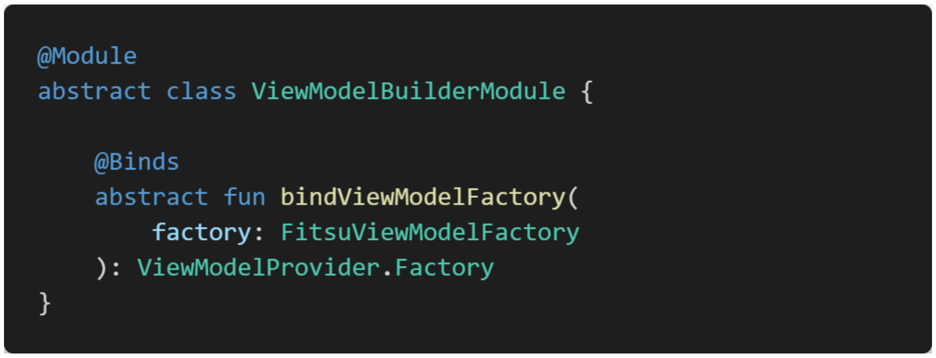
The functionality of the **FitsuViewModelFactory** class is simple. It implements the **create()** method of the **ViewModelProvider.Factory** interface, which is used by the Android framework to instantiate a **ViewModel** object. Using the map of **ViewModel** objects injected by Dagger, **FitsuViewModelFactory** searches for the provider whose class name matches exactly that stored in the **modelClass** parameter. Once the provider is found, **FitsuViewModelFactory** calls the **get()** method to create the desired **ViewModel** object. If there is no provider that meets the requirement, **FitsuViewModelFactory** retries searching. Unlike the first attempt where the **ViewModel**'s class name is used as the search key, it finds the children class of the desired **ViewModel** instead. After the second attempt, if no provider is found, **FitsuViewModelFactory** throws an exception to crash the application immediately. (Android Github 2020.)

Next, Dagger needs to know how to inject the **FitsuViewModelFactory.** One approach is to use the **@Binds** annotation (Section 3.3.2). The **ViewModelModule** class in Figure 37 forces Dagger to inject the Fitsu application's version of **ViewModelProvider.Factory**, which is the **FitsuViewModelFactory** class, instead of the default class of the Android framework.



```
class FitsuViewModelFactory @Inject constructor(
    private val creators: @JvmSuppressWildcards Map<Class<out ViewModel>, Provider<ViewModel>>
) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        var creator: Provider<out ViewModel>? = creators[modelClass]
        if (creator == null) {
            for ((key, value) in creators) {
                if (modelClass.isAssignableFrom(key)) {
                    creator = value
                    break
                }
            }
        }
        if (creator == null) {
            throw IllegalArgumentException("Unknown model class: $modelClass")
        }
        try {
            @Suppress("UNCHECKED_CAST")
            return creator.get() as T
        } catch (e: Exception) {
            throw RuntimeException(e)
        }
    }
}
```

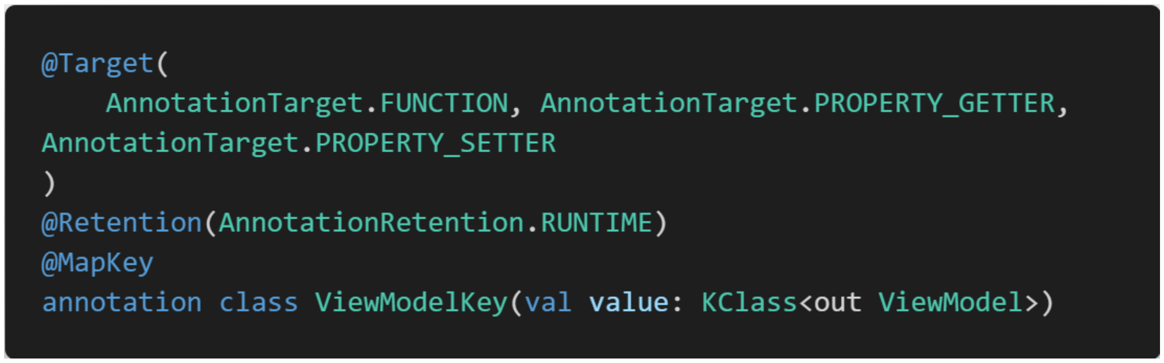
Figure 36. A custom factory that creates **ViewModel** objects using a map



```
@Module
abstract class ViewModelBuilderModule {

    @Binds
    abstract fun bindViewModelFactory(
        factory: FitsuViewModelFactory
    ): ViewModelProvider.Factory
}
```

Figure 37. The **ViewModelBuilderModule** instructs Dagger to inject a **FitsuViewModelFactory** whenever a dependency of type **ViewModelProvider.Factory** is required

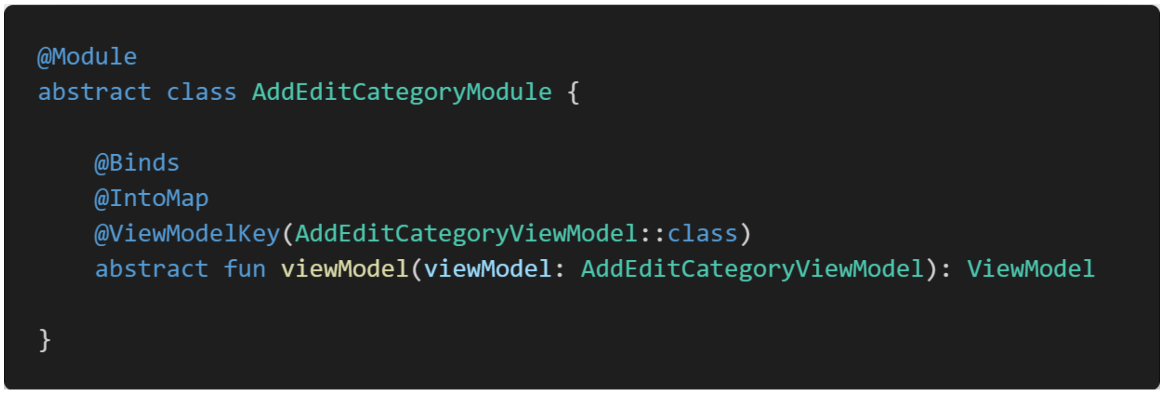
Next, we move to the part where the multi-binding feature shows its potential capability. As mentioned before multi-binding is used in this application to create a map of **ViewModel** objects. This map, just like other maps, is a list of key-value pair where keys are the **ViewModel**'s class name and the values are the corresponding **Provider** objects. Thanks to the Dagger's intelligence, developers only need to specify a unique key for each **ViewModel** object. Such keys can be generated automatically using the annotation class in Figure 38. This class utilizes the class name of the **ViewModel** object to produce a unique key.



```
@Target(
    AnnotationTarget.FUNCTION, AnnotationTarget.PROPERTY_GETTER,
AnnotationTarget.PROPERTY_SETTER
)
@Retention(AnnotationRetention.RUNTIME)
@MapKey
annotation class ViewModelKey(val value: KClass<out ViewModel>)
```

Figure 38. The **ViewModelKey** class generating keys for the **ViewModel** map based on the objects' class name.

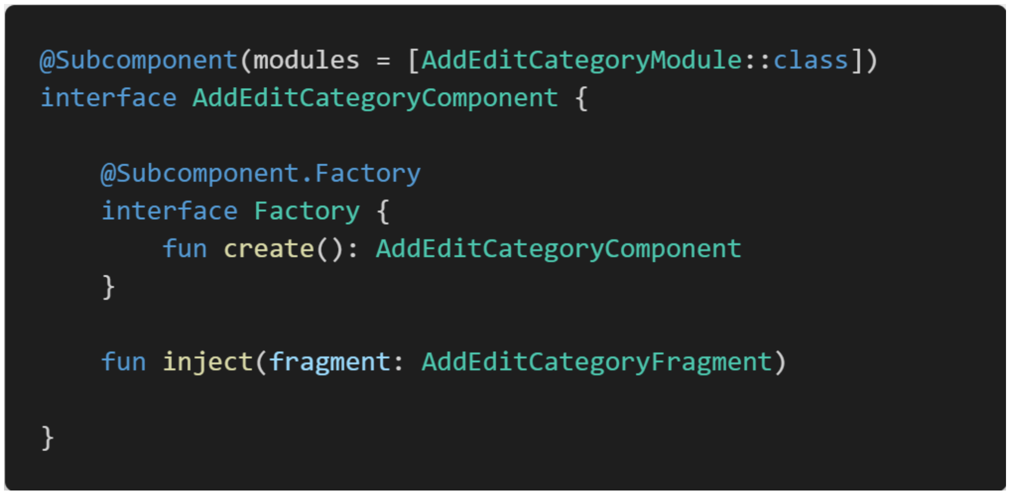
Figure 39 demonstrates how the **ViewModelKey** annotation is used. Upon the method declaration of a **ViewModel** is the **ViewModelKey** annotation with the class name of the **ViewModel** as a parameter. It is vital that the parameter must be nothing than the class name of a **ViewModel** class, meaning the **ViewModelKey** annotation can only be used by the **ViewModel** classes.



```
@Module
abstract class AddEditCategoryModule {

    @Binds
    @IntoMap
    @ViewModelKey(AddEditCategoryViewModel::class)
    abstract fun viewModel(viewModel: AddEditCategoryViewModel): ViewModel

}
```

Figure 39. The **@ViewModelKey** annotation generating a unique key based on the class name of the **AddEditCategoryViewModel** class

With the **@ViewModelKey** annotation, Dagger now fully understands the instantiation of the map of all **ViewModel** objects. It is time to attach all components mentioned above using a component as shown in Figure 40. The **AddEditCategoryComponent** interface instructs Dagger to create a graph that is responsible for creating the **AddEditCategoryViewModel** object**.** The body of the **AddEditCategoryComponent** interface also has the **inject()** function for field injection in the **AddEditCategoryFragment** class**.**

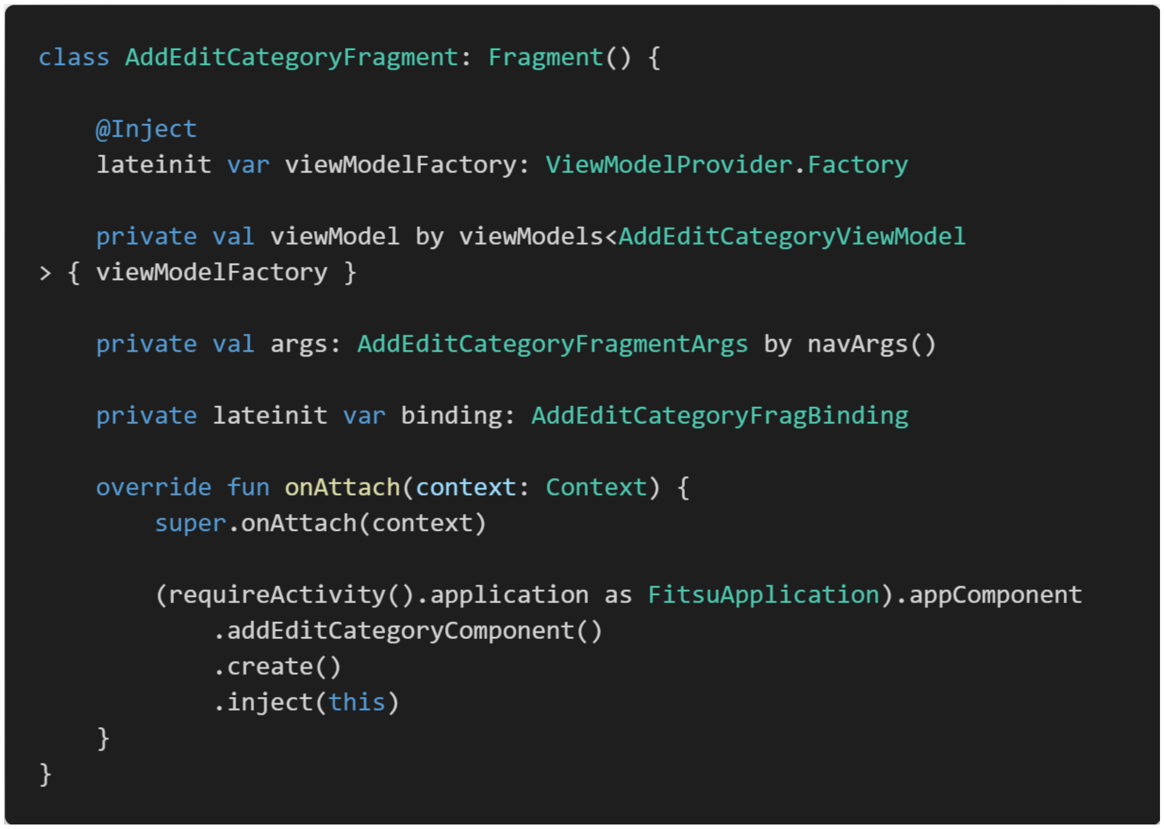


```kotlin
@Subcomponent(modules = [AddEditCategoryModule::class])
interface AddEditCategoryComponent {

    @Subcomponent.Factory
    interface Factory {
        fun create(): AddEditCategoryComponent
    }

    fun inject(fragment: AddEditCategoryFragment)

}
```

Figure 40. The **AddEditCategoryComponent** graph being responsible for creating the **AddEditCategoryViewModel** object and inject it into the **AddEditCategoryFragment** class**.**

It is time to discuss field injection. This feature of Dagger aims for classes whose constructor cannot be interfered, meaning the **@Inject** annotation is unable to be attached to the constructor's signature. In the current context, it is the **AddEditCategoryFragment** class that needs field injection. Obviously, the **AddEditCategoryFragment** class is a fragment where Android OS takes control of the initialization.

Field injection can be easily achieved using the **inject()** method (mentioned in Figure 40) in the **AddEditCategoryFragment** class, as shown in the **onAttach()** method. The length of the Dagger code in the **onAttach()** method seems to be long, but the operation behind the scene is simple. The first line grabs the unique instance of the root Dagger graph, **appComponent**, then references to the sub-graph **AddEditCategoryComponent** in the next two lines, and finally calls the **inject()** method to inject dependencies to the fragment. Last but not least, the **@Inject** annotation is attached with the public variable **viewModelFactory,** so that Dagger knows to which variable the instance of the **FitsuViewModelFactory** class needs to be assigned.



```
class AddEditCategoryFragment: Fragment() {

    @Inject
    lateinit var viewModelFactory: ViewModelProvider.Factory

    private val viewModel by viewModels<AddEditCategoryViewModel
> { viewModelFactory }

    private val args: AddEditCategoryFragmentArgs by navArgs()

    private lateinit var binding: AddEditCategoryFragBinding

    override fun onAttach(context: Context) {
        super.onAttach(context)

        (requireActivity().application as FitsuApplication).appComponent
            .addEditCategoryComponent()
            .create()
            .inject(this)
    }
}
```

Figure 41. The **FitsuViewModelFactory** object injected in the **AddEditCategoryFragment** class using field injection

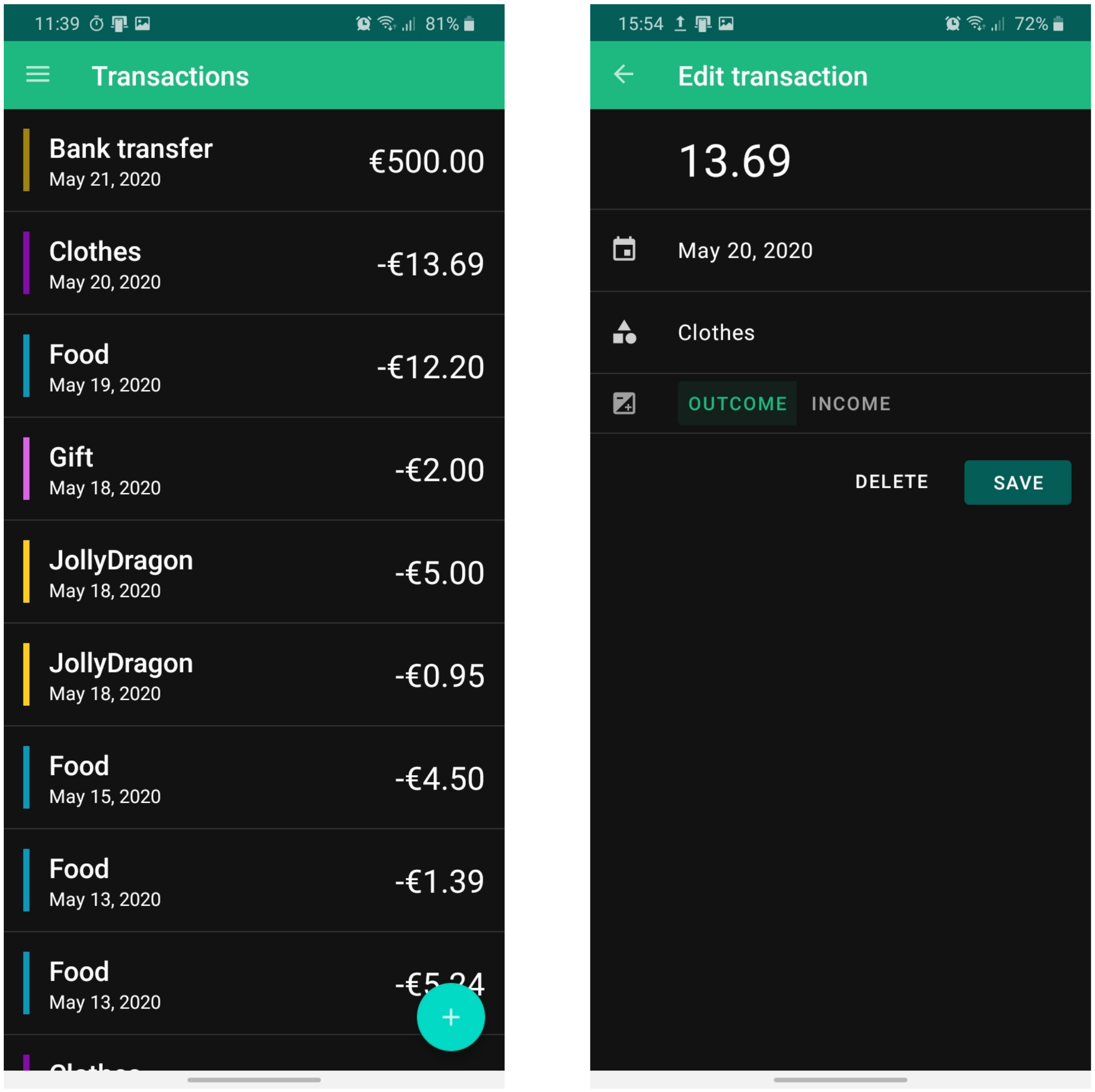
Overall, **ViewModel** objects are injected into every fragment properly, thanks to the multi-binding feature of Dagger and a custom **ViewModel** factory class. Besides the multi-binding feature, this section also demonstrates how dependencies can be injected as fields instead of constructor's parameters.

Moving on to conclude Section 3.3, although this section's purpose is explaining the implementation of dependency injection in the Fitsu application, several new concepts are involved, which may make Dagger library become complicated for readers. However, from the point of view of a person who tried hard-coded dependency and manual dependency injection, Dagger does remove a dramatic amount of boilerplate code. With Dagger, developers are able to view, manage and maintain the overall dependency graph right in the code base.

3.4 The result

The source code of the application can be found on GitHub via this link: https://github.com/buiquanghuy23103/Fitsu

With the Fitsu application, users are free to add as many transactions as they want. All details of a transaction from the money value to the date or type of purchases are taken into account. On the left part of Figure 42, all transactions are listed neatly with easy-to-read text. Percentage rate of each category with colors are shown right below the history of all transactions.



11:39 81%

Transactions

Bank transfer	May 21, 2020	€500.00
Clothes	May 20, 2020	-€13.69
Food	May 19, 2020	-€12.20
Gift	May 18, 2020	-€2.00
JollyDragon	May 18, 2020	-€5.00
JollyDragon	May 18, 2020	-€0.95
Food	May 15, 2020	-€4.50
Food	May 13, 2020	-€1.39
Food	May 13, 2020	-€5.24

15:54 72%

Edit transaction

13.69

May 20, 2020

Clothes

[✓ OUTCOME] INCOME

DELETE SAVE

Figure 42. Fitsu transaction design

On the **Transaction History** screen, whenever a transaction is clicked, a form will appear and allow users to edit every detail of the transaction. These changes can then be either discarded by simply navigating back or saved to the database on the **Save** button click. The transaction can also be deleted on the **Delete** button click.

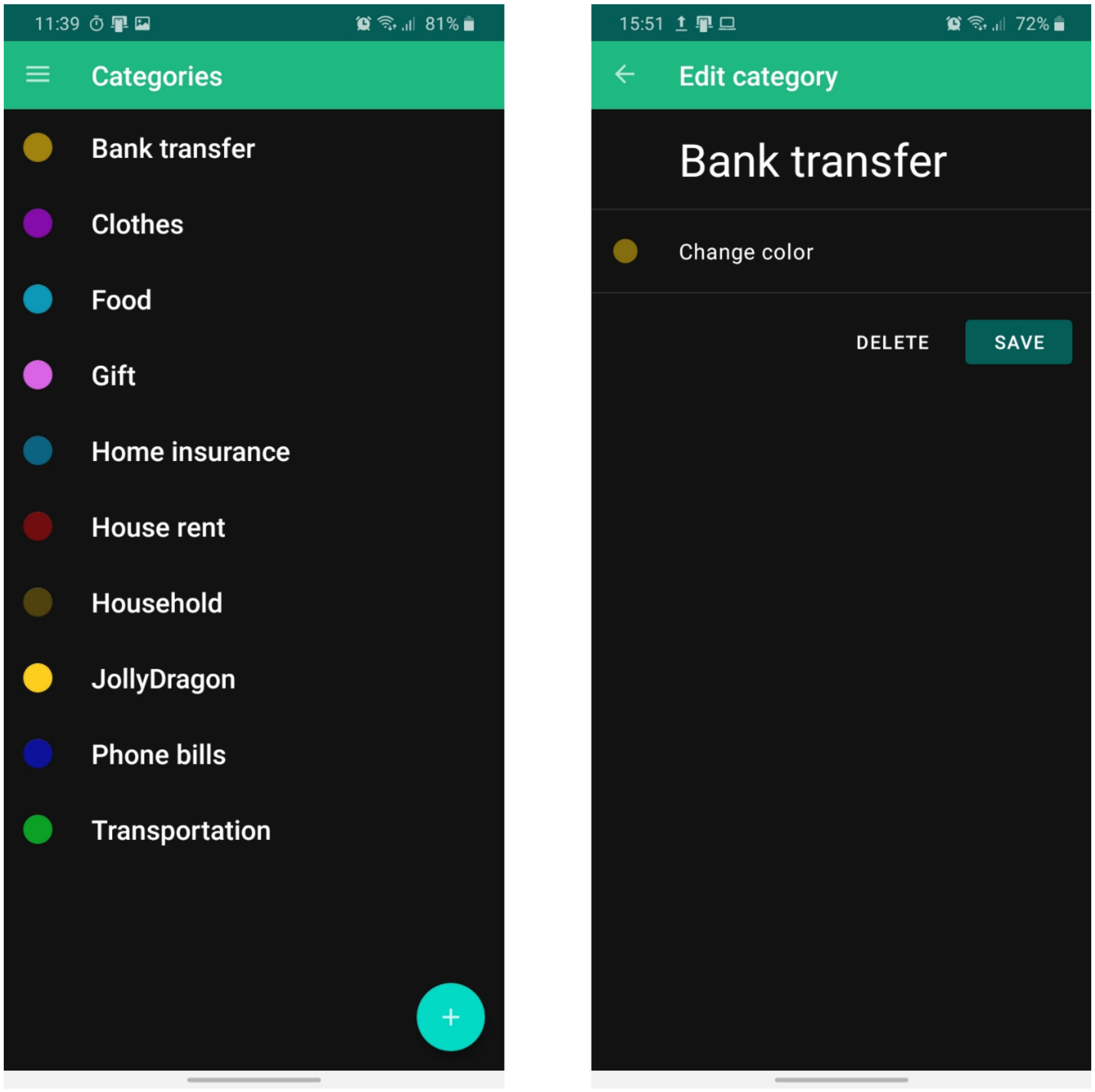
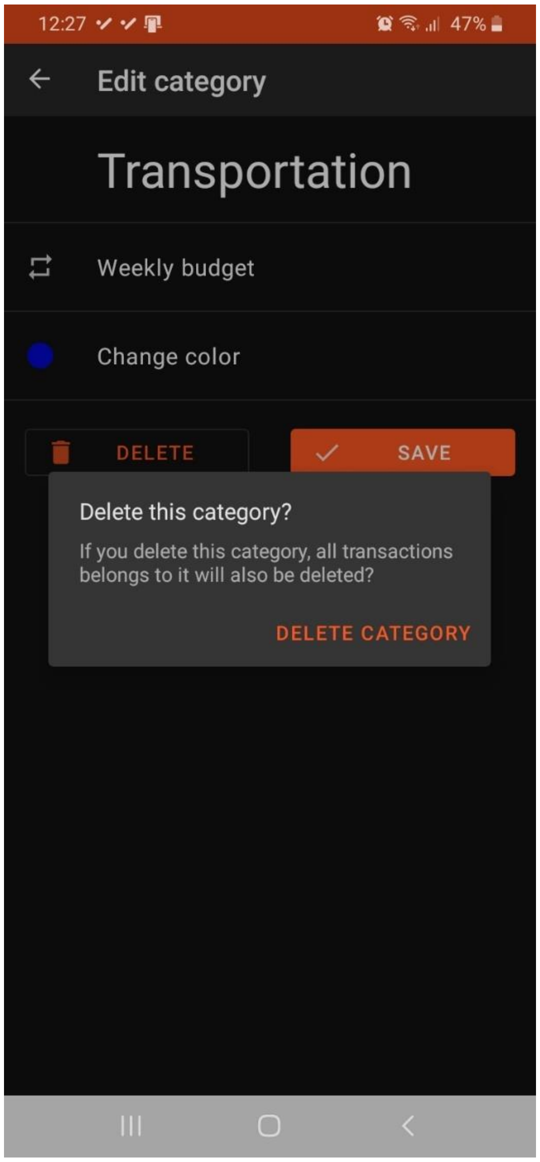


Figure 43. Fitsu category design

Regarding categories of purchases, the **Categories** screen displays all the necessary information about them (Figure 43). Each category is distinguished by colors and names chosen by the users. Users are free to add as many categories as they want by simply clicking the rounded button at the right-bottom corner of the screen. Behind the scene, clicking this button will create an empty category and bring users to the **Edit category** screen for editing.

A click on a certain category will navigate to the form to edit that category. Similarly, to the edit form of a transaction, every detail of a category can be modified in the **Edit category** screen. All changes can be saved by a **Save** button click or discarded when navigating back. Deleting a category, however, is different from doing so with a transaction. Since there may be more than one transaction belonging to a single category, deleting a category results deleting its corresponding transactions. Due to the danger of this action, a dialog box will show up, as shown in Figure 44, to warn the users before any further actions are taken.



Delete this category?

If you delete this category, all transactions belongs to it will also be deleted?

Figure 44. A dialog box will warn the users on the attempt of deleting a category

4 CONCLUSION

The goal of this thesis was to build a simple application that helps students manage their personal finances. Overall, that goal was achieved, meaning that an application with a few features for finance management was able to run successfully on real devices.

4.1 Android development compared to iOS and cross-platform development

Throughout the process, I experienced what people have been saying about the Android development: building and maintaining an Android application is frustrated and time-consuming. The complications of the Android development comes from the fact that the Android operating system runs on multiple devices, screen sizes and platforms. Compared to the iOS development, where target devices are limited to iPads and iPhones only, the Android development takes around 30-40% more time, according to Gupta (2019).

Despite long development time, Android platform still attracts developers due to its large market share. Android platform has beaten the others by 70.68% of the mobile operating system market share worldwide, at the time of writing, according to Global Stats (2020a). Regarding Finland only, the number of Android phones far outweigh its opponents with 72.8% of the market share in April 2020, according to Global Stats (2020b).

Some might wonder why cross-platforms such as React Native, Xamarin and Flutter where a single code base can run on both Android and iOS operating system were not used to deploy this project. To my knowledge, applications developed by the native approach outweigh those developed by either React Native, or Xamarin or Flutter in terms of performance. According to InVerita (2020), when calculating the Borwein algorithm, native Android applications encounter less CPU stress than Flutter applications by two times and React Native application by six times (Figure 45).

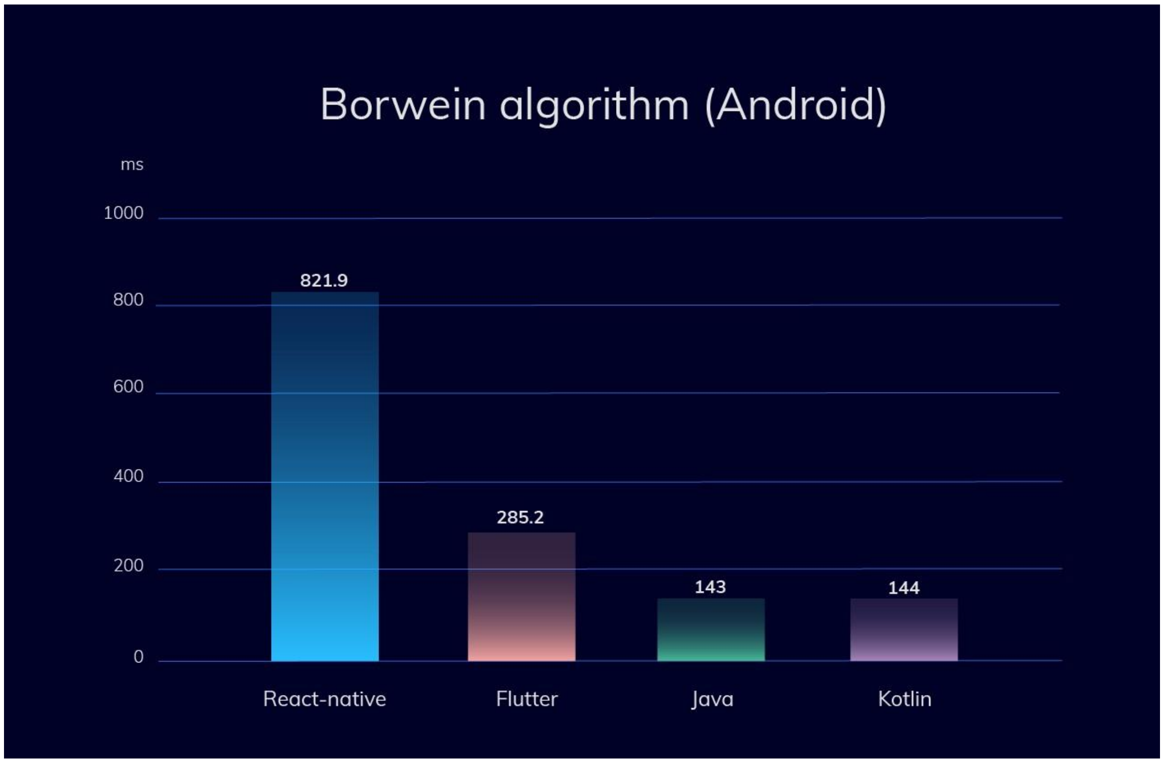


Figure 45. CPU-intensive test (Borwein algorithm) for Android (InVerita, 2020)

Another reason that cross-platform approaches are not taken into account in this thesis is that iOS development requires the Mac OS X operating system. In other words, only Apple products such as Macbook, iMac, or Mac mini are suitable for iOS development. Because such products are unaffordable for a student, I do not have a chance to run cross-platform code on real devices or emulators.

4.2 Improvements

As previously stated, the Fitsu application is only a simple application to manage personal finances. However, it can be considered as the foundation for further developments to achieve a beautiful, stable, high-performance Android application. There are several improvements that can be taken into account including the following:

- Unit tests and instrumented tests should be a part of the code base. These tests automatically check every feature of the application from the user interface to the business logic. Therefore error-prone modifications on the code base will be avoided at the highest possibility.
- More animations and transitions should be taken into account. Movements of graphical object should be both smooth and natural. Although such

graphical effects do not contribute to the feature list of the application, they increase the user experience. According to my observation, most graded applications are the ones with most satisfied user experience on the graphical design and movements. Therefore, making the application more beautiful is not less important than making it useful.

- Adding more custom budget plans might be a good idea. To be more accurate, there will be a dedicated screen for users to create their budget plan such as buying a laptop with EUR 500. This budget plan then is taken into account when calculating monthly expense and savings. Users are able to know how far their savings toward the desired goal is.

In conclusion, the process of building the Fitsu application included many complicated concepts, techniques and effort. Quite a significant amount of basic knowledge about the Android activities, fragments, lifecycles, and database need to be prepared before taking the first steps into the Android development world. Following the sequence of Android concepts, several design patterns were introduced such as the Model-View-ViewModel architecture and dependency injection. These patterns ensure the stability and scalability of the application. With guidance of the patterns, libraries such as Room for database and Dagger for dependency injection were utilised efficiently to reduce the stressfulness of the development. At the end of the thesis, a new application was born with simple features for managing the personal finance. Although the application is not production-qualified, it becomes a firm foundation for further improvements in the near future due to the good practice of architectures.

REFERENCES

Android Documentation, 2019a. Understand the activity lifecycle. WWW document. Available at: https://developer.android.com/guide/components/activities/activity-lifecycle [Accessed 10 December 2019].

Android Documentation, 2019b. Fragments. WWW document. Available at: https://developer.android.com/guide/components/fragments [Accessed 20 December 2019].

Android Documentation, 2019c. Viewmodel overview. WWW document. Available at: https://developer.android.com/topic/libraries/architecture/viewmodel#java [Accessed 31 December 2019].

Android Documentation, 2020a. Save data in a local database using Room. WWW document. Available at https://developer.android.com/training/data-storage/room?hl=en [Accessed 1 March 2020]

Android Documentation, 2020b. Manual dependency injection. WWW document. Available at https://developer.android.com/training/dependency-injection/manual?hl=en [Accessed 29 April 2020]

Android Documentation, 2020c. Manual dependency injection. WWW document. Available at https://developer.android.com/training/dependency-injection/dagger-basics?hl=en [Accessed 29 April 2020]

Android Documentation, 2020d. Dagger basics. WWW document. Available at https://developer.android.com/training/dependency-injection/dagger-basics?hl=en [Accessed 30 April 2020]

Android documentation, 2020e. Guide to architecture. WWW document. Available at https://developer.android.com/jetpack/docs/guide [Accessed 30 April 2020]

Android documentation, 2020f. Dependency injection in Android. Available at https://developer.android.com/training/dependency-injection [Accessed 30 April 2020]

Android Github, 2020. Android Architecture Blueprints v2. WWW document. Available at https://github.com/android/architecture-samples/tree/dagger-android [Accessed 2 May 2020]

Amit Gupta, April 2019. “Android Vs. iOS Development – Which Platform is better and Why?”. WWW document. Available at https://blog.sagipl.com/android-vs-ios-development [Accessed 4 May 2020]

Chugh, 2019. Android Model-View-ViewModel design pattern. WWW document. Available at: https://www.journaldev.com/20292/android-Model-View-ViewModel-design-pattern [Accessed 31 December 2019].

Dagger documentation, 2020a. Multibindings. WWW document. Available at https://dagger.dev/dev-guide/multibindings.html [Accessed 2 May 2020]

Dagger documentation, 2020b. Annotation Type Binds. WWW document. https://dagger.dev/api/latest/dagger/Binds.html [Accessed 3 May 2020]

Global Stats, May 2020a. Mobile Operating System Market Share Worldwide. WWW document. https://gs.statcounter.com/os-market-share/mobile/worldwide [Accessed 4 May 2020]

Global Stats, May 2020b. Mobile Operating System Market Share Finland. WWW document. https://gs.statcounter.com/os-market-share/mobile/finland [Accessed 4 May 2020]

Google I/O, 2017. Architecture Components – Introduction. WWW document. https://www.youtube.com/watch?v=FrteWKKVyzI [Accessed 9 May 2020]

InVerita, March 2020. Flutter vs Native vs React-Native: Examining performance. WWW document. https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980 [Accessed 4 May 2020]

Wikipedia, April 2020. Singleton pattern. WWW document. Available at https://en.wikipedia.org/wiki/Singleton_pattern [Accessed 20 April 2020]

Waters, D. (ed.) 2016. Global logistics. New directions in supply chain management. 5th edition. London: Kogan Pane